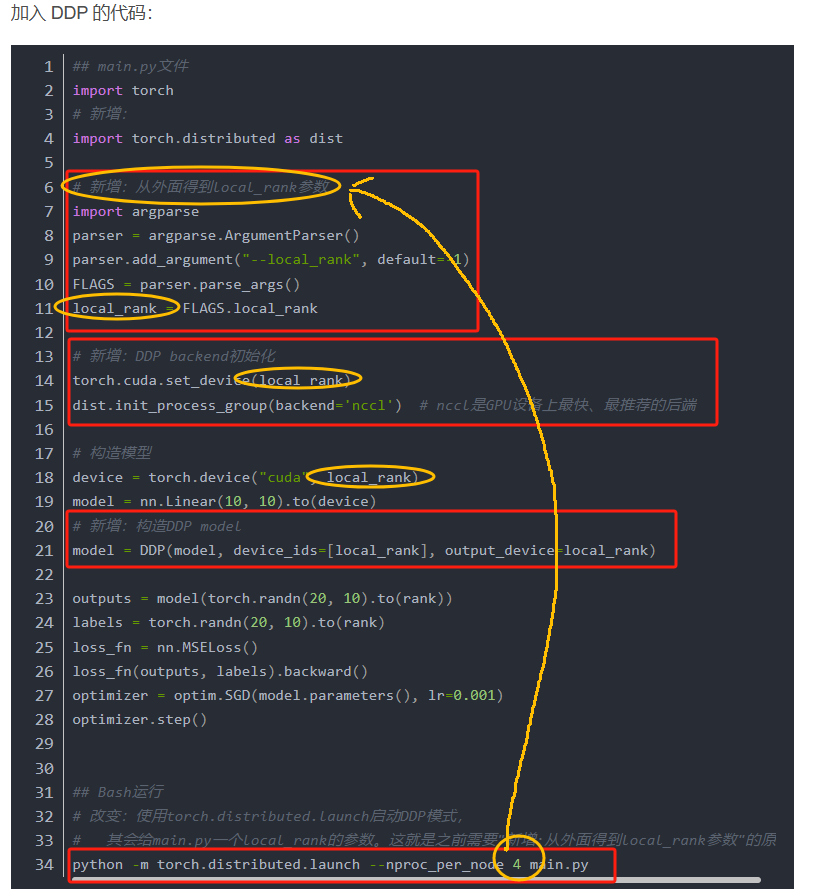
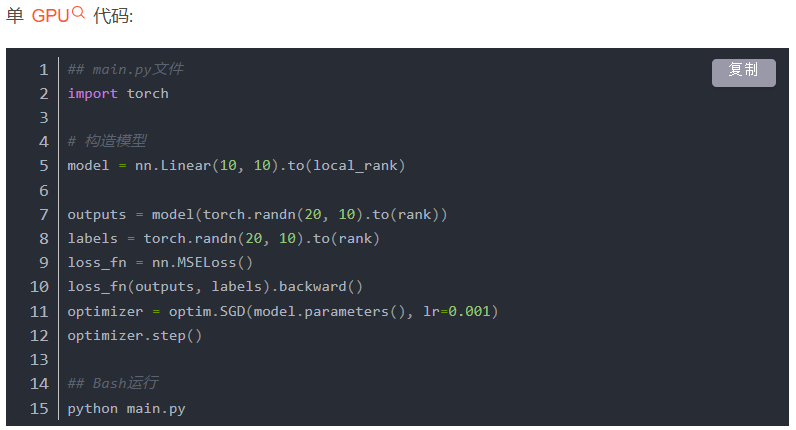
DistributedDataParallel（DDP）是一个支持多机多卡、分布式训练的深度学习工程方法。

* 在分类上，DDP属于Data Parallel。简单来讲，就是通过提高batch size来增加并行度。
* DDP通过Ring-Reduce的数据交换方法提高了通讯效率，并通过启动多个进程的方式减轻Python GIL的限制，从而提高训练速度
* 一般来说，DDP都是显著地比DP快，能达到略低于卡数的加速比（例如，四卡下加速3倍）



**GIL的定义：**

GIL是Python解释器中的一种机制，它是一把全局锁，用于保护解释器免受多线程并发访问的影响。这意味着Python在同一时刻只允许一个线程执行Python字节码。

