GPU 云服务器

最佳实践

产品文档

腾讯云

文档目录

最佳实践
使用 Docker 安装 TensorFlow 并设置 GPU/CPU 支持

最近更新时间：2024-01-11 17:11:13

说明：
本文来自GPU 云服务器用户实践征文，仅供学习和参考。

操作场景

您可通过 Docker 快速在 GPU 实例上运行 TensorFlow，且该方式仅需实例已安装 NVIDIA® 驱动程序，无需安装 NVIDIA® CUDA® 工具包。
本文介绍如何在 GPU 云服务器上，使用 Docker 安装 TensorFlow 并设置 GPU/CPU 支持。

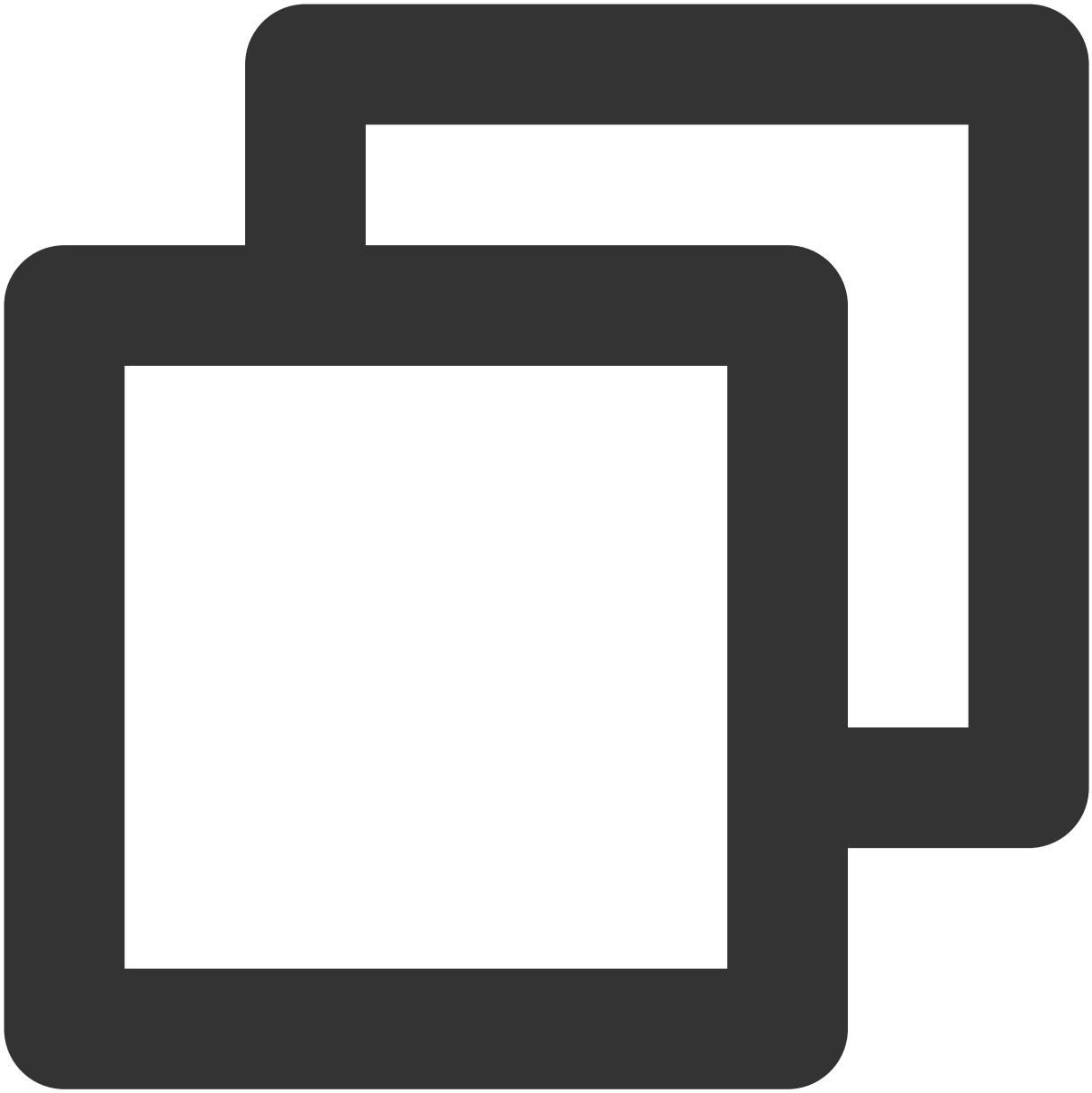
说明事项

本文操作步骤以 Ubuntu 20.04 操作系统的 GPU 云服务器为例。
您的 GPU 云服务器实例需已安装 GPU 驱动。
说明：
建议使用公共镜像创建 GPU 云服务器。若选择公共镜像，则勾选“后台自动安装GPU驱动”即可预装相应版本驱动。
该方式仅支持部分 Linux 公共镜像。

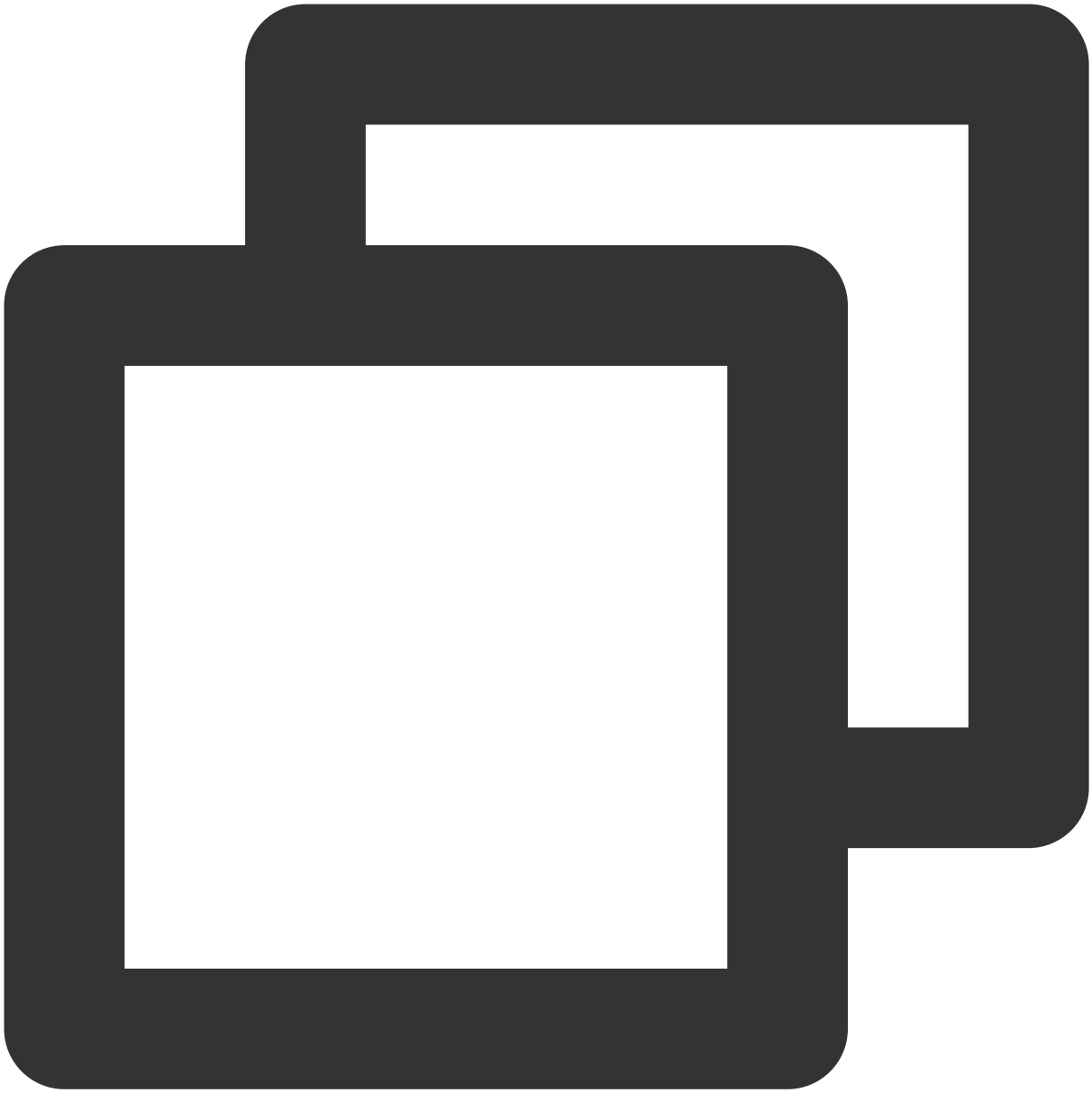
操作步骤

安装 Docker

1. 登录实例，依次执行以下命令，安装所需系统工具。

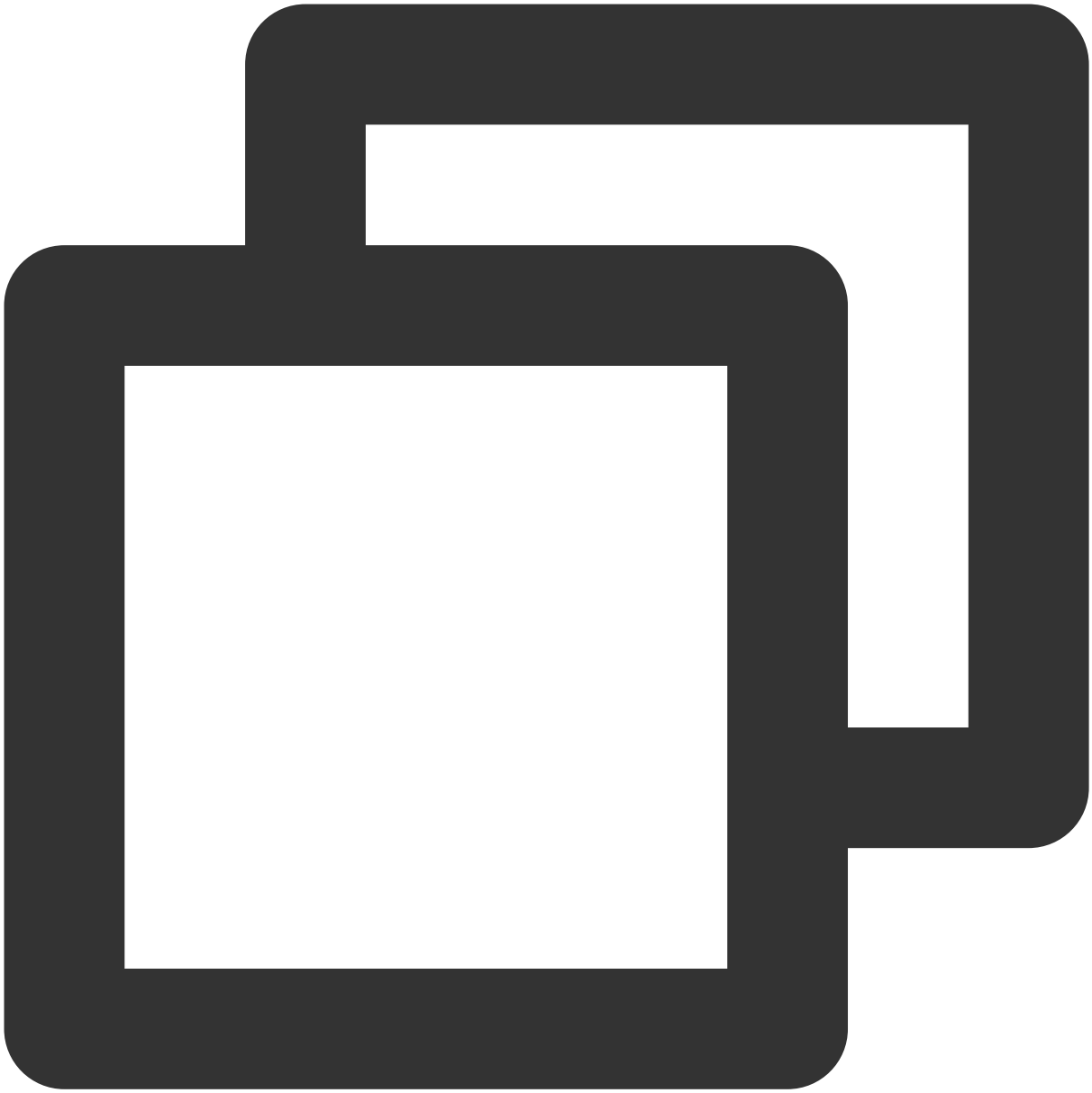


```
sudo apt-get update
```



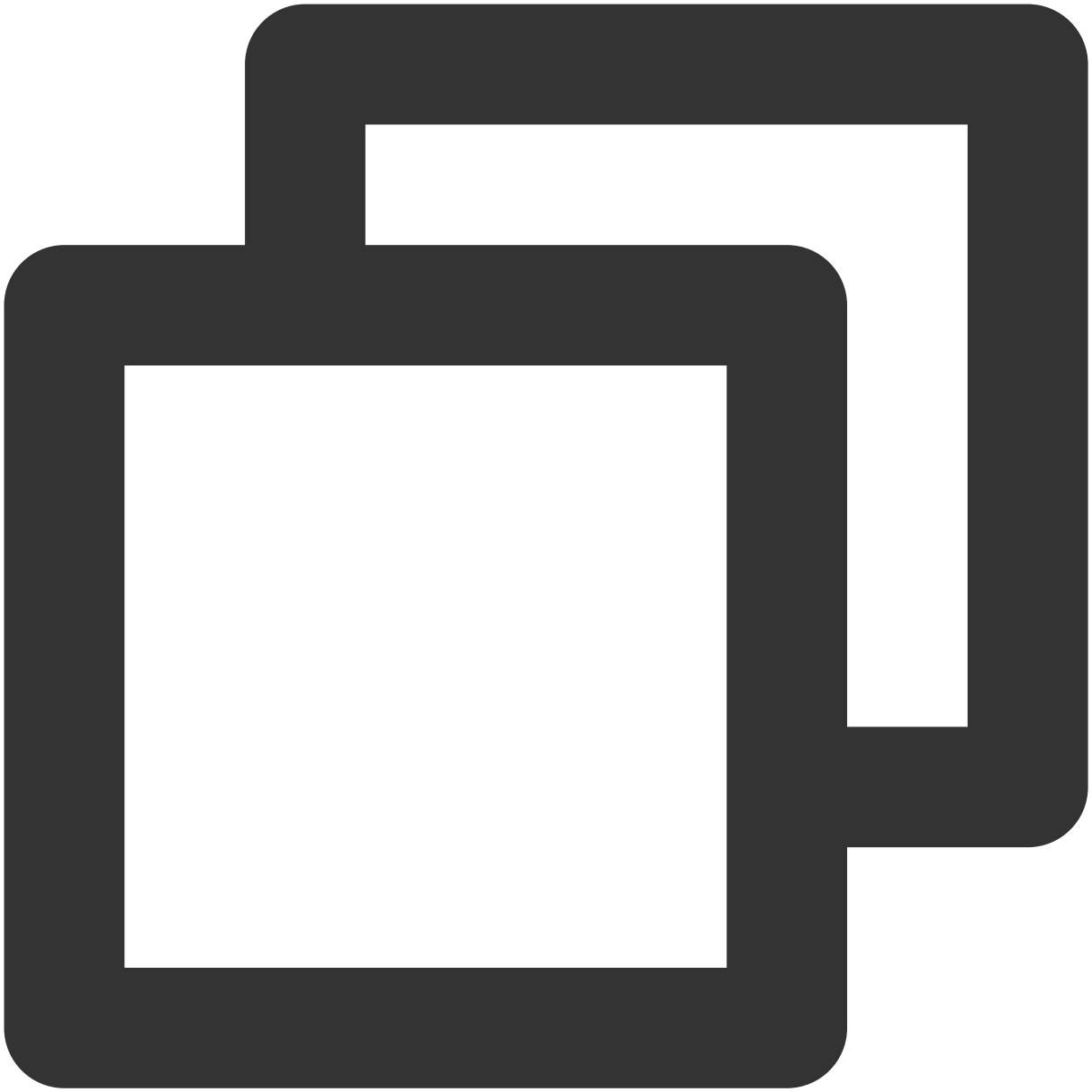
```
sudo apt-get install \\
ca-certificates \\
curl \\
gnupg \\
lsb-release
```

