实操教程 | GPU多卡并行训练总结（以pytorch为例）

CV开发者都爱看的 极市平台 2022-12-26 22:01:00 发表于广东 手机阅读

作者 | 记忆的迷谷@知乎（已授权）

来源 | https://zhuanlan.zhihu.com/p/402198819

编辑 | 极市平台

极市导读

本文的论述分为“为什么要使用多GPU并行训练”、“常见的多GPU训练方法”、“误差梯度如何在不同设备之间通信”、“BN如何在不同设备之间同步”、“两种GPU训练方法：DataParallel和DistributedDataParallel”、“pytorch中常见的GPU启动方式”六部分。>>加入极市CV技术交流群，走在计算机视觉的最前沿

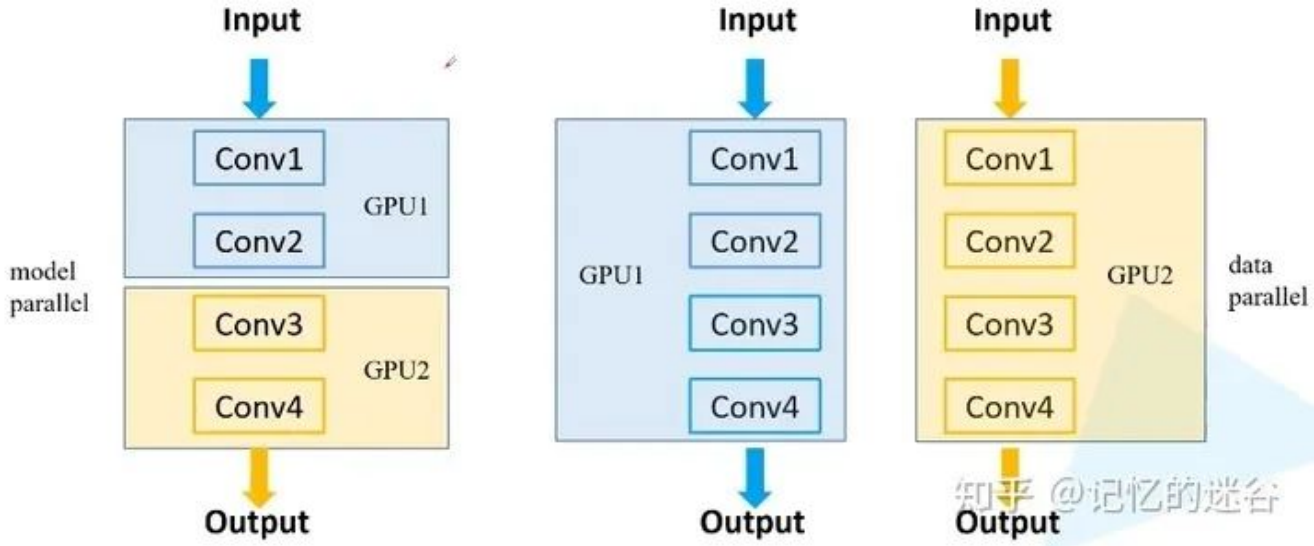
为什么要使用多GPU并行训练

简单来说，有两种原因：第一种是模型在一块GPU上放不下，两块或多块GPU上就能运行完整的模型（如早期的AlexNet）。第二种是多块GPU并行计算可以达到加速训练的效果。想要成为“炼丹大师“，多GPU并行训练是不可或缺的技能。

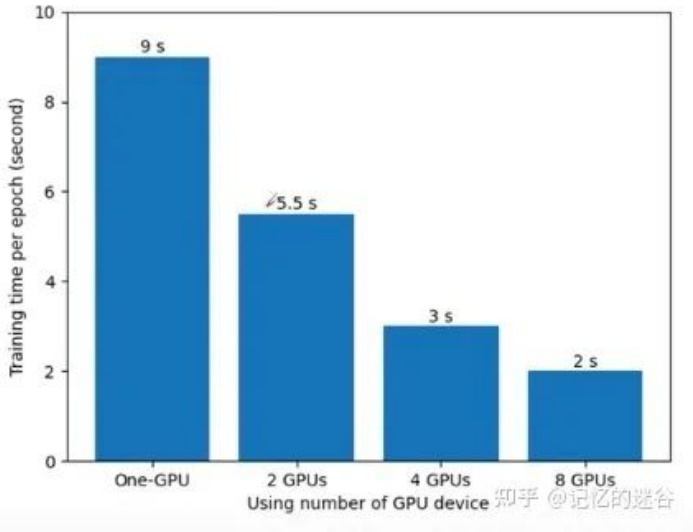
常见的多GPU训练方法：

1.**模型并行方式**：如果模型特别大，GPU显存不够，无法将一个显存放在GPU上，需要把网络的不同模块放在不同GPU上，这样可以训练比较大的网络。（下图左半部分）

