Pytorch版本：1.6.0

上版更新：nn.SyncBatchNorm

当前更新：nn.utils.data

本文档是对Pytorch官方文档和教材学习及查阅过程中的笔记，不仅对原文档做了便于理解的翻译，还对一些重要的部分进行了解释说明。

相比官方文档，本文以最简洁的方式呈现了Pytorch一些功能和API的使用方法，对于这些方法实现的细节和原理不做过多的介绍，而是给出了原文链接，需要了解的读者可自行查看；同时，本文保留了文档中的代码示例，对复杂脚本进行了代码分析，希望可以帮助读者更快地理解相关内容，节省读者的时间；最后，本项目尚未包含官方文档所有内容，仍在持续更新中。

NOTE

- 本人在查看官方文档时，有时看懂一个功能需要很长时间。所以每次都会记录下来，也因此萌生了将其总结到一起，编写一个文档的想法。既是自己的学习历程，也希望能帮到更多的初学者。
- 本文档与官方文档结构类似，对每个函数（API）都有书签直接定位，可以作为学习资料，也可作为速查手册。
- 文中添加了很多链接，以紫色字体显示，便于读者快速转到对应的官方页面，进行进一步的了解。
- 文档仍在持续更新中，由于是作者一个人在编写，又限于本人课题压力，无法做到定时更新。

最后，如果你想对本项目做出贡献，为学习者提供便利，欢迎联系我！

`415091lz@gmail.com`

仓库地址：`https://github.com/liuzhaoo/Pytorch-API-and-Tutorials-CN`

PART 1 API DOCS

TORCH

Tensors

is_tensor

判断对象是否为tensor，相当于`is_instance(obj,Tensor)`,返回bool。用法：`is_tensor(obj)`

Creation Ops

> Random sampling(随机采样)操作在 Random sampling 下，包括`torch.rand()` `torch.rand_like()` `torch.randn()` `torch.randn_like()` `torch.randint()` `torch.randint_like()` `torch.randperm()`

tensor

```
torch.tensor(data, dtype=None, device=None, requires_grad=False, pin_memory=False) → Tensor
```

使用数据来构造一个tensor,此方法会复制数据，如果输入数据是NumPy的`ndarray`,而想避免复制，可使用`torch.as_tensor`

如果数据x是tensor，则`torch.tensor(x)` 等价于`x.clone().detach()`，`torch.tensor(x,requires_grad=True)` 等价于`x.clone().detach().requires_grad_(True)` 此时推荐使用后面的方法。

Parameters

data 为要转换的数据，可以是列表等数据形式

dtype 为数据类型，*device* 用来指定设备，*requires_grad* 表示是否需要计算梯度

as_tensor

> `torch.as_tensor(data, dtype=None, device=None) → Tensor`

将数据转换为 `torch.Tensor` ，如果数据是tensor，而且与此方法指定的 *dtype* 和 *device* 相同，就不会复制tensor（此时应该只 是添加了一个新的指向）

from_numpy

> `torch.from_numpy(ndarray) → Tensor`

将 `ndarray` 类型的数据转换为tensor，返回的tensor使用原来数据的内存 。对 tensor 的修改会对原来数据产生影响，反之亦然。注意不可对返回的 tensor 做改变形状的操作

zeros/ones

> `torch.zeros` (**size*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

产生指定size的用0或1填充的tensor，一般只用第一个参数，其他参数默认

zeros_like/ones_like

> `torch.zeros_like` (*input*, *dtype=None*, *layout=None*, *device=None*, *requires_grad=False*, *memory_format=torch.preserve_format*) → Tensor

生成与 `input` 相同大小的0或1tensor（后面的各种参数也相等），等价于 `torch.zeros(input.size(), dtype=input.dtype, layout=input.layout, device=input.device)` 输入是Tensor

arange

> `torch.arange` (*start=0*, *end*, *step=1*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

返回一个一维的tensor（类似于一维列表），以 *start* 和 *end-1* 为端点，*step* 为步长，若不指定 *start* 则使用默认值0

Example:

```
>>> torch.arange(5)
tensor([ 0,  1,  2,  3,  4])
>>> torch.arange(1, 4)
tensor([ 1,  2,  3])
>>> torch.arange(1, 2.5, 0.5)
tensor([ 1.0000,  1.5000,  2.0000])
```

range

与 `arrange` 类似，但是是以*start* 和 *end* 为端点

linspace

> `torch.linspace` (*start*, *end*, *steps=100*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

生成一维的tensor，其中 *steps* 是返回tensor的元素总数，默认为100，步长计算为 $(end - start) \div (steps - 1)$

Example

```
>>> torch.linspace(3, 10, steps=5)
tensor([  3.0000,   4.7500,   6.5000,   8.2500,  10.0000])
>>> torch.linspace(-10, 10, steps=5)
tensor([-10.,  -5.,   0.,   5.,  10.])
>>> torch.linspace(start=-10, end=10, steps=5)
tensor([-10.,  -5.,   0.,   5.,  10.])
>>> torch.linspace(start=-10, end=10, steps=1)
tensor([-10.])
```

logspace

> `torch.logspace` (*start*, *end*, *steps=100*, *base=10.0*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

生成 `steps` size的一维tensor，起点是 $base^{start}$,终点是$base^{end}$, 步长计算为$base^{(end-start)\div(steps-1)}$

eye

> `torch.eye` (*n*, *m=None*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

生成二维对角矩阵，n是行数，m为可选参数，用于指定列数，默认等于n‘

empty / empty_like

返回未初始化的tensor。 `torch.empty((n,m))` `torch.empty_like(input)` input是tensor

quantize_per_tensor

> `torch.quantize_per_tensor` (*input*, *scale*, *zero_point*, *dtype*) → Tensor

将浮点型张量转换为给定scale和零点的量化张量。量化是指以低于浮点精度的位宽存储张量的技术，即可以减小模型尺寸，降低内存带宽要求，通常用于推理过程，因为不支持后向传播

$$Q(x, scale, zero\_point) = round(\frac{x}{scale} + zero\_point)$$

索引，切片，连接，变异操作

cat

> `torch.cat` (*tensors*, *dim=0*, *out=None*) → Tensor

在指定的维度（必须是给出的tensor已有的维度）上对给出的tensor进行连接

example

```
>>> x = torch.randn(2, 3)
>>> x
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
>>> torch.cat((x, x, x), 0)
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
```

```
>>> torch.cat((x, x, x), 1)
tensor([[ 0.6580, -1.0969, -0.4614,  0.6580, -1.0969, -0.4614,  0.6580,
         -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497, -0.1034, -0.5790,  0.1497, -0.1034,
         -0.5790,  0.1497]])
```

chunk

> `torch.chunk` (*input*, *chunks*, *dim=0*) → List of Tensors

cat 的反操作:在指定维度将tensor分为 chunks个tensor,若该维度的长度不能整除chunks,则最后一个取最小值.

TORCH.NN

Normalization Layers

nn.SyncBatchNorm

在N维的输入（一个具有额外的通道维度的N-2 d的mini-batch）中应用Batch Normalization。在多GPU时使用。

> CLASS `torch.nn.SyncBatchNorm` (*num_features: int*, *eps: float = 1e-05*, *momentum: float = 0.1*, *affine: bool = True*, *track_running_stats: bool = True*, *process_group: Optional[Any] = None*)

计算公式 ：

$$y = \frac{x - E(x)}{\sqrt{Var(x) + \epsilon}} \times \gamma + \beta \tag{1}$$

均值和标准差是在同一进程组里所有mini-batch的每个维度上计算得来的，γ和β是大小为C的向量。默认情况下，γ在U(0，1)上取样，β为0。标准差通过有偏估计量计算，等价于 `torch.var(input, unbiased=False)` 。

同样在默认情况下，在训练期间，这一层将继续运行其计算的平均值和方差的估计值，然后在评估期间使用这些估计值进行归一化。运行估计保持默认 `momentum` 为0.1。

如果 `track_running_stats` 设为False，这一层就不会再保留运行的估计值，而在评估期间会使用批处理统计信息。

注意，这里的 `momentum` 与优化器中和卷积层中的动量不同。

由于此处的BN是对C维中的每一个通道进行的，计算N个Batch中的（N,+）切片统计量。通常称之为容量Batch Normalization或时空Batch Normalization。

当前 `SyncBatchNorm` 只支持每个进程单个GPU的 `DistributedDataParallel` (DDP)，在使用DDP包装网络之前使用 `torch.nn.SyncBatchNorm.convert_sync_batchnorm()` 来将BN层（1d/2d/3d）转换为 `SyncBatchNorm`

参数

- **num_features** -- C (也就是通道数)
- **eps** -- 为数值稳定性而在分母上增加的值（ϵ），默认是1e-5
- **momentum** -- 用于running_mean和running_var计算的值。累积移动平均(即简单平均)可设为None，默认为0.1
- **affine** -- 布尔值，决定该模块是否有可学习的仿射参数。默认为True。
- **track_running_stats** -- 布尔值，为True时，此模块跟踪运行时的平均和方差。为False则不跟踪这些统计量，如果运行的平均值和方差都为None，则在训练和eval模式中使用批处理统计信息。
- **process_group** -- 状态同步在每个进程组中内部进行。

shape

- 输入：(N,C,+)
- 输出：(N,C,+)

实例：

```
>>> # With Learnable Parameters
>>> m = nn.SyncBatchNorm(100)
>>> # creating process group (optional)
>>> # process_ids is a list of int identifying rank ids.
>>> process_group = torch.distributed.new_group(process_ids)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm3d(100, affine=False, process_group=process_group)
>>> input = torch.randn(20, 100, 35, 45, 10)
>>> output = m(input)

>>> # network is nn.BatchNorm layer
>>> sync_bn_network = nn.SyncBatchNorm.convert_sync_batchnorm(network,
process_group)
```

```
>>> # only single gpu per process is currently supported
>>> ddp_sync_bn_network = torch.nn.parallel.DistributedDataParallel(
>>>                         sync_bn_network,
>>>                         device_ids=[args.local_rank],
>>>                         output_device=args.local_rank)
```

> 对象的方法 `convert_sync_batchnorm`(*module*, *process_group=None*)

将模型中的BN层都转换为 `torch.nn.SyncBatchNorm` 的函数

参数

- **module**((*`nn.Module`*) -- 包含BN层的模型
- **process_group** -- 进行同步的进程组，默认为整个组

返回值

包含转换过的BN层的原始模型。

示例：

```
>>> # Network with nn.BatchNorm layer
>>> module = torch.nn.Sequential(
>>>            torch.nn.Linear(20, 100),
>>>            torch.nn.BatchNorm1d(100),
>>>          ).cuda()
>>> # creating process group (optional)
>>> # process_ids is a list of int identifying rank ids.
>>> process_group = torch.distributed.new_group(process_ids)
>>> sync_bn_module = torch.nn.SyncBatchNorm.convert_sync_batchnorm(module,
process_group)
```

DataParallel Layers (multi-GPU, distributed)

DataParallel

> `CLASS torch.nn.DataParallel`(*module*, *device_ids=None*, *output_device=None*, *dim=0*)

在模块级别实现数据并行。

参数

- **module**(*`Module`*) -- 需要并行的模块（通常是整个模型）
- **device_ids**(*list of python:int or* *`torch.device`*) -- CUDA 设备（默认为所有设备）
- **output_device**(*`int`* *or* *`torch.device`*) -- 外部设备（一般用不到）

例子

```
>>> net = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
>>> output = net(input_var)  # input_var can be on any device, including CPU
```

此容器通过在batch维度上将输入分布到指定的设备（gpu）上来实现给定模型的并行（其他对象被复制到每个设备中）。在前向传播过程中，每个设备都有一份完整的model，这些副本分别处理分配到的输入数据。在后向传播时，副本中的的梯度被聚集到原始moudle中。因此batchsize应该大于gpu数量

> ***警告***
>
> 在多GPU训练时，即使只有一个节点，也建议使用
>
> 参见：：`Use nn.parallel.DistributedDataParallel instead of multiprocessing or nn.DataParallel` ， `Distributed Data Parallel` .