2. 执行以下命令，安装 GPG 证书，写入软件源信息。

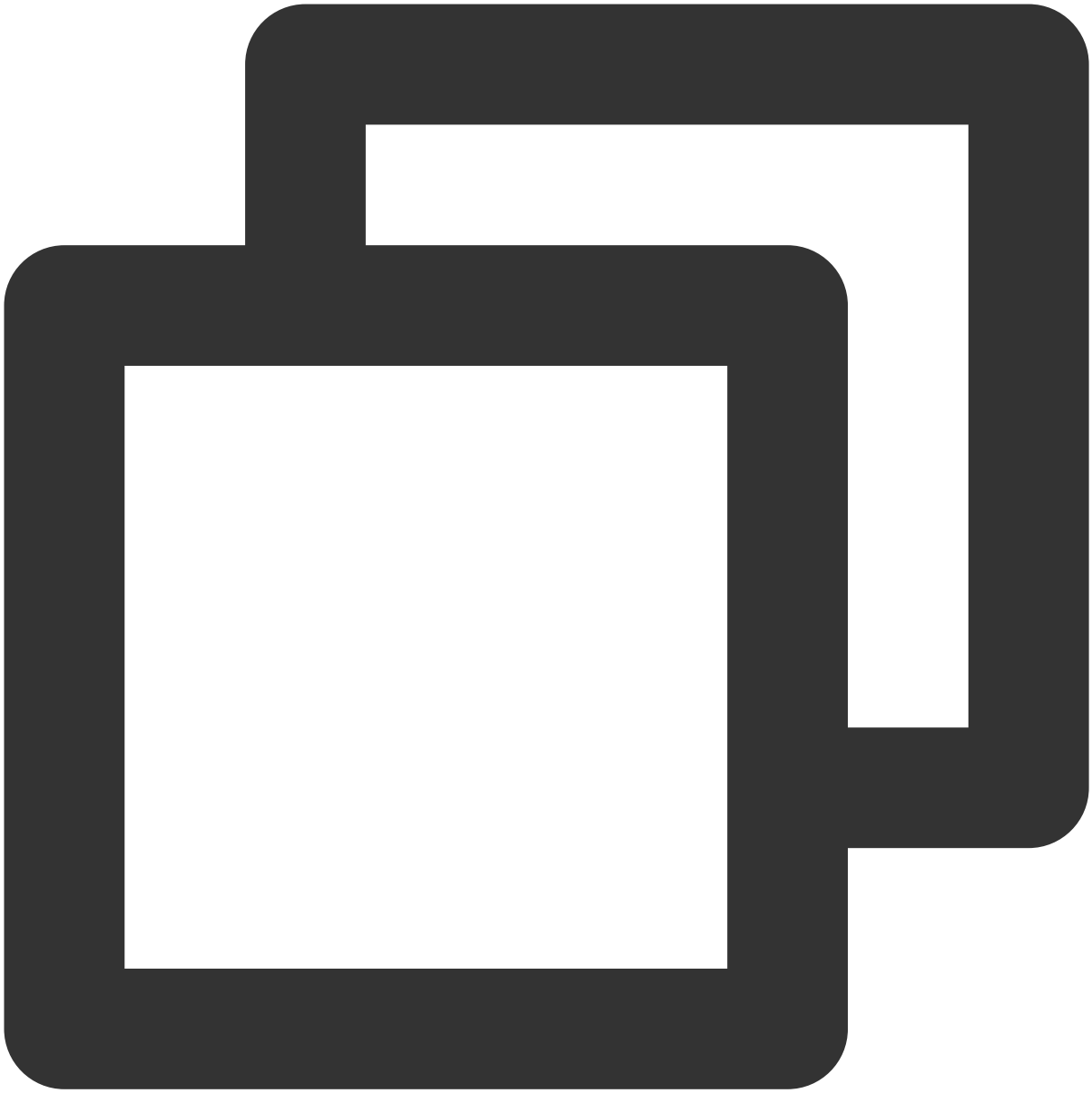


```
sudo mkdir -p /etc/apt/keyrings
 curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /e
 echo \\
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] h
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/
```

3. 依次执行以下命令，更新并安装 Docker-CE。



```
sudo apt-get update
```

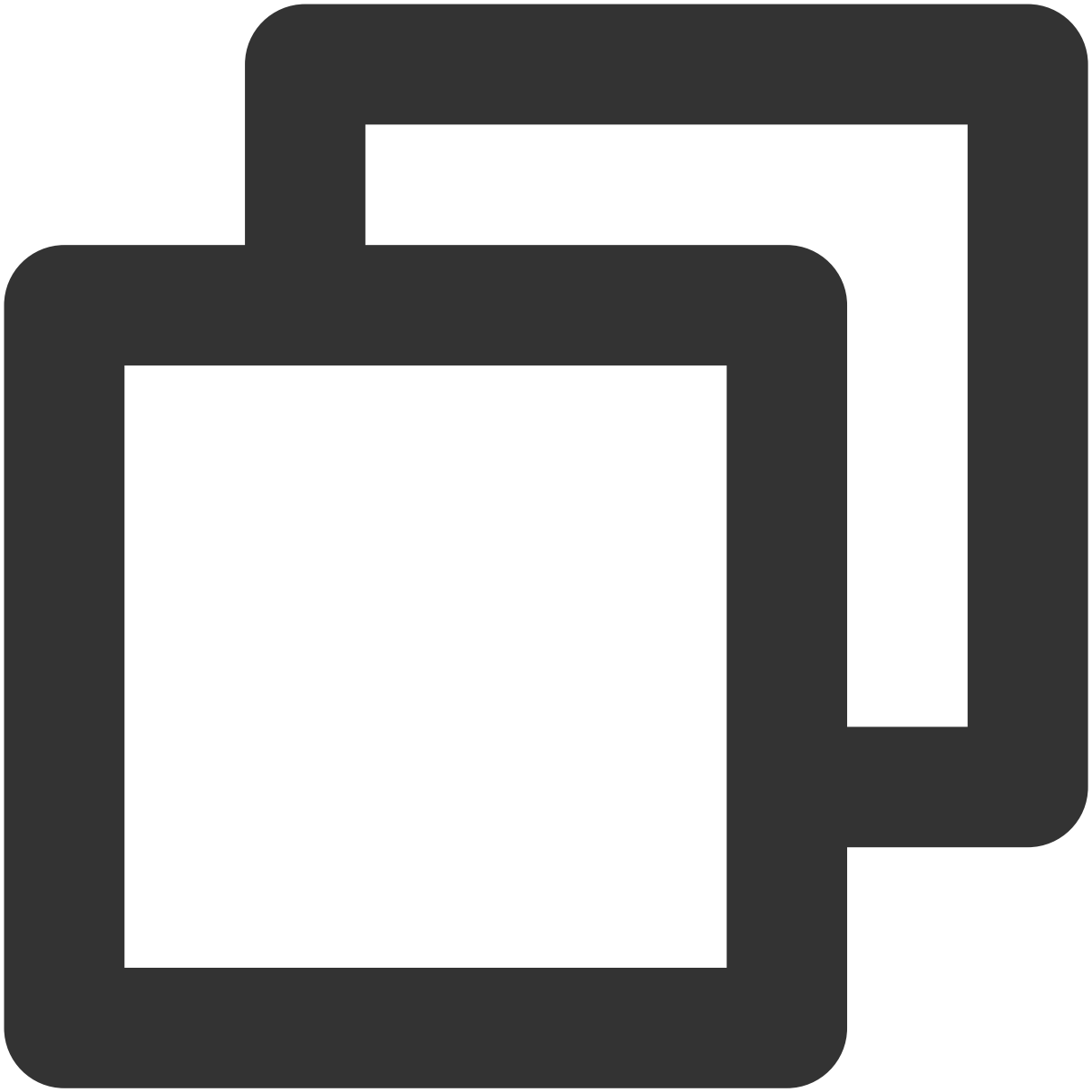


```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

安装 TensorFlow

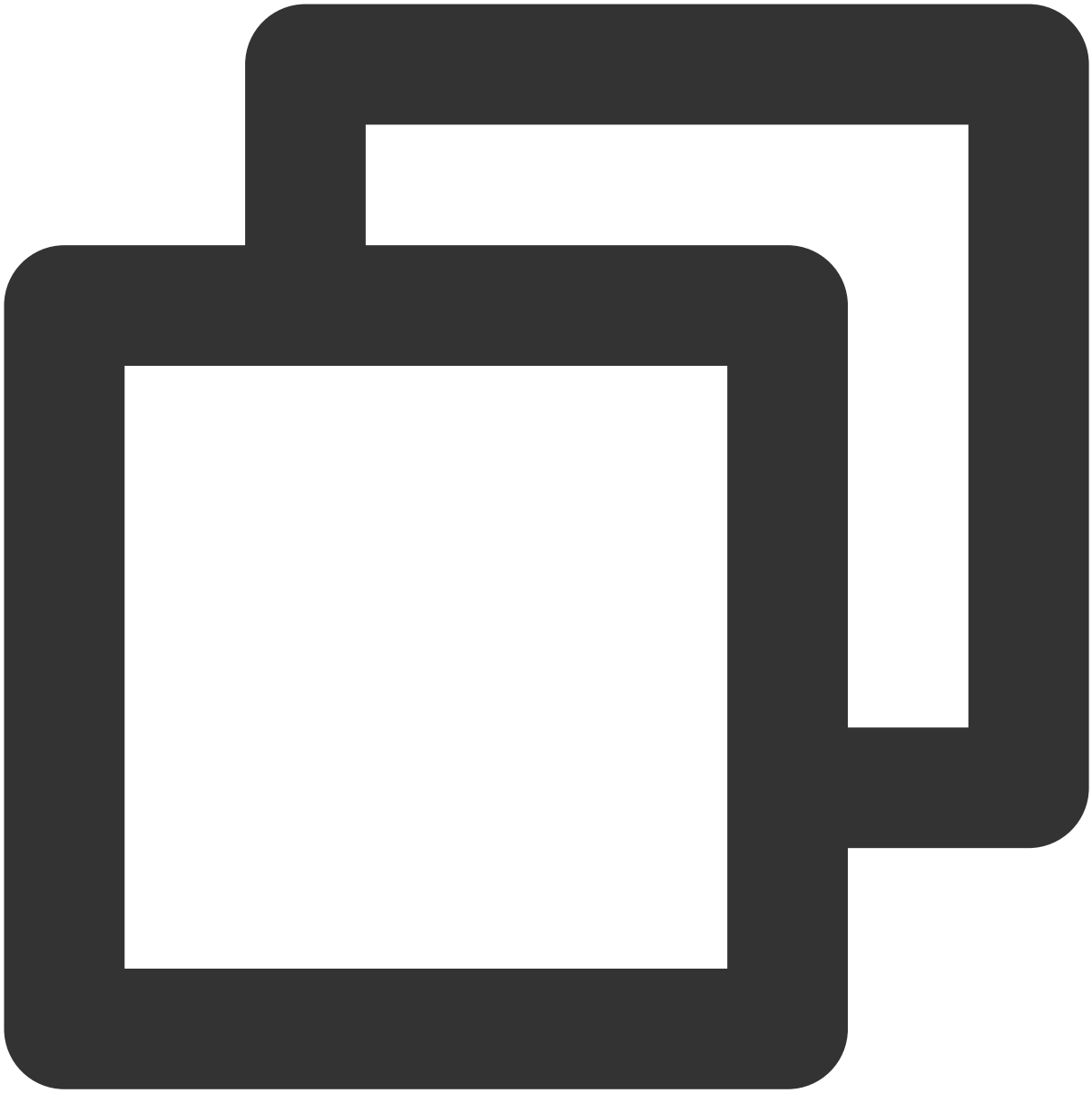
设置 NVIDIA 容器工具包

1. 执行以下命令，设置包存储库和 GPG 密钥。详细信息请参见 Setting up NVIDIA Container Toolkit。

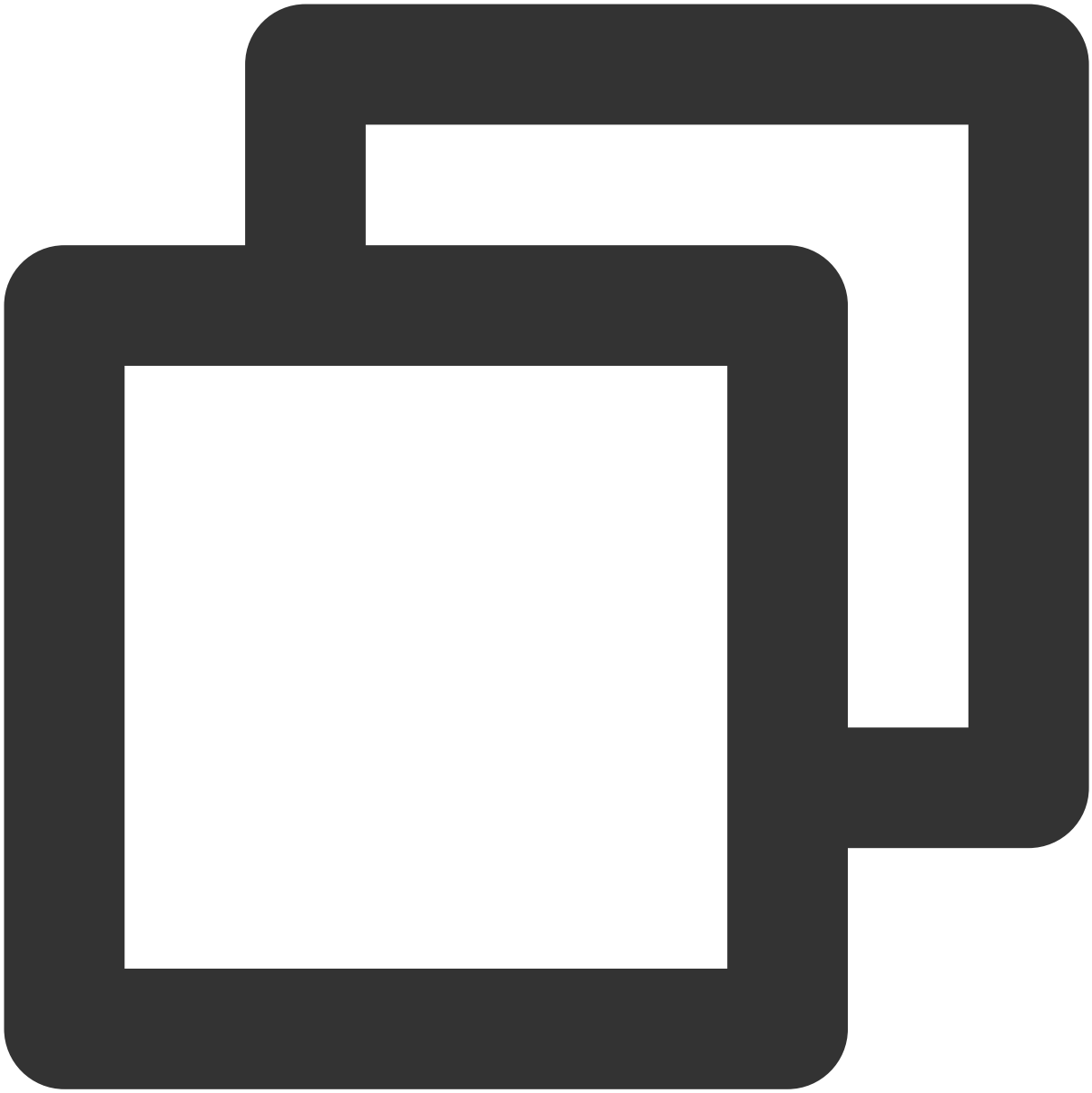


