Pytorch 原生分布式

周家豪

2020 年 11 月 3 日

本文档介绍分布式原理及其训练技巧（以 Pytorch 框架为平台）。

现有的神经网络需要大量的神经元和数据，以往单机单卡的训练模式需要大量的时间进行神经网络的训练，在多机多卡的计算环境下进行分布式训练对于加速神经网络的研究尤为重要。现有神经网络训练平台 TensorFlow、PyTorch、MXNet 等均有自己的分布式训练支持，本文档以 Pytorch 为例介绍分布式原理、实现。

目录

版本说明

版本 *0.1*

介绍 Pytorch 官方 DataParallel（DP）、DistributedDataParallel 分布式（DDP）原理及其实现。

版本 *0.2*

TBA

Pytorch-DP

DP API

如何使用 **class** torch.nn.DataParallel在多卡上跑模型?

单卡，将模型和数据放在指定 GPU 上，Pytorch 默认使用单卡跑程序，如下:

```
import torch
from torchvision.models import resnet18
device = torch.device('cuda:0')
model = resnet18()
model.to(device)
input = torch.randn([1, 3, 224, 224])
input = input.to(device)
output = model(input)
```

多卡，利用 **class** torch.nn.DataParallel，通过将模型用该类包装，如下:

```
model = Model(input_size, output_size)
if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
    model = nn.DataParallel(model)
else:
    model.to(device)
```

nn.DataParallel()包装的模型，自动地将数据划分到多卡上进行执行，在多卡执行完成后，收集和整合数据返回。

DP 实现

class torch.nn.DataParallel（DP）是数据并行的模型级实现。神经网络的训练分为前传和反传，在**每次**前传时，所有运行设备得到原始模型的 **复制**，每个复制处理一部分的输入数据；在反传时，所有运行设备的梯度 **累加**到原始模型上。

DP 包装后的模型支持任意类型的参数，tensor类型按照第一个维度进行进行划分，**tuple**、**list**、**dict**是浅拷贝，其他类型是线程共享的（在前传时修改可能引起冲突）。

```
class DataParallel(Module):
```

```
    r"""
    Args:
        module (Module): module to be parallelized
        device_ids (list of int or torch.device): CUDA devices (default: all devices)
        output_device (int or torch.device): device location of output (default: device_ids[0])

    Attributes:
        module (Module): the module to be parallelized
    """

    def __init__(self, module, device_ids=None, output_device=None, dim=0):
        super(DataParallel, self).__init__()

        if not torch.cuda.is_available():
            self.module = module
            self.device_ids = []
            return

        # 不提供运行设备ids，默认使用全部GPU
        if device_ids is None:
            device_ids = list(range(torch.cuda.device_count()))
        # 不提供输出设备id，默认使用第一个运行设备
        if output_device is None:
            output_device = device_ids[0]

        self.dim = dim
        self.module = module
        self.device_ids = list(map(lambda x: _get_device_index(x, True), device_ids))
        self.output_device = _get_device_index(output_device, True)
        self.src_device_obj = torch.device("cuda:{}".format(self.device_ids[0]))

        # 通过检查不同设备的剩余显存比，检查是否资源不均
        _check_balance(self.device_ids)

        if len(self.device_ids) == 1:
            self.module.cuda(device_ids[0])
```

每次前传，设备对全部参数、缓存验证是否在同一个机器上，然

后将输入分成 N 份（设备个数 N），将模型（参数和缓存）从 0 号设备复制到其余 N-1 设备上，多线程并行执行后，将结果集中中输出设备（默认 0 号设备）。

```
def forward(self, *inputs, **kwargs):
    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 "
                               "on device {} (device_ids[0]) but found one of "
                               "them on device: {}".format(self.src_device_obj, t.device))

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

？反传如何实现？(反传计算图分配在多卡上，是否需要计算？每次均需要同步)

DP 注意

DataParallel 也支持对模型的一个部分通过包装的形式(`self.block=nn.DataParallel(nn.Linear(20, 20))`)进行分布式计算；

DataParallel 包装模块后，无法访问原模块的属性，可以通过继承 DataParallel 类型，例如：