GIL实际上是一个互斥锁，在Python解释器层面上实现。由于GIL的存在，同一时刻只有一个线程能够获得解释器的控制权，其他线程被阻塞，无法执行Python字节码。这意味着在多核CPU上，Python的多线程程序可能无法充分利用多核性能

**GIL的影响：**

* CPU密集型任务受限： 对于CPU密集型任务，由于GIL的存在，多线程并不能有效地提高性能，因为多个线程无法同时执行Python字节码。
* IO密集型任务相对不受限： 在IO密集型任务中，线程在等待IO时会释放GIL，允许其他线程执行Python字节码，因此在这种情况下，多线程能够发挥一定作用。

**GIL的工作原理：**

* 互斥锁机制： GIL是一个互斥锁，它在解释器级别上控制对Python对象和内存管理的访问。只有一个线程能够获得GIL的锁，执行Python字节码，其他线程则被阻塞等待。
* 执行Python字节码的时间片： 当一个线程持有GIL并执行Python字节码时，会执行一段时间，称为时间片。一旦时间片用完或发生阻塞IO操作，线程会释放GIL锁。
* GIL的释放和竞争： 在等待的线程中，如果有线程释放了GIL（例如因为IO等待），其他线程会竞争获取GIL的锁。

**降低GIL的影响：**

* 使用多进程： 多进程允许同时运行多个Python解释器，每个进程都有自己的GIL。这样可以绕过GIL的限制。
* 使用C扩展： 编写一些Python的关键部分为C扩展，这些部分在执行时不受GIL的影响，例如numpy、pandas等。
* 使用异步编程： 使用异步编程模型（例如asyncio库）可以最大程度地减少对线程的依赖，避免GIL对程序性能的影响。

**2. DDP 基本原理**

假如我们有N张显卡：

**缓解GIL限制：**在DDP模式下，会有N个进程被启动，每个进程在一张卡上加载一个模型，这些模型的参数在数值上是相同的

**Ring-Reduce加速：**在模型训练时，各个进程通过一种叫Ring-Reduce的方法与其他进程通讯，交换各自的梯度，从而获得所有进程的梯度；

**实际上是多进程版的Data Parallelism：**各个进程用平均后的梯度更新自己的参数，因为各个进程的初始参数、更新梯度是一致的，所以更新后的参数也是完全相同的。

1. **在 pytorch 中使用 DDP**

DDP有不同的使用模式。DDP的官方最佳实践是，每一张卡对应一个单独的GPU模型（也就是一个进程）。

例如，有两台机子，每台8张显卡，那就是2x8=16个进程，并行数是16。

也可以给每个进程分配多张卡的。总的来说，分为以下三种情况：

* 每个进程一张卡。这是DDP的最佳使用方法。
* 每个进程多张卡，复制模式。一个模型复制在不同卡上面，每个进程都实质等同于DP模式。这样做是能跑得通的，但是，速度不如上一种方法，一般不采用。
* 每个进程多张卡，并行模式。一个模型的不同部分分布在不同的卡上面。例如，网络的前半部分在0号卡上，后半部分在1号卡上。这种场景，一般是因为我们的模型非常大，大到一张卡都塞不下batch size = 1的一个模型。