```
distribution=$(. /etc/os-release;echo $ID$VERSION_ID) \\
   && curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | sudo gpg --d
   && curl -s -L https://nvidia.github.io/libnvidia-container/$distribution/libnvid
         sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-to
         sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list
```

2. 执行以下命令，安装 nvidia-docker2 包及依赖项。

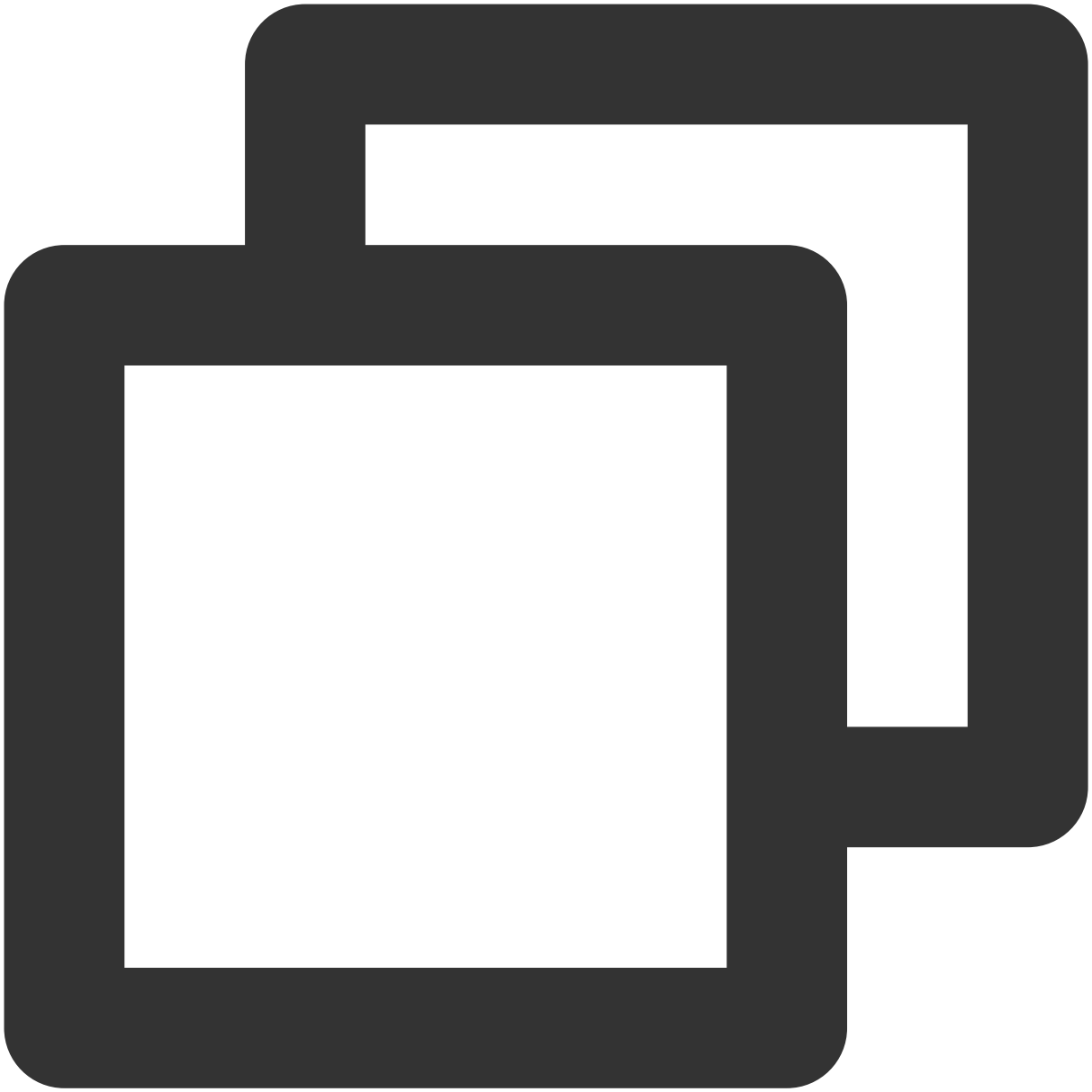


```
sudo apt-get update
```



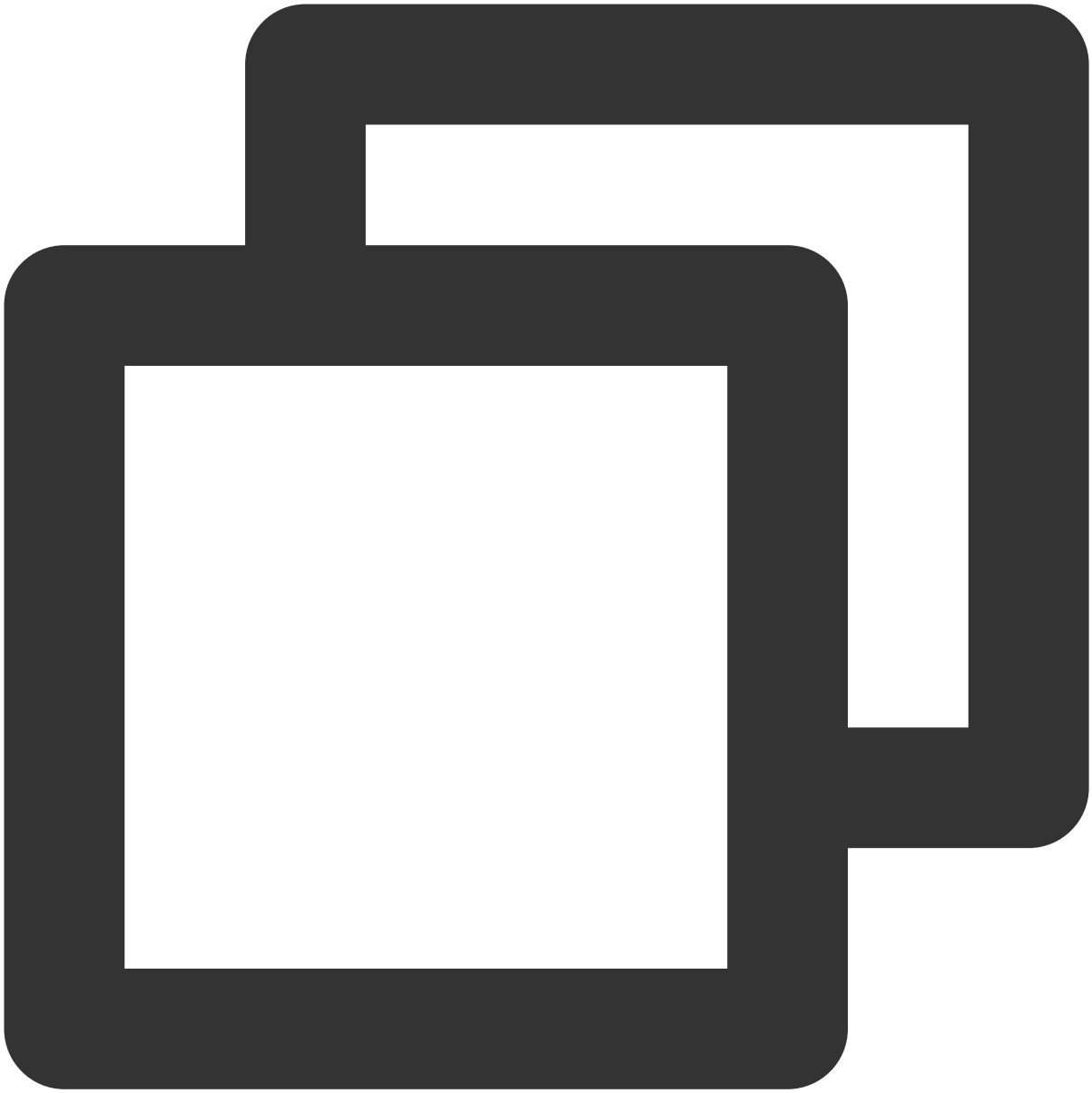
```
sudo apt-get install -y nvidia-docker2
```

3. 执行以下命令，设置默认运行时重启 Docker 守护进程完成安装。



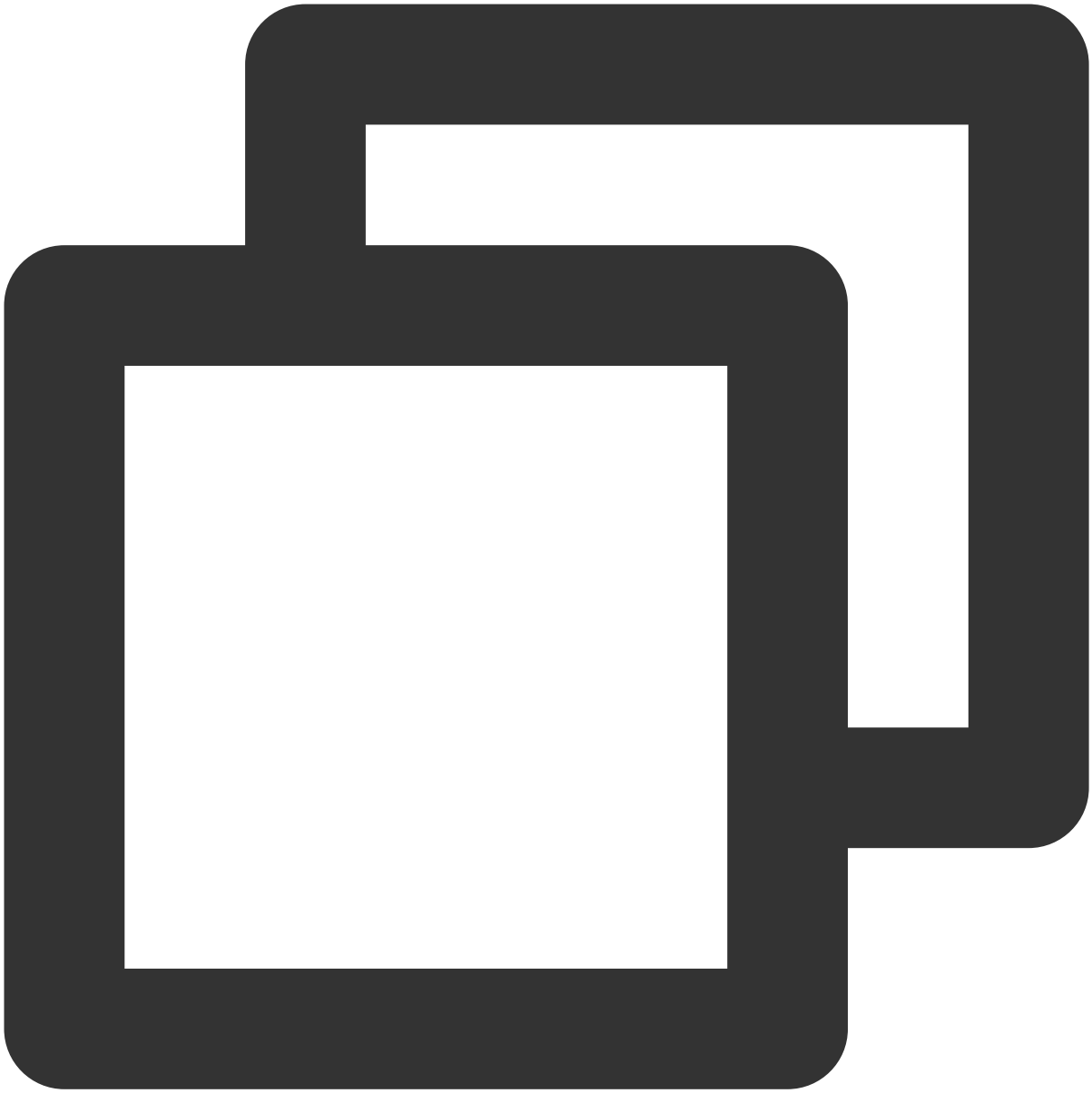
```
sudo systemctl restart docker
```

4. 此时可执行以下命令，通过运行基本 CUDA 容器来测试工作设置。



```
sudo docker run --rm --gpus all nvidia/cuda:11.0.3-base-ubuntu20.04 nvidia-smi
```

返回结果如下所示：



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.51.06    Driver Version: 450.51.06    CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            On   | 00000000:00:1E.0 Off |                    0 |
| N/A   34C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