2.**数据并行方式**：将整个模型放在一块GPU里，再复制到每一块GPU上，同时进行正向传播和反向误差传播。相当于加大了batch_size。（下图右半部分）



在pytorch1.7 + cuda10 + TeslaV100的环境下，使用ResNet34，batch_size=16, SGD对花草数据集训练的情况如下：使用一块GPU需要9s一个epoch，使用两块GPU是5.5s， 8块是2s。这里有一个问题，为什么运行时间不是9/8≈1.1s？因为使用GPU数量越多，设备之间的通讯会越来越复杂，所以随着GPU数量的增加，训练速度的提升也是递减的。

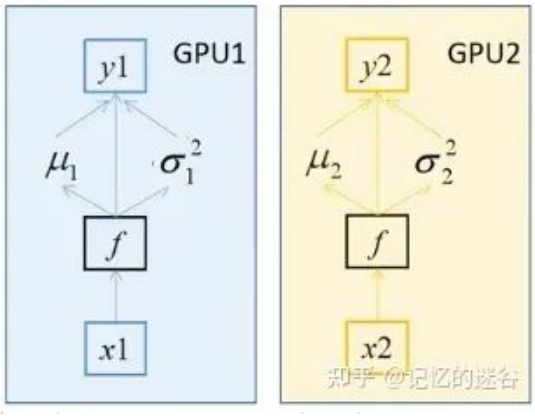


误差梯度如何在不同设备之间通信?

在每个GPU训练step结束后，将每块GPU的损失梯度求平均，而不是每块GPU各计算各的。

BN如何在不同设备之间同步?

假设batch_size=2，每个GPU计算的均值和方差都针对这两个样本而言的。而BN的特性是：batch_size越大，均值和方差越接近与整个数据集的均值和方差，效果越好。使用多块GPU时，会计算每个BN层在所有设备上输入的均值和方差。如果GPU1和GPU2都分别得到两个特征层，那么两块GPU一共计算4个特征层的均值和方差，可以认为batch_size=4。注意：如果不用同步BN，而是每个设备计算自己的批次数据的均值方差，效果与单GPU一致，仅仅能提升训练速度；如果使用同步BN，效果会有一定提升，但是会损失一部分并行速度。



下图为单GPU、以及是否使用同步BN训练的三种情况，可以看到使用同步BN（橙线）比不使用同步BN（蓝线）总体效果要好一些，不过训练时间也会更长。使用单GPU（黑线）和不使用同步BN的效果是差不多的。

两种GPU训练方法：DataParallel 和 DistributedDataParallel：

- DataParallel是单进程多线程的，仅仅能工作在单机中。而DistributedDataParallel是多进程的，可以工作在单机或多机器中。
- DataParallel通常会慢于DistributedDataParallel。所以目前主流的方法是DistributedDataParallel。

pytorch中常见的GPU启动方式：

注：distributed.launch方法如果开始训练后，手动终止程序，最好先看下显存占用情况，有小概率进程没kill的情况，会占用一部分GPU显存资源。

下面以分类问题为基准，**详细介绍使用DistributedDataParallel时的过程：**

首先要初始化各进程环境：

```
def init_distributed_mode(args):
    # 如果是多机多卡的机器，WORLD_SIZE代表使用的机器数，RANK对应第几台机器
    # 如果是单机多卡的机器，WORLD_SIZE代表有几块GPU，RANK和LOCAL_RANK代表第几块GPU
    if'RANK'in os.environ and'WORLD_SIZE'in os.environ:
```

```
        args.rank = int(os.environ["RANK"])
        args.world_size = int(os.environ['WORLD_SIZE'])
        # LOCAL_RANK代表某个机器上第几块GPU
        args.gpu = int(os.environ['LOCAL_RANK'])
    elif'SLURM_PROCID'in os.environ:
        args.rank = int(os.environ['SLURM_PROCID'])
        args.gpu = args.rank % torch.cuda.device_count()
    else:
        print('Not using distributed mode')
        args.distributed = False
        return

    args.distributed = True

    torch.cuda.set_device(args.gpu)  # 对当前进程指定使用的GPU
    args.dist_backend = 'nccl'# 通信后端，nvidia GPU推荐使用NCCL
    dist.barrier()  # 等待每个GPU都运行完这个地方以后再继续
```