更多信息参见官方文档： `LINK`

DistributedDataParallel

在模块级别实现基于 `torch.distributed` 的分布式数据并行结构

此容器通过batch维度中分组，将输入分割到指定的设备上，从而并行化给定模块的应用程序。模块被复制到每台机器（多节点时）和每台设备上，每个这样的副本处理输入的一部分。在后向传播期间，每个节点的梯度被平均。

另请参阅: `Basics` ， `Use nn.parallel.DistributedDataParallel instead of multiprocessing or nn.DataParallel`

想要创建此类需要先对 `torch.distributed.init_process_group()` 进行初始化

以下是使用方法：

在每个有N个GPU 的主机上，都应该创建N个进程。同时确保每个进程分别在从0到N-1的单独的GPU上工作。因此，应该分别指定工作的GPU：

```
>>> torch.cuda.set_device(i) # i为0 - N-1
```

在每个进程中，参考以下内容来构建模块

```
>>> torch.distributed.init_process_group(backend='nccl', world_size=4,
init_method='...')
>>> model = DistributedDataParallel(model, device_ids=[i], output_device=i)
```

为了在每个节点上产生多个进程，您可以使用`torch.distributed.launch`或`torch.multiprocessing.spawn`

请参阅`PyTorch分布式概述`，了解与分布式培训相关的所有功能。更多信息查看`官方文档`

实例 `link`

```
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP


def example(rank, world_size):
    # create default process group
    dist.init_process_group("gloo", rank=rank, world_size=world_size)
    # create local model
    model = nn.Linear(10, 10).to(rank)
    # construct DDP model
    ddp_model = DDP(model, device_ids=[rank])
    # define loss function and optimizer
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    # forward pass
    outputs = ddp_model(torch.randn(20, 10).to(rank))
    labels = torch.randn(20, 10).to(rank)
    # backward pass
    loss_fn(outputs, labels).backward()
    # update parameters
    optimizer.step()

def main():
    world_size = 2
    mp.spawn(example,
        args=(world_size,),
        nprocs=world_size,
        join=True)

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

torch.distributed

Backends(后端)

`torch.distributed` 支持三种内置后端，它们分别有不同的功能，下表显示哪些函数可用于CPU/CUDA张量。仅当用于构建PyTorch的实现支持时，MPI才支持CUDA。

Backend | gloo | mpi | nccl

Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓	✕	✓	?	✕	✕
recv	✓	✕	✓	?	✕	✕
broadcast	✓	✓	✓	?	✕	✓
all_reduce	✓	✓	✓	?	✕	✓
reduce	✓	✕	✓	?	✕	✓
all_gather	✓	✕	✓	?	✕	✓
gather	✓	✕	✓	?	✕	✕
scatter	✓	✕	✓	?	✕	✕
reduce_scatter	✕	✕	✕	✕	✕	✓
all_to_all	✕	✕	✓	?	✕	✕
barrier	✓	✕	✓	?	✕	✓

PyTorch distributed目前只支持Linux。默认情况下，Gloo和NCCL后端是在PyTorch distributed中构建和包含的(只有在使用CUDA构建时才使用NCCL)。MPI是一个可选的后端，只有在从源代码构建PyTorch时才能包含它。(例如，在安装了MPI的主机上构建PyTorch。)

一般来说，使用GPU进行分布式训练时，使用NCCL后端

常用环境变量

- 选择要使用的网络接口

默认情况下，NCCL和Gloo后端都会尝试查找用于通信的网络接口。如果自动检测到的结构不正确，可以可以使用以下环境变量覆盖它(每个变量适用于其各自的后端）：

- **NCCL_SOCKET_IFNAME**, 比如 `export NCCL_SOCKET_IFNAME=eth0`
- **GLOO_SOCKET_IFNAME**, 比如 `export GLOO_SOCKET_IFNAME=eth0`

如果使用Gloo后端，可以指定多个接口，用逗号分隔它们：`export GLOO_SOCKET_IFNAME=eth0,eth1,eth2,eth3` 后端将以循环方式跨这些接口分派操作。所有进程必须在此变量中指定相同数量的接口。

- 其他NCCL环境变量
- NCCL还提供了许多用于微调目的的环境变量

  常用的包括以下用于调试目的：

  - `export NCCL_DEBUG=INFO`
  - `export NCCL_DEBUG_SUBSYS=ALL`

  有关NCCL环境变量的完整列表，请参阅 `NVIDIA NCCL的官方文档`

基础

`torch.distributed` 包为在一台或多台机器上运行的多个计算节点上的多进程并行结构提供PyTorch支持和通信原语。 `torch.nn.parallel.DistributedDataParallel()` 类就是基于此功能构建的，作为任何PyTorch模型的包装来提供同步分布式训练。这不同于 `Multiprocessing package - torch.multiprocessing` 和 `torch.nn.DataParallel()` 提供的并行结构，因为它支持多台联网的机器而且用户必须显式地为每个进程启动主要训练脚本的副本。

在单机情况下， `torch.nn.parallel.DistributedDataParallel()` 与其他数据并行方式相比，仍然具有优势：

- 每个进程都有其对应的优化器（optimizer）并且在每次迭代时都执行完整的优化步骤，虽然这看起来是多余的，但是因为各个进程之间的梯度已经收集到一起并平均，因此对于每个进程都是相同的，这意味着不需要参数广播步骤，减少了节点（GPU或主机）间传输张量的时间。
- 每个进程都包含一个独立的Python解释器，消除了额外的解释器开销和来自单个Python进程驱动多个执行线程，模型副本或GPU的“GIL-thrashing”。这对于大量使用Python运行时的模型尤其重要，包括具有循环层或许多小组件的模型

初始化

在调用其他任何方法之前，需要用 `torch.distributed.init_process_group()` 对此包进行初始化，这将阻止所有进程加入。

> `torch.distributed.is_available` ()

若返回 True 则证明分布式包可以使用。目前， `torch.distributed` 支持Linux和Macos。当从源码构建Pytorch时设置 `USE_DISTRIBUTED=1` Linux的默认值为1，Macos的默认值为0

> `torch.distributed.init_process_group` (*backend*, *init_method=None*, *timeout=datetime.timedelta(0, 1800)*, *world_size=-1*, *rank=-1*, *store=None*, *group_name=''*)

初始化默认的分布式进程组，这也将同时初始化分布式包

初始化进程组的方式有两种

1. 明确指定 `store` , `rank` ,以及 `world_size`

2. 指定 `init_method` (一个URL字符串)，它指示在哪里/如何发现对等点,可以选择指定rank和world_size，或者在URL中 编码所有必需的参数并省略它们。

如果两者都没有指定，则假设 `init_method` 为' 'env://"。

参数

backend（字符串或后端端口名称），用到的后端。取决于构建时的配置，有效值包括 `mpi` , `gloo` , `nccl` ，应该为小写的形 式，也可以通过后端属性访问，比如 `Backend.GLOO` 。如果在每台机器上通过 `nccl` 后端来使用多个进程，每个进程必须独占 访问它使用的每个GPU，因为进程之间共享GPU会导致锁死。

init_method (字符串，可选)，指定如何初始化流程组的URL。如果没有指定 `init_method` 或 `store` ，默认为"env://"。与
`store` 相互排斥

world_size (整数，可选)， 参与工作的进程数，若指定了 `store` ，则此项是必须的。

rank（整数，可选），当前进程的排名，若指定了 `store` ，则此项是必须的。

store （存储，可选），所有任务都可访问的键值对，用来交换连接/地址信息。与 `init_method` 互斥

timout（时间间隔对象，可选），针对进程组执行的操作超时，默认值等于30分钟，这仅适用于`gloo`后端对于。`nccl`，只　　有在环境变量`NCCL_BLOCKING_WAIT`被设置为1时才适用。

group_name (字符串，可选，已弃用) 组名称。

> `CLASS` torch.distributed.Backend

可用后端的类似于枚举的类：GLOO,NCCL,MPI，以及其他注册的后端。

这个类的值时小写字符串，比如"gloo"。可以将它们看作属性来进行访问：`Backend.NCCL`

可以直接使用此类来解析字符串，比如，`Backend(backend_str)`会检查`backend_str`是否有效，如果是，会返回解析后的小写字符串。也可以接受大写字符串。`Backend("GLOO")`返回"`gloo`"

> `torch.distributed.get_backend` (*group= < object object>*)

返回给定进程组的后端

> `torch.distributed.get_backend` (*group= < object object>*)

返回当前进程组的rank

Rank是分配给分布式进程组中的每个进程的唯一标识符。它们是从0到world_size的连续整数。

> `torch.distributed.get_backend` (*group= < object object>*)

返回当前进程组中的进程数

> `torch.distributed.is_initialized` ()

检查默认进程组是否已初始化

> `torch.distributed.is_nccl_available` ()

检查NCCL后端是否可用

目前支持三种初始化方式

TCP 初始化

使用TCP进行初始化的方法有两种，都需要一个所有进程都可以访问的网络地址和一个设定好的的 `world_size` 。

第一种方法需要指定一个属于rank0进程的地址。此初始化方法要求所有进程都手动指定rank。

注意，在最新的分布式包中不再支持多播地址。也不赞成使用 `group_name` 。

不同进程内，均使用主进程的 `ip` 地址和 `port` ，确保每个进程能够通过一个 `master` 进行协作。该 `ip` 一般为主进程所在的主机的 `ip` ，端口号应该未被其他应用占用。实际使用时，在每个进程内运行代码，并需要为每一个进程手动指定一个 `rank` ，进程可以分布于相同或不同主机上。

```
import torch.distributed as dist

# Use address of one of the machines
dist.init_process_group(backend, init_method='tcp://10.1.1.20:23456',
rank=args.rank, world_size=4)
```

共享文件初始化

另一种初始化方法使用一个文件系统，该文件系统与所需的 `world_size` 都可以被同组组中的所有机器共享并可见。URL应以 `file: //` 开头，并包含共享文件系统上不存在的文件(在现有目录中)的路径。如果文件不存在，文件系统初始化将自动创建该文件，但不会删除该文件。因此，下一次在相同的文件路径中初始化 `init_process_group()` 之前，应该确保已经清理了文件。