```
+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

下载 TensorFlow Docker 镜像

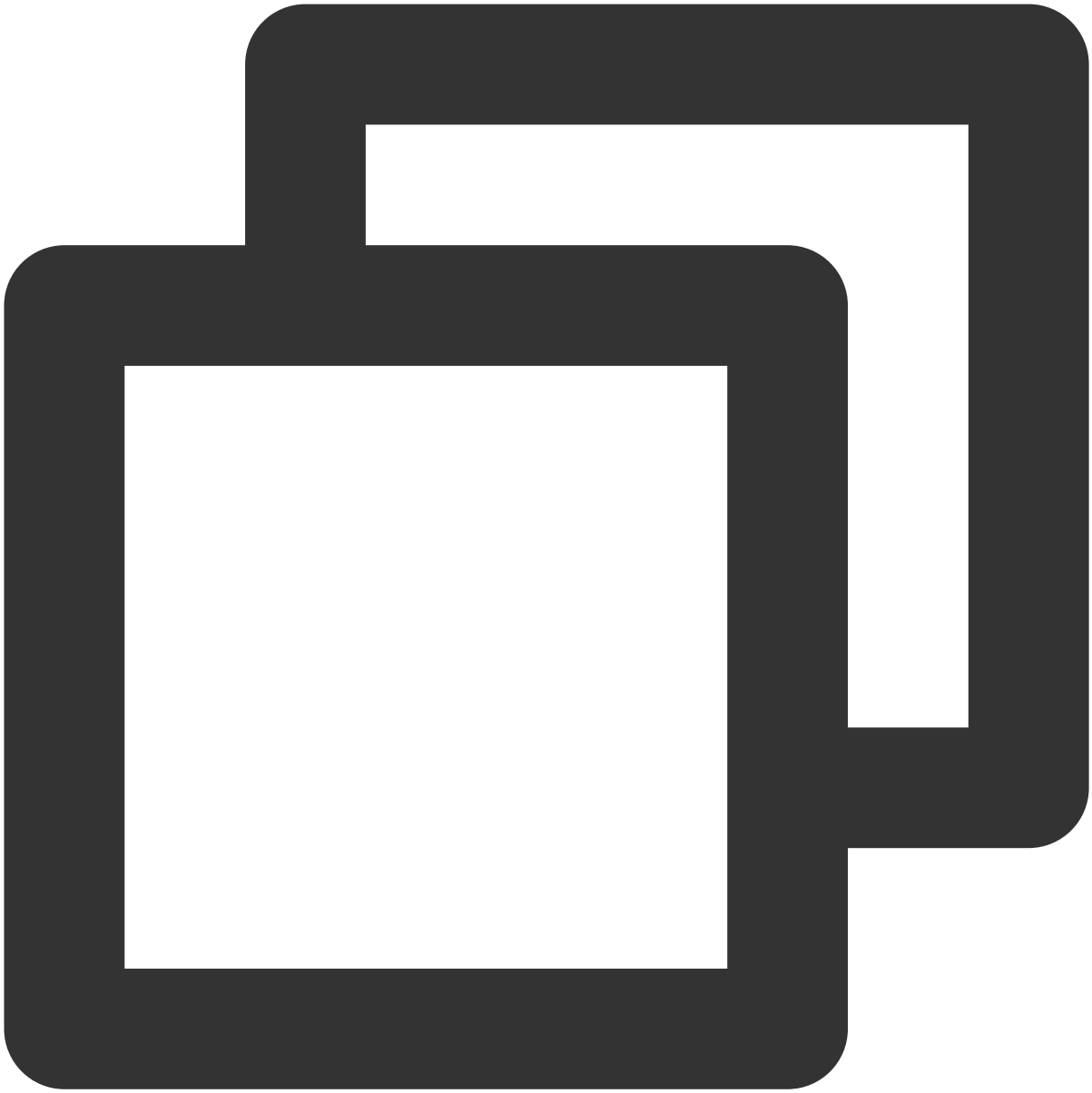
官方 TensorFlow Docker 镜像位于 tensorflow/tensorflow Docker Hub 代码库中。镜像版本按照以下格式进行 标记：

标记	说明
latest	TensorFlow CPU 二进制镜像的最新版本。（默认版本）
nightly	TensorFlow 镜像的每夜版。（不稳定）
version	指定 TensorFlow 二进制镜像的版本，例如 2.1.0。
devel	TensorFlow master 开发环境的每夜版。包含 TensorFlow 源代码。
custom-op	用于开发 TensorFlow 自定义操作的特殊实验性镜像，详情请参见 tensorflow/custom-op。

每个基本标记都有会添加或更改功能的变体：

标记变体	说明
tag -gpu	支持 GPU 的指定标记版本。
tag -jupyter	针对 Jupyter 的指定标记版本（包含 TensorFlow 教程笔记本）。

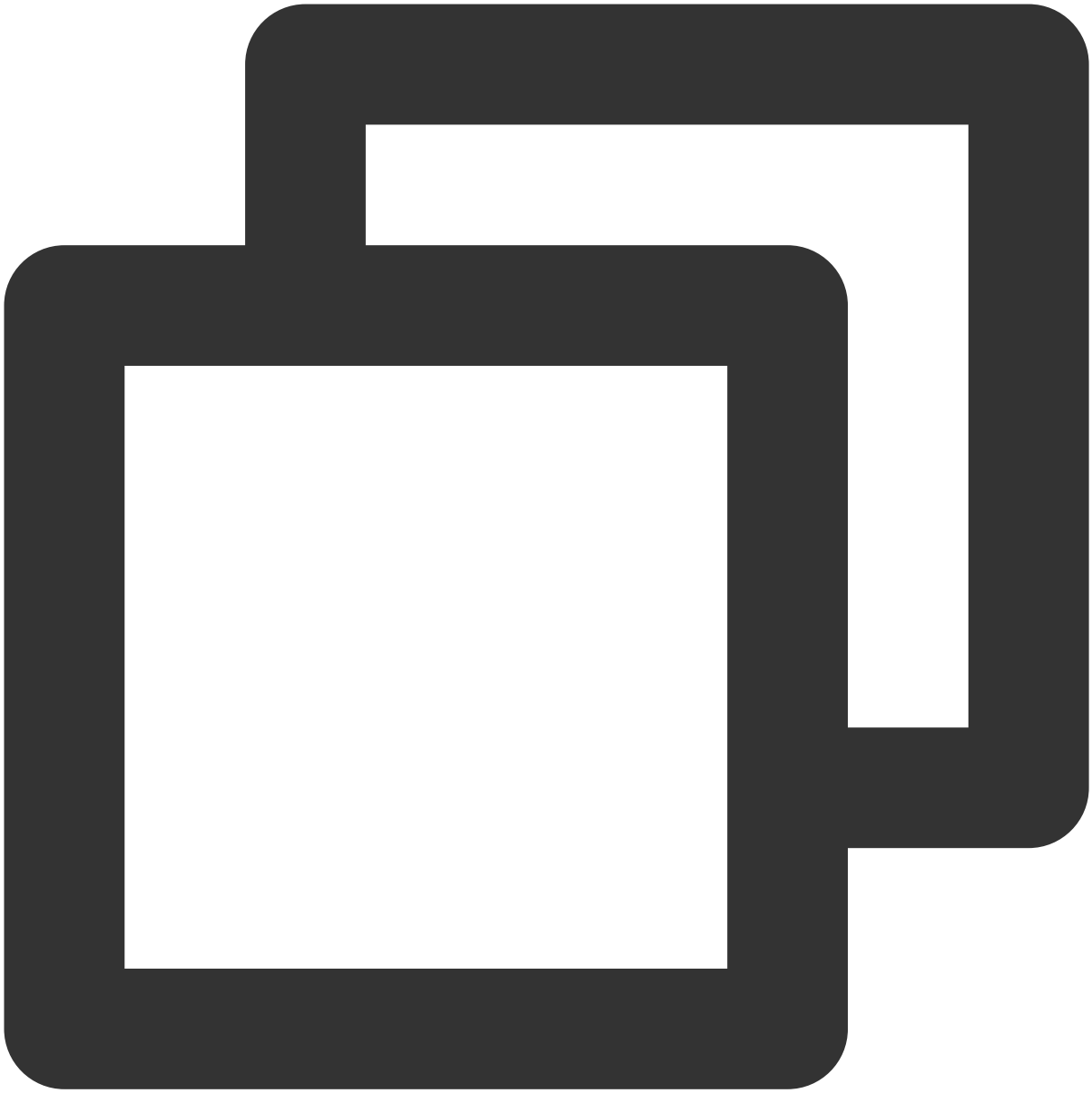
您可以一次使用多个变体。例如，以下命令会将 TensorFlow 版本镜像下载到计算机上：



```
docker pull tensorflow/tensorflow                     # latest stable release
docker pull tensorflow/tensorflow:devel-gpu           # nightly dev release w/ GPU
docker pull tensorflow/tensorflow:latest-gpu-jupyter  # latest release w/ GPU suppo
```

启动 TensorFlow Docker 容器

启动配置 TensorFlow 的容器，请使用以下命令格式。如需了解更多信息，请参见 Docker run reference。

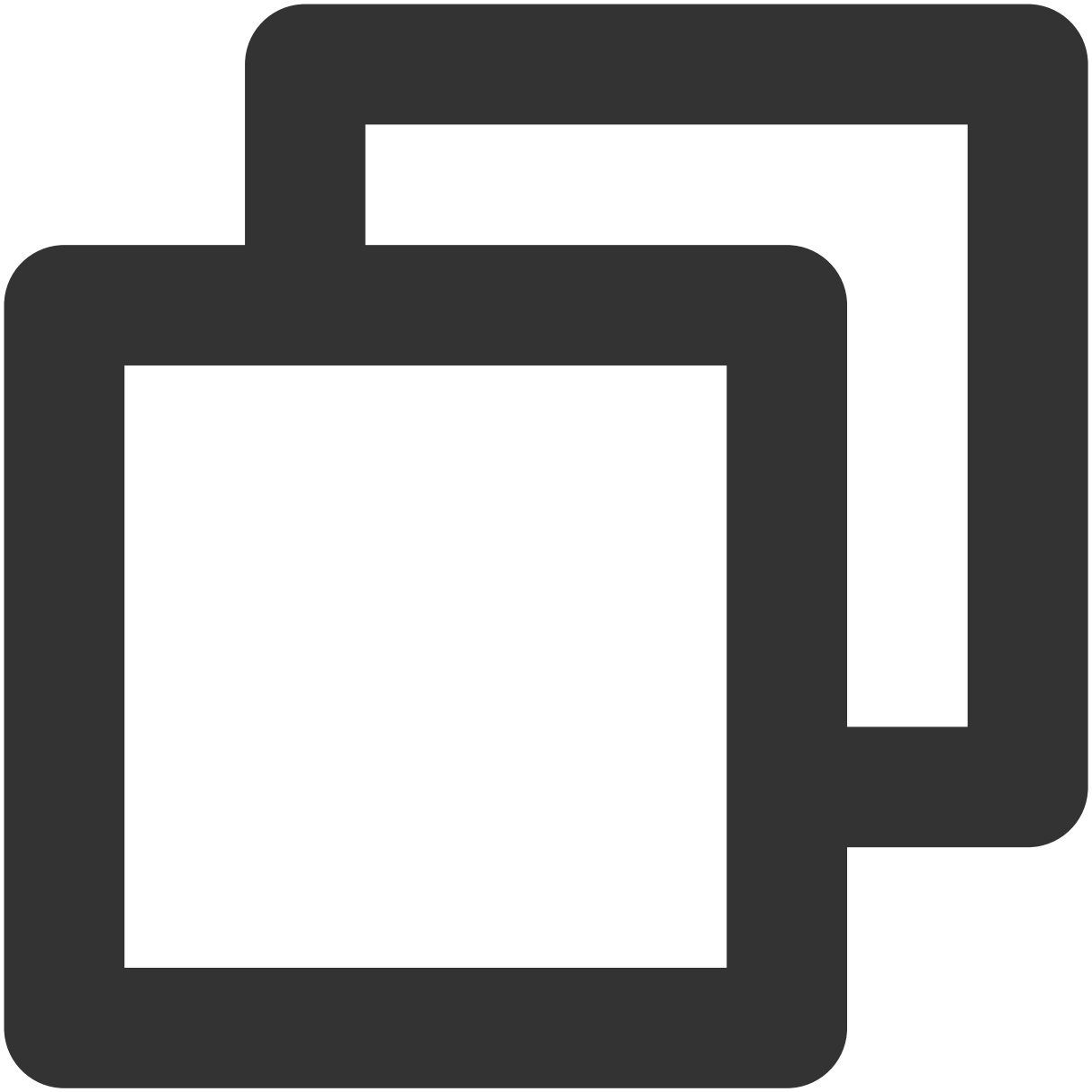


```
docker run [-it] [--rm] [-p hostPort:containerPort] tensorflow/tensorflow[:tag] [co
```

示例

使用仅支持 CPU 的镜像的示例

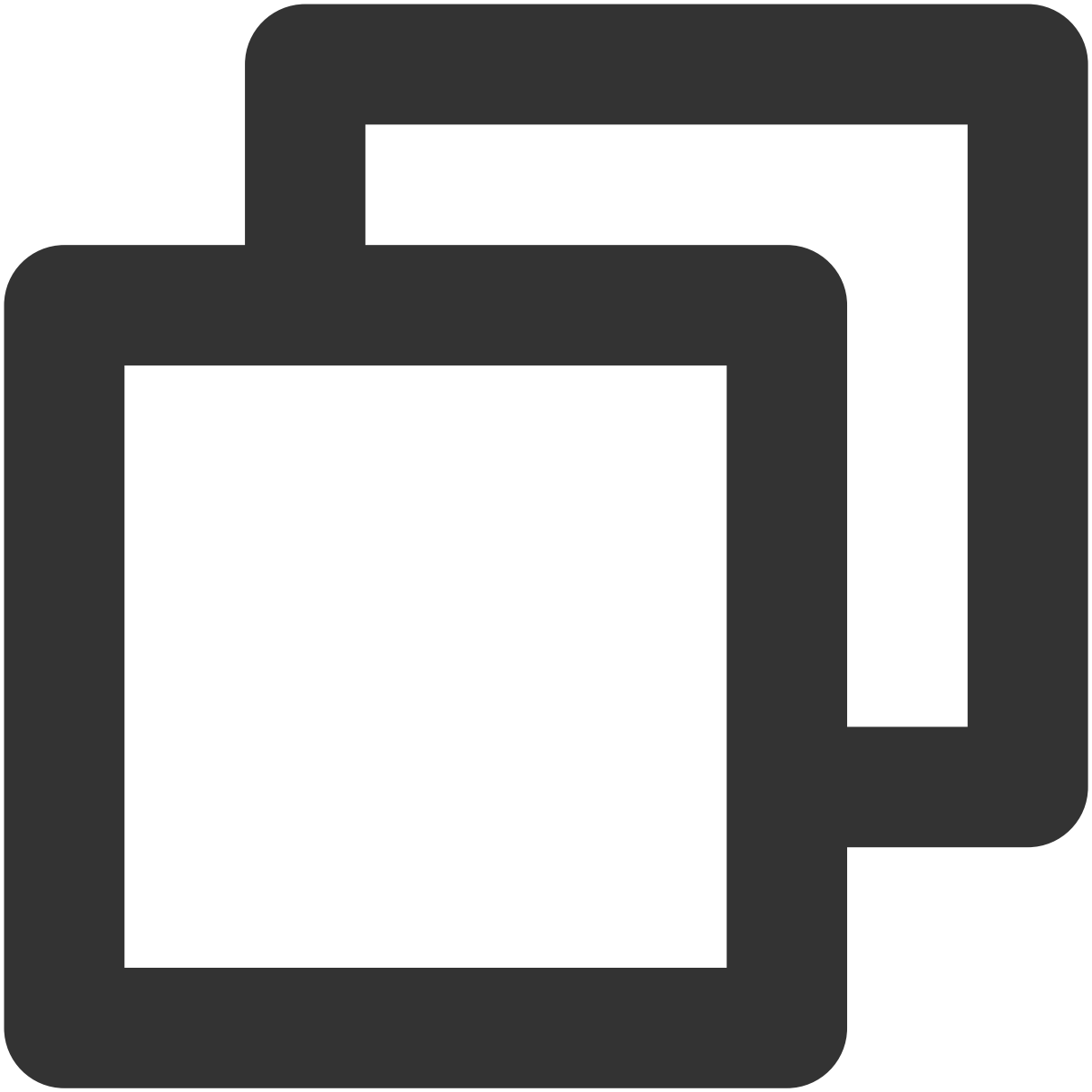
如下所示，使用带 `latest` 标记的镜像验证 TensorFlow 安装效果。Docker 会在首次运行时下载新的 TensorFlow 镜像：



```
docker run -it --rm tensorflow/tensorflow \\
   python -c "import tensorflow as tf; print(tf.reduce_sum(tf.random.normal([1000,
```

