PyTorch 单机多卡操作总结：分布式DataParallel，混合精度，Horovod)

CV开发者都爱看的 极市平台 2022-12-31 22:00:52 发表于广东 手机阅读

作者丨科技猛兽@知乎

来源丨https://zhuanlan.zhihu.com/p/158375055

编辑丨极市平台

极市导读

本文介绍了数种实现单机多卡操作的方法，含有大量代码，并给出了实践中作者踩过的坑及其解决方案。>>加入极市CV技术交流群，走在计算机视觉的最前沿

在上一篇文章中（https://zhuanlan.zhihu.com/p/158375254）我们看到了多GPU训练，也就是最简单的**单机多卡操作nn.DataParallel**。但是很遗憾这种操作还不够优秀，于是就有了今天这篇文章~

写这篇文章的时候看了很多的**tutorials**，附在**文末**了，在此先向文末的**每位作者致敬**，感谢大佬们！

其实单机多卡的办法**还有很多**(如下)，而且上篇的方法是相对**较慢**的。

1、nn.DataParallel 简单方便的 nn.DataParallel

2、torch.distributed 使用 torch.distributed 加速并行训练

3、apex 使用 apex 再加速。

这里，记录了使用 4 块 Tesla V100-PICE 在 ImageNet 进行了运行时间的测试，测试结果发现 **Apex 的加速效果最好，但与 Horovod/Distributed 差别不大**，平时可以直接使用内置的 Distributed。**Dataparallel 较慢，不推荐使用**。那好像上一篇白写了~

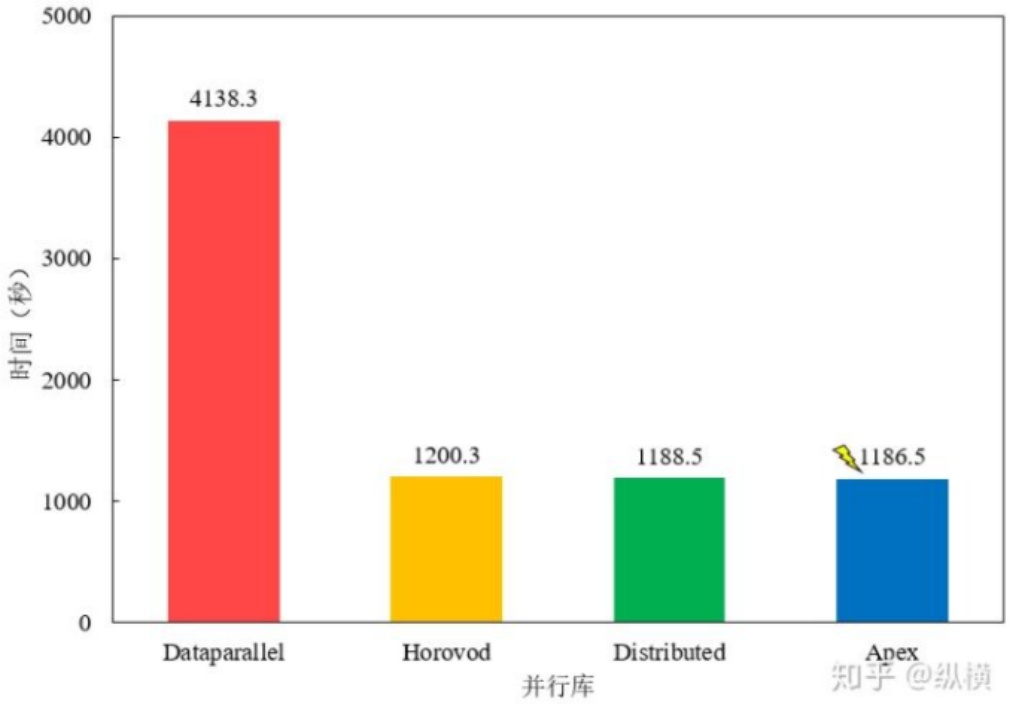


图 1：在 ImageNet 2012 上训练和测试一个 epoch 所需的时间（V100-PICE）

看到这里你可能已经懵逼了，莫慌，下面会分别进行介绍(这里先附上一篇教程，看不懂的话直接当做没看见即可)：

https://yangkky.github.io/2019/07/08/distributed-pytorch-tutorial.html

1. 先问两个问题

问1：为啥非要单机多卡?