```
import torch.distributed as dist

# rank should always be specified
dist.init_process_group(backend, init_method='file:///mnt/nfs/sharedfile',
                        world_size=4, rank=args.rank)
```

环境变量初始化

此方法将从环境变量中读取配置，从而可以完全自定义信息的获取方式。要设置的变量是:

- `MASTER_PORT` - 需要;，必须是机器上的rank为0的空闲端口，。
- `MASTER_ADDR` - 需要，(0级除外）；rank 0节点的地址。
- `WORLD_SIZE` - 需要，可以在这里设置，也可以在调用init函数时设置。
- `RANK` - 需要，可以在这里设置，也可以在调用init函数时设置。

等级为0的机器将用于设置所有连接。

这是默认方法，意味着不必指定 `init_method` (或者可以是 `env: //` ）。

GROUPS 组

默认情况下，集合在默认组（也叫world）上运行，并要求所有进程进入分布式函数调用。然而，一些工作负载可以从更具细粒度的通信中受益，这就是分布式组发挥作用的地方。 `new_group` 函数可以用来创建一个新的组，，这个组具有所有进程的任意子集。它返回一个不透明的组句柄，可以将其作为 `group` 的参数提供给所有集合（在某些众所周知的编程模式中，集合是交换信息的分布式函数）

> `torch.distributed.new_group` (*ranks=None*, *timeout=datetime.timedelta(0, 1800)*, *backend=None*)

创建一个新的分布式组

此函数要求主组中的所有进程(即属于分布式作业的所有进程）都进入此函数，即使它们不是该组（新建的组）的成员。此外，应在所有进程中以相同的顺序创建组。

点对点（P2P）通信

> `torch.distributed.send` (*tensor*, *dst*, *group=*, *tag=0*)

同步发送张量

参数:

- **tensor** (*Tensor*) – 准备发送的张量。
- **dst** (*int*) – 发送的目标的rank。
- **group** (*ProcessGroup__*, *optional*) – 要处理的进程组。
- **tag** (*int* , *optional*) – 用来匹配发送与远程接收的tag。

> `torch.distributed.recv` (tensor, src=None, group=, tag=0)

同步接收张量

参数:

- **tensor** (*Tensor*) – 接收到的数据转换为张量。
- **src** (*int* , *optional*) – 指定接受的来源rank。如果未指定，将接受任何进程的数据。
- **group** (*ProcessGroup__*, *optional*) – 要处理的进程组。
- **tag** (*int* , *optional*) – 用来匹配发送与远程接收的tag。

`isend()` 和 `irecv()` 使用时返回分布式请求对象。通常，此对象的类型未指定，因为它们永远不应手动创建，但它们保证支持两种方法：

- `is_completed()` - 如果操作已完成，则返回True。
- `wait()` - 将阻止该过程，直到操作完成，`is_completed()` 保证一旦返回就返回True。

同步和异步的集体操作

每个集体操作函数都支持以下两种操作：

同步操作——当 `async_op` 被设置为False时的默认模式。当函数返回时，保证执行集体操作(如果它是CUDA操作，则不一定完成，因为所有CUDA操作都是异步的），并且可以调用任何进一步的函数调用，这取决于集体操作的数据。在同步模式下，集体函数不返回任何内容。

当 `async_op` 被设置为True时为异步步操作，集合操作函数返回一个分布式请求对象。一般来说，你不需要手动创建它，它保证支持两种方法:

- `is_completed()` - 如果操作已完成，则返回True。
- `wait()` - 将阻止该过程，直到操作完成。

单GPU集体函数

若每个主机只有一个gpu，则将每个主机都看作一个进程；

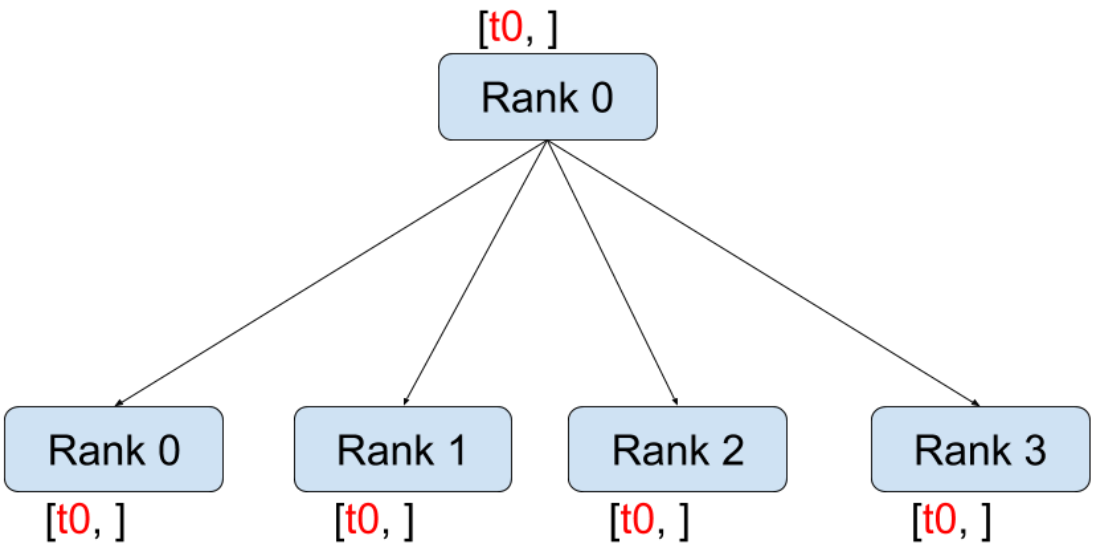
若只有一个主机，多个GPU，则将每个GPU看作一个进程

（以上为个人理解，若有错会改正）

broadcast

> `torch.distributed.broadcast` (*tensor*, *src*, *group=*, *async_op=False*)

将tensor广播到整个组中，该 `tensor` 在该组内所有对应的 `tensor` 尺寸必须一致。



参数:

- **tensor** (`Tensor`) – 如果 `src` 是当前进程的rank，则为发送的数据，否则为要接受的tensor。
- **src** (`int`) – 指定的rank。
- **group** (*ProcessGroup__, optional*) – 该操作所属于的组。
- **async_op** (`bool` , *optional*) – 这个操作是否为异步操作。

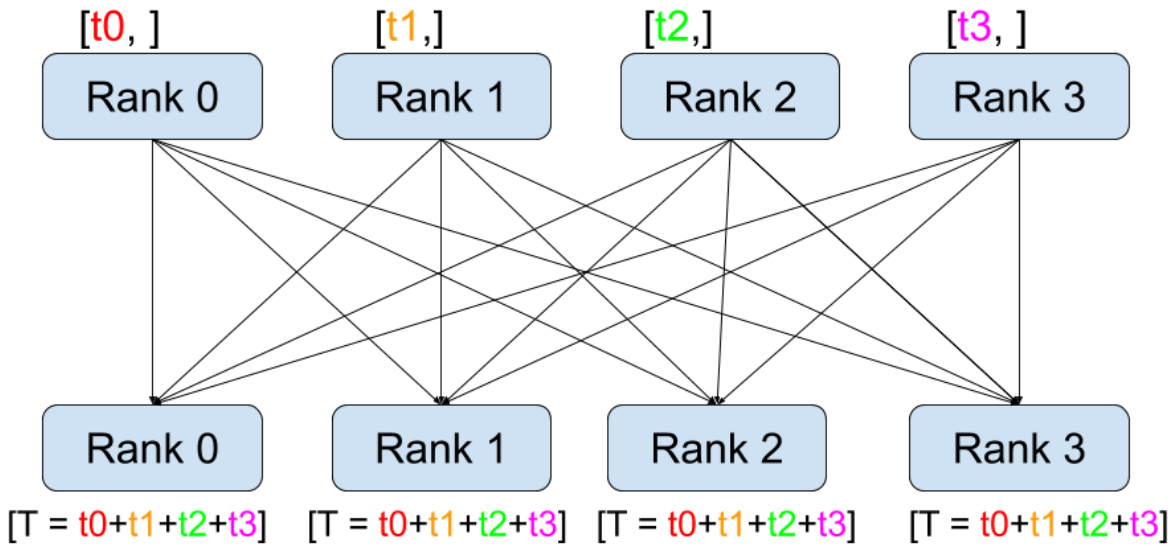
返回值

若async_op设置为True，则返回work句柄，否则为None

all_reduce

> `torch.distributed.all_reduce` (*tensor*, *op=ReduceOp.SUM*, *group=*, *async_op=False*)

对所有进程中的数据进行归纳，所有的进程都获得最终结果，此操作为inplace操作



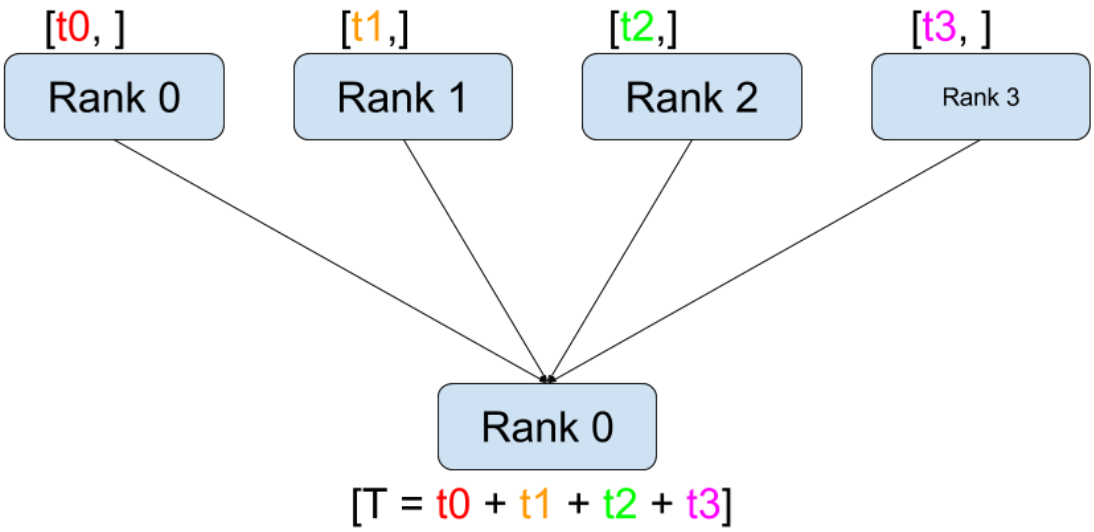
参数:

- **tensor** (`Tensor`) – 集合的输入和输出。该功能就地运行。
- **op** (*optional*) – 来自 `torch.distributed.ReduceOp` 枚举的值之一。指定归纳操作的类型，包括 `SUM, PRODUCT, MIN, MAX,BAND, BOR,BXOR` 。
- **group** (*ProcessGroup__, optional*) – 要处理的进程组。
- **async_op** (`bool` , *optional*) – 这个操作是否是异步操作。

reduce

> `torch.distributed.reduce` (*tensor*, *dst*, *op=ReduceOp.SUM*, *group=*, *async_op=False*)

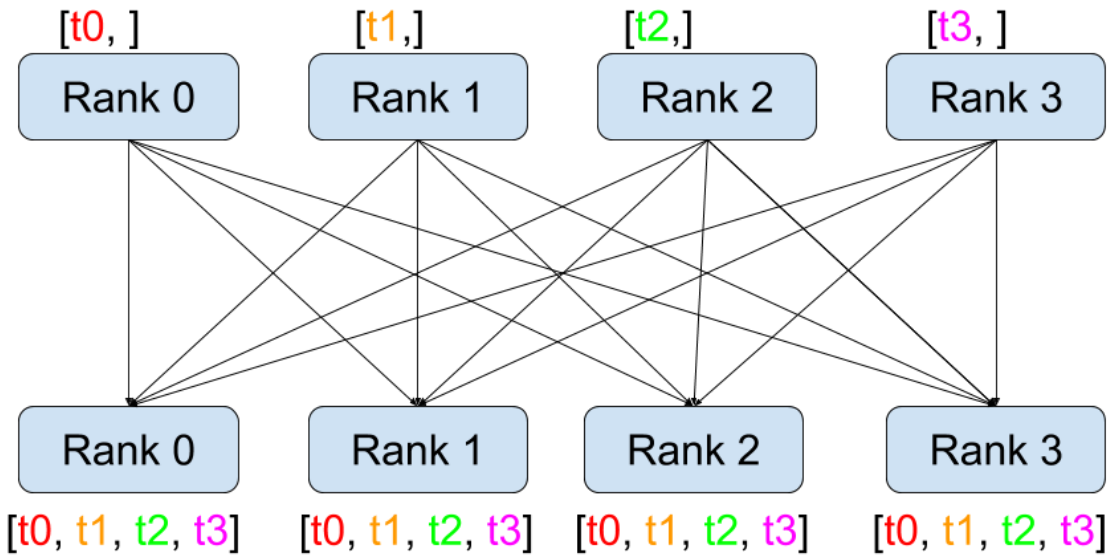
和 `all_reduce` 功能类似，不同的是，只将结果保存到 `dst` 指定的进程里



all_gather

> `torch.distributed.all_gather` (*tensor_list*, *tensor*, *group=*, *async_op=False*)

将整个组中的tensor收集到一个list中，传给所有进程，这里是对tensor进行cat，不同于reduce



参数:

- **tensor_list** (`list` [`Tensor`]) – 输出列表 `[t0,t1,t2,t3]` 。它应该包含用于集合输出的正确大小的张量。
- **tensor** (`Tensor`) – 从当前进程广播的张量 [t0] - tt3]。

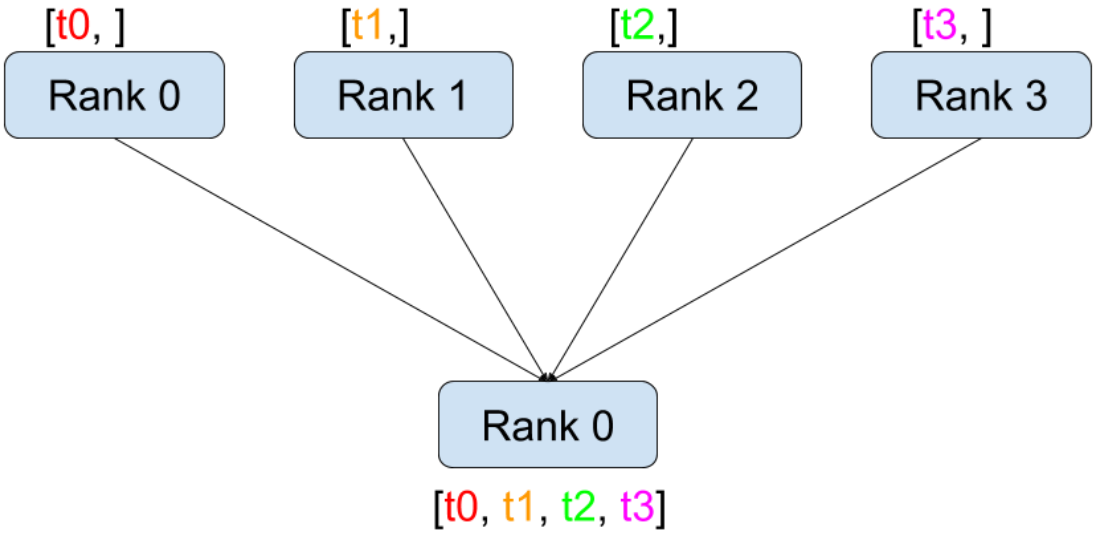
- **group** (*ProcessGroup__, optional*) – 要处理的进程组。
- **async_op** (`bool` , *optional*) – 这个操作是否是异步操作。

gather

> `torch.distributed.gather` (*tensor*, *gather_list=None*, *dst=0*, *group=*, *async_op=False*)

与all_gather类似，但是是指定进程

在发送的进程中用



参数:

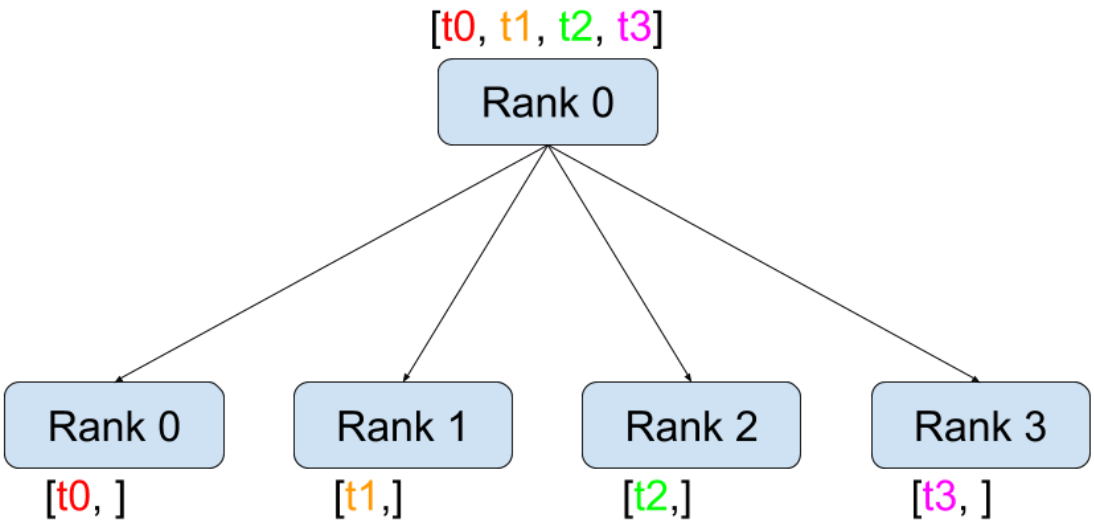
- **tensor** (`Tensor`) – 输入张量。
- **gather_list** (`list` [`Tensor`]) – 用于接收数据的适当大小的张量列表。仅在接收进程程中需要。
- **dst** (`int`) – 指定接收的进程。除接收数据的进程外，在所有进程中都是必需的。
- **group** (*ProcessGroup__, optional*) – 要处理的进程组。
- **async_op** (`bool` , *optional*) – 这个操作是否是异步操作。

scatter

> `torch.distributed.scatter` (*tensor*, *scatter_list=None*, *src=0*, *group=*, *async_op=False*)

将列表中的张量分发到组内的所有进程里（相当于gather的反操作，注意，与broadcast不同）

在接收的进程中用



参数:

- **tensor** (`Tensor`) – 输出张量。
- **scatter_list** (`list` [`Tensor`]) – 要分散的张量列表。仅在发送数据的进程中需要。
- **src** (`int`) – 指定来源进程。除发送数据的进程外，在所有进程中都是必需的。
- **group** (*ProcessGroup__, optional*) – 要处理的进程组。
- **async_op** (`bool` , *optional*) – 这个操作是否是异步操作。

> `torch.distributed.reduce_scatter` (*output*, *input_list*, *op=ReduceOp.SUM*, *group=*, *async_op=False*)

先reduce再scatter

多GPU collective函数

根据实例可以看出，在初始化时，将当前组作为rank0，也就是把这个节点当作了单进程。所以此方法应该是单进程多GPU 的方法（个人猜测）

在使用NCCL和Gloo后端时，如果每个节点（主机）上有多个GPU， `broadcast_multigpu()` `all_reduce_multigpu()` `reduce_multigpu()` `all_gather_multigpu()` 和 `reduce_scatter_multigpu()` 支持每个节点内多个GPU之间的分布式集合操作。这些函数可以潜在地提高整体分布式训练性能，并且因为它们传递的是一个张量列表，所以很容易使用。假如一个节点（主机）要调用一个函数，则该函数所传递的tensor列表的每个tensor都应该位于此主机的单独的GPU上（）。注意每个进程上的 `tensor list` 长度都必须相同

例如，假设用于训练的系统包含 `2` 个节点(`node`)，也就是主机，每个节点有 `8` 个 `GPU` 。在这 `16` 个 `GPU` 上的每一个中，有一个需要进行 `all_reduce` 的 `tensor` 。那么可以参考如下代码：

Node 0