在main函数初始阶段，进行以下初始化操作。需要注意的是，学习率需要根据使用GPU的张数增加。在这里使用简单的倍增方法。

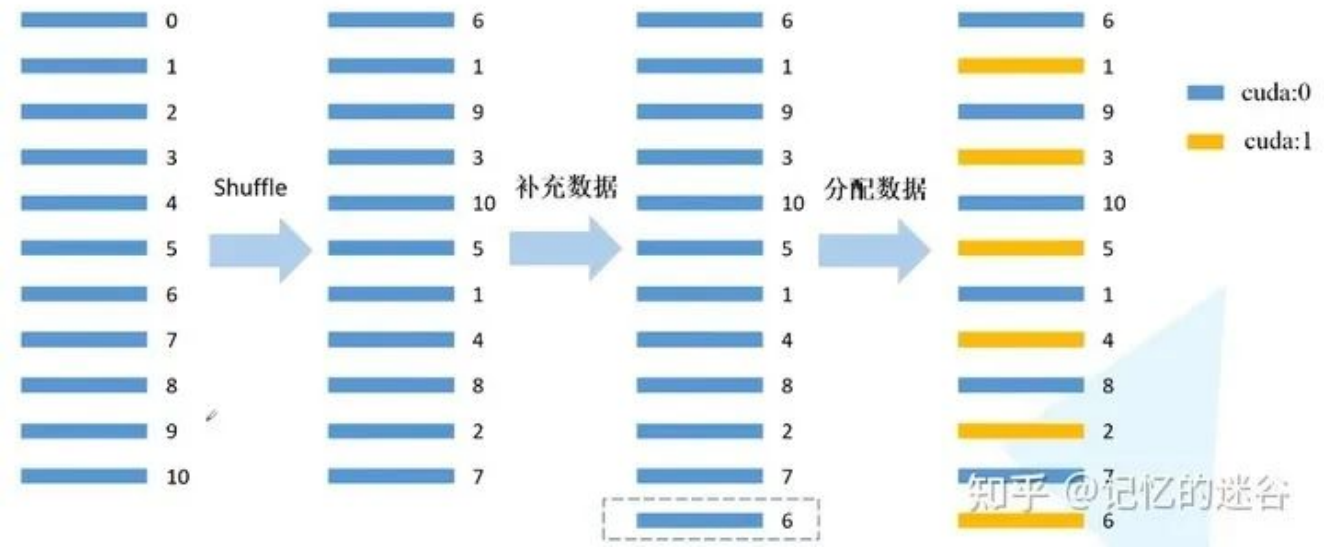
```
def main(args):
            if torch.cuda.is_available() isFalse:
                raise EnvironmentError("not find GPU device for training.")

            # 初始化各进程环境
            init_distributed_mode(args=args)

            rank = args.rank
            device = torch.device(args.device)
            batch_size = args.batch_size
            num_classes = args.num_classes
            weights_path = args.weights
            args.lr *= args.world_size  # 学习率要根据并行GPU的数倍增
```

实例化数据集可以使用单卡相同的方法，但在sample样本时，和单机不同，需要使用DistributedSampler和BatchSampler。

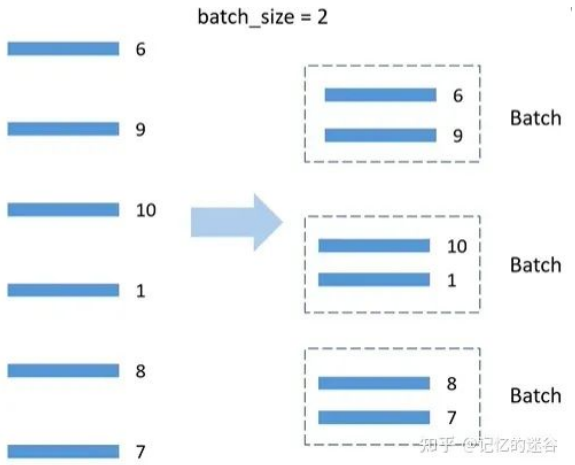
```
#给每个rank对应的进程分配训练的样本索引
train_sampler=torch.utils.data.distributed.DistributedSampler(train_data_set)
```

```
val_sampler=torch.utils.data.distributed.DistributedSampler(val_data_set)
#将样本索引每batch_size个元素组成一个list
train_batch_sampler=torch.utils.data.BatchSampler(
train_sampler,batch_size,drop_last=True)
```