答1：加速神经网络训练最简单的办法就是上GPU，如果一块GPU还是不够，就多上几块。

事实上，比如BERT和GPT-2这样的大型语言模型甚至是在上百块GPU上训练的。

为了实现多GPU训练，我们必须想一个办法在多个GPU上分发数据和模型，并且协调训练过程。

问2：上一篇讲得单机多卡操作nn.DataParallel，哪里不好?

答2：要回答这个问题我们得先简单回顾一下nn.DataParallel，要使用这玩意，我们将模型和数据加载到多个 GPU 中，控制数据在 GPU 之间的流动，协同不同 GPU 上的模型进行并行训练。具体怎么操作?

我们只需要用 DataParallel 包装模型，再设置一些参数即可。需要定义的参数包括：

- **参与训练的 GPU 有哪些**，device_ids=gpus。
- **用于汇总梯度的 GPU 是哪个**，output_device=gpus[0] 。

DataParallel 会自动帮我们将数据切分 load 到相应 GPU，将模型复制到相应 GPU，进行正向传播计算梯度并汇总：

```
model = nn.DataParallel(model.cuda(), device_ids=gpus, output_device=gpus[0])
```

值得注意的是，模型和数据都需要先 load 进 GPU 中，DataParallel 的 module 才能对其进行处理，否则会报错：

```
# main.py
import torch
import torch.distributed as dist

gpus = [0, 1, 2, 3]
torch.cuda.set_device('cuda:{}'.format(gpus[0]))

train_dataset = ...

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=...)

model = ...
model = nn.DataParallel(model.to(device), device_ids=gpus, output_device=gpus[0])

optimizer = optim.SGD(model.parameters())

for epoch in range(100):
   for batch_idx, (data, target) in enumerate(train_loader):
      images = images.cuda(non_blocking=True)
      target = target.cuda(non_blocking=True)
      ...
      output = model(images)
      loss = criterion(output, target)
      ...
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
```

> 稍微解释几句：model.to(device)将模型迁移到GPU里面，images.cuda，target.cuda把数据迁移到GPU里面。
> nn.DataParallel(model.to(device)，device_ids=gpus，output_device=gpus[0]) 包 装模型。

缺点：

- 在每个训练批次（batch）中，因为模型的权重都是在一个进程上先算出来，然后再把他们分发到每个GPU上，所以网络**通信**就成为了一个瓶颈，而GPU使用率也通常很低。
- 除此之外，nn.DataParallel 需要所有的GPU都在一个节点（一台机器）上，且并不支持 Apex 的 混合精度训练。

一句话，**一个进程算权重使通信成为瓶颈，**nn.DataParallel慢而且不支持混合精度训练。

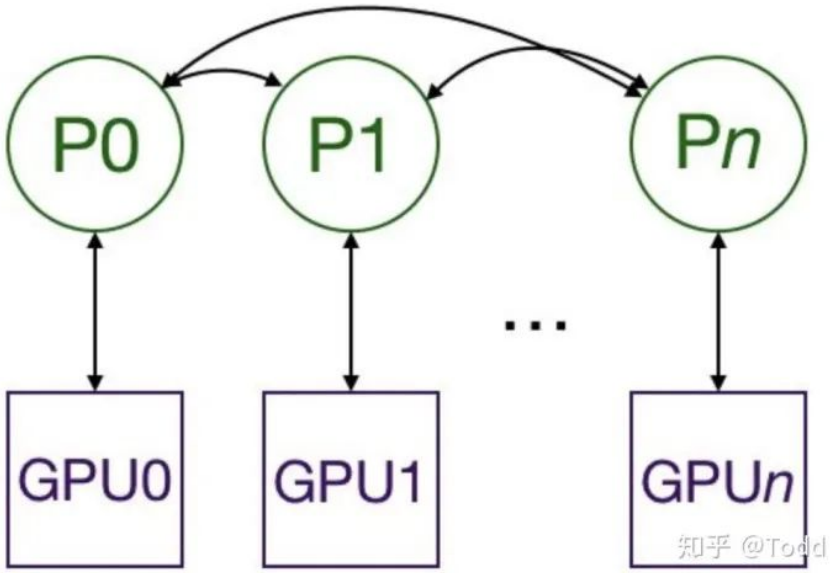
2. 使用 torch.distributed 加速并行训练：

DataParallel：单进程控制多 GPU。

DistributedDataParallel：多进程控制多 GPU，一起训练模型。

2.1 介绍

在 1.0 之后，官方终于对分布式的常用方法进行了封装，支持 all-reduce，broadcast，send 和 receive 等等。通过 MPI 实现 CPU 通信，通过 NCCL 实现 GPU 通信。官方也曾经提到用 `DistributedDataParallel` 解决 DataParallel 速度慢，GPU 负载不均衡的问题，目前已经很成熟了。



