

# 分布式机器学习 实验指导书

课程号：85990072

授课教师：王智副教授

编者：袁新杰助教

清华大学  
清华大学深圳国际研究生院



**清华大学深圳国际研究生院**  
Tsinghua Shenzhen International Graduate School

[最后编辑于 2023 年 3 月 8 日]

# 前言

可以介绍一下指导书结构，可以介绍一下我为什么想写这本指导书。感谢一下前任助教们希望我这个是第一版，以后每年助教都可以持续更新迭代。

本指导书着重讲解环境配置与程序运行，对于具体分布式机器学习的算法实现，已由老师在课上讲解，实验指导书里不再赘述。【当然如果之后有人有兴趣把这部分补充上那就更好了】

助教袁新杰于 2023 年 3 月初

# 目录

前言	i
目录	iii
<b>1 环境配置</b>	<b>1</b>
1.1 使用本地环境与 GPU	1
1.1.1 安装 CUDA 工具箱	1
1.1.2 安装 Anaconda	5
1.1.3 创建虚拟环境并安装 PyTorch	5
1.1.4 安装其他包	6
1.2 使用虚拟环境与本地 GPU	6
1.2.1 安装并配置 Docker 引擎	7
1.2.2 搜索并下载 PyTorch 镜像	8
1.2.3 启动容器	10
1.2.4 安装新包后重新打包成镜像	10
1.2.5 限制 Docker 内存占用 (可选)	11
1.3 使用华为云计算资源	12
1.4 使用深研院计算资源	12
1.4.1 创建开发环境	12
1.4.2 安装其他软件或包	14
1.4.3 创建镜像环境	14
<b>2 实验一：梯度下降单机优化</b>	<b>16</b>
2.1 实验内容与要点介绍	16
2.1.1 实验内容与要求	16
2.1.2 PyTorch 优化器	16
2.1.3 几种算法回顾	18
2.2 使用 VSCode 与本地环境调试运行	18
2.3 使用 VSCode 与本地容器调试运行	20
2.3.1 启动容器并挂载本地文件夹	20
2.3.2 在 VSCode 中使用容器	20
2.4 使用 VSCode 与远程服务器调试运行	21
<b>3 实验二：通信模型与参数聚合</b>	<b>23</b>
3.1 实验内容与要点介绍	23
3.1.1 实验内容与要求	23
3.1.2 多节点通信	23
3.1.3 记录 GPU 上任务的运行时间	25

3.2	使用进程模拟多节点	25
3.2.1	手动运行多进程	25
3.2.2	使用 torch.multiprocessing 自动创建多进程	26
3.3	使用容器模拟多节点	26
3.3.1	Docker compose 介绍	26
3.3.2	通过 Docker compose 启动容器	28
<b>4</b>	<b>实验三：数据并行</b>	<b>29</b>
4.1	实验内容与要点介绍	29
4.1.1	实验内容与要求	29
4.1.2	数据集、加载器和采样器	30
<b>5</b>	<b>实验四：模型并行实验</b>	<b>32</b>
5.1	实验内容	32

# 第 1 章 环境配置

本章将主要介绍分别使用本地计算资源、深研院计算资源和华为云计算资源时构建环境的方法。

## 1.1 使用本地环境与 GPU

使用本地计算资源可以不受网络链接状况约束，随时随地调试程序，对于简单的项目，本地调试也可能更省时间。

本节以助教所使用的计算机为例，展示环境配置过程。助教使用的计算机系统与配置为：

- 系统：windows10 专业教育版；22H2
- 处理器：Intel(R) Core(TM) i7-8700 CPU
- 内存：16GB
- 显卡：NVIDIA GeForce RTX 2060
- 编辑器：Visual Studio Code

### 1.1.1 安装 CUDA 工具箱

TODO 这一部分只是贴了网址, 没有写成教程

对于包含英伟达显卡的计算机，我们推荐首先安装 CUDA 工具包以使用 GPU 加速计算。

注意，仅包含英伟达 GPU 的计算机需要安装 CUDA 工具箱以使用 GPU 加速计算。  
使用核显或 AMD 显卡的计算机再后续步骤中使用 CPU 计算即可

GPU 型号、CUDA 工具包、PyTorch 版本相互关联。因此需要一起规划好。

<https://developer.nvidia.com/zh-cn/cuda-gpus> 查看得到我的显卡的 2060 的算力为 7.5。



### 支持 CUDA 的 GeForce 和 TITAN 产品

GPU	计算能力	GPU
GeForce RTX 3090	8.6	GeForce
GeForce RTX 3080	8.6	GeForce
GeForce RTX 3070	8.6	GeForce
NVIDIA TITAN RTX	7.5	GeForce
GeForce RTX 2080 Ti	7.5	GeForce
GeForce RTX 2080	7.5	GeForce
GeForce RTX 2070	7.5	GeForce
GeForce RTX 2060	7.5	GeForce
NVIDIA TITAN V	7.0	GeForce

Figure 1.1: CAPTION holder

<https://en.wikipedia.org/wiki/CUDA> 查看得到支持我显卡的 CUDA 版本为  $\geq 10.0$

<https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html> 同时 CUDA 对显卡驱动的最低版本也提出了要求，但显卡驱动对 CUDA 向下兼容，因此一般安装了最近发布的显卡驱动版本即可，无需与 CUDA 版本特别对应。

<https://pytorch.org/get-started/locally/> 最后要注意，PyTorch 并不一定支持最新的 CUDA 版本，因此安装前再去 PyTorch 上看一眼 PyTorch 支持哪些 CUDA 版本。

我们发现 PyTorch 最高支持到 CUDA 11.7，满足显卡算力对 CUDA 版本  $\geq 10.0$  的要求，因此我们可以选择安装 CUDA 11.7。

## GPUs supported [\[ edit \]](#)

Supported CUDA Compute Capability versions for CUDA SDK version and Microarchitecture (by code name):

Compute Capability (CUDA SDK support vs. Microarchitecture)											
CUDA SDK version(s)	Tesla	Fermi	Kepler (early)	Kepler (late)	Maxwell	Pascal	Volta	Turing	Ampere	Ada Lovelace	Hopper
1.0 <sup>[29]</sup>	1.0 – 1.1										
1.1	1.0 – 1.1+x										
2.0	1.0 – 1.1+x										
2.1 - 2.3.1 <sup>[30][31][32][33]</sup>	1.0 – 1.3										
3.0 - 3.1 <sup>[34][35]</sup>	1.0 –	2.0									
3.2 <sup>[36]</sup>	1.0 –	2.1									
4.0 - 4.2	1.0 –	2.1+x									
5.0 - 5.5	1.0 –			3.5							
6.0	1.0 –			3.5							
6.5	1.1 –				5.x						
7.0 - 7.5		2.0 –			5.x						
8.0		2.0 –				6.x					
9.0 - 9.2			3.0 –				7.0				
10.0 - 10.2			3.0 –					7.5			
11.0 <sup>[37]</sup>				3.5 –					8.0		
11.1 - 11.4 <sup>[38]</sup>				3.5 –					8.6		
11.5 - 11.7.1 <sup>[39]</sup>				3.5 –					8.7		
11.8 <sup>[40]</sup>				3.5 –							9.0
12.0					5.0 –						9.0

Figure 1.2: CAPTION holder

PyTorch Build	Stable (1.13.1)	Preview (Nightly)		
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python	C++ / Java		
Compute Platform	CUDA 11.6	CUDA 11.7	ROCm 5.2	CPU
Run this Command:	<pre>conda install pytorch torchvision torchaudio pytorch-cuda=11.7 -c pytorch -c nvidia</pre>			