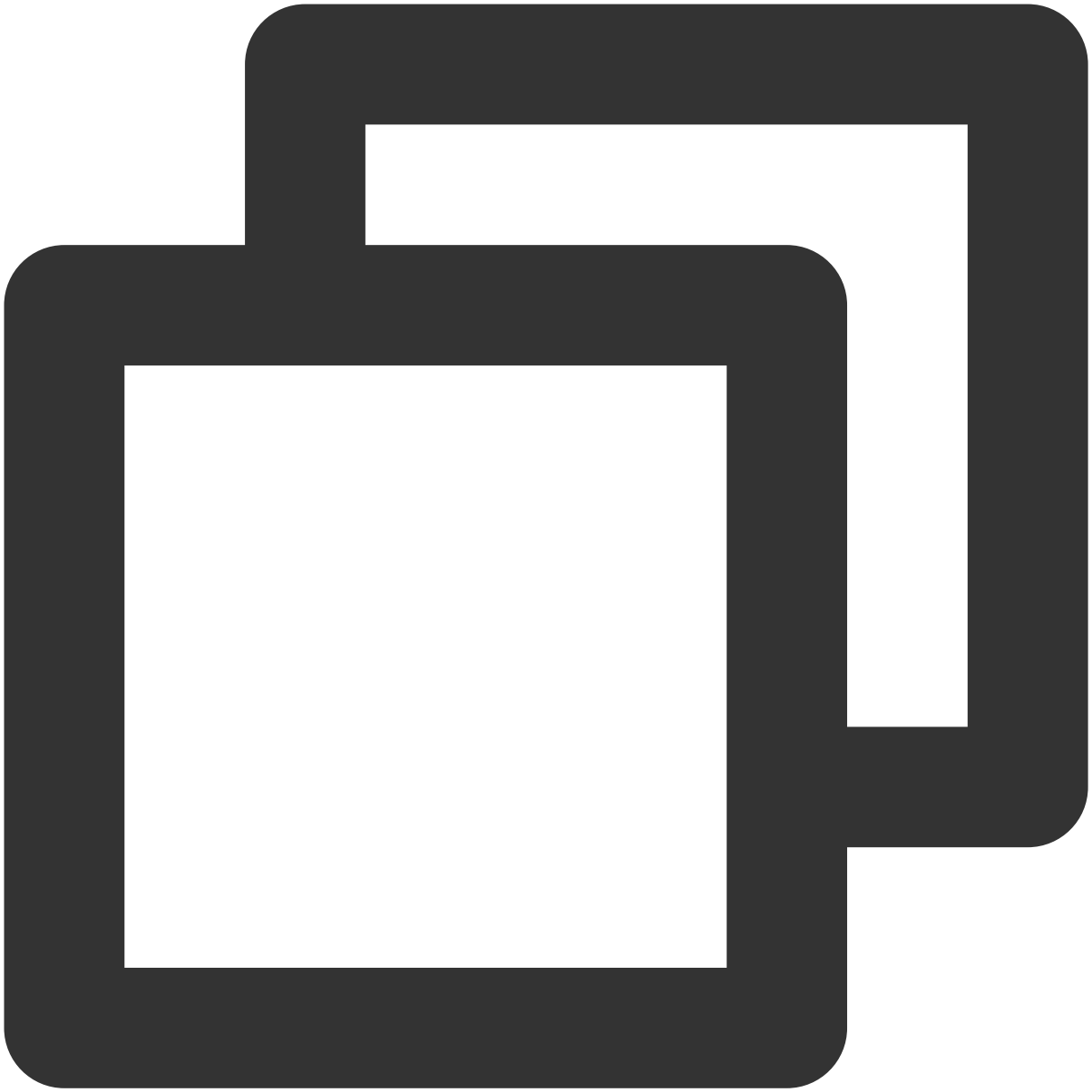
其他 TensorFlow Docker 方案示例如下：

在配置 TensorFlow 的容器中启动 `bash` shell 会话：



```
docker run -it tensorflow/tensorflow bash
```

如需在容器内运行在主机上开发的 TensorFlow 程序，请通过 `-v hostDir:containerDir -w workDir` 参数，装载主机目录并更改容器的工作目录。示例如下：

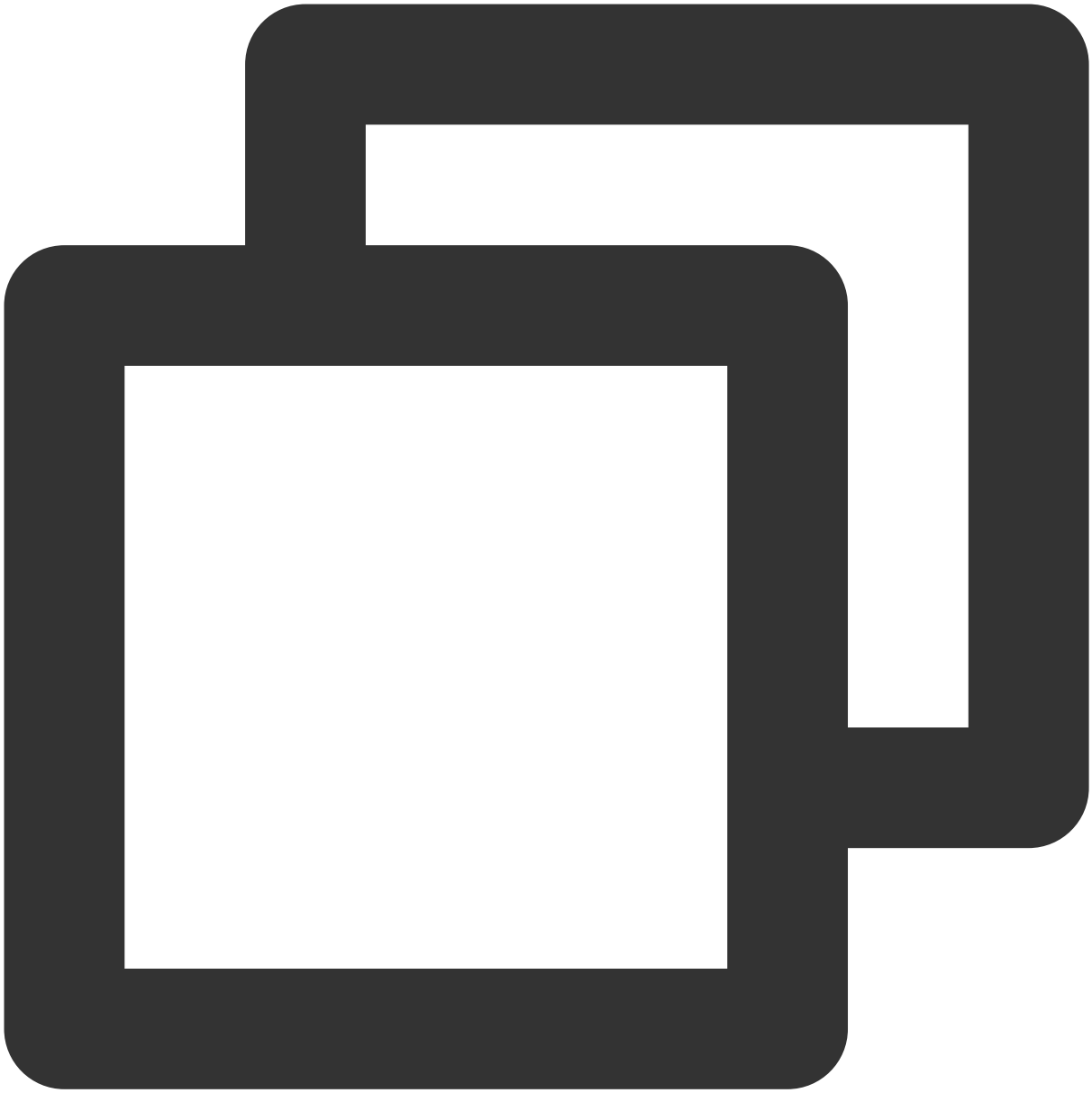


```
docker run -it --rm -v $PWD:/tmp -w /tmp tensorflow/tensorflow python ./script.py
```

说明：

向主机公开在容器中创建的文件时，可能会出现权限问题。通常情况下，最好修改主机系统上的文件。

使用 nightly 版 TensorFlow 启动 Jupyter 笔记本服务器：

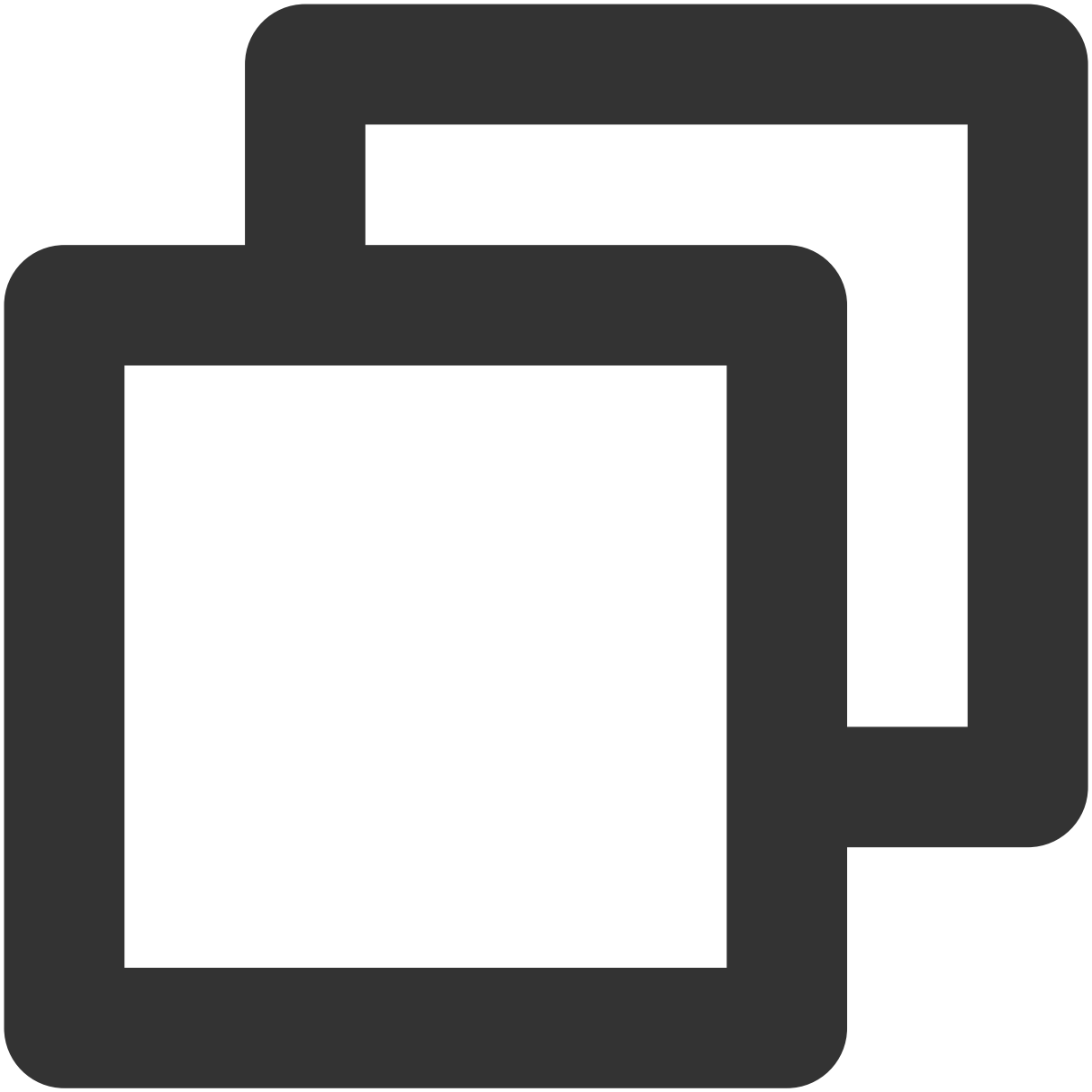


```
docker run -it -p 8888:8888 tensorflow/tensorflow:nightly-jupyter
```