```
import torch
import torch.distributed as dist

dist.init_process_group(backend="nccl",
                        init_method="file:///distributed_test",
                        world_size=2,
                        rank=0)
tensor_list = []
for dev_idx in range(torch.cuda.device_count()):
    tensor_list.append(torch.FloatTensor([1]).cuda(dev_idx))

dist.all_reduce_multigpu(tensor_list)
```

Node 1

```
import torch
import torch.distributed as dist

dist.init_process_group(backend="nccl",
                        init_method="file:///distributed_test",
                        world_size=2,
                        rank=1)
tensor_list = []
for dev_idx in range(torch.cuda.device_count()):
    tensor_list.append(torch.FloatTensor([1]).cuda(dev_idx))

dist.all_reduce_multigpu(tensor_list)
```

其他函数参考官方文档： `LINK`

Launch utility

`torch.distributed` 包还在 `torch.distributed.launch` 中提供了一个启动实用程序。用于在每个单节点上启动多个分布式进程。其同时支持 `Python2` 和 `Python 3` 。

`torch.distributed.launch` 是一个模块，它在每个训练节点上产生多个分布式训练进程。

该模块可用于单节点分布式培训，其中每个节点将派生一个或多个进程。它可以用于CPU训练或GPU训练。如果该实用程序用于GPU训练，则每个分布式进程将在单个GPU上运行。这样可以很好地提高单节点的训练性能。它也可以用于多节点分布式训练，通过在每个节点上生成多个进程，也可以很好地提高多节点分布式训练的性能。这对于具有直接gpu支持的多个Infiniband接口的系统尤其有利，因为所有这些接口都可以用于聚合通信带宽。

在单节点分布式训练或多节点分布式训练时，该实用程序将启动每个节点给定数量的进程(`–nproc_per_node`)。如果用于GPU训练，这个数字需要小于或等于当前系统(主机)的GPU数量(`nproc_per_node`)，GPU 0到GPU (`nproc_per_node - 1`)的每个GPU都只运行一个进程。

使用方式

1. 单节点多进程

```
>>> python -m torch.distributed.launch --nproc_per_node=NUM_GPUS_YOU_HAVE
           YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3 and all other
           arguments of your training script)
```

2. 多节点多进程

   Node 1 *(IP: 192.168.1.1, and has a free port: 1234)*

```
>>> python -m torch.distributed.launch --nproc_per_node=NUM_GPUS_YOU_HAVE
           --nnodes=2 --node_rank=0 --master_addr="192.168.1.1"
           --master_port=1234 YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --
arg3
           and all other arguments of your training script)
```

   Node 2

```
>>> python -m torch.distributed.launch --nproc_per_node=NUM_GPUS_YOU_HAVE
           --nnodes=2 --node_rank=1 --master_addr="192.168.1.1"
           --master_port=1234 YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --
arg3
           and all other arguments of your training script)
```

3. 如何查找这个模块提供的可选参数:

```
>>> python -m torch.distributed.launch --help
```

重要提示

1. 这个utilty和多进程分布式(单节点或多节点)GPU训练目前只能通过使用NCCL分布式后端来达到最好的性能。因此NCCL后端是GPU培训推荐使用的后端。

2. 在您的训练程序中，您必须解析命令行参数： `--local_rank = LOCAL_PROCESS_RANK` ，这将由此模块提供。如果您的训练程序使用GPU，则应确保您的代码仅在LOCAL_PROCESS_RANK的GPU设备上运行。这可以通过以下方式完成：

   解析local_rank参数

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument("--local_rank", type=int)
>>> args = parser.parse_args()
```

使用其中一种方法将设备设置为local rank

```
>>> torch.cuda.set_device(arg.local_rank)  # before your code runs
```

或

```
>>> with torch.cuda.device(arg.local_rank):
>>>    # your code to run
```

3. 在训练程序中，应该在开头先调用以下函数来启动分布式后端，保证 `init_method` 使用 `env: //` ，因为此模块只支持此种方法

```
torch.distributed.init_process_group(backend='YOUR BACKEND',
                                     init_method='env://')
```

4. 在您的训练计划中，您可以使用常规分布式功能或使用 `torch.nn.parallel.DistributedDataParallel()` 模块。如果您的训练计划使用GPU进行训练，并且您希望使用 `torch.nn.parallel.DistributedDataParallel()` 模块。以下是配置方式：

```
model = torch.nn.parallel.DistributedDataParallel(model,
                                                   device_ids=
[arg.local_rank],

 output_device=arg.local_rank)
```

保证 `device_ids` 参数是运行此代码的GPU的唯一的id

`local_rank` 不是全局唯一的：它只对机器上的每个进程唯一。

TORCH.UTILS.DATA

`torch.utils.data.DataLoader` 类是PyTorch数据加载功能的核心，它表示在数据集上的一个Python迭代，支持

- `map-style and iterable-style datasets`，
- `customizing data loading order`，
- `automatic batching`，

- `single- and multi-process data loading`,
- `automatic memory pinning`.

这些选项是由 `DataLoader` 的构造函数参数配置的：

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
           batch_sampler=None, num_workers=0, collate_fn=None,
           pin_memory=False, drop_last=False, timeout=0,
           worker_init_fn=None)
```

以下各节将详细描述这些选项的效果和用法。

数据集类型

`DataLoader` 构造函数最重要的参数是 `dataset`，它指示要加载数据的数据集对象。PyTorch支持两种不同类型的数据集：

- `map-style datasets`

  一种映射型的数据集，使用 `__getitem__()` 和 `__len__()` 协议，表示一种从indices/keys（可能为非整型）到数据样本的映射

  比如有这样一个数据集，当访问 `dataset[idx]` 时，可以从磁盘上的文件夹读取到第 `idx` 个图像以及与它相关的标签。

  参考 `Dataset` 以学习更多细节

- `iterable-style datasets`

  这类数据集是 `IterableDataset` 的子类的一个实例，使用 `__iter__()` 协议，表示可在数据样本上迭代。这种类型的数据集特别适合于很难甚至无法进行随机读取，以及BatchSize的大小取决于获取的数据的情况。

  比如调用 `iter(dataset)` 时，可以返回从数据库、远程服务器读取的数据流，甚至实时生成的日志。

  参考 `IterableDataset` 以学习更多细节

- ***NOTE***

  在使用 `multi-process data loading` 的情况下使用 `IterableDataset` 时，由于在每个工作进程上都复制相同的数据集对象，因此必须对副本进行不同的配置，以避免数据重复。

数据加顺序以及 `Sampler`

对于 `iterable-style` 数据集，数据加载顺序完全由用户定义的iterable控制，这使得更加容易实现块读取和动态batch size (例如，每次分批取样)。

本节剩下的部分是关于 `map-style` 数据集的， `torch.utils.data.Sampler` 类用于指定数据加载时使用的索引/键的顺序。 这些类表示数据集索引上的可迭代对象（迭代索引）。例如，在常见的随机梯度下降（SGD）的情况下，一个 `Sample` r可以随机排列一列索引值并且每次产生一个或少量的索引来进行mini-batchSGD

根据 `DataLoader` 类的 `shuffle` 参数可以自动构造一个 `sequential` 或 `shuffled` 的 `sampler` 。另外，用户也可以使用 `sampler` 参数来指定一个自定义的的 `Sampler` 对象，每次它都会产生下一个要获取的 index/key。

每次生成一个 batch indices 列表的自定义的 `Sampler` 可以作为 `batch_sampler` 参数传入 `DataLoader` 类。还可以通过 `batch_size` 和 `drop_last` 参数启用自动批处理(Automatic batching)。下一节会给出更多细节

NOTE

`sampler` 和 `batch_sampler` 都不兼容 `iterable-style` 的数据集，因为这样的数据集没有键或索引的概念。

加载分批和未分批的数据

`DataLoader` 支持通过参数 `batch_size` 、 `drop_last` 和 `batch_sampler` 自动将获取的单个数据样本排序成批。

自动成批

这是最常见的情况，对应于获取一小批数据并将它们整理成分批的样本，此样本的第一维是Batch维度。

如果 `batch_size` （默认是1）的值不是 `None` ，数据加载器会生成成批的样本，而不是单个样本。 `batch_size` 和 `drop_last` 参数用于指定数据加载器如何获得成批的数据集keys。对于 `map-style` 的数据集，用户可以选择 `batch_sampler` ，它会每次生成一个键列表。

NOTE

- `batch_size` 和 `drop_last` 参数本质上是用来从sampler参数中构造batch_sampler的。对于 `map-style` 的数据集，sampler可以由用户提供，也可以基于 `shuffle` 参数构造。

- 在使用多进程从 `iterable-style` 数据集中获取数据时， `drop_last` 参数会删除每个worker的数据集副本的最后一个非完整批。

当在sampler中使用indices获取一个列表的样本时，作为 `collate_fn` 参数传递的函数（这个函数作为参数传进类中）会将这个列表的样本排列成批。

禁用自动成批

在某些情况下，用户可能希望在数据集代码中手动处理批，或者只是加载单个示例。比如直接加载成批的数据更省力（算力）（例如，批量读取数据库或连续读取内存块），或者批大小是依赖于数据的，或者程序是设计来处理单个样本的。在这些场景下，最好不要使用自动批处理(其中collate_fn用于对样本进行排序)，而是让数据加载器直接返回数据集对象的每个成员。

当 `batch_size` 和 `batch_sampler` 都为None (`batch_sampler` 的默认值已经为None)时，自动批处理将被禁用。此时使用作为 `collate_fn` 参数传递的函数来处理从数据集获得的每个示例。这时，这个函数只是将Numpy数组转换维PyTorch的Tensor，其他保持不变。

`collate_fn` 的作用

在禁用和不禁用自动批处理时， `collate_fn` 的作用是不同的。

禁用批处理时 `collate_fn` 调用每个单独的数据样本，输出是从数据加载器迭代器中产生的。这时，这个函数只是将Numpy数组转换维PyTorch的Tensor

不禁用时 `collate_fn` 每次调用一个列表里的数据样本，它需要将输入样本整理为批，以便从数据加载器迭代器生成。本节的剩余部分描述 `collate_fn` 在这种情况下的行为。

例如，如果每个数据样本由一个3通道图像和一个完整的类标签组成，也就是说数据集的每个元素都返回一个元组（ `image, class_index` ），默认的 `collate_fn` 会将包含这样的元组的列表整理成一个批处理过的图像tensor的单独的元组以及一个批处理过的类标签Tensor。具体来说， `collate_fn` 有以下特点：

- 它总是添加一个新维度作为批处理维度。
- 它自动将NumPy数组和Python数值转换为PyTorch张量。
- 它保留了数据结构，例如，如果每个样本是一个字典，它输出具有相同键集但批处理过的张量作为值的字典(如果值不能转换成张量，则值为列表)

用户可以使用自定义的 `collate_fn` 来实现自定义批处理，例如沿第一个维度以外的维度排序、各种长度的填充序列或添加对自定义数据类型的支持。

单进程和多进程加载数据

`DataLoader` 默认使用单进程加载数据。

在一个Python进程中，全局解释器锁 `Global Interpreter Lock (GIL)` 防止跨线程完全并行化Python代码。为避免数据加载阻塞计算代码，PyTorch提供了一个简单的开关来执行多进程数据加载，只需将参数num_workers设置为一个正整数。

单进程加载数据（默认）

在此模式下，在初始化DataLoader的进程中程中完成数据获取。因此，数据加载可能会阻塞计算。然而，当用于进程间数据共享的资源（如共享内存、文件描述符)有限（不多）时，或者整个数据集很小并且可以完全加载到内存中时，此模式是首选方式。此外，单进程加载通常会显示更多可读的错误跟踪，因此对调试很有用。

多进程加载数据

将参数 `num_workers` 设置为一个正整数会开启具有指定数量的数据加载工作进程的多进程数据加载模式。

在此模式下，每当创建一个 `DataLoader` 的迭代器时(例如，当调用 `enumerate(dataLoader)` 时)，会创建 `num_workers` 个工作进程。此时， `dataset` ， `collate_fn` 和 `worker_init_fn` 被传递给每个worker，它们被用于初始化和获取数据。这意味着数据集访问和它的内部IO，以及转换(包括collate_fn)都在工作进程中运行。

`torch.utils.data.get_worker_info()` 返回工作进程中的各种有用信息(包括工作进程id、数据集副本、初始种子等)，并在主进程中返回None。用户可以在数据集代码和/或worker_init_fn中使用此函数来单独配置每个数据集副本，并确定代码是否在工作进程中运行.例如，这在对数据集分区时特别有用。

对于 `map-style` 的数据集，主进程使用 `sampler` 生成索引并将它们发送给workers。因此，随机shuffle 是在主进程中完成的，通过分配指标来引导加载。

对于 `iterable-style` 数据集，由于每个工作进程都获得了数据集对象的副本，单纯的多进程加载通常会导致重复的数据。使用 `torch.utils.data.get_worker_info()` 和/或worker_init_fn，用户可以独立配置每个副本。(参见IterableDataset文档了解如何实现这一点。)出于类似的原因，在多进程加载中， `drop_last` 参数会删除每个worker的 iterable-style数据集副本的最后一批非完整数据。

WARNING

通常不建议在多进程加载时返回CUDA张量，因为在多进程中使用CUDA和共享CUDA张量有很多微妙之处。相反，我们建议使用自动内存固定(例如，设置 `pin_memory=True`)，这可以使数据快速传输到支持cuda的gpu。

不同系统的配置

- 在Unix中， `fork()` 是默认的开始多进程的方法。使用 `fork()` ，子进程通常可以通过克隆的地址空间直接访问数据集和Python参数函数。
- 对于Windows， `spawn()` 是默认的开始多进程的方法。使用 `spawn()` ，另一个解释器会被启动，它运行main script，然后是通过pickle序列化接收数据集、collate_fn和其他参数的内部辅助函数。

这个单独的序列化意味着你应该采取两个步骤，以确保兼容Windows，同时使用多进程数据加载：

- 使用 `if __name__ == '__main__':` 包装主函数
- 确保任何自定义的 `collate_fn, worker_init_fn` 或数据集代码被声明为顶层定义，在其余的 `__main__` 检查之外。这确保了它们在辅助进程中可用。

多进程数据加载的随机性

默认情况下，每个worker都将其PyTorch种子设置为 `base_seed + worker_id` ，其中 `base_seed` 是主进程使用其RNG生成的长线程(因此，强制使用RNG状态)。然而，其他库的种子可能会在初始化worker时被复制(例如NumPy)，导致每个worker返回相同的随机数。

在 `worker_init_fn` 中，您可以使用 `torch.util .data.get_worker_info()` 或 `torch.initial_seed()` 访问每个worker的PyTorch种子集。并使用它在数据加载之前为其他库设置种子。

Memory Pinning

主机到GPU的拷贝在来自固定(页面锁定)内存时要快得多。

在加载数据时，将 `pin_memory=True` 传递给 `DataLoader` 会自动将获取的数据张量放到固定的内存中，从而使数据更快地传输到支持cuda的gpu。

默认的内存固定逻辑只能识别张量、映射和包含张量的迭代。默认情况下，如果pinning logic看到一个自定义类型的批(如果使用collate_fn返回一个自定义类型的批),或者如果这个批的每个元素都是自定义类型, pinning logic 不会识别出他们,它会返回这批没有固定的内存(或这些元素)。要为自定义的批处理或数据类型启用Memory Pinning，请在自定义类型上定义 `pin_memory()` 方法。

> CLASS `torch.utils.data.DataLoader` (*dataset*, *batch_size=1*, *shuffle=False*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=None*, *pin_memory=False*, *drop_last=False*, *timeout=0*, *worker_init_fn=None*, *multiprocessing_context=None*, *generator=None*)

数据据加载程序。组合dataset 和 sampler，并为给定的数据集提供可迭代性。

`DataLoader` 支持 `map-style` 和 `iterable-style` 的数据集，具有单进程或多进程加载、自定义加载顺序和可选的自动批处理(排序)和内存固定。

参数

- **dataset** (*`Dataset`*) – 要从中加载数据的数据集。
- **batch_size** (*`int`*, *optional*) – 每个Batch中的样本数(default: `1`).
- **shuffle** (*`bool`*, *optional*) – 设置为 `True` 来将每个epoch中的数据打乱 (default: `False`).
- **sampler** (*`Sampler`* *or Iterable*,*optional*) – 定义从数据集中抽取样本的策略.可以是任何使用了 `__len__` 的 `Iterable` 。如果指定了此项，就不能再指定 `shuffle` 。
- **batch_sampler** (*`Sampler`* *or Iterable***, *optional*) – 类似于 `sampler` , 但是每次返回一个batch的indices。 与 `batch_size` , `shuffle` , `sampler` , 以及 `drop_last` 是相互排斥的。
- **num_workers** (*`int`*, *optional*) –加载数据用的子进程数量. `0` 代表在主进程中加载数据 (default: `0`)
- **collate_fn** (*callable, optional*) – 将一个列表中的样本排合并来生成一个mini-batch的 Tensor(s). Used when using batched loading from 当在map-style 的数据集中使用批处理加载时使用。
- **pin_memory** (*`bool`*, *optional*) – 若为 `True` , 数据加载器会在返回张量之前将其复制到CUDA固定内存中。如果数据元素是自定义类型，或者 `collate_fn` 返回的是自定义类型的批，请参见下面的示例。
- **drop_last** (*`bool`*, *optional*) – 设置为 `True` 时，如果数据集size不能被batch size整除，会丢弃最后一个不完整的batch。若为 `False` 数据集size不能被batch size整除,最后一个batch会比其他的小。 (default: `False`)
- **timeout** (*numeric, optional*) –如果为正，则为从workers收集一批数据的超时的值. 永远为非负数. (default: `0`)
- **worker_init_fn** (*callable, optional*) – 如果不是 `None` ，在设置seed之后，加载数据之前，将会在每个具有id（ `[0, num_workers - 1]` 中的一个整数）子进程中调用，然后作为输入。 (default: `None`)

WARNING

- 若使用 `spawn()` 方法来启动子进程，则 `worker_init_fn` 不能为unpicklable对象（比如lambda函数）

> CLASS `torch.utils.data.Dataset`

表示数据集的抽象类。

所有表示从keys到数据样本的映射的数据集都应该继承此类。所有的子类都应该重写 `__getitem__()`，支持给定一个key，获取对应的数据样本。子类也可以选择性地重写 `__len__()`，这是通过许多Sampler实现和DataLoader的默认选项期望的返回的数据集的大小。

NOTE

`Dataloader` 默认构建一个生成整数indices的index sampler，为使它在使用非整数indices/keys的map-style的数据集时也能工作，必须提供一个自定义的sampler。

> CLASS `torch.utils.data.IterableDataset`

一个可迭代的数据集

所有表示可迭代数据样本的数据集都应该继承此类.当数据来自流时，这种形式的数据集特别有用。

所有的子类应该覆盖剩余的 `iter__()`，这会返回这个数据集中样本的迭代器。

当一个子类和DataLoader一起使用时,数据集中的每一项都将从DataLoader迭代器中生成。当 `num_workers > 0` 时，每个工作进程将拥有数据集对象的不同副本，因此通常需要独立配置每个副本，以避免从工作进程返回重复数据。 `get_worker_info()` 在工作进程中调用时，返回有关工作进程的信息。它可以用在数据集的 `__iter__()` 方法或 `DataLoader` 的 `worker_init_fn` 选项中来修改每个副本的行为。

`examples`

> CLASS `torch.utils.data.TensorDataset` (**tensors*)

包装tensors的数据集

通过沿第一维索引张量来检索每个样本。

参数

- **tensors** (*`Tensor`*) -- 和第一个维度size相同的tensors

> CLASS `torch.utils.data.ConcatDataset` (*datasets*)

数据集是多个数据集的拼接。

这个类用于组装不同的现有数据集。

参数

- **datasets** (*sequence*) -- 需要拼接的数据集的列表

> CLASS `torch.utils.data.ChainDataset` (*datasets*)

用于链接多个IterableDataset的数据集。

这个类对于组装现有的不同数据集流非常有用。链合操作是动态完成的，因此用这个类连接大型数据集将是有效的。

参数

- **datasets** (*iterable of IterableDataset*) -- 要链接在一起的数据集

> CLASS `torch.utils.data.Subset` (*dataset*, *indices*)

指定了indices的子集

参数

- **dataset** (`Dataset`) – The whole Dataset
- **indices** (*sequence*) – 在整个数据集中为子集选择的索引

> `torch.utils.data.get_worker_info` ()

返回关于当前DataLoader迭代器工作进程的信息。

在一个worker中调用时,将返回一个保证具有以下属性的对象：

- `id` ： 当前进程id
- `num_workers` ： 进程数量
- `seed` ： 当前worker的随机种子集。 这个值由主进程RNG和worker id决定。 See `DataLoader` 's documentation for more details.
- `dataset` ： 此进程中的数据集对象的副本注意，这是与主进程不同的进程中的另一个对象。

NOTE

在将 `worker_init_fn` 传进 `DataLoader` 时，这种方法可以用于以不同的方式设置每个辅助进程。例如，使用 `worker_id` 将数据集对象配置为只读取分片数据集的特定部分，或者使用seed对数据集代码中使用的其他库进行seed.

> `torch.utils.data.random_split` (*dataset*, *lengths*, *generator=<torch._C.Generator object>*)

随机地将一个数据集分割成不重叠的给定长度的新数据集。generator是可选的，以生成可复写的结果，例如:

```
>>> random_split(range(10), [3, 7],
generator=torch.Generator().manual_seed(42))
```

参数

- **dataset** (*Dataset*) – 要分割的数据集
- **lengths** (*sequence*) – 要产生的数据集的长度
- **generator** (*Generator*) –用于随机排列的生成器

CLASS `torch.utils.data.Sampler` (*data_source*)

所有Sampler的基类。

每个Sampler子类必须提供一个 `__iter__()` 方法，提供一种遍历数据集元素的indices的方法，以及一个 `__len__()` 方法，返回返回的迭代器的长度。

`__iter__()` 方法不是DataLoader严格要求的，但在涉及到DataLoader长度的任何计算中都是必需的。

CLASS `torch.utils.data.SequentialSampler` (*data_source*)

按顺序取样元素，总是按相同的顺序。

参数

- **data_source** (*Dataset*) – 要进行取样的数据集

CLASS `torch.utils.data.RandomSampler` (*data_source*, *replacement=False*, *num_samples=None*, *generator=None*)

随机取样。如果 `replacement` 为False，则从一个打乱的数据集中采样。如果 `replacement` 为True，则用户可以指定要抽取的 `num_samples` 。

参数

- **data_source** (*Dataset*) – 要进行取样的数据集
- **replacement** (*bool*) – 如果为True，抽取样品进行替换， default= `False`
- **num_samples** (*int*) –要抽取的样本数, default= `len(dataset)` . 只有当 replacement为True时，才应该指定此参数
- **generator** (*Generator*) – 用于采样的发生器。

CLASS `torch.utils.data.SubsetRandomSampler` (*indices*, *generator=None*)

从给定的索引列表中随机抽取元素，不进行替换。

参数

- **indices** (*sequence*) –indices 序列
- **generator** (`Generator`) – 用于采样的发生器。

CLASS `torch.utils.data.WeightedRandomSampler` (*weights*, *num_samples*, *replacement=True*, *generator=None*)

对给定的概率(权重)从 `[0,..,len(weights)-1]` 中取样

参数

- **weights** (*sequence*) – 一组权值序列，不需要让它们的和为1
- **num_samples** (`int`) – 要抽取的样本数
- **replacement** (`bool`) –如果为真，抽取样品进行替换。否则，不对样本进行替换，意味着当为某一行抽取样本索引时，将无法再次为该行抽取该索引。
- **generator** (`Generator`) – 用于采样的发生器。

Example

```
>>> list(WeightedRandomSampler([0.1, 0.9, 0.4, 0.7, 3.0, 0.6], 5,
replacement=True))
[4, 4, 1, 4, 5]
>>> list(WeightedRandomSampler([0.9, 0.4, 0.05, 0.2, 0.3, 0.1], 5,
replacement=False))
[0, 1, 4, 3, 2]
```

CLASS `torch.utils.data.BatchSampler` (*sampler*, *batch_size*, *drop_last*)

封装另一个采样器以生成一小批索引。

参数

- **sampler** (`Sampler` *or Iterable*) – 基础 sampler. 可以是使用了 `__len__` 的任何可迭代对象
- **batch_size** (`int`) – Size of mini-batch.
- **drop_last** (`bool`) – 若 `True` , 如果最后一批的大小小于batch_size，采样器将删除它
- Example

```
>>> list(BatchSampler(SequentialSampler(range(10)), batch_size=3,
drop_last=False))
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
>>> list(BatchSampler(SequentialSampler(range(10)), batch_size=3,
drop_last=True))
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

CLASS `torch.utils.data.distributed.DistributedSampler` (*dataset*, *num_replicas=None*, *rank=None*, *shuffle=True*, *seed=0*)