Figure 1.3: CAPTION holder

选择对应版本 CUDA 安装包并下载安装, 安装过程略。<https://developer.nvidia.com/cuda-toolkit-archive>

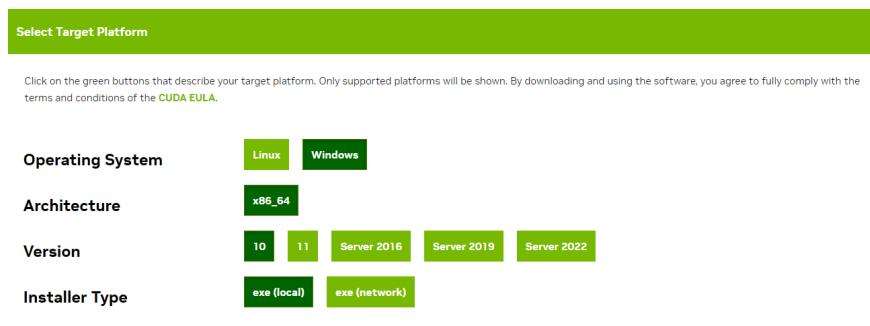


Figure 1.4: CAPTION holder

在这一步完成后, 我们打开终端输入 `nvcc -V` 以及 `nvidia-smi` 应当分别能看到图1.5和图1.6类似的输出, 这说明我们安装完成。

```
(base) PS C:\Users\MMLab_Cantjie> nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Tue_Mar__8_18:36:24_Pacific_Standard_Time_2022
Cuda compilation tools, release 11.6, V11.6.124
Build cuda_11.6.r11.6/compiler.31057947_0
```

Figure 1.5: caption:nvcc-v-install-success

```
(base) PS C:\Users\MMLab_Cantjie> nvidia-smi
Sun Mar  5 11:03:16 2023

+-----+
| NVIDIA-SMI 531.18              Driver Version: 531.18      CUDA Version: 12.1   |
+-----+-----+-----+-----+-----+-----+
| GPU  Name           TCC/WDDM | Bus-Id      Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap|  Memory-Usage | GPU-Util  Compute M. |
|                                           MIG M. |
+-----+-----+-----+-----+-----+-----+
|   0   NVIDIA GeForce RTX 2060    WDDM | 00000000:01:00.0  On  |          N/A         |
| 45%   36C    P8             13W / 160W| 1071MiB / 6144MiB |      5%      Default |
|                                           |                      | N/A |
+-----+-----+-----+-----+-----+-----+

+-----+
| Processes: |
| GPU   GI    CI          PID    Type    Process name                  GPU Memory |
| ID     ID     ID                     |              Usage   |
+-----+-----+-----+-----+-----+-----+
|   0   N/A   N/A         6964     C+G     C:\Windows\explorer.exe       N/A       |
+-----+-----+-----+-----+-----+-----+
```

Figure 1.6: caption:nvidia-smi-install-success



### 1.1.2 安装 Anaconda

我们可能同时有多个项目或作业在处理，而不同的项目或作业可能使用了不同 python 版本、不同的工具包等，为了避免冲突，我们通常会为每一个项目或作业指定一个虚拟环境，以使得各个环境之间互不干扰。为此，我们 Anaconda 以创建并管理虚拟环境。

安装过程参考官网文档即可：<https://docs.anaconda.com/anaconda/install/windows/>

安装完成后启动终端，输入 `conda -V`，如正确显示 conda 版本则说明安装成功。

```
(base) PS C:\Users\MMLab_Cantjie> conda -V
conda 22.9.0
```

Figure 1.7: CAPTION holder

### 1.1.3 创建虚拟环境并安装 PyTorch

安装完成 conda 后，我们新建一个预装了 Python 的、用来完成本门课程的虚拟环境。

需要注意的是，PyTorch 和 Python 版本也需要对应，在<https://github.com/pytorch/vision#installation>中，我们发现 torch 1.13 要求 python 介于 3.7.2 和 3.10 之间。

torch	torchvision	python
main / nightly	main / nightly	>=3.8 , <=3.10
1.13.0	0.14.0	>=3.7.2 , <=3.10
1.12.0	0.13.0	>=3.7 , <=3.10
1.11.0	0.12.0	>=3.7 , <=3.10
1.10.2	0.11.3	>=3.6 , <=3.9
1.10.1	0.11.2	>=3.6 , <=3.9

Figure 1.8: CAPTION holder

打开终端，输入下面命令以利用 conda 新建环境，

```
1 $ conda create --name <envname> python=3.9
```

将其中 `<envname>` 改成自定义的环境名称，如助教自己选择的 `distributedml`。

新建完成后，通过 `conda activate <envname>` 进入环境。在 pytorch 官网安装页面<https://pytorch.org/get-started/locally/>选择对应的 pytorch 版本、系统版本等，复制给出的命令并运行。

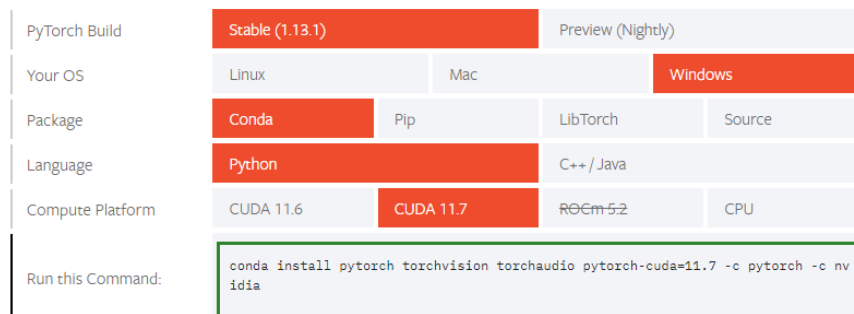


Figure 1.9: CAPTION holder

安装完成后，进入 Python 就可以 `import torch` 了，如图1.10。

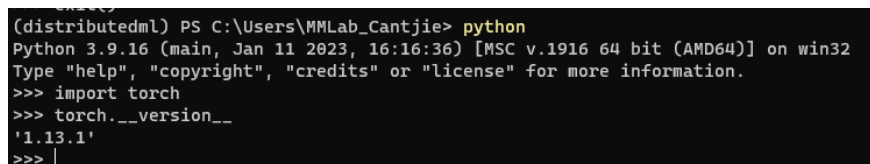


Figure 1.10: caption:pytorch-install-success

### 1.1.4 安装其他包

如果我们还想要安装其他包，比如同学们画图常用的 `matplotlib` 包，该怎么办呢？在 `conda activate <envname>` 进入环境后，直接通过 `conda install matplotlib` 就可以了。

## 1.2 使用虚拟环境与本地 GPU

上面的本地环境配置不可为不复杂，CUDA、显卡型号、显卡驱动、PyTorch、Python 等版本需要手动一一对应起来安装。那有没有什么更简单的利用本机 GPU 计算资源的方法呢？

在这一节，我们介绍直接利用 Docker 镜像搭配环境的方法。

### 1.2.1 安装并配置 Docker 引擎

首先在官网下载安装包<https://docs.docker.com/desktop/install/windows-install/>，安装过程略。

在安装完成后启动 Docker Desktop，在 windows 下，很可能会报错（具体内容是啥助教忘了截图了），一般错误的原因是缺少 wsl2 和 hyper-v。