请参考 Jupyter 官网 相关说明，使用浏览器访问 `http://127.0.0.1:8888/?token=...` 。

使用支持 GPU 的镜像的示例

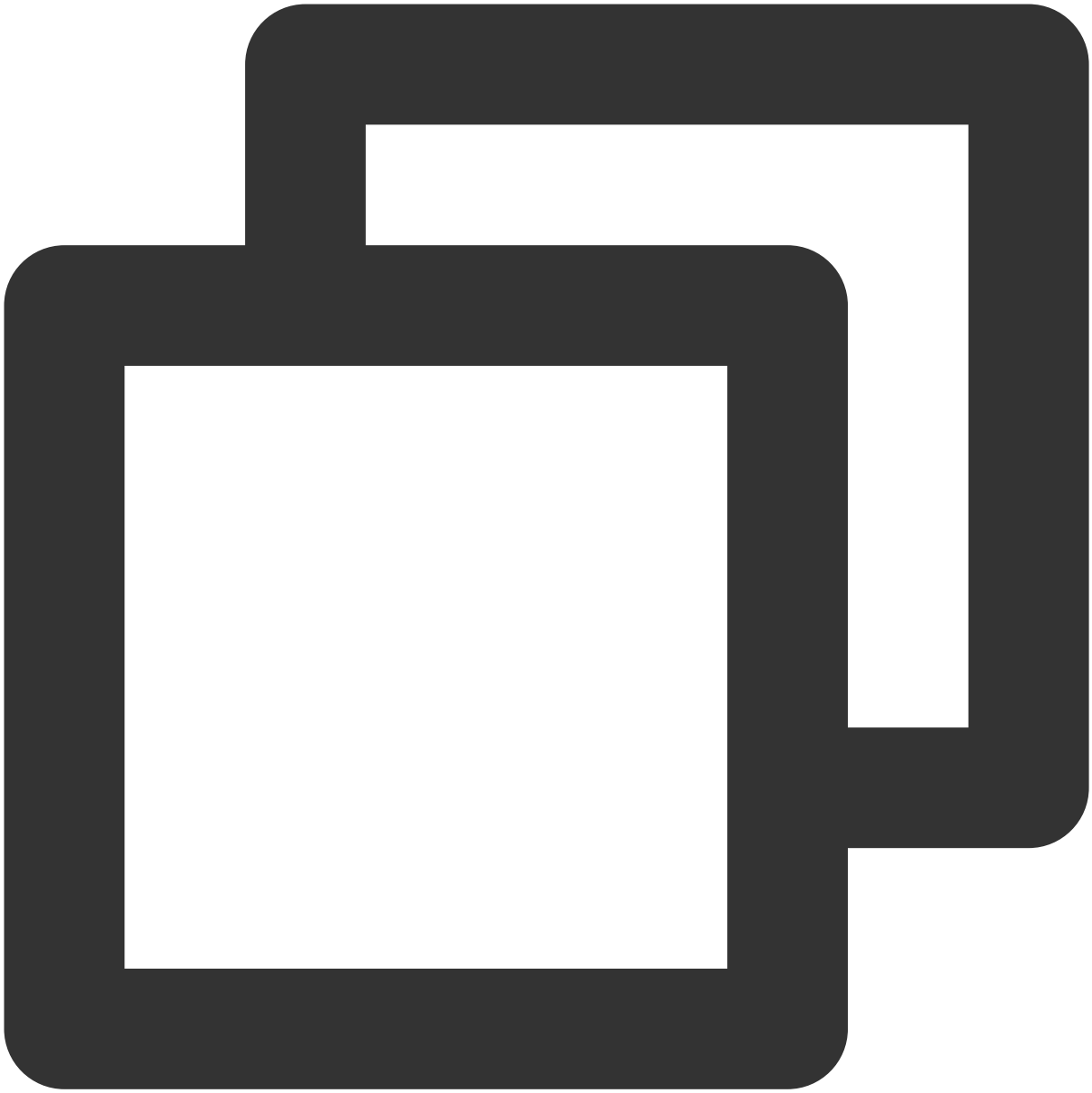
执行以下命令，下载并运行支持 GPU 的 TensorFlow 镜像。



```
docker run --gpus all -it --rm tensorflow/tensorflow:latest-gpu \\
   python -c "import tensorflow as tf; print(tf.reduce_sum(tf.random.normal([1000,
```

设置支持 GPU 镜像可能需要一段时间。如果重复运行基于 GPU 的脚本，您可以使用 `docker exec` 重复使用容器。

执行以下命令，使用最新的 TensorFlow GPU 镜像在容器中启动 `bash` shell 会话：



```
docker run --gpus all -it tensorflow/tensorflow:latest-gpu bash
```

使用 GPU 云服务器训练 ViT 模型

最近更新时间：2024-01-11 17:11:13

说明：

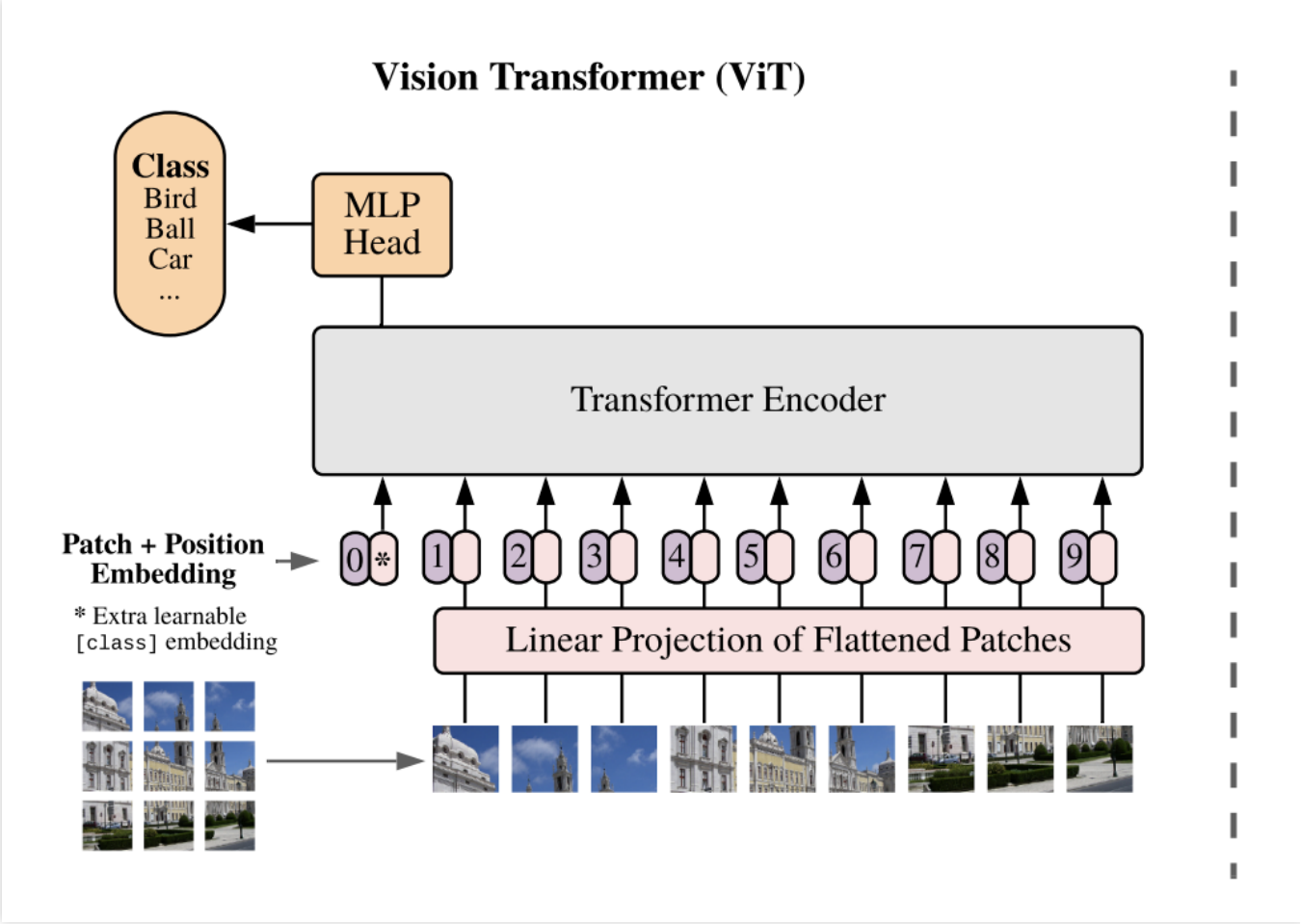
本文来自GPU 云服务器用户实践征文，仅供学习和参考。

操作场景

本文介绍如何使用 GPU 云服务器进行 ViT 模型离线训练，完成简单的图像分类任务。

ViT 模型简介

ViT 全称 Vision Transformer，该模型由 Alexey Dosovitskiy 等人提出，在多个任务上取得 SoTA 结果。示意图如下：



对于一幅输入的图像，ViT 将其划分为多个子图像 patch，每个 patch 拼接 position embedding 后，和类别标签一起作为 Transfomer Encoder 的一组输入。而类别标签位置对应的输出层结果通过一个网络后，即得到 ViT 的输出。在预训练状态下，该结果对应的 ground truth 可以使用掩码的某个 patch 作为替代。

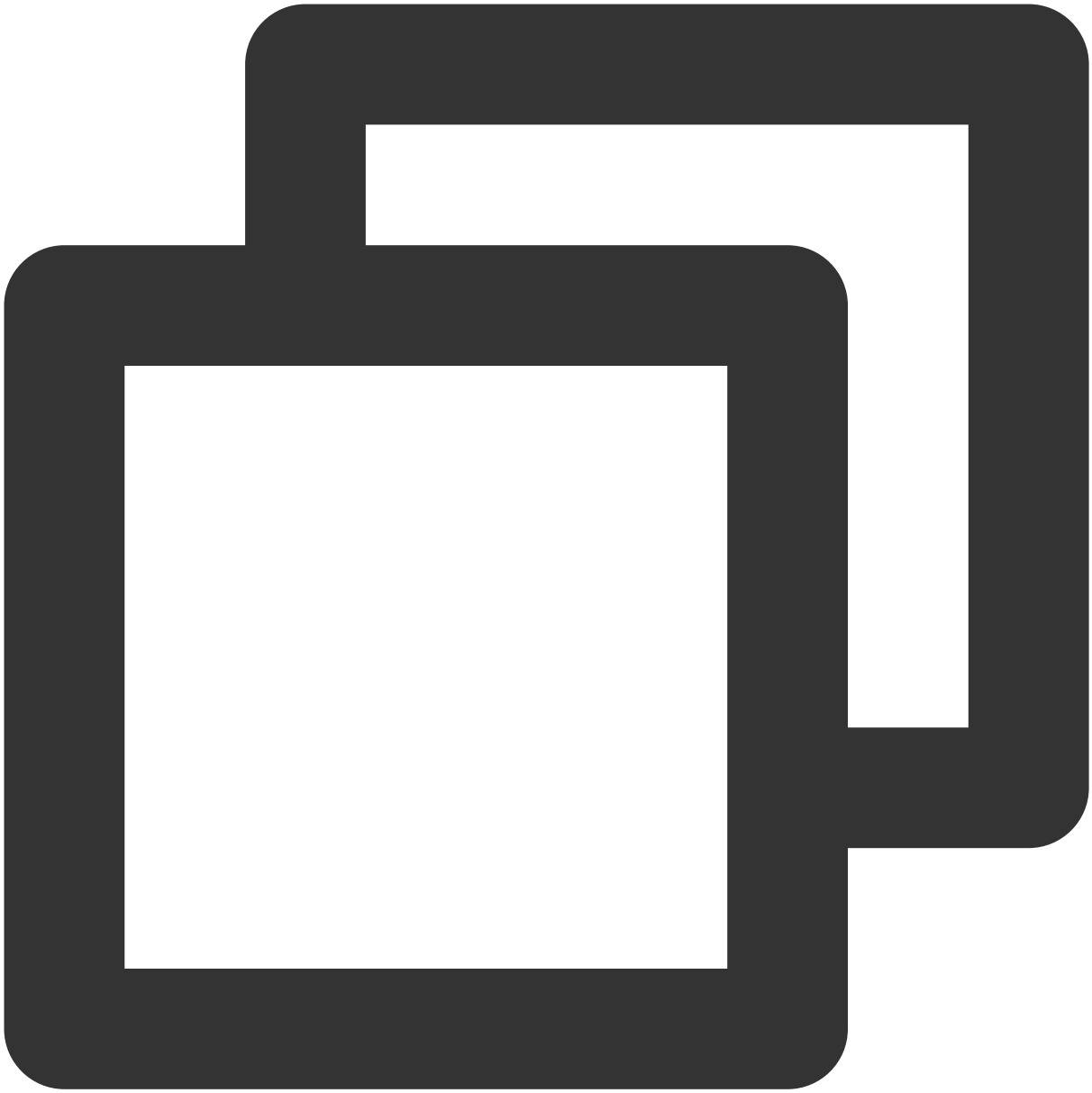
示例环境

实例类型：本文可选实例为 GN7 与 GN8，结合 Technical 提供的 GPU 对比，Turing 架构的 T4 性能优于 Pascal 架构的 P40。本文最终选用 GN7.5XLARGE80。
所在地域：由于可能需上传一些尺寸较大的数据集，需优先选择延迟最低的地域。本文使用 在线 Ping 工具测试，所在位置到提供 GN7 的重庆区域延迟最小，因此选择重庆区域。
系统盘：100GB 高性能云硬盘。
操作系统：Ubuntu 18.04
带宽：5M
本地操作系统：MacOS