```
class MyDataParallel(nn.DataParallel):
    def __getattr__(self, name):
        return getattr(self.module, name)
```

DataParallel 支持单独使用多个自功能，实现自定义的分布式功能，其中包括数据划分、模型复制、数据集中和并行执行。

DataParallel 不支持与 ModelParallel 同时使用。

DDP API

DistributedDataParallel（DPI）支持进程内和进程间的数据并行，支持与模型并行同时使用。DDP 的每个进程有自己独立的 Python 解释器和网络优化器，在一个迭代步中，完整地执行优化步骤。

1. 步骤一，设置进程的后端支持、并行执行参数、初始化种子（全局梯度同步的前提）。
   Pytorch 支持 Gloo、MPI 和 NCCL，其中 MPI 支持需要源码 Pytorch 编译时支持。
   后端的选择原则：对于 GPU 分布式训练，选择 NCCL；对于 CPU 分布式训练，选择 Gloo（IP over IB, 其他选则 MPI）。
   通过`torch.distributed.init_process_group()`函数初始化 DistributedDataParallel 使用，具体的使用方式分为两种：

   (a) 通过明确地设定 store、rank 和 world_size 参数；

   (b) 通过设定 init_method 参数（默认'env://'），指定进程间互相发现的方式。

   初始化方式分为以下三种：

   (a) TCP，需要多进程间可以访问指定网络地址（属于 0 号进程）和 world_size 参数，该初始方式需要各自进程指定进行的具体 rank，例如：`dist.init_process_group(backend, init_method='tcp://10.1.1.20:23456', rank=args.rank, world_size=4)`

   (b) 共享文件，需要多进程通过共享文件系统，对于一个不存在的文件进行读写，需要指定共享文件名称和 world_size 参数。（需要共享文件系统支持使用 ftncl 锁定），该初始方式需要各自进程指定的 rank，例如：
   `dist.init_process_group(backend, init_method='file:///mnt/nfs/sharedfile', world_size=4, rank=args.rank)`

   (c) 环境变量，需要在环境中指定`MASTER_PORT`、`MASTER_ADDR`、 `WORLD_SIZE`、`RANK`参数，Rank 为 0 的进程负责建立进程间的连接。

```
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'

    # 初始化进程组，使用gloo/nccl组件，rank当前进程序号，world_size总共进程个数
    dist.init_process_group("gloo", rank=rank, world_size=world_size)

    # 初始化时使用相同的种子，保证模型一致
    torch.manual_seed(42)

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

2. 步骤二示例分布式数据并行，通过(torch.multiprocessing.spawn)或者torch.distributed.launch 打开多进程。

```
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):
        setup(rank, world_size)

        # 设置进程使用的设备
        n = torch.cuda.device_count() // world_size
        device_ids = list(range(rank * n, (rank + 1) * n))

        # 创建模型，并将模型放置在该进程的0号卡
```

```
    model = ToyModel().to(device_ids[0])
    # 包装模型，在当前设备上执行
    ddp_model = DDP(model, device_ids=device_ids)

    loss_fn = nn.MSELoss()
    # 对当前进程的参数进行参数更新
    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(device_ids[0])
    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)
```

DDP 实现

DistributedDataParallel（DDP）是基于torch.distributed的模块级的数据并行的实现，可以进行单进程多卡、多进程单卡两种模式的数据并行。在进程初始化torch.distributed.init_process_group后，模型被复制到各个设备或者进程中，在接下来的训练过程中，前传的输入被分块在不同的设备或者进程中分别处理，而在反传的过程中，各个设备或者进程的梯度被平均。具体地两种使用形式如下 (支持单节点或者多节点):

1. 单进程多卡，每个节点一个进程，该进程利用所有设备进行运算。

```
torch.distributed.init_process_group(backend="nccl")
model = DistributedDataParallel(model) # device_ids will include all GPU devices by default
```

2. 多进程单卡，多个进程，每个进程利用一个设备进行运算。(需要保证每个进程使用单一的卡，且每个进程的初始参数一致，DDP不会广播参数)

```
# N processes with N GPUs, i \in [0, N-1]
torch.cuda.set_device(i)

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

# use ``torch.distibuted.launch`` or ``torch.multiprocessing.spawn`` to spawn up multiple pr
```