为了启用 hyper-v，在控制面板中按照图1.11中的操作选中 Hyper-V 并确定。

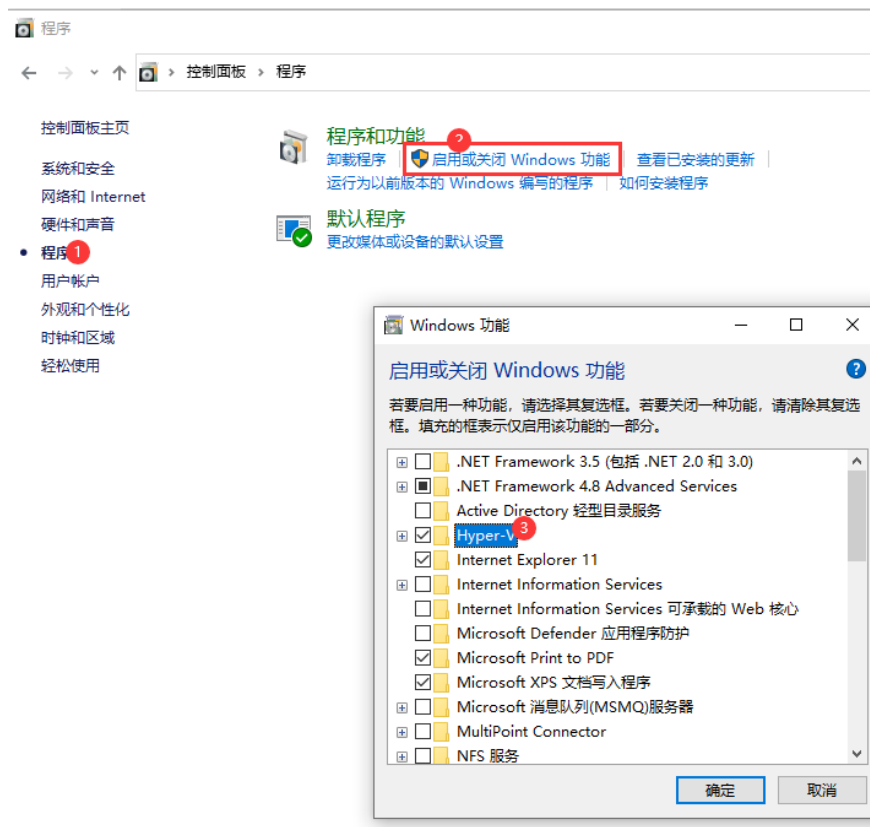


Figure 1.11: caption:turn-on-hyper-v

为了启用 wsl2，参考<https://learn.microsoft.com/en-us/windows/wsl/install>，在终端下输入 `wsl --install` 等待安装完成即可。

安装完成后启动 Docker Desktop，为了加速下载，可以按照图1.12所示方法为 Docker 指定国内镜像服务器，即在原本的配置中加入如下内容。

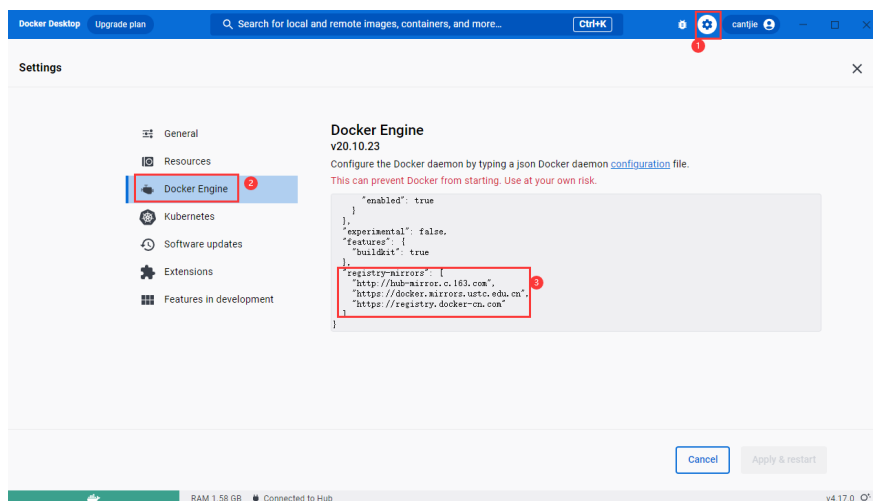


Figure 1.12: caption:docker-mirrors-setting

```

1  "registry-mirrors": [
2    "http://hub-mirror.c.163.com",
3    "https://docker.mirrors.ustc.edu.cn",
4    "https://registry.docker-cn.com"
5  ]

```

启动终端，输入 `docker --version`，如图1.13，正常返回 Docker 版本就说明安装成功了。

```

(distributedml) PS C:\Users\MMLab_Cantjie> docker --version
Docker version 20.10.23, build 7155243

```

Figure 1.13: caption:docker-install-success

## 1.2.2 搜索并下载 PyTorch 镜像

Dockerhub 是一个共享镜像的平台<https://hub.docker.com/>。所谓镜像，类似于一个操作系统的 iso 文件：我们拿到 iso 文件后可以创建使用该操作系统的虚拟机；而当我们拿到镜像后，也可以利用该镜像创建一个使用该镜像的容器，即容器是一个镜像的实例。

因此，如果有人在某个容器中把 CUDA、PyTorch、Python 等环境都配置好，并打包成镜像共享给我们，我们就可以免去复杂的安装过程，从而直接使用镜像生成容器，在容器中直接运行我们所写的脚本。

在 DockerHub 中，我们搜索 `pytorch/pytorch`，可以找到对应的这个镜像[https://hub.](https://hub.docker.com/r/pytorch/pytorch/)

[docker.com/r/pytorch/pytorch](https://docker.com/r/pytorch/pytorch)。点击网页中的 Tags 标签页，我们可以从图1.14看到这个镜像就是已经把 PyTorch 和 CUDA 安装好了的，我们直接使用这个镜像就好啦！

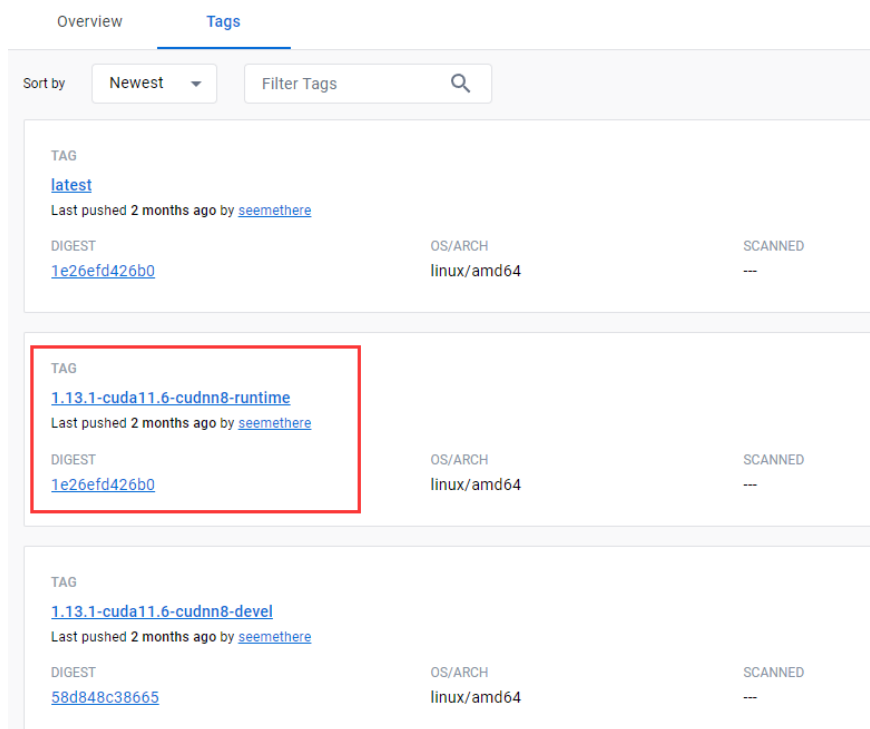


Figure 1.14: caption:pytorch-image-tags-web

下载这个镜像前，还需要登录的。首先去注册个账号，然后打开终端，输入 `docker login` 登录。

然后就可以通过这条命令下载这个镜像了：

```
1 $ docker pull pytorch/pytorch:1.13.1-cuda11.6-cudnn8-runtime
```

这个镜像比较大，下载需要一点时间。完成后，我们再输入 `docker image list` 就可以看到这个镜像了，见图1.15。

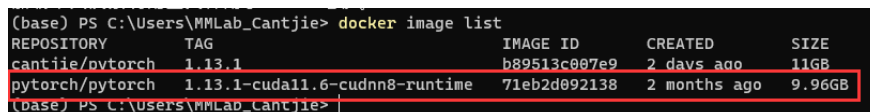


Figure 1.15: caption:docker-image-list-pytorch

### 1.2.3 启动容器

下载完镜像，我们该通过这个镜像启动一个容器了，我们需要到容器里看看这个容器里面是不是有我们需要的环境。

打开终端，输入

```
1 $ docker run -it pytorch/pytorch:1.13.1-cuda11.6-cudnn8-runtime
```