**基本概念**

在16张显卡，16的并行数下，DDP会同时启动16个进程。

Group：进程组。默认情况下，只有一个组。这个可以先不管，一直用默认的就行。

world size：表示全局的并行数，简单来讲，就是2x8=16。



Rank：表现当前进程的序号，用于进程间通讯。对于16的world sizel来说，就是0,1,2,…,15。

注意：rank=0的进程就是master进程。



local rank：每台机子上的进程的序号。机器一上有0,1,2,3,4,5,6,7，机器二上也有0,1,2,3,4,5,6,7

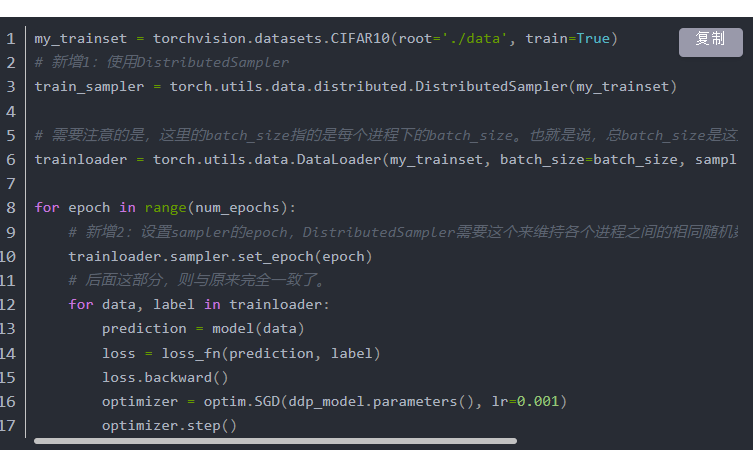


具体流程：程序虽然会在16个进程上跑起来，但是它们跑的是同一份代码，所以在写程序的时候要处理好不同进程的关系。



**前向与后向传播：**

DDP同时起了很多个进程，但是他们用的是同一份数据，那么就会有数据上的冗余。也就是说，你平时一个epoch如果是一万份数据，现在就要变成1\*16=16万份数据了。那么，我们需要使用一个特殊的sampler，来使得各个进程上的数据各不相同，进而让一个epoch还是1万份数据。（实际上，就是需要把一个epoch的一万份数据分成16份，给到 16 张卡上。）



保存参数：

# 1. save模型的时候，和DP模式一样，有一个需要注意的点：保存的是model.module而不是model。

# 因为model其实是DDP model，参数是被`model=DDP(model)`包起来的。

# 2. 我只需要在进程0上保存一次就行了，避免多次保存重复的东西。

if dist.get\_rank() == 0:

torch.save(model.module, "saved\_model.ckpt")

调用方式：

DDP模型下，python源代码的调用方式和原来的不一样。现在，需要用**torch.distributed.launch**来启动训练。

在这里，给出分布式训练的重要参数：

nnodes，机器数量

node\_rank，当前是哪台机器

nproc\_per\_node，每台机器有多少个进程

通讯相关参数，在多机的时候会用到

通讯的address

通讯的port

实现方式

需要在每一台机子（总共m台）上都运行一次torch.distributed.launch。每个torch.distributed.launch会启动n个进程，并给每个进程一个–local\_rank=i的参数

这样就得到nm个进程，world\_size=nm

单机模式：

## Bash运行

# 假设我们只在一台机器上运行，可用卡数是8

**python -m torch.distributed.launch --nproc\_per\_node 8 main.py**

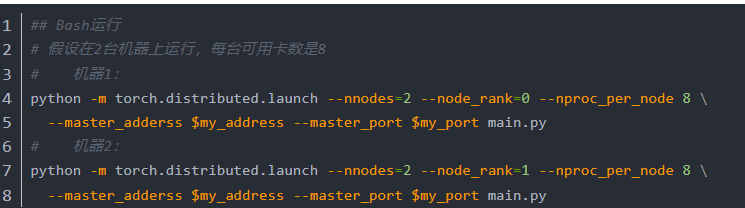
多机模式：

**通讯的address**

* master\_address，也就是master进程的网络地址 ，默认是：127.0.0.1，只能用于单机。

**通讯的port**

* master\_port，也就是master进程的一个端口，要先确认这个端口没有被其他程序占用。一般情况下用默认的就行，默认是：29500



# 假设只用4,5,6,7号卡

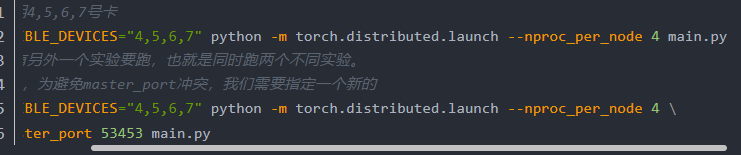
CUDA\_VISIBLE\_DEVICES="4,5,6,7" python -m torch.distributed.launch --nproc\_per\_node 4 main.py

# 假如还有另外一个实验要跑，也就是同时跑两个不同实验。

# 这时，为避免master\_port冲突，我们需要指定一个新的

CUDA\_VISIBLE\_DEVICES="4,5,6,7" python -m torch.distributed.launch --nproc\_per\_node 4 \