限制数据将其加载到数据集的一个子集的sampler。

它与 `torch.nn.parallel.DistributedDataParallel` 结合起来特别有用。在这种情况下，每个进程都可以将 `torch.utils.data.DistributedSampler` 类实例作为一个 `DataLoader` 的sampler进行传递，并加载其独占的原始数据集的子集。

注意假设数据集的大小是恒定的。

参数

- **dataset** –要进行取样的数据集
- **num_replicas** (*int* , *optional*) – 参与分布式训练的进程数量. By default, `rank` is retrieved from the current distributed group.
- **rank** (*int* , *optional*) –当前进程在 `num_replicas` 的Rank，默认 `rank` 从当前分布式组中检索
- **shuffle** (*bool* , *optional*) – If `True` (default), sampler 会打乱indices
- **seed** (*int* , *optional*) – 在 `shuffle=True` 时，用来打乱采样器的随机种子，这个数字在分布式组中的所有进程之间应该是相同的Default: `0` 。

在分布式模式下，在创建DataLoader迭代器之前，在每个epoch开始时调用 `set_epoch(epoch) <set_epoch>` 方法是必要的，这样可以使 shuffling在多个epoch之间正常工作。否则，将始终使用相同的顺序。

Example:

```
>>> sampler = DistributedSampler(dataset) if is_distributed else None
>>> loader = DataLoader(dataset, shuffle=(sampler is None),
...                     sampler=sampler)
>>> for epoch in range(start_epoch, n_epochs):
...     if is_distributed:
...         sampler.set_epoch(epoch)
...     train(loader)
```

PART 2 TUTORIALS

Distributed Overview

pytorch中，torch.distributed 的特性主要可以分为三个部分:

- `Distributed Data-Parallel Training` (DDP) 是一种广泛采 用的单程序多数据训练范例，在每个进程上复制模型，每个模型副本都输入不同的数据样本。DDP注重于梯度通信，以保持模型副本之间的同步，同时将梯度与梯度计算重叠来加速训练。

  相关api： `torch.nn.DataParallel` ， `torch.nn.parallel.DistributedDataParallel`

  DataParallel

  可以进行单机器多GPU的设置，只需一行代码，但是此方法性能不是最好的，因为其实现方法是在每个前向传播中都重复模型，它的单进程多线程并行结构自然会受到GIL竞争的影响。

  DistributedDataParallel

  相比于 `DataParallel` ， `DistributedDataParallel` 需要更多步骤来进行设置，比如调用 `init_process_group` ，DDP使用多进程并行结构，因此模型副本之间不存在GIL争用。此外，模型在构建DDP时广播（到各进程），而不是在每一次前向传递时广播，这也有助于加快训练速度

- `RPC-Based Distributed Training` (RPC，Remote Procedure Call) 概括来说就是不同机器训练同一个模型，使用一个一个更高级的API来自动区分在多台机器上分布的模型

  相关api： `torch.distributed.rpc`

  此方法没遇到过

- `Collective Communication` (c10d) 库支持在一组中跨进程发送张量，它支持集体通信API（如 `torch.distributed.all_reduce` 和 `torch.distributed.all_gather` ）和P2P通信API（如 `torch.distributed.send` 和 `torch.distributed.isend` ）。DDP和RPC（ `ProcessGroup Backend` ）在1.6.0版本中的c10d上建立，前者使用集体通信，后者使用P2P通信。通常，开发人员不需要直接使用这个原始通信API（c10d），因为上面的DDP和RPC特性可以用于许多分布式训练场景

数据并行训练

pytorch为数据并行训练提供了很多方法，对于从简单到复杂、从原型到生产的应用程序，通常的开发顺序是:

1. 如果GPU能够容纳模型和数据，而且不考虑训练速度，使用单设备训练。
2. 若有多个GPU并且想修改少量代码就可以加速训练，使用单个机器多GPU（`torch.nn.DataParallel`）
3. 若想更好地加速训练，可以使用单机器多GPU `torch.nn.parallel.DistributedDataParallel` 的方法，但是需要写更多代码
4. 若需要在多个机器上使用多个GPU，可以使用 `torch.nn.parallel.DistributedDataParallel` 和 `launching script`
5. 使用 `torchelastic` 来进行分布式训练（如果错误可以预测或者在训练过程中可以动态地添加或减少资源）

单机模型的最优并行方法

模型并行在分布式训练技术中得到了广泛的应用， `DataParallel` 是一种简单的并行训练方法，该特性将相同的模型复制到所有GPU，其中每个GPU处理分配给它的输入数据（平均到所有gpu）。虽然它可以显著地加速训练过程，但它在某些情况下无法工作，因为模型太大，无法装入一个GPU。这部分展示了如何通过使用并行模型解决这个问题，它将单个模型分配给在不同的GPU,而不是在每个GPU中都复制整个模型

模型并行的高级思想是将模型的不同子网络放到不同的设备上，并实现相应的前向方法，使中间输出在设备之间移动。由于模型只有一部分在单个设备上运行，因此一组设备可以共同服务于更大的模型。

基本用法

从一个包含两个线性层的模型开始。要在两个GPU上运行此模型，只需将每个线性层放在不同的GPU上，并相应地移动输入和中间输出以匹配设备。

```
import torch
import torch.nn as nn
import torch.optim as optim


class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = torch.nn.Linear(10, 10).to('cuda:0')
        self.relu = torch.nn.ReLU()
        self.net2 = torch.nn.Linear(10, 5).to('cuda:1')
```

```
    def forward(self, x):
        x = self.relu(self.net1(x.to('cuda:0')))
        return self.net2(x.to('cuda:1'))
```

上面的模型看起来很像只在一个gpu上运行，除了在合适的位置进行了 `to（device）` 调用。这也是模型中唯一需要更改的地方，后向传播和参数优化都是自动进行的（在各自的gpu上）。但是在调用loss函数时，应该确保标签与输出在同一设备上。

```
model = ToyModel()
loss_fn = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)

optimizer.zero_grad()
outputs = model(torch.randn(20, 10))  # (输出在cuda1上)
labels = torch.randn(20, 5).to('cuda:1')
loss_fn(outputs, labels).backward()
optimizer.step()
```

更多内容请参见 官方文档

DistributedDataParallel

`DistributedDataParallel` (DDP)实现了模块级的数据并行，可以跨多台机器运行。使用DDP的应用程序应该生成多个进程，并为每个进程创建一个DDP实例。DDP使用 `torch.distributed` 包中的集体通信来同步梯度和缓冲。更具体的说，DDP为 `model.parameters()` 给出的每个参数注册了一个hook，当后向传播工程中计算到相关的梯度时，这个hook就会触发。然后，DDP使用该信号（hook触发）触发进程间的梯度同步。参考 `DDP design note` 查看更多细节。

推荐使用DDP的方法是为每个模型副本（一般一个副本占用一个设备）生成一个进程，其中一个模型副本可以跨多个设备。DDP进程可以放置在同一台机器上或跨机器，但GPU设备不能跨进程共享（即一个gpu只能有一个进程）。

DataParallel 和 DistributedDataParallel的比较

尽管增加了复杂性，但是 `DistributedDataParallel` 还是更受欢迎

- 首先， `DataParallel` 是单进程、多线程的，只能在一台机器上工作，而 `DistributedDataParallel` 是多进程的，可以在单台和多台机器上训练。即使在单台机器上，由于线程之间的GIL竞争、每个迭代的复制模型以及分散输入和收集输出所带来的额外开销， `DataParallel` 通常比 `DistributedDataParallel` 慢。
- 回想一下之前的教程，如果你的模型太大，一个GPU装不下，就必须使用模型并行来将它分割到多个GPU。 `DistributedDataParallel` 支持模型并行，而目前， `DataParallel` 不支持。

当DDP与模型并行相结合时，每个DDP进程将使用模型并行，所有进程共同使用并行数据。

基本用法

要创建DDP模块，首先要正确地设置进程组。更多细节可以看`使用PyTorch编写分布式应用程序`。

```
import os
import tempfile
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
import torch.multiprocessing as mp

from torch.nn.parallel import DistributedDataParallel as DDP


def setup(rank, world_size):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'

    # initialize the process group
    dist.init_process_group("gloo", rank=rank, world_size=world_size)


def cleanup():
    dist.destroy_process_group()
```

现在创建一个模块，用DDP包装它，并为提供一些虚拟输入数据。注意，由于DDP将rank0进程的模型状态 broadcasts 给DDP构造函数中的所有其他进程，不需要担心不同的DDP进程会从不同的模型参数初始值开始运行（所有的进程初始值都是相同的）。

```
class ToyModel(nn.Module):
    def __init__(self):
        super(ToyModel, self).__init__()
        self.net1 = nn.Linear(10, 10)
        self.relu = nn.ReLU()
        self.net2 = nn.Linear(10, 5)

    def forward(self, x):
        return self.net2(self.relu(self.net1(x)))


def demo_basic(rank, world_size):
    print(f"Running basic DDP example on rank {rank}.")
```

```
    setup(rank, world_size)

    # create model and move it to GPU with id rank
    model = ToyModel().to(rank)
    ddp_model = DDP(model, device_ids=[rank])

    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(rank)
    loss_fn(outputs, labels).backward()
    optimizer.step()

    cleanup()


def run_demo(demo_fn, world_size):
    mp.spawn(demo_fn,
             args=(world_size,),
             nprocs=world_size,
             join=True)
```

在DDP中，构造函数、前向传递和后向传递都是分布式的同步点。期望不同的进程启动相同数量的同步，并以相同的顺序到达这些同步点，并在大致相同的时间进入每个同步点。否则，快速进程可能会提前到达并超时等待延迟者。因此，用户负责平衡进程之间的工作负载分布。有时，由于网络延迟、资源竞争、不可预测的工作负载等原因，处理速度不可避免地会出现偏差

保存和加载检查点

在训练和从检查点恢复时，使用 `torch.save` 和 `torch.load` 来保存和加载模块是很常见的，具体可查看 `SAVING AND LOADING MODELS` 在使用DDP时，一种优化是只将模型保存在一个进程中，然后将其加载到所有进程中，从而减少写开销。这是正确的，因为所有进程都是从相同的参数开始的，并且在向后传递时梯度是同步的，因此优化器应该保持参数设置为相同的值。如果使用这种优化，请确保在保存完成之前不会开始加载所有进程。此外，在加载模块时，需要提供一个适当的map_location参数，以防止进程进入其他设备。如果map_location丢失，`torch.load` 将首先将模块加载到CPU，然后将每个参数复制到保存它的位置，这将导致同一台机器上的所有进程使用同一组设备.

```
def demo_checkpoint(rank, world_size):
    print(f"Running DDP checkpoint example on rank {rank}.")
    setup(rank, world_size)
```

```
    model = ToyModel().to(rank)
    ddp_model = DDP(model, device_ids=[rank])

    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    CHECKPOINT_PATH = tempfile.gettempdir() + "/model.checkpoint"
    if rank == 0:
        # All processes should see same parameters as they all start from
same
        # random parameters and gradients are synchronized in backward
passes.
        # Therefore, saving it in one process is sufficient.
        torch.save(ddp_model.state_dict(), CHECKPOINT_PATH)

    # Use a barrier() to make sure that process 1 loads the model after
process
    # 0 saves it.
    dist.barrier()
    # configure map_location properly
    map_location = {'cuda:%d' % 0: 'cuda:%d' % rank}
    ddp_model.load_state_dict(
        torch.load(CHECKPOINT_PATH, map_location=map_location))

    optimizer.zero_grad()
    outputs = ddp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(rank)
    loss_fn = nn.MSELoss()
    loss_fn(outputs, labels).backward()
    optimizer.step()

    # Not necessary to use a dist.barrier() to guard the file deletion below
    # as the AllReduce ops in the backward pass of DDP already served as
    # a synchronization.

    if rank == 0:
        os.remove(CHECKPOINT_PATH)

    cleanup()
```