我们发现我们进入了一个 linux 系统，进去运行一下 `nvidia-smi` 试试，诶，怎么 `command not found`，看不到显卡。这是因为容器启动时没有给他指定 GPU。我们输入 `exit`，然后加上 GPU 参数再试一下

```
1 $ docker run --gpus all -it pytorch/pytorch:1.13.1-cuda11.6-cudnn8-runtime
```

进入容器后，我们输入 `nvidia-smi` 等命令，查看运行结果，如图1.16所示，发现正是我们所需要的环境。

```
(base) PS C:\Users\NM\Lab_Cantjie> docker run --gpus all -it pytorch/pytorch:1.13.1-cuda11.6-cudnn8-runtime
root@6ea34f37e644:/workspace# nvidia-smi
Sun Mar  5 03:38:13 2023

+-----+
| NVIDIA-SMI 530.30.02                Driver Version: 531.18      CUDA Version: 12.1     |
| GPU  Name Persistence-M| Bus-Id  Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|  Memory-Usage | GPU-Util  Compute M. |
|=====+=====+
| 0  NVIDIA GeForce RTX 2060      On   00000000:01:00:00 On      1168MiB / 6144MiB      9%      Default |
| 45%   36C   P8             13W / 160W|              |              |
+-----+-----+

+-----+
| Processes: |
| GPU   GI   CI        PID   Type   Process name          GPU Memory |
| ID   ID   ID             |              |           Usage       |
+-----+-----+
| 0   N/A  N/A         78      G   /Xwayland              N/A       |
| 0   N/A  N/A        16291      G   /Xwayland              N/A       |
+-----+-----+

root@6ea34f37e644:/workspace# nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Tue_Mar_ 8_18:18:20_PST_2022
Cuda compilation tools, release 11.6, V11.6.124
Build cuda_11.6.r11.6/compiler.31957947_0
root@6ea34f37e644:/workspace# python
Python 3.10.8 (main, Nov  4 2022, 13:48:29) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.__version__
'1.13.1'
```

Figure 1.16: caption:docker-pytorch-container-env-check

### 1.2.4 安装新包后重新打包成镜像

但是，也有一些包在默认的镜像里是没有的，万一我们需要这些包，比如 `matplotlib` 包，难道我们要每次启动新的容器之后都手动通过 `conda install` 安装一下么？不用的，我们安

装一次之后，将这个容器重新打包成一个新的镜像就好了！我们之后再用，就用新的镜像了。

```
(distributedml) PS C:\Users\MMLab_Cantjie> docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED          STATUS          PORTS          NAMES
23dcfbd17a23   71eb2d    "bash"                  3 minutes ago   Exited (0)     About a minute ago   hopeful_nobel
(distributedml) PS C:\Users\MMLab_Cantjie> docker commit 23dcfbd17a23 new-image:1.13.1
sha256:ee3d894b2f0685c3eb0429923792b2840ddd69026ba30b91ad09d61d27f26e40
(distributedml) PS C:\Users\MMLab_Cantjie> docker image list
REPOSITORY    TAG          IMAGE ID      CREATED          SIZE
new-image     1.13.1      ee3d894b2f06  13 seconds ago  9.98GB
cantjie/pytorch 1.13.1      b89513c007e9  3 days ago     11GB
pytorch/pytorch 1.13.1-cuda11.6-cudnn8-runtime 71eb2d092138  2 months ago   9.96GB
```

Figure 1.17: caption:docker-commit-new-image

在刚才启动的容器里，我们输入 `conda install matplotlib`，安装完成后，输入 `exit` 退出容器。回到 windows 下的命令行，输入 `docker ps -a` 查看所有容器，如图1.17所示，我们发现刚刚安装了 matplotlib 的容器的 id 为 23dcfbd17a23。接下来，我们运行

```
1 $ # docker commit <containerID> <new-image-name>:<tags>
2 $ docker commit 23dcfbd17a23 new-image:1.13.1
```

便将容器打包成了一个镜像。输入 `docker image list`，便可以看到我们新建的容器了。

以后就都可以用这个新镜像了，可是如何使用这个环境呢，我们留到完成具体实验内容的时候再来讲。

### 1.2.5 限制 Docker 内存占用 (可选)

由于 docker 占用内存很大，对于内存不足的电脑可能造成卡顿现象，可以通过修改配置文件限制其内存占用。

修改 C:\users\<username>\.wslconfig

```
1 [wsl2]
2 memory=6GB
3 swap=6GB
4 swapfile=E:\\wsl-swap.vhdx
```

## 1.3 使用华为云计算资源

## 1.4 使用深研院计算资源

注意,在该平台启动的开发环境需要首先手动调整一下 DNS 服务器,即在 `/etc/resolv.conf` 最上方添加一行 “`nameserver 114.114.114.114`” 即可。

助教会为同学们下发用户名和密码。使用自己的账户信息登录网址: <http://10.103.9.27:30000/>(或使用助教的域名<http://sigs-gpu.cantjie.com:30000>)

### 1.4.1 创建开发环境

登录系统后,我们需要创建一个开发环境。如图1.18所示,首先在左侧边栏选择 AI 分区->开发环境,然后点击创建。



Figure 1.18: caption:sigs-platform-dev-env-before-create

在弹出的窗口图1.19中,设置环境名称、镜像、储存位置、资源规格等信息。其中,当与小组同学共享账号时,请不要直接选择默认的资源规格,请通过自定义,限制环境所需的CPU 核心数量、储存空间大小和显卡数量。对于我们的实验,系统自带的镜像已基本可以满足需求,因此在镜像选择中,按图1.20所示选择 `superadmin/pytorch:1.12.0-cuda11.3-cudnn8-runtime-2` 即可(注意末尾的-2,带-2 的版本为本学期初根据课程需求定制的版本,不带-2 的版本使用 ssh 连接进入后会找不到合适环境。)



创建开发环境

\* 开发环境名称

env-test

容器类型

自定义

\* 镜像选择

matrix-registry.h3c.com:8088/

选择

\* 存储位置选择

/

选择

\* 资源规格

4核|40G|1卡

开发环境描述

请输入开发环境描述，长度0~255个字符

确定

取消

Figure 1.19: caption:sigs-platform-dev-env-create

镜像选择

我的镜像

本地HUB

共享镜像

请输入镜像名称

Q

Q

镜像名称	镜像分类	镜像作者	镜像标签	操作
horovod	DEEP LEARNING	superadmin	0.18.0-1f1.14.0-torch1.2	选择
superadmin/tensorflow	DEEP LEARNING	superadmin	2.2.3-gpu-py3	选择
superadmin/pytorch	DEEP LEARNING	superadmin	1.12.0-cuda11.3-cudnn	选择
hpal-tf	DEEP LEARNING,MACHINE LEARNING	superadmin	1.14	选择

共4条

< 1 >

10条/页

前往 1 页

Figure 1.20: caption:sigs-platform-dev-env-image-selection

创建时指定的储存路径会挂载到环境的 `/private` 目录下。

### 1.4.2 安装其他软件或包

新建完成后，点击启动，等待镜像状态变为运行中，访问方式一栏会出现 SSH、远程桌面和 VNC 三种连接方式，操作一栏的打开按钮也会有灰变蓝。

当我们需要使用镜像中尚未安装的工具或包时，可以通过访问方式提供的三种方式安装，譬如，在本地使用 ssh 连接服务器后，运行 `conda install matplotlib` 或 `apt install git` 等；也可通过点击打开按钮，在弹出页面中进入终端并安装，见图1.21。