DistributedSampler原理如图所示：假设当前数据集有0~10共11个样本，使用2块GPU计算。首先打乱数据顺序，然后用 11/2 =6（向上取整），然后6乘以GPU个数2 = 12，因为只有11个数据，所以再把第一个数据（索引为6的数据）补到末尾，现在就有12个数据可以均匀分到每块GPU。然后分配数据：间隔将数据分配到不同的GPU中。



BatchSampler原理: DistributedSmpler将数据分配到两个GPU上，以第一个GPU为例，分到的数据是6，9，10，1，8，7，假设batch_size=2，就按顺序把数据两两一组，在训练时，每次获取一个batch的数据，就从组织好的一个个batch中取到。注意：只对训练集处理，验证集不使用BatchSampler。



接下来使用定义好的数据集和sampler方法加载数据：

```
train_loader = torch.utils.data.DataLoader(train_data_set,
                                           batch_sampler=train_batch_sampler,
                                           pin_memory=True,    # 直接加载到显存中，达到加速
                                           num_workers=nw,
                                           collate_fn=train_data_set.collate_fn)

val_loader = torch.utils.data.DataLoader(val_data_set,
                                         batch_size=batch_size,
                                         sampler=val_sampler,
                                         pin_memory=True,
                                         num_workers=nw,
                                         collate_fn=val_data_set.collate_fn)
```

如果有预训练权重的话，需要保证每块GPU加载的权重是一模一样的。需要在主进程保存模型初始化权重，在不同设备上载入主进程保存的权重。这样才能保证每块GOU上加载的权重是一致的：

```
# 实例化模型
    model = resnet34(num_classes=num_classes).to(device)

    # 如果存在预训练权重则载入
    if os.path.exists(weights_path):
        weights_dict = torch.load(weights_path, map_location=device)
        # 简单对比每层的权重参数个数是否一致
        load_weights_dict = {k: v for k, v in weights_dict.items()
                             if model.state_dict()[k].numel() == v.numel()}
        model.load_state_dict(load_weights_dict, strict=False)
    else:
        checkpoint_path = os.path.join(tempfile.gettempdir(), "initial_weights.pt")
        # 如果不存在预训练权重，需要将第一个进程中的权重保存，然后其他进程载入，保持初始化权重一致
        if rank == 0:
            torch.save(model.state_dict(), checkpoint_path)

        dist.barrier()
        # 这里注意，一定要指定map_location参数，否则会导致第一块GPU占用更多资源
        model.load_state_dict(torch.load(checkpoint_path, map_location=device))
```

如果需要冻结模型权重，和单GPU基本没有差别。如果不需要冻结权重，可以选择是否同步BN层。然后再把模型包装成DDP模型，就可以方便进程之间的通信了。多GPU和单GPU的优化器设置没有差别，这里不再赘述。

```
# 是否冻结权重
    if args.freeze_layers:
        for name, para in model.named_parameters():
            # 除最后的全连接层外，其他权重全部冻结
            if"fc"notin name:
                para.requires_grad_(False)
    else:
        # 只有训练带有BN结构的网络时使用SyncBatchNorm采用意义
        if args.syncBN:
            # 使用SyncBatchNorm后训练会更耗时
            model = torch.nn.SyncBatchNorm.convert_sync_batchnorm(model).to(device)

        # 转为DDP模型
         model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[args.gpu])

         # optimizer使用SGD+余弦淬火策略
            pg = [p for p in model.parameters() if p.requires_grad]
            optimizer = optim.SGD(pg, lr=args.lr, momentum=0.9, weight_decay=0.005)
            lf = lambda x: ((1 + math.cos(x * math.pi / args.epochs)) / 2) * (1 - args.lrf
    scheduler = lr_scheduler.LambdaLR(optimizer, lr_lambda=lf)
```