操作步骤

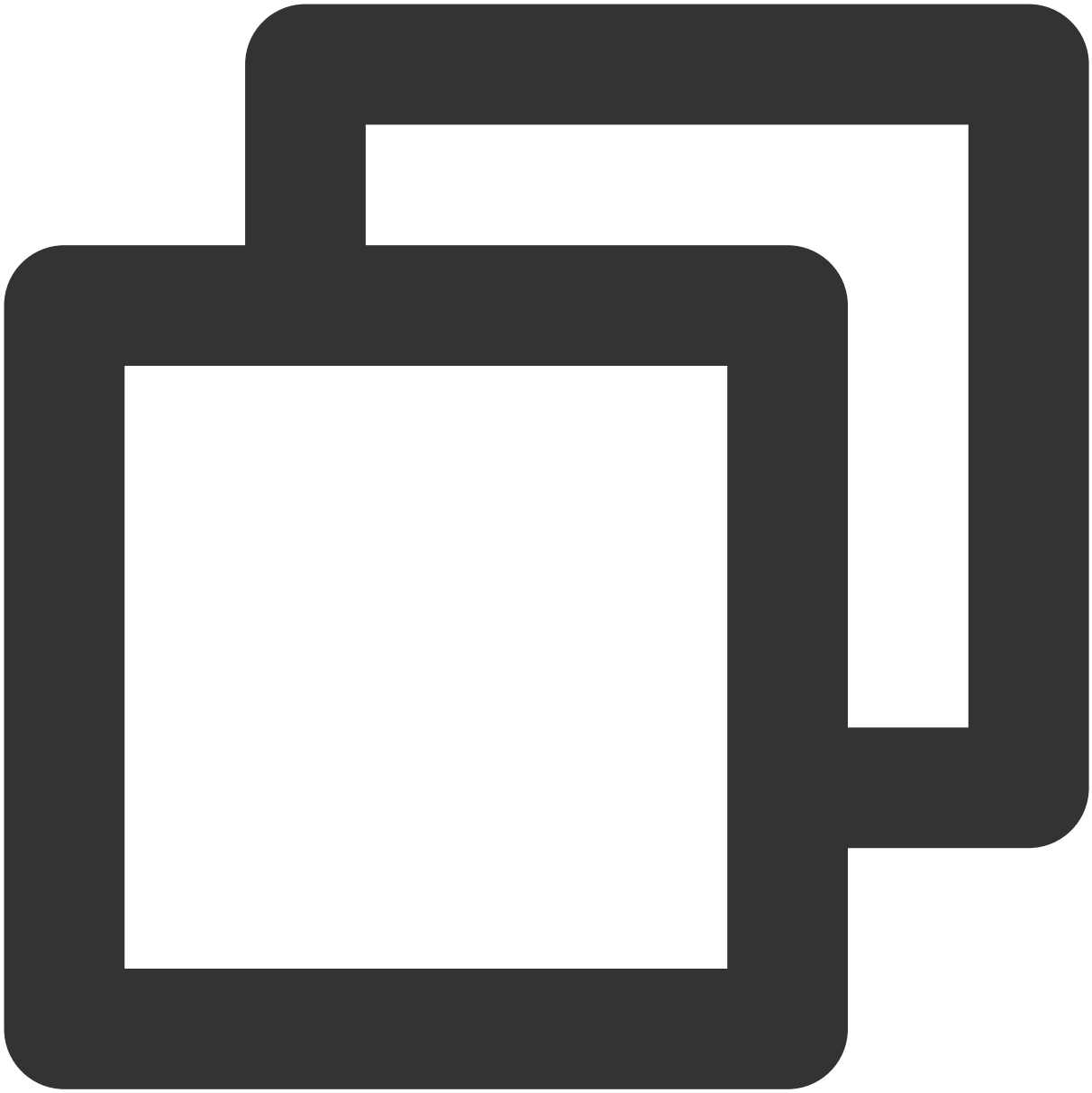
设置实例免密登录（可选）

1. （可选）您可在本机 `~/.ssh/config` 中，配置服务器的别名。本文创建别名为 `tcg` 。
2. 通过 `ssh-copy-id` 命令，将本机 SSH 公钥复制至 GPU 云服务器。
3. 在 GPU 云服务器中执行以下命令，关闭密码登录以增强安全性。



```
echo 'PasswordAuthentication no' | sudo tee -a /etc/ssh/ssh\\_config
```

4. 执行以下命令，重启 SSH 服务。



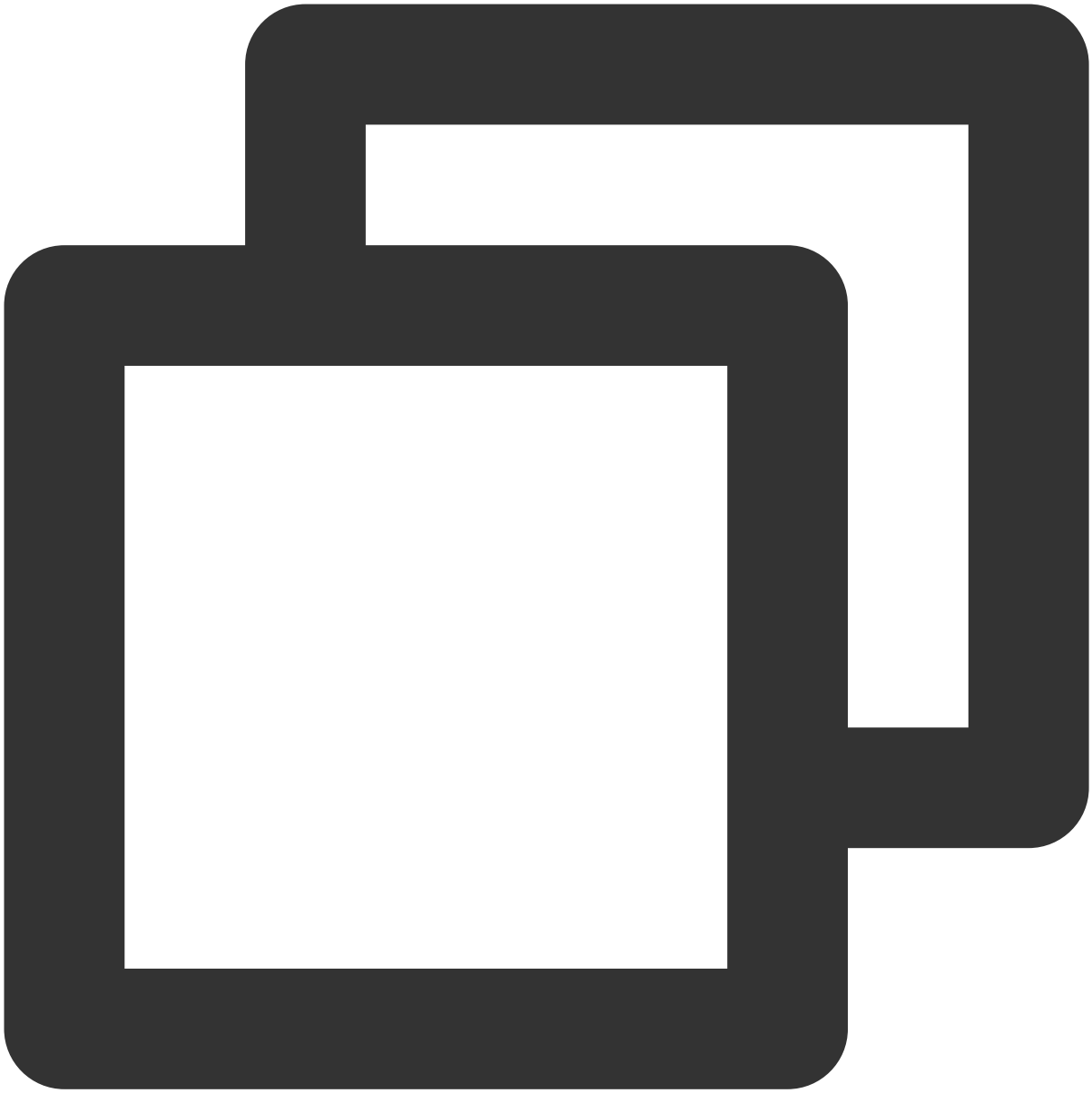
```
sudo systemctl restart sshd
```

PyTorch-GPU 开发环境配置

若使用 GPU 版本的 PyTorch 进行开发，则需要进行一些环境配置。步骤如下：

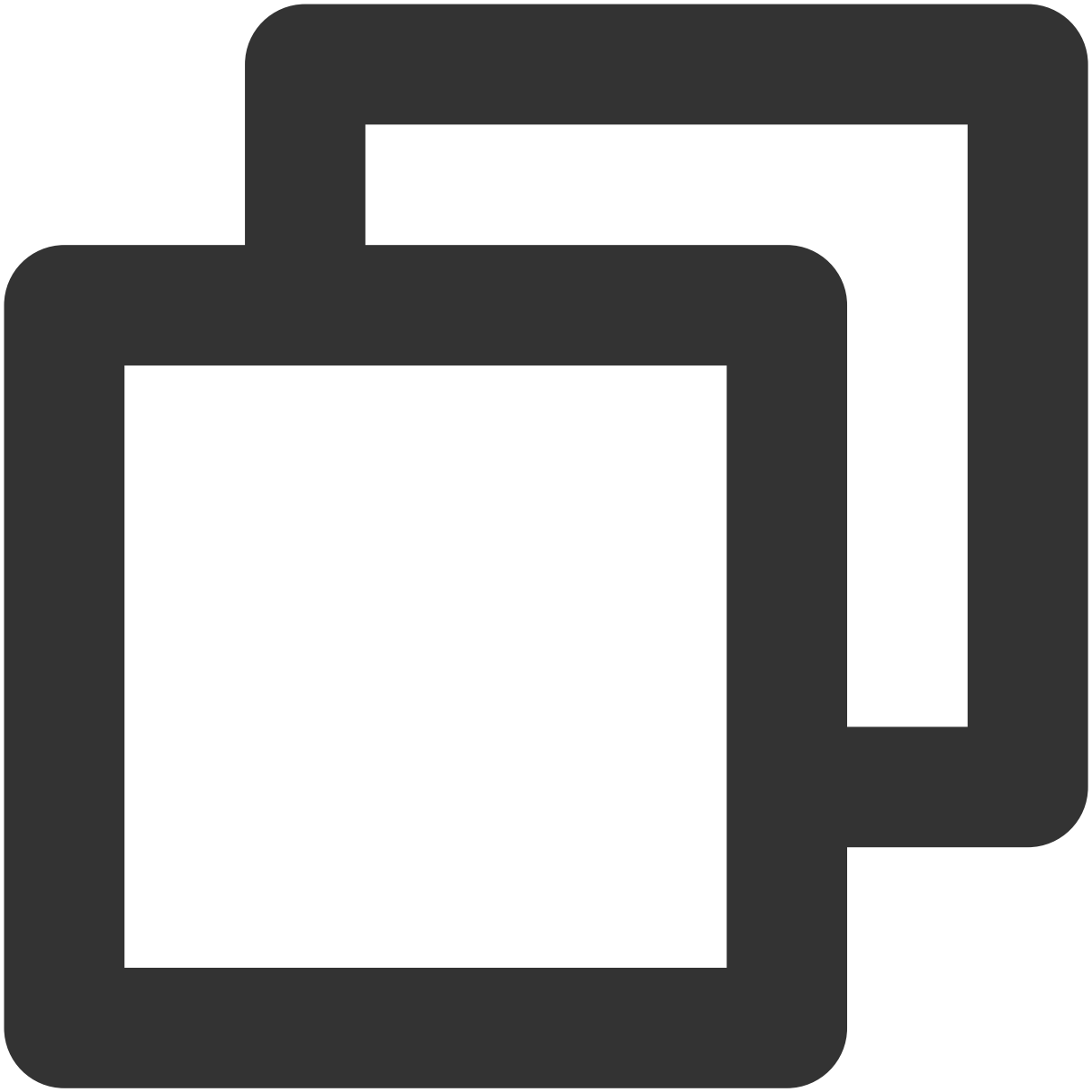
1. 安装 Nvidia 显卡驱动。

执行以下命令，安装 Nvidia 显卡驱动。



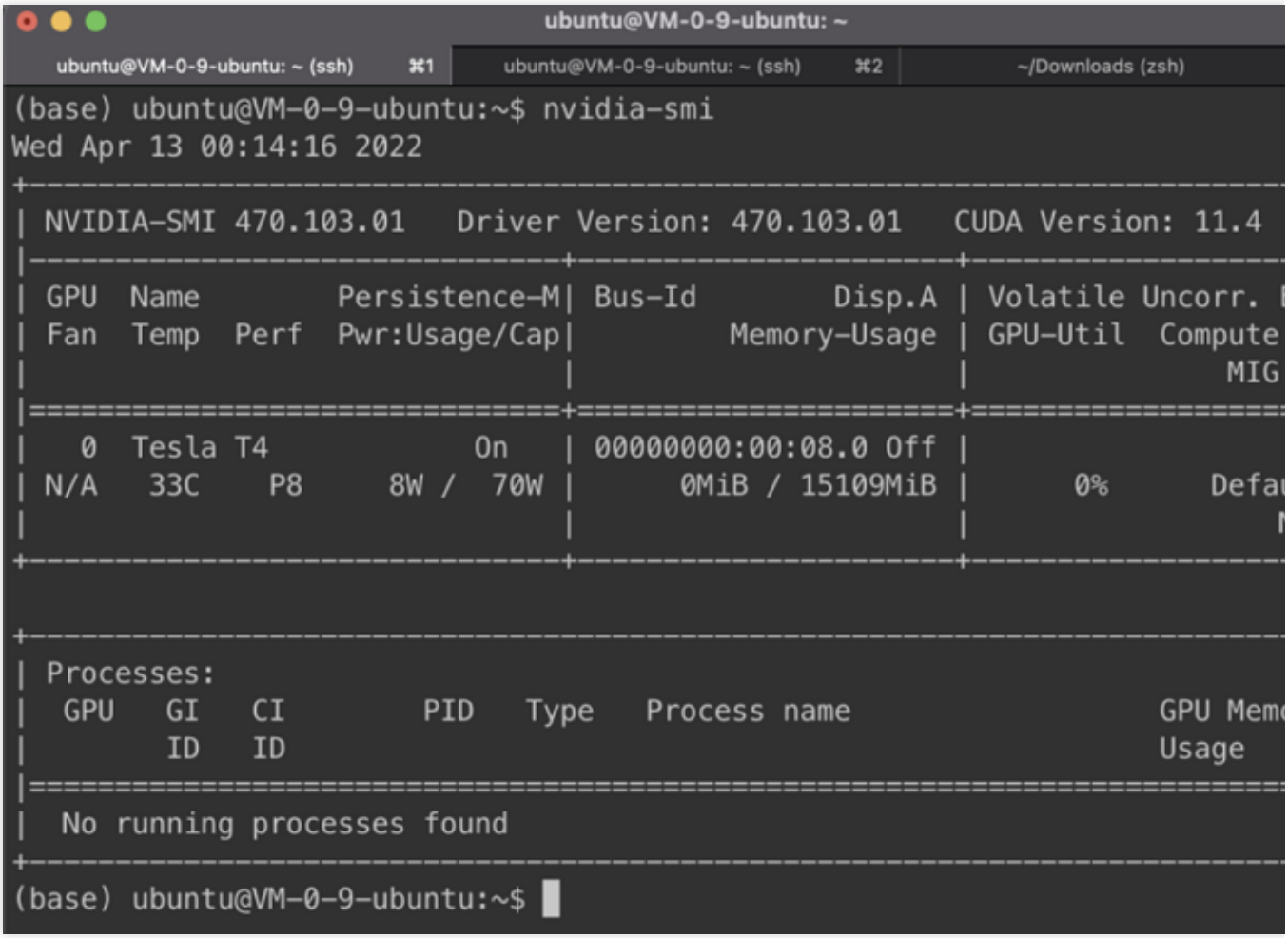
```
sudo apt install nvidia-driver-418
```