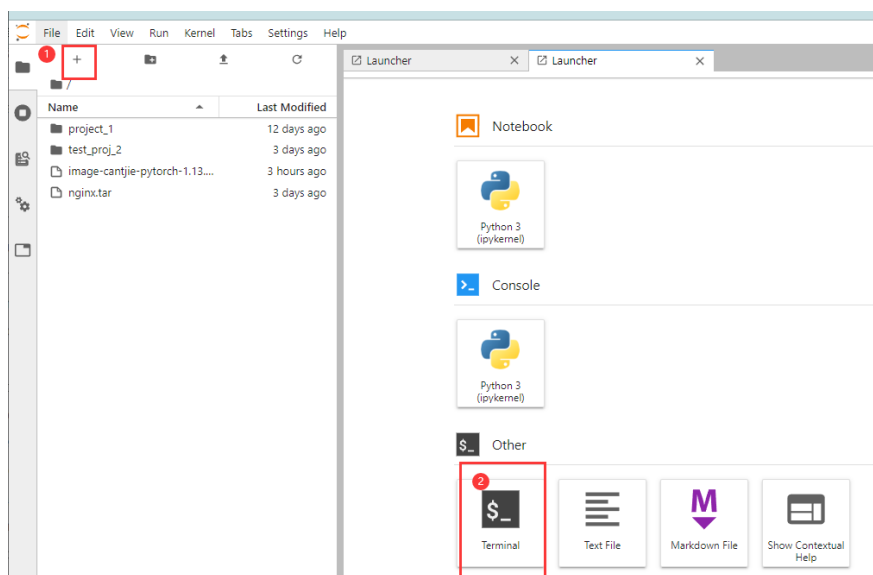


Figure 1.21: caption:sigs-platform-env-install-packages-via-web

### 1.4.3 创建镜像环境

注意，截止本指导书发布时，由于平台缺乏技术文档，个人制作的镜像可能无法让容器正常启动。该小节仍有待完善。

注意，该平台为 *x86* 架构，*arm* 平台上创建的镜像无法运行在该平台

镜像创建方法可参考 §1.2.4。

在本地创建镜像后，使用 `docker save -o filename.tar imagename:tag`<sup>1</sup> 命令导出镜像文件，然后上传至文件管理-> 我的文件栏目下（图1.22）。然后在镜像服务-> 我的镜像中创建镜像（图1.23）。

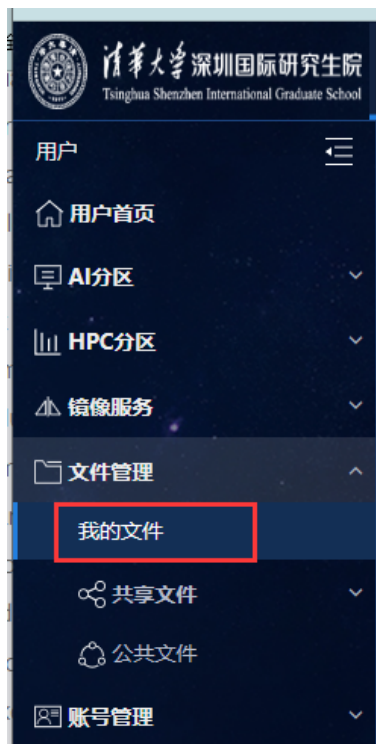


Figure 1.22: caption:sigs-platform-upload-file

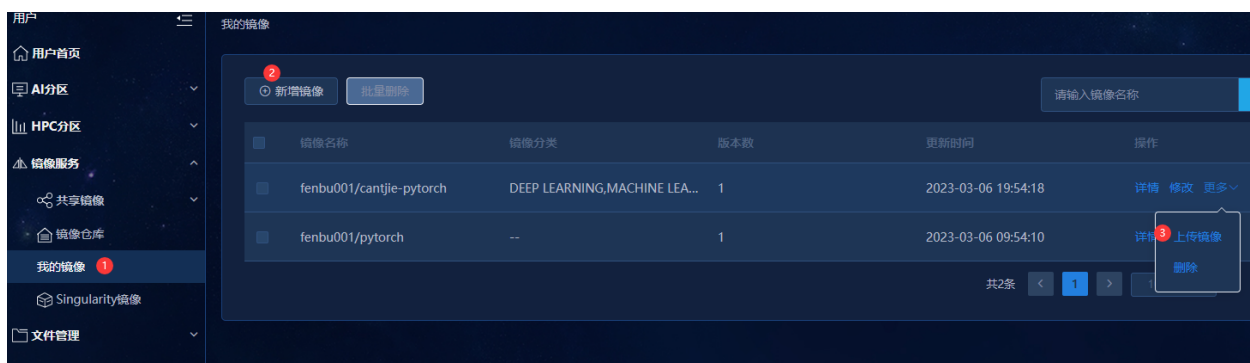


Figure 1.23: caption:sigs-platform-create-custom-image

<sup>1</sup>注意，此处不可使用 sha256 格式的 imageID 代替 <imagename>:<tag> 的格式，不然平台可能无法正确处理该镜像

## 第 2 章 实验一：梯度下降单机优化

### 2.1 实验内容与要点介绍

#### 2.1.1 实验内容与要求

##### 实验内容

- 了解优化器的作用与构建方式（以 PyTorch 为例）
- 构建一阶确定性、一阶随机性优化算法，实现 GD、SGD、Adam 优化算法
- 分析确定性优化算法与随机性优化算法实验结果

##### 实验要求

- 在 MNIST 数据集上完成图像分类任务
- 实现 GD、SGD、ADAM 三种基于梯度的优化方法，写出三个优化器类
- 绘制三种优化方法下的 loss 函数变化图像，通过 loss 图像及其他实验结果，分析三种优化方法的特点

#### 2.1.2 PyTorch 优化器

##### 优化器是干什么用的

下面展示了一段简单的网络训练过程的代码，我们通过这段代码来理解 PyTorch 中优化器所发挥的作用。

```

1 def train_loop(dataloader, model, loss_fn, optimizer):
2     size = len(dataloader.dataset)
3     for batch, (X, y) in enumerate(dataloader):
4         # Compute prediction and loss
5         pred = model(X)
6         loss = loss_fn(pred, y)
7
8         # Backpropagation
9         optimizer.zero_grad()
10        loss.backward()
11        optimizer.step()

```

在这段代码中 `model` 为神经网络模型，通过 `model(X)` 调用了 `model` 中的 `forward` 方法，即进行正向传播，获得神经网络输出（第 5 行）。然后通过损失函数 `loss_fn` 计算神经网络输出 `pred` 与数据真实值或标签 `y` 的差距得到损失值 `loss`（第 6 行）。得到损失值后，通过反向传播（第 10 行），网络 `model` 中的各个参数对应的梯度将会得到更新，得到各个参数的梯度后，优化器 `optimizer` 便可以根据既定的优化算法来更新参数（第 11 行）。需要注意的是，神经网络的梯度参数并不是储存最近一次反向传播（即调用 `loss.backward()`）的结果，而是会将反向传播得到的梯度与当前储存的值相加。因此，我们需要第 9 行 `optimizer.zero_grad()` 来将神经网络 `model` 中储存的梯度值置为 0。

如果你是第一次看到类似代码，你可能还会疑惑上述代码中优化器 `optimizer` 和 `model` 似乎并没有建立联系，那为什么优化器能处理 `model` 中的参数呢？这是因为在这个函数之外，`model` 中的参数 `model.parameters()` 早就被喂给 `optimizer` 了：

```

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

```

## 如何在优化器中实现自己的算法

从上面的例子中可以看到，除了构建函数外，一个最简单的优化器只需要实现 `zero_grad` 和 `step` 方法即可。此处需要注意的有这几点：

- 当我们手动更改中参数或梯度的值时候，需要将其从计算图中分离。即在 `zero_grad` 方法中，应包含 `param.grad.detach_()`。
- 使用 Adam 算法时，由于还需要上一步优化得到的状态，因此可在初始化函数中构建一个字典用来储存状态。

