# PyTorch 深度剖析：并行训练的 DP 和 DDP 分别在什么情况下使用及实例

原创 CV开发者都爱看的 极市平台 2021-11-27 22:00:00 手机阅读







作者丨科技猛兽
编辑丨极市平台

极市导读

这篇文章从应用的角度出发，介绍 DP 和 DDP 分别在什么情况下使用，以及各自的使用方法。以及 DDP 的保存和加载模型的策略，和如何同时使用 DDP 和模型并行 (model parallel)。 >>加入极市CV技术交流群，走在计算机视觉的最前沿

## 目录



这篇文章从应用的角度出发，介绍 DP 和 DDP 分别在什么情况下使用，以及各自的使用方法。以及 DDP 的保存和加载模型的策略，和如何同时使用 DDP 和模型并行 (model parallel)。

## 1 DP 和 DDP 分别在什么情况下使用

### 1.1 几种并行训练的选项

PyTorch 提供了几种并行训练的选项。

- 如果：(1) 训练速度无所谓。(2) 模型和数据能够 fit 进一个 GPU 里面：这种情况建议**不要分布式训练**。
- 如果：(1) 想提升训练速度。(2) 非常不想过多地修改代码。(3) 有1台机器 (machine 或者叫做 node) (只能在单机上使用，俗称 "**单机多卡**")，机器上有多张 GPU：这种情况建议使用 **Data Parallel 分布式训练**。

- 如果：(1) 想进一步提升训练速度。(2) 可以适当多地修改代码。(3) 有1台或者多台的机器 (machine 或者叫做 node) (可以在多机上使用，俗称 "**多机多卡**")，机器上有多张 GPU：这种情况建议使用 **Distributed Data Parallel 分布式训练**。

### 1.2 DP 和 DDP 的比较

- Data Parallel：单进程，多线程，只能适用于1台机器的情况。Distributed Data Parallel：多进程，可以适用于多台机器的情况。
- 当模型太大，一个 GPU 放不下时，Data Parallel：不能结合模型并行的方法。Distributed Data Parallel：可以结合模型并行的方法。

## 2 Data Parallel 介绍

### 2.1 简介

Data Parallel 这种方法允许我们以最小的代码修改代价实现有1台机器上的多张 GPU 的训练。只需要修改1行代码。但是尽管 Data Parallel 这种方法使用方便，但是 Data Parallel 的性能却不是最好的。我们先介绍下 `torch.nn.DataParallel` 这个 PyTorch class。

**定义：**

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

> 在 module 层面实现数据并行。

torch.nn.DataParallel 要输入一个 `module` ，在前向传播过程中，这个 `module` 会在每个 device 上面复制一份。同时输入数据在 batch 这个维度被分块，这些数据会被按块分配在不同的 device 上面。最后形成的局面就是：所有的 GPU 上面都有一样的 `module` ，每个 GPU 都有单独的数据。在反向传播过程中，每一个 GPU 上得到的 gradient 会汇总到主 GPU (server) 上面。主 GPU (server) 更新参数之后，还会把新的参数模型参数 broadcast 到每个其它的 GPU 上面。

**DP 使用的是 Parameter Server (PS) 架构。** Parameter Server 架构 (PS 模式) 由 server 节点和 worker 节点组成，server 节点的主要功能是初始化和保存模型参数、接受 worker 节点计算出的局部梯度、汇总计算全局梯度，并更新模型参数。

worker 节点的主要功能是各自保存部分训练数据，初始化模型，从 server 节点拉取最新的模型参数 (pull)，再读取参数，根据训练数据计算局部梯度，上传给 server 节点 (push)。

PS 模式下的 DP，会造成负载不均衡，因为充当 server 的 GPU 需要一定的显存用来保存 worker 节点计算出的局部梯度；另外 server 还需要将更新后的模型参数 broadcast 到每个 worker，server 的带宽就成了 server 与worker 之间的通信瓶颈，server 与 worker 之间的通信成本会随着 worker 数目的增加而线性增加。

**所以读完了以上的分析，自然而然的2个要求就是：**

1. 训练的 batch size 要能够被 GPU 数量整除。
2. 在使用 DataParallel 之前，输入的 `module` 必须首先已经在 `device_ids[0]` 上面了。