使用PyTorch编写分布式应用程序（torch.distributed）

在这个简短的教程中，我们将介绍PyTorch的分布式包。我们将看到如何设置分布式设置，使用不同的通信策略，并回顾包的一些内部内容。

设置

PyTorch中包含的分布式包(即 `torch.distributed`)使研究人员和实践者能够轻松地跨进程和机器集群并行化他们的计算。为此，它利用消息传递语义，允许每个进程将数据通信给任何其他进程。与multiprocessing (`torch.multiprocessing`)包相反，使用此包，进程可以使用不同的通信后端，并且不受限于在同一台机器上执行。

出于本教程的目的，我们将使用以下模板使用一台机器和派生多个进程。

```
"""run.py:"""
#!/usr/bin/env python
import os
import torch
import torch.distributed as dist
from torch.multiprocessing import Process

def run(rank, size):
    """ Distributed function to be implemented later. """
    pass

def init_process(rank, size, fn, backend='gloo'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=size)
    fn(rank, size)


if __name__ == "__main__":
    size = 2
    processes = []
    for rank in range(size):
        p = Process(target=init_process, args=(rank, size, run))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()
```

代码分析：

`init_process` 为初始化函数，先指定了master节点的地址和端口（本机），这保证通过它初始化的所有进程（本例中是两个） 都使用相同的地址和端口来通过master进行协调。然后调用 `init_process_group` ，指定了后端，rank以及总进程数。然后运行 `run` 函数（用来传递tensor）。

指定总进程数为2，然后以循环的方式创建两个进程，每个进程的内容是 `init_process` 函数（也就是两个进程中都进行初化） 得到列表 processes，包含两个进程，用 `.join` 方法添加进程。

总之，生成两个进程，它们分别设置分布式环境、初始化进程组(`dists .init_process_group`)，最后执行给定的run函数。

点对点通信

上面已有对函数的介绍 `link`

send 和 recv

从一个进程到另一个进程的数据传输称为点对点通信。这些是通过 `send` 和 `recv` 函数或它们的直接对等部分，isend和irecv实现的。

```
"""Blocking point-to-point communication."""

def run(rank, size):
    tensor = torch.zeros(1)
    if rank == 0:
        tensor += 1
        # Send the tensor to process 1
        dist.send(tensor=tensor, dst=1)
    else:
        # Receive tensor from process 0
        dist.recv(tensor=tensor, src=0)
    print('Rank ', rank, ' has data ', tensor[0])
```

代码分析:

在每个进程中都会运行此函数，若当前进程rank==0，则此进程为发送端，另一进程为接收端，对两个函数分别设置参数即可

点对点通信不再做过多叙述，见 官方文档

集体通信

与点对点通信相反，collectives可以在组内的所有流程之间通信。组是所有流程的子集。要创建一个组，我们可以向 `dist.new_group(group)` 传递一个rank的列表（指明哪些进程放入这个新组）。默认情况下，collectives在所有进程(也称为world)上执行，也就是把所有进程作为一个组。例如，为了得到所有进程中所有张量的和，我们可以使用 `dist.all_reduce(tensor, op, group)` 。

```
""" All-Reduce example."""
def run(rank, size):
    """ Simple point-to-point communication. """
    group = dist.new_group([0, 1])
    tensor = torch.ones(1)
    dist.all_reduce(tensor, op=dist.reduce_op.SUM, group=group)
    print('Rank ', rank, ' has data ', tensor[0])
```

代码分析:

首先新建一个组，使用0，1进程，然后调用 `all_reduce` 函数，将两个进程的tensor相加后再放到两个进程中。由于在每个进程 都会运行此函数，所以为了清楚地显示，在函数的最后打印出此进程的rank

具体的各种函数见 上文

分布式训练

很简单，我们想实现随机梯度下降的分布式版本。我们的脚本将让所有进程计算它们的模型（每个进程的模型相同或是部分模型）在它们的batch上的梯度，然后平均它们的梯度。为了在改变进程数量时确保类似的收敛结果，我们首先必须对数据集进行分区。（也可以使用 `tnt.dataset.SplitDataset` ）

```
""" Dataset partitioning helper """
class Partition(object):

    def __init__(self, data, index):
        self.data = data
        self.index = index

    def __len__(self):
        return len(self.index)

    def __getitem__(self, index):
        data_idx = self.index[index]
        return self.data[data_idx]
```

```
class DataPartitioner(object):

    def __init__(self, data, sizes=[0.7, 0.2, 0.1], seed=1234):
        self.data = data
        self.partitions = []
        rng = Random()
        rng.seed(seed)
        data_len = len(data)
        indexes = [x for x in range(0, data_len)]
        rng.shuffle(indexes)

        for frac in sizes:
            part_len = int(frac * data_len)
            self.partitions.append(indexes[0:part_len])
            indexes = indexes[part_len:]

    def use(self, partition):
        return Partition(self.data, self.partitions[partition])
```

实现将数据集分为三部分的功能

```
""" Partitioning MNIST """
def partition_dataset():
    dataset = datasets.MNIST('./data', train=True, download=True,
                             transform=transforms.Compose([
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.1307,), (0.3081,))
                             ]))
    size = dist.get_world_size()
    bsz = 128 / float(size)
    partition_sizes = [1.0 / size for _ in range(size)]
    partition = DataPartitioner(dataset, partition_sizes)
    partition = partition.use(dist.get_rank())
    train_set = torch.utils.data.DataLoader(partition,
                                            batch_size=bsz,
                                            shuffle=True)
    return train_set, bsz
```

代码分析

先取dataset，然后算出进程数 `size` ， `bsz` 是每个进程的batchsize， `partition_sizes` 是一个列表，有 `size` 个元素，每个元素 的值为1/size。然后调用上面的函数，将dataset分为size份，每份占1/size，再根据rank取出对应每个进程的那份。最后通过 `DataLoader` 取得 `train_set` ，注意此 `train_set` 是当前进程的数据

假设有2个副本，那么每个进程将有一个train_set为60000 / 2 = 30000个样本。上面提到，用batchsize除以副本的数量，以便保持整体的batchsize为128。

现在可以编写我们通常的前向后向优化训练代码，并添加一个函数调用来平均我们模型的梯度。

```
def run(rank, size):
    torch.manual_seed(1234)
    train_set, bsz = partition_dataset()
    model = Net()
    optimizer = optim.SGD(model.parameters(),
                          lr=0.01, momentum=0.5)

    num_batches = ceil(len(train_set.dataset) / float(bsz))
    for epoch in range(10):
        epoch_loss = 0.0
        for data, target in train_set:
            optimizer.zero_grad()
            output = model(data)
            loss = F.nll_loss(output, target)
            epoch_loss += loss.item()
            loss.backward()
            average_gradients(model)
            optimizer.step()
        print('Rank ', dist.get_rank(), ', epoch ',
              epoch, ': ', epoch_loss / num_batches)
```

代码分析

注意传入的参数中有rank，用来指定当前的循环进程。首先取出当前进程的`train_set`和`bsz`，调用模型和优化方法。算出每个 batch（每个进程的batch）里的数据量，然后进行epoch循环。在后向传播完成后，进行梯度平均，此代码如下：

```
""" Gradient averaging. """
def average_gradients(model):
    size = float(dist.get_world_size())
    for param in model.parameters():
        dist.all_reduce(param.grad.data, op=dist.reduce_op.SUM)
        param.grad.data /= size
```

`dist.all_reduce`函数将每个参数的值都加起来放到当前进程中，然后再除以平均数（进程数）。

MY NOTE

- 1. 对输入图像的预处理可以在dataset里留一个transform，然后用torchvision里的transforms：

```
from torchvision import transforms
preprocess = transforms.Compose([
transforms.Resize(256),
transforms.CenterCrop(224),
transforms.ToTensor(),
transforms.Normalize(
mean=[0.485, 0.456, 0.406],
std=[0.229, 0.224, 0.225]
)])
```

2. tensor类型之间的转换

   - 直接指定类型

```
short_points = torch.tensor([[1, 2], [3, 4]], dtype=torch.short)
double_points = torch.ones(10, 2, dtype=torch.double)
```

   - 使用api

```
double_points = torch.zeros(10, 2).double()  # 直接转换
short_points = torch.ones(10, 2).short()

double_points = torch.zeros(10, 2).to(torch.double)  # .to
short_points = torch.ones(10, 2).to(dtype=torch.short)
```

3. 加载预训练模型

```
model_path = '../data/p1ch2/horse2zebra_0.4.0.pth'
model_data = torch.load(model_path)
netG.load_state_dict(model_data)
```

4. 计算数据集图片的均值和标准差，以cifa10为例，加载出来的图片为tensor，在dim3上进行堆叠，然后只留下通道维度，再对各个通道的数据求均值和标准差。

   cifa10 的均值和标准差为：

> tensor([0.4915, 0.4823, 0.4468])
>
> tensor([0.2470, 0.2435, 0.2616])

```
from dataset import cifa10
  abspath = './'
train_dataloader = cifa10(abspath, train=True,
transform=transforms.ToTensor())
imgs = torch.stack([img_t for img_t,t in train_dataloader],dim=3)

imgs.view(3, -1).mean(dim=1)
imgs.view(3, -1).std(dim=1)
```

5. `torch.max` 需要指定维度，返回两个值，最大值及其index

6. 假如 x 为tensor ，则 `x.mean(dim)` 就是求对应维度的均值， `x.sum(dim)` 是求对应维度的数值之和。同时，这个维度会消失。

7. transform里的ToTensor除了将pic和narray类型的数据转换为tensor外，还会将最后一维的数据与第二维数据换位置。

8. `transpose` 函数转换维度：

```
# 方式1
a = torch.ones(3, 2)
a_t = torch.transpose(a, 0, 1)   # 0, 1维度互换

# 方式2
a = torch.ones(3, 2)
a_t = a.transpose(0, 1)
```

9. tensor放到GPU上，可以在定义时指定： `points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device='cuda')` 也可以使用.to操作： `points_gpu = points.to(device='cuda:0')`

10. 与numpy的交互

```
# tensor转换为numpy
points = torch.ones(3, 4)
points_np = points.numpy()

# numpy转换为tensor
points = torch.from_numpy(points_np)

points = torch.tensor(points_up) # 这种方式适用于list等类型，numpy应该也可以
```

11. tensor保存与加载

```
 # 保存
torch.save(points, '../data/p1ch3/ourpoints.t')  # 直接保存

with open('../data/p1ch3/ourpoints.t','wb') as f:     #将文件名传递给一个
描述符
    torch.save(points, f)

# 加载
points = torch.load('../data/p1ch3/ourpoints.t')
#或者
with open('../data/p1ch3/ourpoints.t','rb') as f:
    points = torch.load(f)
```