### 2.1.3 几种算法回顾

梯度下降 Gradient Descent:

$$w_{t+1} = w_t - \eta \nabla f(w_t) \quad (2.1)$$

随机梯度下降 Stochastic gradient descent:

$$w_{t+1} = w_t - \eta \nabla f_i(w_t) \quad (2.2)$$

Adam:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) \nabla f(w_t) \quad (2.3)$$

$$g_{t+1} = \beta_2 g_t + (1 - \beta_2) (\nabla f(w_t))^2 \quad (2.4)$$

## 2.2 使用 VSCode 与本地环境调试运行

如果你已经完成了本地环境配置 (§1.1), 那就可以打开 VSCode 进行下面的操作了:

1. 安装 Python 插件, 如图2.1所示。
2. 选择 Python 解释器, 按下 **F1** 或 **Ctrl + Shift + P**, 输入 "select interpreter" 并选择 "Python: Select Interpreter" 项 (图2.2)。然后选择: select at work space level。最后选择你在 §1.1.3一节中创建的环境对应的解释器 (图2.3中为助教自己创建的 distributedml 环境)。
3. 最后, 打开自己的 .py 文件, 可以在编辑器右上角看到一个播放形状的三角, 点击它或在下拉列表中选择运行或调试, 即可开始运行或调试啦。

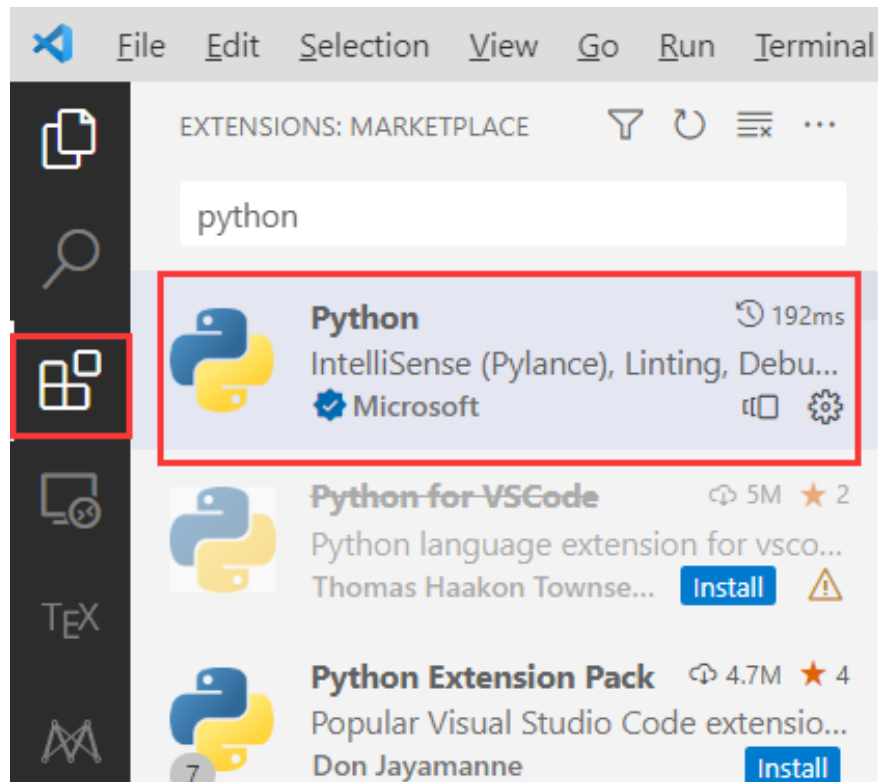


Figure 2.1: caption:task1-vscode-extension-install-python

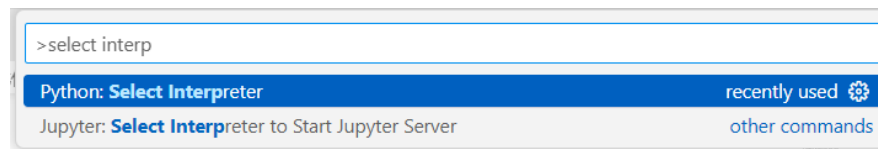


Figure 2.2: caption:task1-vscode-local-select-interpreter

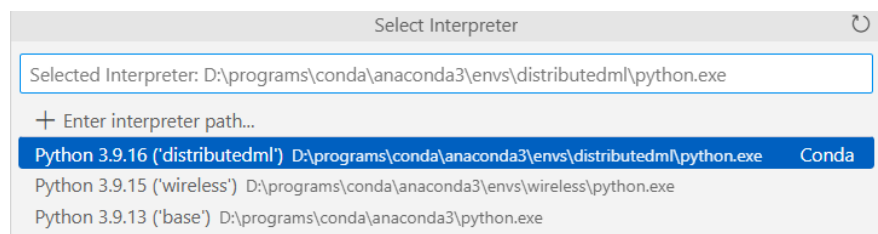


Figure 2.3: caption:task1-vscode-local-select-my-env

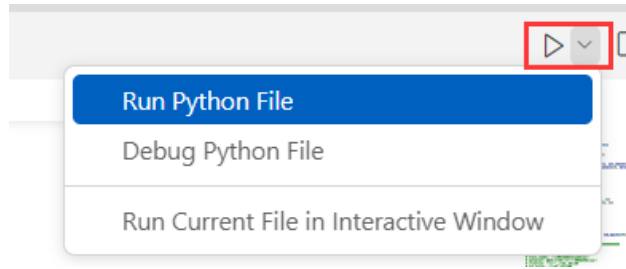


Figure 2.4: caption:task1-vscode-local-run-or-debug

## 2.3 使用 VSCode 与本地容器调试运行

### 2.3.1 启动容器并挂载本地文件夹

在 §1.2.4 一小节中，我们创建了自己的镜像，现在，我们需要先启动这个镜像（对于助教而言是 `cantjie/pytorch:1.13.1`）。但是，目前镜像里面可没有我们写好的代码，而且，就算我们在镜像里面写好代码，该怎么拿出来交作业呢？

为了解决这个问题，我们就需要将本地的目录挂在到容器上，在启动容器是，我们使用 `-v <host-dir>:<container-dir>` 参数，参考下面命令执行：

```
1 $ docker run -it --gpus all -v $pwd/relative/path/to/code:/workspace  
cantjie/pytorch:1.13.1
```

现在进入容器后，我们可以看到，如图2.5所示，本地的代码已经被挂在到了 `workspace` 文件夹下。

```
> docker run -it --gpus all -v $pwd\:/workspace cantjie/pytorch:1.13.1  
root@2573b476cbd1:/workspace# ls  
MyOptimizer.py  pyCache  model.py
```

Figure 2.5: caption:task1-docker-run-with-mount

### 2.3.2 在 VSCode 中使用容器

首先安装 Dev Container 插件，然后按下 `Ctrl + Shift + P`，并找到 `Attatch to Running Container` 命令，如图2.6。

接下来会弹出一个新窗口，在这个新窗口中，就像在本地环境下调试运行一样在容器



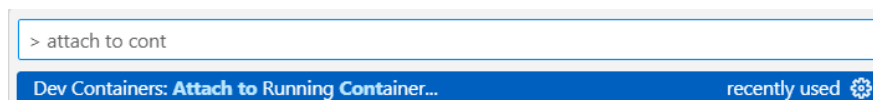


Figure 2.6: caption:task1-vscode-attach-to-container-quick-search

里调试运行即可。余下的步骤基本参考上一小节 §2.2 中的操作即可。即

- 在 VSCode 侧边栏 Explorer 栏目中打开 `/workspace` 目录。
- 在 VSCode 安装 Python 插件。
- 选择编译器为 `/opt/conda/bin/python`

## 2.4 使用 VSCode 与远程服务器调试运行

深研院平台和华为云平台的远程服务器使用方法类似，此处以学校的环境为例。在学校的计算平台创建了开发环境后，平台会提供 ssh 链接地址以及用户名和密码，我们使用该信息链接远程环境。

首先在 VSCode 中安装 Remote SSH 插件，然后按下 `Ctrl + Shift + P`，搜索 Remote-SSH: Open SSH Configuration File 命令（图2.7）。