--master\_port 53453 main.py



完整代码：################

## main.py文件

import argparse

from tqdm import tqdm

import torch

import torchvision

import torch.nn as nn

import torch.nn.functional as F

# 新增：

import torch.distributed as dist

from torch.nn.parallel import DistributedDataParallel as DDP

**### 1. 基础模块 ###**

# 假设我们的模型是这个，与DDP无关

class ToyModel(nn.Module):

def \_\_init\_\_(self):

super(ToyModel, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 6, 5)

self.pool = nn.MaxPool2d(2, 2)

self.conv2 = nn.Conv2d(6, 16, 5)

self.fc1 = nn.Linear(16 \* 5 \* 5, 120)

self.fc2 = nn.Linear(120, 84)

self.fc3 = nn.Linear(84, 10)

def forward(self, x):

x = self.pool(F.relu(self.conv1(x)))

x = self.pool(F.relu(self.conv2(x)))

x = x.view(-1, 16 \* 5 \* 5)

x = F.relu(self.fc1(x))

x = F.relu(self.fc2(x))

x = self.fc3(x)

return x

**# 假设我们的数据是这个**

def get\_dataset():

transform = torchvision.transforms.Compose([

torchvision.transforms.ToTensor(),

torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

])

my\_trainset = torchvision.datasets.CIFAR10(root='./data', train=True,

download=True, transform=transform)

# DDP：使用DistributedSampler，DDP帮我们把细节都封装起来了。

# 用，就完事儿！sampler的原理，第二篇中有介绍。

train\_sampler = torch.utils.data.distributed.DistributedSampler(my\_trainset)

# DDP：需要注意的是，这里的batch\_size指的是每个进程下的batch\_size。

# 也就是说，总batch\_size是这里的batch\_size再乘以并行数(world\_size)。

trainloader = torch.utils.data.DataLoader(my\_trainset,

batch\_size=16, num\_workers=2, sampler=train\_sampler)

return trainloader

**### 2. 初始化我们的模型、数据、各种配置 ####**

**# DDP：从外部得到local\_rank参数**

parser = argparse.ArgumentParser()

parser.add\_argument("--local\_rank", default=-1, type=int)

FLAGS = parser.parse\_args()

local\_rank = FLAGS.local\_rank

**# DDP：DDP backend初始化**

torch.cuda.set\_device(local\_rank)

dist.init\_process\_group(backend='nccl') # nccl是GPU设备上最快、最推荐的后端

**# 准备数据，要在DDP初始化之后进行**

trainloader = get\_dataset()

**# 构造模型**

model = ToyModel().to(local\_rank)

**# DDP: Load模型要在**构造DDP模型之前，且只需要在master上加载就行了。

ckpt\_path = None

if dist.get\_rank() == 0 and ckpt\_path is not None:

model.load\_state\_dict(torch.load(ckpt\_path))

**# DDP: 构造DDP model**

model = DDP(model, device\_ids=[local\_rank], output\_device=local\_rank)

**# DDP:** 要在构造DDP model之后，才能**用model初始化optimizer。**

optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

# 假设我们的loss是这个

loss\_func = nn.CrossEntropyLoss().to(local\_rank)

**（把DDP初始化准备好，然后准备数据，构造模型加载模型，然后套DDP壳子也就是构造DDP模型，然后用套好的DDP模型初始化optimizer）**

**### 3. 网络训练 ###**

model.train()

iterator = tqdm(range(100))

for epoch in iterator:

# DDP：设置sampler的epoch，

# DistributedSampler需要这个来指定shuffle方式，

# 通过维持各个进程之间的相同随机数种子使不同进程能获得同样的shuffle效果。

trainloader.sampler.set\_epoch(epoch)

# 后面这部分，则与原来完全一致了。

for data, label in trainloader:

data, label = data.to(local\_rank), label.to(local\_rank)

optimizer.zero\_grad()

prediction = model(data)

loss = loss\_func(prediction, label)

loss.backward()

iterator.desc = "loss = %0.3f" % loss

optimizer.step()

# DDP:

# 1. save模型的时候，和DP模式一样，有一个需要注意的点：保存的是model.module而不是model。

# 因为model其实是DDP model，参数是被`model=DDP(model)`包起来的。

# 2. 只需要在进程0上保存一次就行了，避免多次保存重复的东西。

if dist.get\_rank() == 0:

torch.save(model.module.state\_dict(), "%d.ckpt" % epoch)

################

## Bash运行

# DDP: 使用torch.distributed.launch启动DDP模式

# 使用CUDA\_VISIBLE\_DEVICES，来决定使用哪些GPU

# CUDA\_VISIBLE\_DEVICES="0,1" python -m torch.distributed.launch --nproc\_per\_node 2 main.py