与 DataParallel 的单进程控制多 GPU 不同，在 distributed 的帮助下，我们只需要编写一份代码，torch 就会自动将其分配给n个进程，分别在n个 GPU 上运行。

和单进程训练不同的是，多进程训练需要注意以下事项：

- 在喂数据的时候，一个batch被分到了好几个进程，每个进程在取数据的时候要确保拿到的是不同的数据（ `DistributedSampler` ）；

- 要告诉每个进程自己是谁，使用哪块GPU（ `args.local_rank` ）；
- 在做BatchNormalization的时候要注意同步数据。

2.2 使用方式

2.2.1 启动方式的改变

在多进程的启动方面，我们不用自己手写 multiprocess 进行一系列复杂的CPU、GPU分配任务，PyTorch为我们提供了一个很方便的启动器 `torch.distributed.launch` 用于启动文件，所以我们运行训练代码的方式就变成了这样：

```
CUDA_VISIBLE_DEVICES=0,1,2,3 python \-m torch.distributed.launch \--nproc_per_node=4 main.py
```

其中的 `--nproc_per_node` 参数用于指定为当前主机创建的进程数，由于我们是单机多卡，所以这里node数量为1，所以我们这里设置为所使用的GPU数量即可。

2.2.2 初始化

在启动器为我们启动python脚本后，在执行过程中，启动器会将当前进程的（其实就是 GPU 的）index 通过参数传递给 python，我们可以这样获得当前进程的 index：即通过参数 `local_rank` 来告诉我们**当前进程使用的是哪个GPU**，用于我们在每个进程中指定不同的device：

```
def parse():
    parser = argparse.ArgumentParser()
    parser.add_argument('--local_rank', type=int, default=0, help='node rank for dis
    args = parser.parse_args()
    return args

def main():
    args = parse()
    torch.cuda.set_device(args.local_rank)
    torch.distributed.init_process_group(
        'nccl',
        init_method='env://'
    )
    device = torch.device(f'cuda:{args.local_rank}')
    ...
```

其中 torch.distributed.**init_process_group** 用于**初始化GPU通信方式（NCCL）和参数的获取方式（env代表通过环境变量）**。使用 init_process_group 设置GPU之间通信使用的后端和端口，通过 NCCL 实现 GPU 通信。

2.2.3 DataLoader

在读取数据的时候，我们要保证一个batch里的数据**被均摊到每个进程上**，每个进程都能获取到不同的数据，但如果我们手动去告诉每个进程拿哪些数据的话太麻烦了，PyTorch也为我们封装好了这一方法。之后，使用 **DistributedSampler 对数据集进行划分**。如此前我们介绍的那样，它能帮助我们将每个 batch 划分成几个 partition，在当前进程中只需要获取和 rank 对应的那个 partition 进行训练。

所以我们在初始化 `data loader` 的时候需要使用到 `torch.utils.data.distributed.DistributedSampler` 这个特性：

```
train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=..., sampler=tr
```

这样就能给**每个进程一个不同的 sampler**，告诉每个**进程自己分别取哪些数据**。

2.2.4 模型的初始化

和 `nn.DataParallel` 的方式一样，我们对于模型的初始化也是简单的一句话就行了

```
model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[ar
gs.local_rank])
```

使用 DistributedDataParallel 包装模型，它能帮助我们为不同 GPU 上求得的梯度进行 all reduce（即汇总不同 GPU 计算所得的梯度，并同步计算结果）。all reduce 后不同 GPU 中模型的梯度均为 all reduce 之前各 GPU 梯度的均值。

2.2.5 同步BN

为什么要同步BN?

现有的标准 Batch Normalization 因为使用数据并行（Data Parallel），是单卡的实现模式，只对单个卡上对样本进行归一化，相当于减小了批量大小（batch-size）（详见BN工作原理部分）。对于比较消耗显存的训练任务时，往往单卡上的相对批量过小，影响模型的收敛效果。之前在我们在图像语义分割的实验中，Jerry和我就发现使用大模型的效果反而变差，实际上就是BN在作怪。跨卡同步 Batch Normalization 可以使用全局的样本进行归一化，这样相当于‘增大‘了批量大小，这样训练效果不再受到使用 GPU 数量的影响。最近在图像分割、物体检