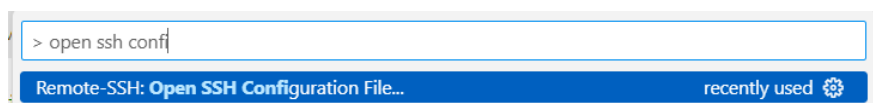


Figure 2.7: caption:task1-open-ssh-config-file

在下拉列表中选择 `C:\Users\<username>\.ssh.`

在打开的 .ssh 文件中，按照图 2.8 给出的格式，添加一个主机。其中 Host 对应昵称，HostName 为远程主机 IP。

最后，`Ctrl + Shift + P` 并搜索 Remote-SSH: Connect to Host 命令，并在后续选择刚刚创建的主机信息。

在弹出的窗口等待连接，并输入密码。余下的步骤又和上一小节一样了：打开文件夹、安装 Python 扩展、指定解释器。此处不再赘述。

```
2 Host raspi
3   HostName 192.168.1.201
4   User base
5
6 Host fenbu001-default-pytorch
7   HostName 10.103.9.38
8   Port 53211
9   User root
```

Figure 2.8: caption:task1-ssh-config-file-demo

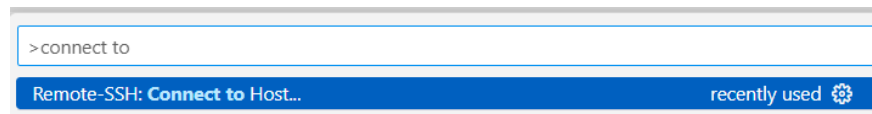


Figure 2.9: caption:task1-vscode-connect-to-host-quick-search

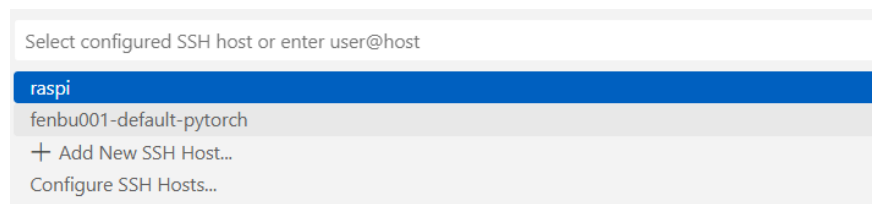


Figure 2.10: caption:taks1-vscode-connect-to-certain-host-quick-search

## 第 3 章 实验二：通信模型与参数聚合

### 3.1 实验内容与要点介绍

#### 3.1.1 实验内容与要求

##### 实验内容

- 了解并掌握分布式算法中的集体通信和参数聚合策略，了解并掌握集体通信中常用的消息传递接口 (Reduce, AllReduce, Gather, AllGather, Scatter, etc.)
- 实现模型参数或模型参数梯度的聚合，尝试使用不同聚合方式 (Sum, Mean, etc.) 对模型的参数和梯度进行聚合，并分析不同聚合策略对模型性能的影响。

##### 实验要求

- 实现集体通信下的参数/梯度聚合，基于至少 2 种集体通信原语实现梯度平均的聚合方法，并比较它们的通信时间开销
- 在框架下设置瓶颈节点，并讨论瓶颈节点对集体通信的影响

#### 3.1.2 多节点通信

在本次实验中，我们采用 PyTorch 中的 `torch.distributed` 模块（下面将简写为 `dist`）作为多节点通信的支持工具。

## 初始化进程组

多节点通信的第一步，是初始化进程组。因此每个节点在训练之前，需要先调用 `dist.init_process_group()` 函数来初始化进程。这个函数会阻塞当前进程，并等待其他进程加入，阻塞持续至直到所有节点（进程）都加入了进来。

对于本次实验，我们需要关注这个函数的三个输入：`backend`，`world_size` 和 `rank`。

- `backend` 指定通信后端，即实现多节点通信的底层的通信协议，对于本次实验，当在 Linux 环境下且使用 GPU 时，一般选择 `nccl`，其他情况下，一般使用 `gloo` 即可（每种后端支持的设备类型和功能有所不同，更多内容可阅读官方文档<https://pytorch.org/docs/stable/distributed.html#backends>）。
- `world_size` 为进程总数。
- `rank` 指定当前进程的优先级。启动多节点时，需要为每个进程指定 `rank`，一般为每个进程赋值为 0 到进程总数-1 中的整数。

但是光有这三个参数还不足以让多节点（进程）之间可以发现彼此，其实还需要告诉节点主进程的通信地址和端口。这里我们采用设置环境变量的方式告诉节点们如何找到 `rank=0` 的主节点。下面这段代码展示了 `rank=0` 的节点的初始化方式。

```
1 import torch.distributed as dist
2
3 # change it to the corresponding ip addr
4 os.environ['MASTER_ADDR'] = 'localhost'
5 os.environ['MASTER_PORT'] = 12355
6
7 # initialize the process group
8 dist.init_process_group(backend="nccl", rank=0, world_size=2)
```

## 广播模型参数

完成进程组初始化后，在神经网络模型训练开始前，需要确保所有节点上的模型是一样的，因此需要将主节点上的模型的参数广播给其他所有节点。我们使用 `dist.broadcast()` 函数来同步所有节点的参数。

下面这段代码展示了广播过程，`dist.broadcast()` 需要两个参数：

- `tensor`，为需要广播或接收的数据。当广播源为当前进程时，`tensor` 将被发送给其

他节点，当广播源不是当前进程时，`tensor` 将被赋为接收到的数据。

- `src`，为广播源的 rank。

```
1 if get_world_size() > 1:
2     for param in model.parameters():
3         dist.broadcast(tensor=param.data,src=0)
```

## 参数聚合

神经网络模型训练过程中，就要实现参数聚合了。该部分请同学们自行完成。

### 3.1.3 记录 GPU 上任务的运行时间

利用 `torch.cuda.Event.elapsed_time()` 记录，示例代码如下：

```
1 start_evt = torch.cuda.Event(enable_timing=True)
2 end_evt = torch.cuda.Event(enable_timing=True)
3 start_evt.record()
4 # the time between start_evt and end_evt will be caculated
5 end_evt.record()
6 torch.cuda.synchronize()
7 whole_time = start_evt.elapsed_time(end_evt)
```

## 3.2 使用进程模拟多节点

为了模拟多节点通信，我们可以在同一台机器上使用不同进程或容器来实现。本节介绍多节点模拟的方法。通过进程模拟的方法在本地、在本地容器中、在华为云、在学校的计算平台都是通用的。

### 3.2.1 手动运行多进程

启动两个终端，分别指定不同的 rank 即可。例如：

```
1 # first process:
2 $ python model.py --n_devices=2 --rank=0
3 # second process:
4 $ python model.py --n_devices=2 --rank=1
```

### 3.2.2 使用 torch.multiprocessing 自动创建多进程

通过 `torch.multiprocessing` 中的 `spawn()` 函数即可让该函数自动帮我们创建多个进程，其中，我们需要关注该函数的三个参数：

- `fn` 为函数名，将作为生成的进程的入口。
- `args` 为 tuple 元组类型。每个进程将通过 `fn(i, *args)` 的方式调用，其中 `i` 即为所生成进程的 rank，从 0 开始逐次递增 1。
- `nprocs` 为生成的进程总数，即前文所指的 `worldsize` 或 `n_devices`。

下面一段代码简要说明了 `spawn()` 函数的使用方法。详情可参考 `model-mp.py`。

```
1 import torch.multiprocessing as mp
2 def main(rank, args):
3     pass
4 if __name__ == "__main__":
5     args = parse_args()
6     mp.spawn(main, (args,), nprocs=args.n_devices)
```

## 3.3 使用容器模拟多节点

“容器就类似于虚拟机了，那通过容器模拟多节点岂不是更真实？”不知道有多少同学也和助教一样这样以为过。那我们就来尝试一下看起来更高端的容器模拟多节点吧。

注意，由于我们只在本地安装了 docker 并自定义了镜像 (§1.2.4)，所以下面的内容针对的是在本地使用容器模拟的过程。当然只要你掌握了方法，在远程的平台上也是一样使用的。