**下面是2条重要的注意信息：**

1. 每次 Forward 的时候， `module` 会在每个 device 上面被浅复制。也就是说，DataParellel 保证了 device[0] 上的这个 replica (参数和 buffer) 和其他 device 上的 replica (参数和 buffer) 拥有着相同的存储位置。也就是说，只有那些 **in-place** 的操作才能够实现牵一发而动全身的效果，即：in-place 操作改变 device[0] 上的某个参数，会改变其他所有 device 上的参数。常见的 in-place 操作，比如有： `[BatchNorm2d](https://link.zhihu.com/?target=`

https%3A//pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html%23torch.nn.BatchNorm2d) 和 [spectral_norm()](https://link.zhihu.com/?target=https%3A//pytorch.org/docs/stable/generated/torch.nn.utils.spectral_norm.html%23torch.nn.utils.spectral_norm) 。

2. module 内部定义的 Forward 和 backward hooks，一共会被激活 len(device_ids) 次。每次激活时输入就依照当前 device 上的 input 执行。而且 hook 注册和激活的顺序无法控制。只能保证在当前 GPU 上面， [register_forward_pre_hook()](https://link.zhihu.com/?target=https%3A//pytorch.org/docs/stable/generated/torch.nn.Module.html%23torch.nn.Module.register_forward_pre_hook) 先于 [forward()](https://link.zhihu.com/?target=https%3A//pytorch.org/docs/stable/generated/torch.nn.Module.html%23torch.nn.Module.forward) 被执行，而无法保证它先于所有的 [forward()](https://link.zhihu.com/?target=https%3A//pytorch.org/docs/stable/generated/torch.nn.Module.html%23torch.nn.Module.forward) 被执行。

**参数定义：**

- **module** (Module) – module to be parallelized
- **device_ids** (list of python:int or torch.device) – CUDA devices (default: all devices)
- **output_device** (int or torch.device) – device location of output (default: device_ids[0])

**使用：**

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

### 2.2 用法示例

这一节通过具体的例子展示 DataParallel 的用法。

1) 首先 Import PyTorch modules 和超参数。

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader

# Parameters and DataLoaders
input_size = 5
output_size = 2

batch_size = 30
data_size = 100
```

2) 设置 device。

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

3) 制作一个dummy (random) dataset，这里我们只需要实现 getitem 方法。

```
class RandomDataset(Dataset):

    def __init__(self, size, length):
        self.len = length
        self.data = torch.randn(length, size)

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

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

rand_loader = DataLoader(dataset=RandomDataset(input_size, data_size),
                         batch_size=batch_size, shuffle=True)
```

4) 制作一个示例模型。

```
class Model(nn.Module):
    # Our model

    def __init__(self, input_size, output_size):
        super(Model, self).__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, input):
        output = self.fc(input)
        print("\tIn Model: input size", input.size(),
              "output size", output.size())

        return output
```

5) 创建 Model 和 DataParallel，首先要把模型实例化，再检查下我们是否有多块 GPU。最后是 put model on device：

```
model = Model(input_size, output_size)
if torch.cuda.device_count() > 1:
  print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
  model = nn.DataParallel(model)

model.to(device)
```

输出：

```
Let's use 2 GPUs!
```

6) Run the Model：

```
for data in rand_loader:
    input = data.to(device)
    output = model(input)
    print("Outside: input size", input.size(),
          "output_size", output.size())
```

输出：

```
# on 2 GPUs
Let's use 2 GPUs!
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([5, 5]) output size torch.Size([5, 2])
    In Model: input size torch.Size([5, 5]) output size torch.Size([5, 2])
Outside: input size torch.Size([10, 5]) output_size torch.Size([10, 2])
```

以上就是 DataParellel 的极简示例，注意我们并没有告诉程序我们要使用多少块 GPU，因为 torch.cuda.device_count() 会自动地计算出当前的所有可用的 GPU 数，假设电脑里面是8块，那么输出就会是：

```
Let's use 8 GPUs!
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
Outside: input size torch.Size([10, 5]) output_size torch.Size([10, 2])
```

## 3 Distributed Data Parallel 介绍

### 3.1 简介

Distributed Data Parallel 这种方法允许我们在有1台或者多台的机器上分布式训练。与 Data Parallel 的不同之处是：

- 需要启动这一步：init_process_group（https://pytorch.org/docs/stable/distributed.html#torch.distributed.init_process_group）
- 模型在创建的时候就已经复制到各个 GPU 上面，而不是在 Forward 函数里面复制的。

我们先介绍下 `torch.nn.parallel.DistributedDataParallel` 这个 PyTorch class。

**定义：**

CLASS `torch.nn.parallel.DistributedDataParallel` (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,gradient_as_bucket_view=False)

> 在 module 层面实现分布式数据并行。

torch.nn.DistributedDataParallel

torch.nn.DataParallel 要输入一个 module ，在模型构建的过程中，这个 module会在每个 device 上面复制一份。同时输入数据在 batch 这个维度被分块，这些数据会被按块分配在不同的 device 上面。最后形成的局面就是：所有的 GPU 上面都有一样的 module，每个 GPU 都有单独的数据。在反向传播过程中，每一个 GPU 上得到的 gradient 会被平均。

使用这个 class 需要 torch.distributed 的初始化，所以需要调用 [torch.distributed.init_process_group()](https://link.zhihu.com/?target=https%3A//pytorch.org/docs/stable/distributed.html%23torch.distributed.init_process_group) 。

如果想在一个有 N 个 GPU 的设备上面使用 DistributedDataParallel，则需要 spawn up N 个进程，每个进程对应0-N-1 的一个 GPU。这可以通过下面的语句实现：