测的论文中，使用跨卡BN也会显著地提高实验效果，所以跨卡 BN 已然成为竞赛刷分、发论文的必备神器。

可惜 PyTorch 并没有为我们实现这一功能，在接下来的介绍中我们会在 apex 中看到这一功能。

2.3 汇总

至此，我们就可以使用 torch.distributed 给我们带来的多进程训练的性能提升了，汇总代码结果如下：

```
# main.py
import torch
import argparse
import torch.distributed as dist

parser = argparse.ArgumentParser()
parser.add_argument('--local_rank', default=-1, type=int,
                    help='node rank for distributed training')
args = parser.parse_args()

dist.init_process_group(backend='nccl')
torch.cuda.set_device(args.local_rank)

train_dataset = ...
#每个进程一个sampler
train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=..., sampler=tr

model = ...
model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.local_rank

optimizer = optim.SGD(model.parameters())

for epoch in range(100):
   for batch_idx, (data, target) in enumerate(train_loader):
      images = images.cuda(non_blocking=True)
      target = target.cuda(non_blocking=True)
      ...
      output = model(images)
      loss = criterion(output, target)
      ...
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
```

在使用时，调用 torch.distributed.launch 启动器启动：

```
CUDA_VISIBLE_DEVICES=0,1,2,3 python -m torch.distributed.launch --nproc_per_node=4 r
```

在 ImageNet 上的完整训练代码，请点击：

https://github.com/tczhangzhi/pytorch-distributed/blob/master/distributed.py

3. 使用 apex 再加速(混合精度训练、并行训练、同步BN)：

3.1 介绍

注：需要使用到 Volta结构 的GPU，目前只有Tesla V100和TITAN V系列支持。

Apex 是 NVIDIA 开源的用于混合精度训练和分布式训练库。Apex 对混合精度训练的过程进行了封装，改两三行配置就可以进行混合精度的训练，从而大幅度降低显存占用，节约运算时间。此外，Apex 也提供了对分布式训练的封装，针对 NVIDIA 的 NCCL 通信库进行了优化。

混合精度训练是在尽可能减少精度损失的情况下利用**半精度浮点数**加速训练。它使用FP16即半精度浮点数**存储权重和梯度**。在减少占用内存的同时起到了**加速训练**的效果。

总结下来就是两个原因：**内存占用更少，计算更快**。

- 内存占用更少：这个是显然可见的，通用的模型 fp16 占用的内存只需原来的一半。memory-bandwidth 减半所带来的好处：
  - **模型占用的内存更小**，训练的时候可以用更大的batchsize。
  - 模型训练时，**通信量（特别是多卡，或者多机多卡）大幅减少**，大幅减少等待时间，加快数据的流通。
- 计算更快：
  - 目前的不少GPU都有针对 fp16 的计算进行优化。论文指出：在近期的GPU中，半精度的计算吞吐量可以是单精度的 2-8 倍；从下图我们可以看到混合精度训练几乎没有性能损失。

Model	Baseline	Mixed Precision
AlexNet	56.77%	56.93%
VGG-D	65.40%	65.43%
GoogLeNet (Inception v1)	68.33%	68.43%
Inception v2	70.03%	70.02%
Inception v3	73.85%	74.13%
Resnet50	75.92%	76.04%

3.2 使用方式

3.2.1 混合精度

在混合精度训练上，Apex 的封装十分优雅。直接使用 `amp.initialize` 包装模型和优化器，apex 就会自动帮助我们管理模型参数和优化器的精度了，根据精度需求不同可以传入其他配置参数。

```
from apex import amp

model, optimizer = amp.initialize(model, optimizer, opt_level='O1')
```

其中 opt_level 为精度的优化设置，O0（第一个字母是大写字母O）：

- O0：纯FP32训练，可以作为accuracy的baseline；
- O1：混合精度训练（推荐使用），根据黑白名单自动决定使用FP16（GEMM, 卷积）还是FP32（Softmax）进行计算。
- O2："几乎FP16"混合精度训练，不存在黑白名单，除了Batch norm，几乎都是用FP16计算。
- O3：纯FP16训练，很不稳定，但是可以作为speed的baseline；

3.2.2 并行训练

Apex也实现了并行训练模型的转换方式，改动并不大，主要是优化了NCCL的通信，因此代码和 torch.distributed 保持一致，换一下调用的API即可：

```
from apex import amp
from apex.parallel import DistributedDataParallel

model, optimizer = amp.initialize(model, optimizer, opt_level='O1')
model = DistributedDataParallel(model, delay_allreduce=True)

# 反向传播时需要调用 amp.scale_loss，用于根据loss值自动对精度进行缩放
with amp.scale_loss(loss, optimizer) as scaled_loss:
    scaled_loss.backward()
```