与单GPU不同的地方：rain_sampler.set_epoch(epoch)，这行代码会在每次迭代的时候获得一个不同的生成器，每一轮开始迭代获取数据之前设置随机种子，通过改变传进的epoch参数改变打乱数据顺序。通过设置不同的随机种子，可以让不同GPU每轮拿到的数据不同。后面的部分和单GPU相同。

```
for epoch in range(args.epochs):
                train_sampler.set_epoch(epoch)

                mean_loss = train_one_epoch(model=model,
                                            optimizer=optimizer,
                                            data_loader=train_loader,
                                            device=device,
                                            epoch=epoch)

                scheduler.step()

                sum_num = evaluate(model=model,
                                   data_loader=val_loader,
                                   device=device)
                acc = sum_num / val_sampler.total_size
```

我们详细看看每个epoch是训练时和单GPU训练的差异（上面的train_one_epoch）

```
def train_one_epoch(model, optimizer, data_loader, device, epoch):
            model.train()
            loss_function = torch.nn.CrossEntropyLoss()
            mean_loss = torch.zeros(1).to(device)
            optimizer.zero_grad()

            # 在进程0中打印训练进度
            if is_main_process():
                data_loader = tqdm(data_loader)

            for step, data in enumerate(data_loader):
                images, labels = data

                pred = model(images.to(device))

                loss = loss_function(pred, labels.to(device))
                loss.backward()
                loss = reduce_value(loss, average=True)  #  在单GPU中不起作用，多GPU时，获得所
                mean_loss = (mean_loss * step + loss.detach()) / (step + 1)  # update mear

                # 在进程0中打印平均loss
                if is_main_process():
                    data_loader.desc = "[epoch {}] mean loss {}".format(epoch, round(mean_

                ifnot torch.isfinite(loss):
                    print('WARNING: non-finite loss, ending training ', loss)
                    sys.exit(1)

                optimizer.step()
                optimizer.zero_grad()

            # 等待所有进程计算完毕
            if device != torch.device("cpu"):
                torch.cuda.synchronize(device)

            return mean_loss.item()

        def reduce_value(value, average=True):
            world_size = get_world_size()
            if world_size < 2:  # 单GPU的情况
                return value
```

```
        with torch.no_grad():
            dist.all_reduce(value)   # 对不同设备之间的value求和
            if average:  # 如果需要求平均，获得多块GPU计算loss的均值
                value /= world_size

        return value
```

接下来看一下验证阶段的情况，和单GPU最大的额不同之处是预测正确样本个数的地方。

```
    @torch.no_grad()
    def evaluate(model, data_loader, device):
        model.eval()

        # 用于存储预测正确的样本个数，每块GPU都会计算自己正确样本的数量
        sum_num = torch.zeros(1).to(device)

        # 在进程0中打印验证进度
        if is_main_process():
            data_loader = tqdm(data_loader)

        for step, data in enumerate(data_loader):
            images, labels = data
            pred = model(images.to(device))
            pred = torch.max(pred, dim=1)[1]
            sum_num += torch.eq(pred, labels.to(device)).sum()

        # 等待所有进程计算完毕
        if device != torch.device("cpu"):
            torch.cuda.synchronize(device)

        sum_num = reduce_value(sum_num, average=False)  # 预测正确样本个数

    return sum_num.item()
```

需要注意的是：保存模型的权重需要在主进程中进行保存。

```
if rank == 0:
            print("[epoch {}] accuracy: {}".format(epoch, round(acc, 3)))
            tags = ["loss", "accuracy", "learning_rate"]
            tb_writer.add_scalar(tags[0], mean_loss, epoch)
```

```
            tb_writer.add_scalar(tags[1], acc, epoch)
            tb_writer.add_scalar(tags[2], optimizer.param_groups[0]["lr"], epoch)

            torch.save(model.module.state_dict(), "./weights/model-{}.pth".format(epoch))
```

如果从头开始训练，主进程生成的初始化权重是以临时文件的形式保存，需要训练完后移除掉。最后还需要撤销进程组。

```
if rank == 0:# 删除临时缓存文件         if os.path.exists(checkpoint_path) is True:
```

> 鸣谢：本博客内容借鉴于up主：霹雳吧啦Wz

极市干货

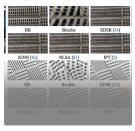
技术干货：数据可视化必须注意的30个小技巧总结 | 如何高效实现矩阵乘？万文长字带你从CUDA初学者的角度入门

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

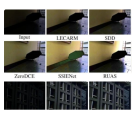
喜欢此内容的人还喜欢

ICCV 2023 | 南开程明明团队提出适用于SR任务的新颖注意力机制（已开源）

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

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