```
torch.cuda.set_device(i)
```

i from 0-N-1，每个进程中都需要：

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

为了在每台设备 (节点) 上建立多个进程，我们可以使用 torch.distributed.launch 或者 torch.multiprocessing.spawn 。

如果你在一个进程中使用 torch.save 来保存模型，并在其他一些进程中使用 torch.load 来加载模型，请确保每个进程的 map_location 都配置正确。如果没有 map_location，torch.load 会将从保存的设备上加载模型。

**几点注意：**

- **减少优化器显存：** DistributedDataParallel 可以搭配 [torch.distributed.optim.ZeroRedundancyOptimizer](https://link.zhihu.com/?target=https%3A//pytorch.org/docs/stable/distributed.optim.html%23torch.distributed.optim.ZeroRedundancyOptimizer) 一起使用来减少 optimizer states memory，具体这里就不过多介绍，可以参考下面链接：

- **封装模型：** 在用 DistributedDataParallel 封装模型之后，千万不要试图改变你的模型的参数。因为，当用DistributedDataParallel 包装模型时，DistributedDataParallel 的构造函数会在构造时对模型本身的所有参数注册额外的梯度还原函数 (gradient reduction functions)。如果你事后改变了模型的参数，梯度还原函数就没法再与正确的参数集匹配。
- **梯度同步的机制：** DistributedDataParallel 在 module 层面实现了数据并行，可以在多台机器上运行。使用 DDP 的应用程序应该 spawn up 多个进程，并在每个进程中创建一个 DDP 实例。DDP 使用 Torch.distributed 包中的 collective communications 来同步梯度和缓冲区 (synchronize gradients and buffers)。更具体地说，DDP 为model.parameters() 给出的每个参数注册了一个 autograd hook，当在反向传播中计算出相应的梯度时，该 hook 将被触发。然后 DDP 使用该信号来触发跨进程的梯度同步。

**参数定义：**

- **module** (Module) – module to be parallelized
- **device_ids** (list of python:int or torch.device) –CUDA devices.

- 1) For single-device modules, `device_ids` can contain exactly one device id, which represents the only CUDA device where the input module corresponding to this process resides. Alternatively, `device_ids` can also be `None` .
- 2) For multi-device modules and CPU modules, `device_ids` must be `None` .

  When `device_ids` is `None` for both cases, both the input data for the forward pass and the actual module must be placed on the correct device. (default: `None` )
- **output_device** (int or torch.device) – Device location of output for single-device CUDA modules. For multi-device modules and CPU modules, it must be `None` , and the module itself dictates the output location. (default: `device_ids[0]` for single-device modules)
- **broadcast_buffers** (bool) – Flag that enables syncing (broadcasting) buffers of the module at beginning of the `forward` function. (default: `True` )
- **process_group** – The process group to be used for distributed data all-reduction. If `None` , the default process group, which is created by `[torch.distributed.init_process_group()](https://link.zhihu.com/?target=https%3A//pytorch.org/docs/stable/distributed.html%23torch.distributed.init_process_group)` , will be used. (default: `None` )
- **bucket_cap_mb** – `DistributedDataParallel` will bucket parameters into multiple buckets so that gradient reduction of each bucket can potentially overlap with backward computation. `bucket_cap_mb` controls the bucket size in MegaBytes (MB). (default: 25)
- **find_unused_parameters** (bool) – Traverse the autograd graph from all tensors contained in the return value of the wrapped module's `forward` function. Parameters that don't receive gradients as part of this graph are preemptively marked as being ready to be reduced. In addition, parameters that may have been used in the wrapped module's `forward` function but were not part of loss computation and thus would also not receive gradients are preemptively marked as ready to be reduced. (default: `False` )
- **check_reduction** – This argument is deprecated.
- **gradient_as_bucket_view** (bool) – When set to `True` , gradients will be views pointing to different offsets of `allreduce` communication buckets. This can reduce peak memory usage, where the saved memory size will be equal to the total gradients size. Moreover, it avoids the overhead of copying between gradients and `allreduce` communication buckets. When gradients are views, `detach_()` cannot be called on the gradients. If hitting such errors, please fix it by referring to the `[zero_grad()](https://link.zhihu.com/?target=https%3A//pytorch.org/docs/stable/generated/torch.optim.Optimizer.zero_grad.html%23torch.optim.Optimizer.zero_grad)` function in `torch/optim/optimizer.py` as a solution.

### 3.2 用法示例

这一节通过具体的例子展示 DistributedDataParallel 的用法，这个例子假设我们有一个8卡 GPU。

1) 首先初始化进程：