3.2.3 同步BN

Apex为我们实现了同步BN，用于解决单GPU的minibatch太小导致BN在训练时不收敛的问题。

```
from apex.parallel import convert_syncbn_model
from apex.parallel import DistributedDataParallel

# 注意顺序：三个顺序不能错
model = convert_syncbn_model(UNet3d(n_channels=1, n_classes=1)).to(device)
model, optimizer = amp.initialize(model, optimizer, opt_level='O1')
model = DistributedDataParallel(model, delay_allreduce=True)
```

调用该函数后，Apex会自动遍历model的所有层，将BatchNorm层替换掉。

3.3 汇总

Apex的并行训练部分主要与如下代码段有关：

```
# main.py
import torch
import argparse
import torch.distributed as dist

from apex.parallel import convert_syncbn_model
from apex.parallel import DistributedDataParallel

parser = argparse.ArgumentParser()
parser.add_argument('--local_rank', default=-1, type=int,
                    help='node rank for distributed training')
args = parser.parse_args()

dist.init_process_group(backend='nccl')
torch.cuda.set_device(args.local_rank)

train_dataset = ...
train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=..., sampler=tr

model = ...
#同步BN
model = convert_syncbn_model(model)
#混合精度
model, optimizer = amp.initialize(model, optimizer)
#分布数据并行
model = DistributedDataParallel(model, device_ids=[args.local_rank])

optimizer = optim.SGD(model.parameters())

for epoch in range(100):
   for batch_idx, (data, target) in enumerate(train_loader):
      images = images.cuda(non_blocking=True)
      target = target.cuda(non_blocking=True)
      ...
```

```
        output = model(images)
        loss = criterion(output, target)
        optimizer.zero_grad()
        with amp.scale_loss(loss, optimizer) as scaled_loss:
           scaled_loss.backward()
        optimizer.step()
```

使用 launch 启动：

```
CUDA_VISIBLE_DEVICES=0,1,2,3 python -m torch.distributed.launch --nproc_per_node=4 r
```

4 多卡训练时的数据记录（TensorBoard、torch.save）

4.1 记录Loss曲线

在我们使用多进程时，每个进程有自己计算得到的Loss，我们在进行数据记录时，希望**对不同进程上的Loss取平均**（也就是 map-reduce 的做法），对于其他需要记录的数据也都是一样的做法：

```
def reduce_tensor(tensor: torch.Tensor) -&gt; torch.Tensor:
    rt = tensor.clone()
    distributed.all_reduce(rt, op=distributed.reduce_op.SUM)
    rt /= distributed.get_world_size()#总进程数
    return rt

# calculate loss
loss = criterion(predict, labels)
reduced_loss = reduce_tensor(loss.data)
train_epoch_loss += reduced_loss.item()
```

注意在写入TensorBoard的时候只让一个进程写入就够了：

```
# TensorBoard
if args.local_rank == 0:
    writer.add_scalars('Loss/training', {
        'train_loss': train_epoch_loss,
        'val_loss': val_epoch_loss
    }, epoch + 1)
```

4.2 torch.save

在保存模型的时候，由于是Apex混合精度模型，我们需要使用Apex提供的保存、载入方法（见Apex README）：

```
# Save checkpoint
checkpoint = {
    'model': model.state_dict(),
    'optimizer': optimizer.state_dict(),
    'amp': amp.state_dict()
}
torch.save(checkpoint, 'amp_checkpoint.pt')
...

# Restore
model = ...
optimizer = ...
checkpoint = torch.load('amp_checkpoint.pt')

model, optimizer = amp.initialize(model, optimizer, opt_level=opt_level)
model.load_state_dict(checkpoint['model'])
optimizer.load_state_dict(checkpoint['optimizer'])
amp.load_state_dict(checkpoint['amp'])

# Continue training
...
```

5 多卡后的 batch_size 和 learning_rate 的调整

见：https://www.zhihu.com/question/64134994/answer/217813386

从理论上来说，lr = batch_size * base lr，因为 batch_size 的增大会导致你 update 次数的减少，所以为了达到相同的效果，应该是同比例增大的。

但是更大的 lr 可能会导致收敛的不够好，尤其是在刚开始的时候，如果你使用很大的 lr，可能会直接爆炸，所以可能会需要一些 warmup 来逐步的把 lr 提高到你想设定的 lr。