乍一看上去很复杂，多个容器之间的通信怎么处理呢？其实完全不用担心，我们只要使用 docker compose 就可以了，它会帮我们自动配置好同一组容器下的网络。

### 3.3.1 Docker compose 介绍

我们以助教发给大家的 `docker-compose.yml` 为例，我们取其中的一部分先简单分析一下这个文件的内容。

```
1 services:
2     node01:
```

```

3     # container_name: node01
4     image: cantjie/pytorch:1.13.1
5     volumes:
6         - ../workspace          # <host(local) dir (should start with . or
/)>:<dir in container>
7     command:                    # python /workspace/model.py --n_devices=1
--rank=0 --gpu=0
8         - python
9         - -u
10        - /workspace/model.py
11        - --n_devices=2
12        - --rank=0
13        - --gpu=0
14        - --master_addr=localhost
15        - --master_port=12378
16    deploy:                      # make GPU accessible in container
17        resources:
18            reservations:
19                devices:
20                    - driver: nvidia
21                      count: 1
22                      capabilities: [gpu]

```

文件中定义了两个 `services`，每个 `service` 就对应了一个容器，对于每个容器的配置，以 `node01` 为例，我们通过 `image` 指定了镜像，通过 `volumes` 指定了文件挂载路径（参考 §2.3.1），通过 `command` 指定了容器启动后需要执行的命令，下面这个写法实际上是告诉容器执行这条语句：

```

1     $ python -u /workspace/model.py --n_devices=2 --rank=0 --gpu=0 \
2     --master_addr=localhost --master_port=12378

```

其中 `-u` 表示将 Python 中 `print` 命令的输出以 `unbuffer` 的方式输出，这是 docker 容器的一个特性，如果不加 `-u`，我们只能在训练完成、代码跑完之后才能看到程序输出的结果啦。

最后的 `deploy` 则是让容器能够使用宿主机的 GPU（`deploy` 这一段是网上复制来哒，细问我也不懂啦）。

至于 `node02`，除了 `rank` 外，只有 `master_addr` 与 `node01` 不同，对于 `node01` 来说，`master` 就是自己了，而对于 `node02` 来说，`master` 当然是 `node01` 了。

你可能要问，那为什么这里不是写 `master` 的 `ip`，而是写 “`node01`” 就行了呢？这就是 Docker compose 的方便之处了，他会自动修改容器内的 `hosts`，也就是 “`node01`” 就是 `node01` 这个节点的 “域名” 了。

### 3.3.2 通过 Docker compose 启动容器

将这个文件和 `model.py` 放到同一个目录下，然后终端到这里，输入 `docker compose up` 就完成啦，我们就可以看到程序已经开始训练了！如图3.1所示。

```
> docker compose up
[+] Running 2/0
- Container demo-node01-1 Created
- Container demo-node02-1 Created
Attaching to demo-node01-1, demo-node02-1
demo-node01-1 | Device 0 starts training ...
demo-node02-1 | Device 1 starts training ...
demo-node01-1 | Device: 0 epoch: 1, iters: 20, loss: 2.299
demo-node02-1 | Device: 1 epoch: 1, iters: 20, loss: 2.297
demo-node02-1 | Device: 1 epoch: 1, iters: 40, loss: 2.279
demo-node01-1 | Device: 0 epoch: 1, iters: 40, loss: 2.282
demo-node02-1 | Device: 1 epoch: 1, iters: 60, loss: 2.229
demo-node01-1 | Device: 0 epoch: 1, iters: 60, loss: 2.214
demo-node01-1 | Device: 0 epoch: 1, iters: 80, loss: 1.879
```

Figure 3.1: caption:task2-docker-compose-up

这里还需要注意的是，由于我们在 `yaml` 中，并没有使用 `container_name` 为容器指定名字，因此 Docker 生成容器的时候，会按照 `<dir>-<service-name>-<number>` 命名方式为我们的容器命名，如果你的 `docker-compose.yml` 处在一个中文名称的文件夹下，系统很可能会报错的。放到英文命名的文件夹下就好了。



## 第 4 章 实验三：数据并行

### 4.1 实验内容与要点介绍

#### 4.1.1 实验内容与要求

##### 实验内容

- 学习掌握分布式策略中的数据并行训练策略
- 掌握数据并行的原理
- 掌握常用的数据划分策略
- 构建随机采样和随机划分方式将数据集分发到不同的设备，并进行模型训练

##### 实验要求

- 模拟多节点的 DML 系统，将一个完整数据集以不同的划分方式划分，并分配给系统中的节点。
- 实现包括随机采样和随机划分的划分方式，并实现数据并行地训练模型
- 分析数据并行相对于单机训练的性能指标提升，分析不同数据划分方法对模型性能的影响

### 4.1.2 数据集、加载器和采样器

在 Pytorch 中，分布式训练加载数据过程中涉及到这样三个重要的类，即 `Dataset`、`Sampler`、`DataLoader`：

- `Dataset` 类：它直接接触源数据，将数据总数目交给 `Sampler`，将提取数据的接口交给 `DataLoader`。
- `Sampler` 类：定义 `DataLoader` 遍历数据索引的方式。
- `DataLoader` 类：在得到 `Sampler` 提供的索引后，去 `Dataset` 中提取数据，并将得到的数据用于训练。

他们之间的关系可以用图4.1来表示。

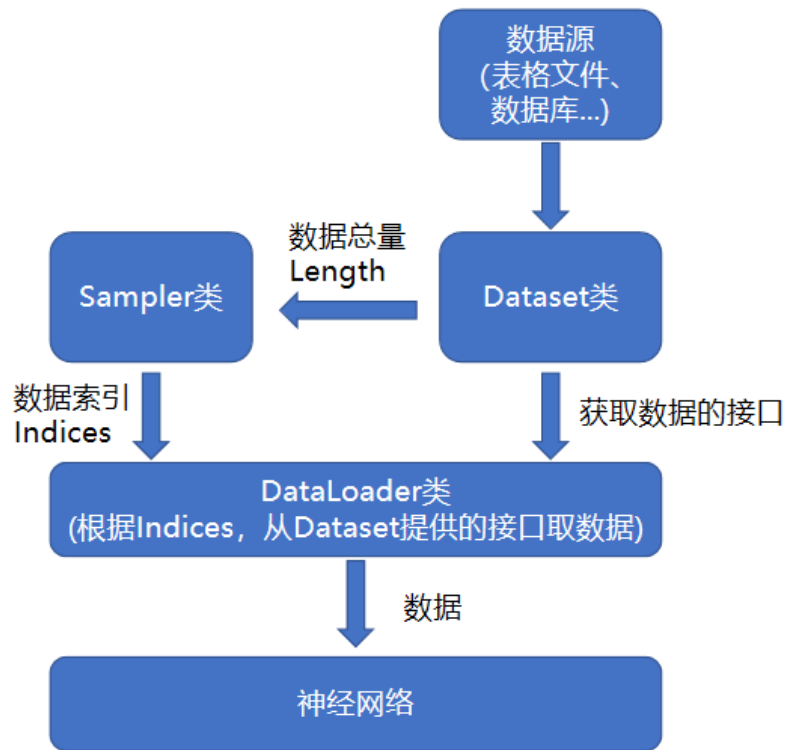


Figure 4.1: caption:task3-sampler-dataset-dataloader

在我们的实验中，我们主要需要实现 `Sampler` 类，除了构造函数外，还需要实现该类中的 `__iter__()` 方法和 `__len__()` 方法。在 `__iter__()` 方法中，需要返回包含样本序列号的一个迭代器，譬如，一段最简单的迭代器可以这样实现：

```
1 def __iter__(self):
2     indices=list(range(len(self.dataset)))
3     return iter(indices)
```

`__len__()` 方法则应当返回上述迭代器中的数据个数。除此之外，我们也可以尝试在 `Sampler` 中实现其他功能，譬如 `set_epoch()` 方法，并在每一轮训练前调用该方法，以避免每一轮训练都得到同样的 `indices` 序列。

## 第 5 章 实验四：模型并行实验

### 5.1 实验内容