```
import os
import sys
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
```

```
# On Windows platform, the torch.distributed package only
# supports Gloo backend, FileStore and TcpStore.
# For FileStore, set init_method parameter in init_process_group
# to a local file. Example as follow:
# init_method="file:///f:/libtmp/some_file"
# dist.init_process_group(
#    "gloo",
#    rank=rank,
#    init_method=init_method,
#    world_size=world_size)
# For TcpStore, same way as on Linux.

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()
```

2) 创建一个 toy module，叫它 ToyModel，用 DDP 去包裹它。注意，由于 DDP 在构造函数中把模型状态从第rank 0 的进程广播给所有其他进程，所以我们无需担心不同的 DDP 进程从不同的参数初始值启动。PyTorch提供了 `mp.spawn` 来在一个节点启动该节点所有进程，每个进程运行 `train(i, args)` ，其中 `i` 从0到 `args.gpus \- 1` 。所以有以下 code。

执行代码时，GPU 数和进程数都是 world_size。

```
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)
```

### 3.3 保存和加载模型

当使用 DDP 时，我们只在一个进程中保存模型，然后将其加载到所有进程中，以减少写的开销。这也很好理解，因为所有进程从相同的参数开始，梯度在后向传递中是同步的，因此，所有进程的梯度是相同的。所以读者请确保所有进程在保存完成之前不要开始加载。此外，在加载模块时，我们需要提供一个适当的 map_location 参数，以防止一个 process 踏入其他进程的设备。如果缺少 map_location，torch.load 将首先把 module 加载到 CPU，然后把每个参数复制到它被保存的地方，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()
```

### 3.4 与模型并行的结合 (DDP + model parallel)

有关模型并行的介绍可以参考：

DDP 也适用于 **multi-GPU 模型**。DDP 包裹着 **multi-GPU 模型**，在用海量数据训练大型模型时特别有帮助。

```
class ToyMpModel(nn.Module):
    def __init__(self, dev0, dev1):
        super(ToyMpModel, self).__init__()
        self.dev0 = dev0
        self.dev1 = dev1
        self.net1 = torch.nn.Linear(10, 10).to(dev0)
        self.relu = torch.nn.ReLU()
        self.net2 = torch.nn.Linear(10, 5).to(dev1)
```

```
        def forward(self, x):
            x = x.to(self.dev0)
            x = self.relu(self.net1(x))
            x = x.to(self.dev1)
            return self.net2(x)
```

当把一个 **multi-GPU 模型**传递给 DDP 时，device_ids 和 output_device 不能被设置。输入和输出数据将被应用程序或模型 forward() 方法放在适当的设备中。

```
def demo_model_parallel(rank, world_size):
    print(f"Running DDP with model parallel example on rank {rank}.")
    setup(rank, world_size)

    # setup mp_model and devices for this process
    dev0 = (rank * 2) % world_size
    dev1 = (rank * 2 + 1) % world_size
    mp_model = ToyMpModel(dev0, dev1)
    ddp_mp_model = DDP(mp_model)

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

    optimizer.zero_grad()
    # outputs will be on dev1
    outputs = ddp_mp_model(torch.randn(20, 10))
    labels = torch.randn(20, 5).to(dev1)
    loss_fn(outputs, labels).backward()
    optimizer.step()

    cleanup()


if __name__ == "__main__":
    n_gpus = torch.cuda.device_count()
    assert n_gpus >= 2, f"Requires at least 2 GPUs to run, but got {n_gpus}"
    world_size = n_gpus
    run_demo(demo_basic, world_size)
    run_demo(demo_checkpoint, world_size)
    run_demo(demo_model_parallel, world_size)
```

**参考：**

https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html

https://pytorch.org/docs/stable/notes/ddp.html

**算法trick**：目标检测比赛中的tricks集锦 | 从39个kaggle竞赛中总结出来的图像分割的Tips和Tricks

**技术综述：**一文弄懂各种loss function | 工业图像异常检测最新研究总结（2019-2020）



阅读原文