实际应用中发现不一定要同比例增长，有时候可能增大到 batch_size/2 倍的效果已经很不错了。

在我的实验中，使用8卡训练，则增大batch_size 8倍，learning_rate 4倍是差不多的。

6 完整代码示例（Apex混合精度的Distributed DataParallel，用来训练3D U-Net的）

```
import os
import datetime
import argparse
from tqdm import tqdm
import torch
from torch import distributed, optim
from torch.utils.data import DataLoader
```

```
#每个进程不同sampler
from torch.utils.data.distributed import DistributedSampler
from torch.utils.tensorboard import SummaryWriter
#混合精度
from apex import amp
#同步BN
from apex.parallel import convert_syncbn_model
#Distributed DataParallel
from apex.parallel import DistributedDataParallel

from models import UNet3d
from datasets import IronGrain3dDataset
from losses import BCEDiceLoss
from eval import eval_net

train_images_folder = '../../datasets/IronGrain/74x320x320/train_patches/images/'
train_labels_folder = '../../datasets/IronGrain/74x320x320/train_patches/labels/'
val_images_folder = '../../datasets/IronGrain/74x320x320/val_patches/images/'
val_labels_folder = '../../datasets/IronGrain/74x320x320/val_patches/labels/'


def parse():
    parser = argparse.ArgumentParser()
    parser.add_argument('--local_rank', type=int, default=0)
    args = parser.parse_args()
    return args


def main():
    args = parse()

#设置当前进程的device，GPU通信方式为NCCL
    torch.cuda.set_device(args.local_rank)
    distributed.init_process_group(
        'nccl',
        init_method='env://'
    )

#制作Dataset和sampler
    train_dataset = IronGrain3dDataset(train_images_folder, train_labels_folder)
    val_dataset = IronGrain3dDataset(val_images_folder, val_labels_folder)
    train_sampler = DistributedSampler(train_dataset)
    val_sampler = DistributedSampler(val_dataset)

    epochs = 100
    batch_size = 8
    lr = 2e-4
    weight_decay = 1e-4
    device = torch.device(f'cuda:{args.local_rank}')

#制作DataLoader
    train_loader = DataLoader(train_dataset, batch_size=batch_size, num_workers=4,
                              pin_memory=True, sampler=train_sampler)
    val_loader = DataLoader(val_dataset, batch_size=batch_size, num_workers=4,
```

```
                                pin_memory=True, sampler=val_sampler)

    #3步曲：同步BN，初始化amp，DistributedDataParallel封装
        net = convert_syncbn_model(UNet3d(n_channels=1, n_classes=1)).to(device)
        optimizer = optim.Adam(net.parameters(), lr=lr, weight_decay=weight_decay)
        net, optimizer = amp.initialize(net, optimizer, opt_level='O1')
        net = DistributedDataParallel(net, delay_allreduce=True)
        scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[25, 50, 75], g
        criterion = BCEDiceLoss().to(device)

        if args.local_rank == 0:
            print(f'''Starting training:
                Epochs:          {epochs}
                Batch size:      {batch_size}
                Learning rate:   {lr}
                Training size:   {len(train_dataset)}
                Validation size: {len(val_dataset)}
                Device:          {device.type}
            ''')
            writer = SummaryWriter(
                log_dir=f'runs/irongrain/unet3d_32x160x160_BS_{batch_size}_{datetime.dat
            )
        for epoch in range(epochs):
            train_epoch_loss = 0
            with tqdm(total=len(train_dataset), desc=f'Epoch {epoch + 1}/{epochs}', unit

                images = None
                labels = None
                predict = None

                # train
                net.train()
                for batch_idx, batch in enumerate(train_loader):
                    images = batch['image']
                    labels = batch['label']
                    images = images.to(device, dtype=torch.float32)
                    labels = labels.to(device, dtype=torch.float32)

                    predict = net(images)

                    # calculate loss
                    # reduce不同进程的loss
                    loss = criterion(predict, labels)
                    reduced_loss = reduce_tensor(loss.data)
                    train_epoch_loss += reduced_loss.item()

                    # optimize
                    optimizer.zero_grad()
                    with amp.scale_loss(loss, optimizer) as scaled_loss:
                        scaled_loss.backward()
                    optimizer.step()
                    scheduler.step()

                    # set progress bar
```

```
                pbar.set_postfix(**{'loss (batch)': loss.item()})
                pbar.update(images.shape[0])

            train_epoch_loss /= (batch_idx + 1)

            # eval
            val_epoch_loss, dice, iou = eval_net(net, criterion, val_loader, device,

            # TensorBoard
            if args.local_rank == 0:
                writer.add_scalars('Loss/training', {
                    'train_loss': train_epoch_loss,
                    'val_loss': val_epoch_loss
                }, epoch + 1)

                writer.add_scalars('Metrics/validation', {
                    'dice': dice,
                    'iou': iou
                }, epoch + 1)

                writer.add_images('images', images[:, :, 0, :, :], epoch + 1)
                writer.add_images('Label/ground_truth', labels[:, :, 0, :, :], epoch
                writer.add_images('Label/predict', torch.sigmoid(predict[:, :, 0, :,

            if args.local_rank == 0:
                torch.save(net, f'unet3d-epoch{epoch + 1}.pth')


def reduce_tensor(tensor: torch.Tensor) -> torch.Tensor:
    rt = tensor.clone()
    distributed.all_reduce(rt, op=distributed.reduce_op.SUM)
    rt /= distributed.get_world_size()#进程数
    return rt


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

7 单机多卡正确打开方式Horovod

Horovod是Uber开源的跨平台的分布式训练工具，名字来自于俄国传统民间舞蹈，舞者手牵手围成一个圈跳舞，与Horovod设备之间的通信模式很像，有以下几个特点：

1. 兼容TensorFlow、Keras和PyTorch机器学习框架。
2. 使用Ring-AllReduce算法，对比Parameter Server算法，有着无需等待，负载均衡的优点。
3. 实现简单，五分钟包教包会。（划重点）

Uber官方在git上给了很详细的例子：https://github.com/horovod/horovod/tree/master/exa

mples，所以这里只简单讲一下大概的使用方法：

TensorFlow

以TF的Custom Training Loop API为例：