DDP 支持稠密混合精度（FP16/FP32）的分布训练，暂时只支持 gloo 和 nccl 两种后端，其中 nccl 是 GPU 分布式训练推荐的后端。

DDP 的具体实现如下：

```
class DistributedDataParallel(Module):
    def __init__(self, module, device_ids=None,
                 output_device=None, dim=0, broadcast_buffers=True,
                 process_group=None, bucket_cap_mb=25,
                 find_unused_parameters=False,
                 check_reduction=False):

    super(DistributedDataParallel, self).__init__()

    # 不需要反传的模型，不需要DDP
    assert any((p.requires_grad for p in module.parameters())), (
        "DistributedDataParallel is not needed when a module "
        "doesn't have any parameter that requires a gradient."
    )

    # 模型并行
    self.is_multi_device_module = len({p.device for p in module.parameters()}) > 1
    self.is_cuda = all([p.device.type == 'cuda' for p in module.parameters()])

    if not self.is_cuda or self.is_multi_device_module:
        assert not device_ids and not output_device, (
            "DistributedDataParallel device_ids and output_device arguments "
            "only work with single device CUDA modules, but got "
            "device_ids {}, output_device {}, and module parameters {}."
        ).format(device_ids, output_device, {p.device for p in module.parameters()})
```

```
        self.device_ids = None
        self.output_device = None
    else:
        # Use all devices by default for single-device CUDA modules
        if device_ids is None:
            device_ids = list(range(torch.cuda.device_count()))

        self.device_ids = list(map(lambda x: _get_device_index(x, True), device_ids))

        if output_device is None:
            output_device = device_ids[0]

        self.output_device = _get_device_index(output_device, True)

    if self.is_multi_device_module:
        assert self.is_cuda, (
            "DistributedDataParallel with multi device module only works "
            "with CUDA devices, but module parameters locate in {}."
        ).format({p.device for p in module.parameters()})

    if process_group is None:
        self.process_group = _get_default_group()
    else:
        self.process_group = process_group

    self.dim = dim
    self.module = module
    self.broadcast_buffers = broadcast_buffers
    self.find_unused_parameters = find_unused_parameters
    self.require_backward_grad_sync = True
    self.require_forward_param_sync = True

    if check_reduction:
        # This argument is no longer used since the reducer
        # will ensure reduction completes even if some parameters
        # do not receive gradients.
        pass
```

```
        MB = 1024 * 1024

        # used for intra-node param sync and inter-node sync as well
        self.broadcast_bucket_size = int(250 * MB)

        # reduction bucket size
        self.bucket_bytes_cap = int(bucket_cap_mb * MB)

        # 创建时，进行模型的参数、缓存的同步
        module_states = list(self.module.state_dict().values())
        if len(module_states) > 0:
            self._distributed_broadcast_coalesced(
                module_states,
                self.broadcast_bucket_size)

        self._ddp_init_helper()

    def forward(self, *inputs, **kwargs):
        if self.require_forward_param_sync:
            self._sync_params()

        if self.device_ids:
            inputs, kwargs = self.scatter(inputs, kwargs, self.device_ids)
            if len(self.device_ids) == 1:
                output = self.module(*inputs[0], **kwargs[0])
            else:
                outputs = self.parallel_apply(self._module_copies[:len(inputs)], inputs, kwargs)
                output = self.gather(outputs, self.output_device)
        else:
            output = self.module(*inputs, **kwargs)

        if torch.is_grad_enabled() and self.require_backward_grad_sync:
            self.require_forward_param_sync = True
            # We'll return the output object verbatim since it is a freeform
            # object. We need to find any tensors in this object, though,
            # because we need to figure out which parameters were used during
```

```
        # this forward pass, to ensure we short circuit reduction for any
        # unused parameters. Only if 'find_unused_parameters' is set.
        if self.find_unused_parameters:
            self.reducer.prepare_for_backward(list(_find_tensors(output)))
        else:
            self.reducer.prepare_for_backward([])
    else:
        self.require_forward_param_sync = False

    return output
```

DDP 注意

1. 用户负责不同进程的负载，在 DDP 中，创建、前传和反传是分布式同步节点，进程进入同步节点执行的顺序是一致的，不同的负载造成进程间的等待。
2. 存档点，使用一个进程进行保存、多个进程进行读取，需要保证模型读取在保存完成之后。读取模型需要提供相应的设备（map_location，默认是 CPU，会将模型读取到存入的设备上，导致同一机器的所有进程用相同的设备）。

```
    ....

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

    CHECKPOINT_PATH = tempfile.gettempdir() + "/model.checkpoint"
        if rank == 0:
            # 所用进程从相同的初始点进行，仅保存一份在0号进程
            torch.save(ddp_model.state_dict(), CHECKPOINT_PATH)

    # barrier()保证1号进程在0号保存完成后执行
    dist.barrier()
    # 配置不同卡的设备
    rank0_devices = [x   rank * len(device_ids) for x in device_ids]
    device_pairs = zip(rank0_devices, device_ids)
    map_location = {'cuda:%d' % x: 'cuda:%d' % y for x, y in device_pairs}
    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(device_ids[0])
    loss_fn = nn.MSELoss()
    loss_fn(outputs, labels).backward()
    optimizer.step()

    # barrier()保证全部进程完成权重的读取
    dist.barrier()

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

    cleanup()
```

3. 参数读取的 map_location 需要指定，默认读取到写入参数的设备；

4. 由于 DDP 进行 all_reduce 操作时按照注册参数的反向顺序进行，需要保证各个进程中模型和参数注册的顺序是一致的。由于梯度 all_reduce 函数在创建时进行，任何之后的参数或者缓存修改都会造成未定义的行为。

ModelParallel API

Horovod