```
import tensorflow as tf
import horovod.tensorflow as hvd

# 1. 初始化horovod
hvd.init()
# 2. 给当前进程分配对应的gpu，local_rank()返回的是当前是第几个进程
config = tf.ConfigProto()
config.gpu_options.visible_device_list = str(hvd.local_rank())
# 3. Scale学习率，封装优化器
opt = tf.train.AdagradOptimizer(0.01 * hvd.size())
opt = hvd.DistributedOptimizer(opt)
# 4. 定义初始化的时候广播参数的hook，这个是为了在一开始的时候同步各个gpu之间的参数
hooks = [hvd.BroadcastGlobalVariablesHook(0)]
# 搭建model，定义loss
loss = ...
train_op = opt.minimize(loss)
# 5. 只保存一份ckpt就行
checkpoint_dir = '/tmp/train_logs' if hvd.rank() == 0 else None
# 7. 用MonitoredTrainingSession实现初始化，读写ckpt
with tf.train.MonitoredTrainingSession(checkpoint_dir=checkpoint_dir,
                                       config=config,
                                       hooks=hooks) as mon_sess:
  while not mon_sess.should_stop():
    # Perform synchronous training.
    mon_sess.run(train_op)
```

具体的代码看 tensorflow_mnist.py ： https://github.com/horovod/horovod/blob/master/examples/tensorflow_mnist.py

单机双卡训练输入以下命令：

```
CUDA_VISIBLE_DEVICES=6,7 horovodrun -np 2 -H localhost:2 python tensorflow_mnist.py
```

这里 -np 指的是进程的数量。

执行之后可以看到如下的结果，因为多线程，每个step都打印了两遍。

```
[1,0]<stderr>:INFO:tensorflow:loss = 0.13126025, step = 300 (0.191 sec)
[1,1]<stderr>:INFO:tensorflow:loss = 0.01396352, step = 310 (0.177 sec)
[1,0]<stderr>:INFO:tensorflow:loss = 0.063738815, step = 310 (0.182 sec)
[1,1]<stderr>:INFO:tensorflow:loss = 0.044452004, step = 320 (0.215 sec)
```

```
[1,0]<stderr>:INFO:tensorflow:loss = 0.028987963, step = 320 (0.212 sec)
[1,0]<stderr>:INFO:tensorflow:loss = 0.09094897, step = 330 (0.206 sec)
[1,1]<stderr>:INFO:tensorflow:loss = 0.11366991, step = 330 (0.210 sec)
[1,0]<stderr>:INFO:tensorflow:loss = 0.08559138, step = 340 (0.200 sec)
[1,1]<stderr>:INFO:tensorflow:loss = 0.037002128, step = 340 (0.201 sec)
[1,0]<stderr>:INFO:tensorflow:loss = 0.15422738, step = 350 (0.181 sec)
[1,1]<stderr>:INFO:tensorflow:loss = 0.06424393, step = 350 (0.179 sec)
```

PyTorch

Torch下也是类似的套路，但是由于PyTorch本身单机多卡训练已经够简单了，API也稳定，所以笔者一般做的时候就是直接用Torch自己的 `DP` 和 `DDP` 了。

```
import torch
import horovod.torch as hvd

# 1. 初始化horovod
hvd.init()
# 2. 给当前进程分配对应的gpu，local_rank()返回的是当前是第几个进程
torch.cuda.set_device(hvd.local_rank())
# Define dataset...
train_dataset = ...
# 3. 用DistributedSampler给各个worker分数据
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset, num_replicas=hvd.size(), rank=hvd.rank())
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=..., sampler=tr
# Build model...
model = ...
model.cuda()
# 4. 封装优化器
optimizer = optim.SGD(model.parameters())
optimizer = hvd.DistributedOptimizer(optimizer, named_parameters=model.named_paramet
# 5. 初始化的时候广播参数，这个是为了在一开始的时候同步各个gpu之间的参数
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
# 训练
for epoch in range(100):
   for batch_idx, (data, target) in enumerate(train_loader):
       optimizer.zero_grad()
       output = model(data)
       loss = F.nll_loss(output, target)
       loss.backward()
       optimizer.step()
       if batch_idx % args.log_interval == 0:
           print('Train Epoch: {} [{}/{}]\tLoss: {}'.format(
               epoch, batch_idx * len(data), len(train_sampler), loss.item()))
```

总体而言，用了All-Reduce算法的API，速度应该都差不多(一开始的图。

8 把踩过的一些坑和解决办法列举在这，以避免大家以后重复踩坑

tf.contrib.distributed.MirroredStrategy 需要optimizer支持merge_call（bert实现的optimizer是直接修改apply_gradient的，所以会报错），这个时候就需要正确地修改optimizer里的_apply_dense、_apply_sparse(参考Issue 23986 和 JayYip)。或者用horovod，就可以避免这个问题。

Effective batch size，不同的多卡工具对输入的batch size的操作不一样，要确定最后进模型的effective batch size才有意义。一般来说，多进程的batch size指的是每张卡的batch size。

Learning rate scale，学习率要根据effective batch size调整。

All-Reduce由于是多进程的，数据流各自独立，为了防止同一个step多gpu的batch重叠，最好的的办法是在每个进程里根据local_rank设置shard的数据，保证各个gpu采样的数据不重叠。

为了使用horovod，新建docker container时，要加--privileged，否则会疯狂报warning，虽然没影响，但是看着难受。

Pytorch的DP多卡要注意最后一个batch的batch size不能小于gpu的数量，否则会报错，最保险的做法是drop_last，扔掉最后的batch。

并不是所有情况下All-Reduce都比PS好，比如当卡间通信用的是NVLink的时候，在gpu数量不多的情况下，数据传输的时间不是瓶颈，All-Reduce的提升就几乎没有了。

DP和DDP有一个区别在于BatchNorm。

DDP封装model后不能再改动model。

参考资料

Nicolas：分布式训练单机多卡的正确打开方式（四）

纵横：当代研究生应当掌握的并行训练方法（单机多卡）zhuanlan.zhihu.com

薰风初入弦：Pytorch中的Distributed Data Parallel与混合精度训练（Apex）

Todd：PyTorch Parallel Training（单机多卡并行、混合精度、同步BN训练指南文档）

李沐：跨卡同步 Batch Normalization

公众号

极市干货

技术干货：数据可视化必须注意的30个小技巧总结 | 如何高效实现矩阵乘？万文长字带你从CUDA初学者的角度入门

实操教程：Nvidia Jetson TX2使用TensorRT部署yolov5s模型 | 基于YOLOV5的数据集标注&训练，Windows/Linux/Jetson Nano多平台部署全流程

点击阅读原文进入CV社区

获取更多技术干货

阅读原文

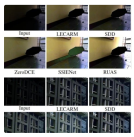
喜欢此内容的人还喜欢

YOLOv5帮助母猪产仔？南京农业大学研发母猪产仔检测模型并部署到Jetson Nano开发板

极市平台

ICCV23｜将隐式神经表征用于低光增强，北大张健团队提出NeRCo

极市平台



9个数据科学中常见距离度量总结以及优缺点概述

极市平台