安装完成后执行如下命令，查看是否安装成功。



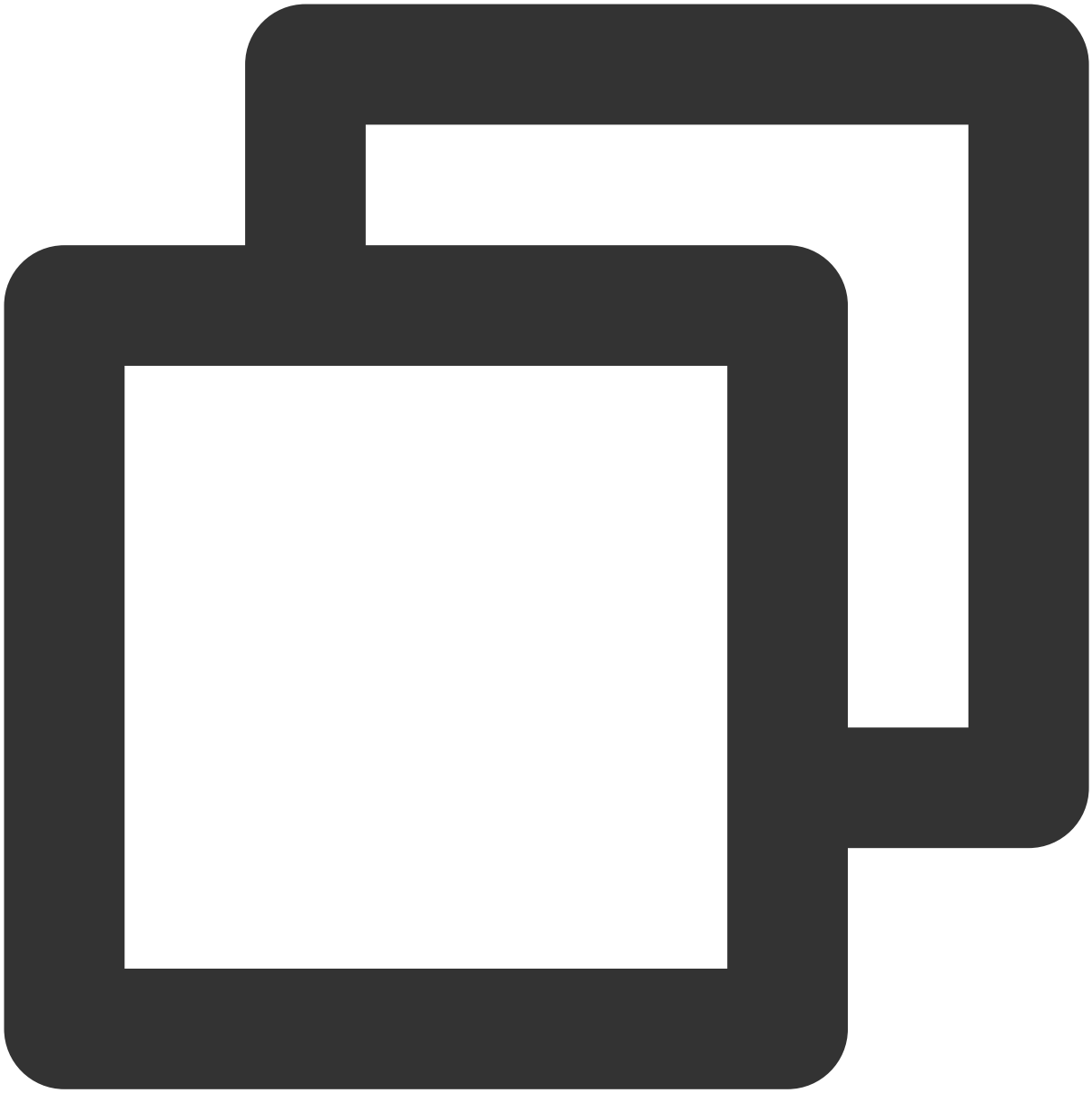
```
nvidia-smi
```

返回结果如下图所示，表示已安装成功。

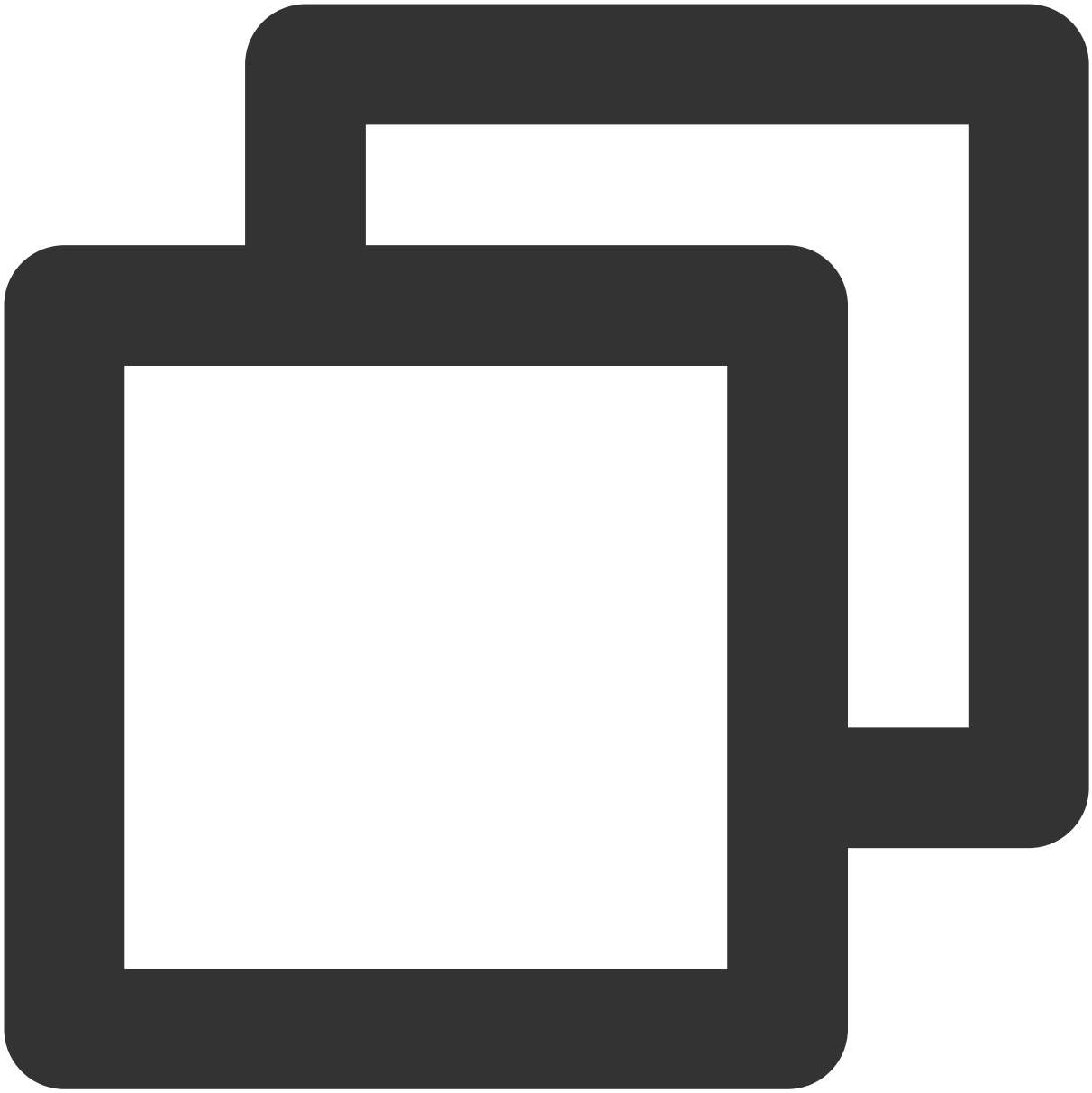


2. 配置 conda 环境。

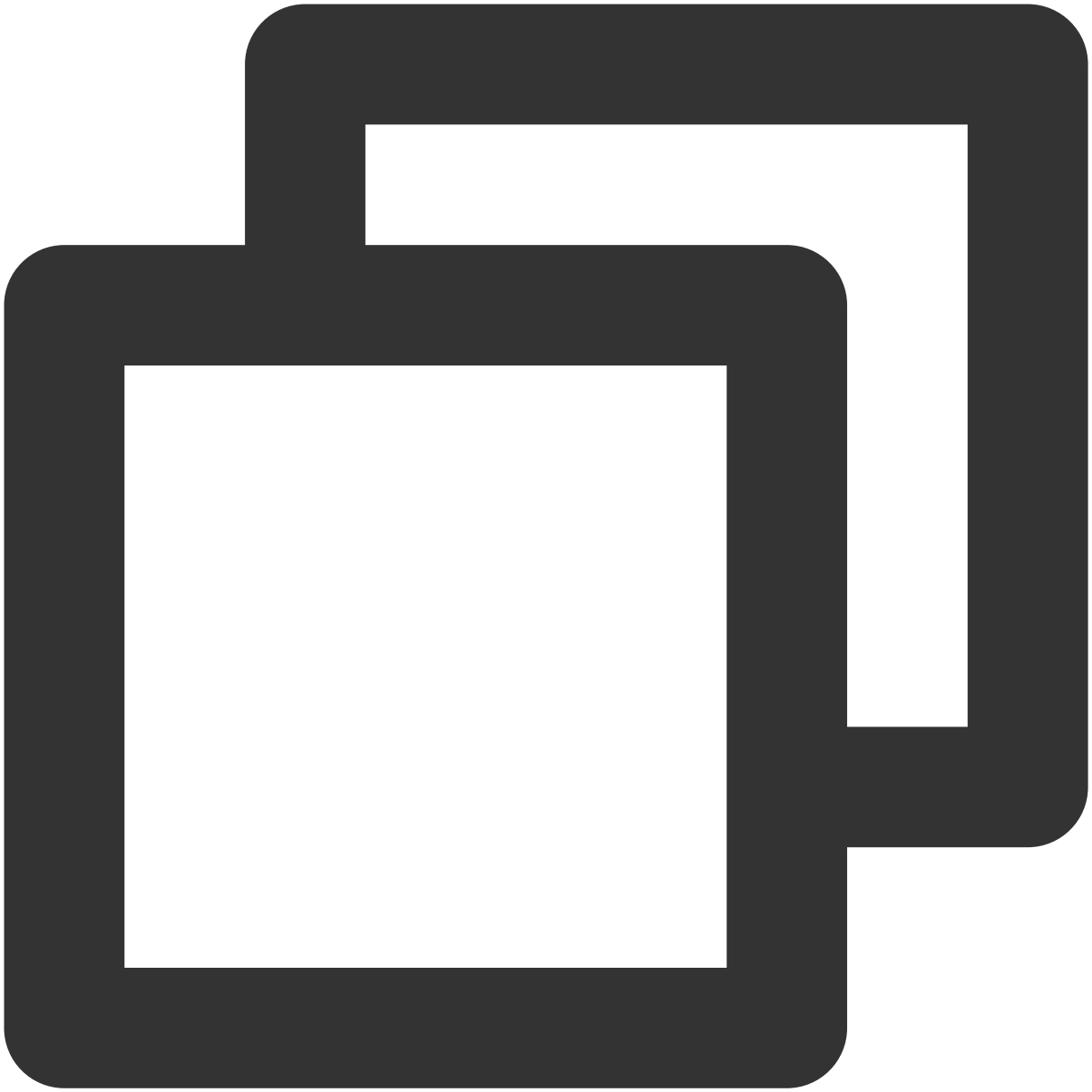
依次执行以下命令，配置 conda 环境。



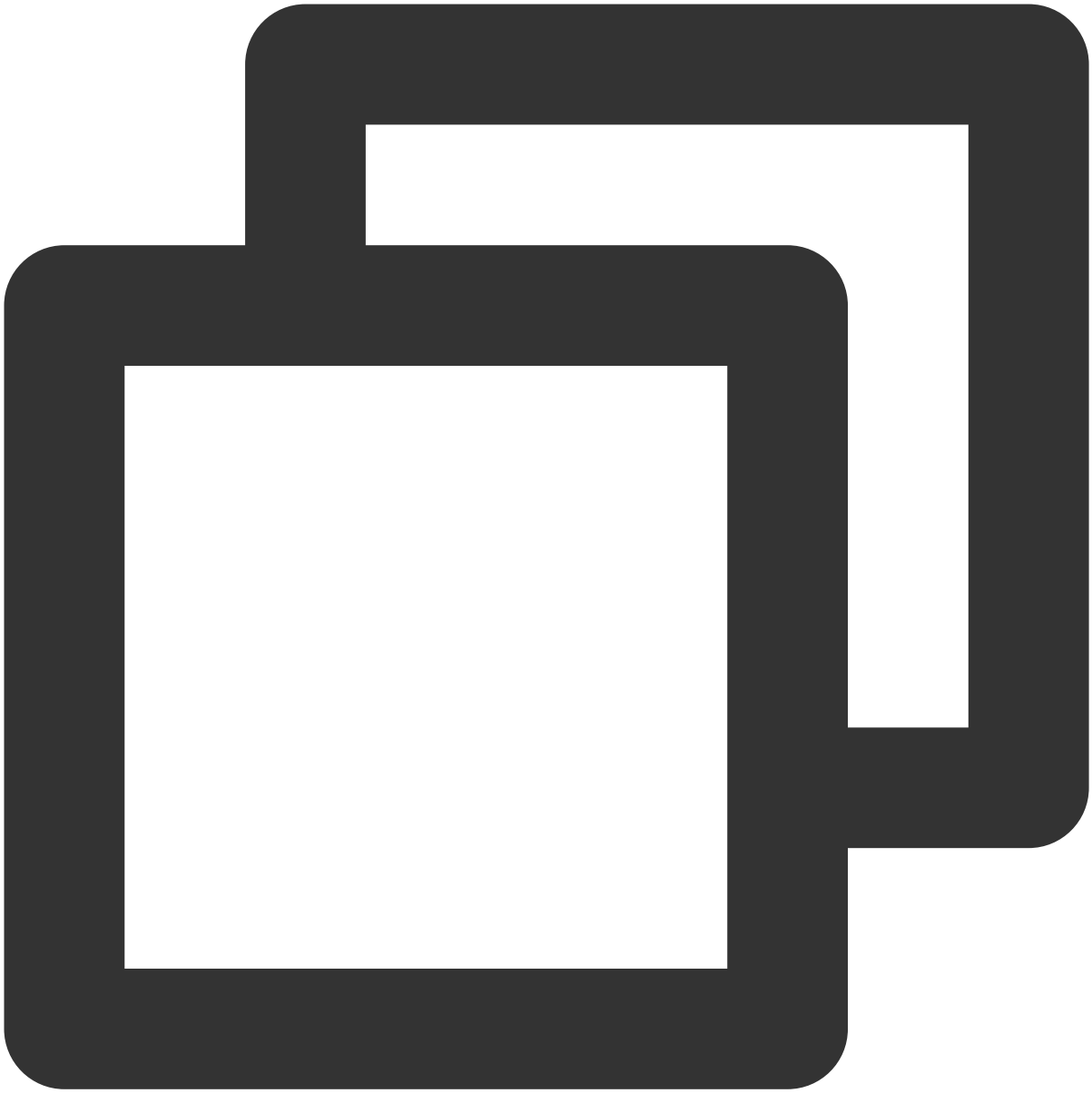
```
wget https://repo.anaconda.com/miniconda/Miniconda3-py39\\_4.11.0-Linux-x86\\_64.sh
```



```
chmod +x Miniconda3-py39\\_4.11.0-Linux-x86\\_64.sh
```

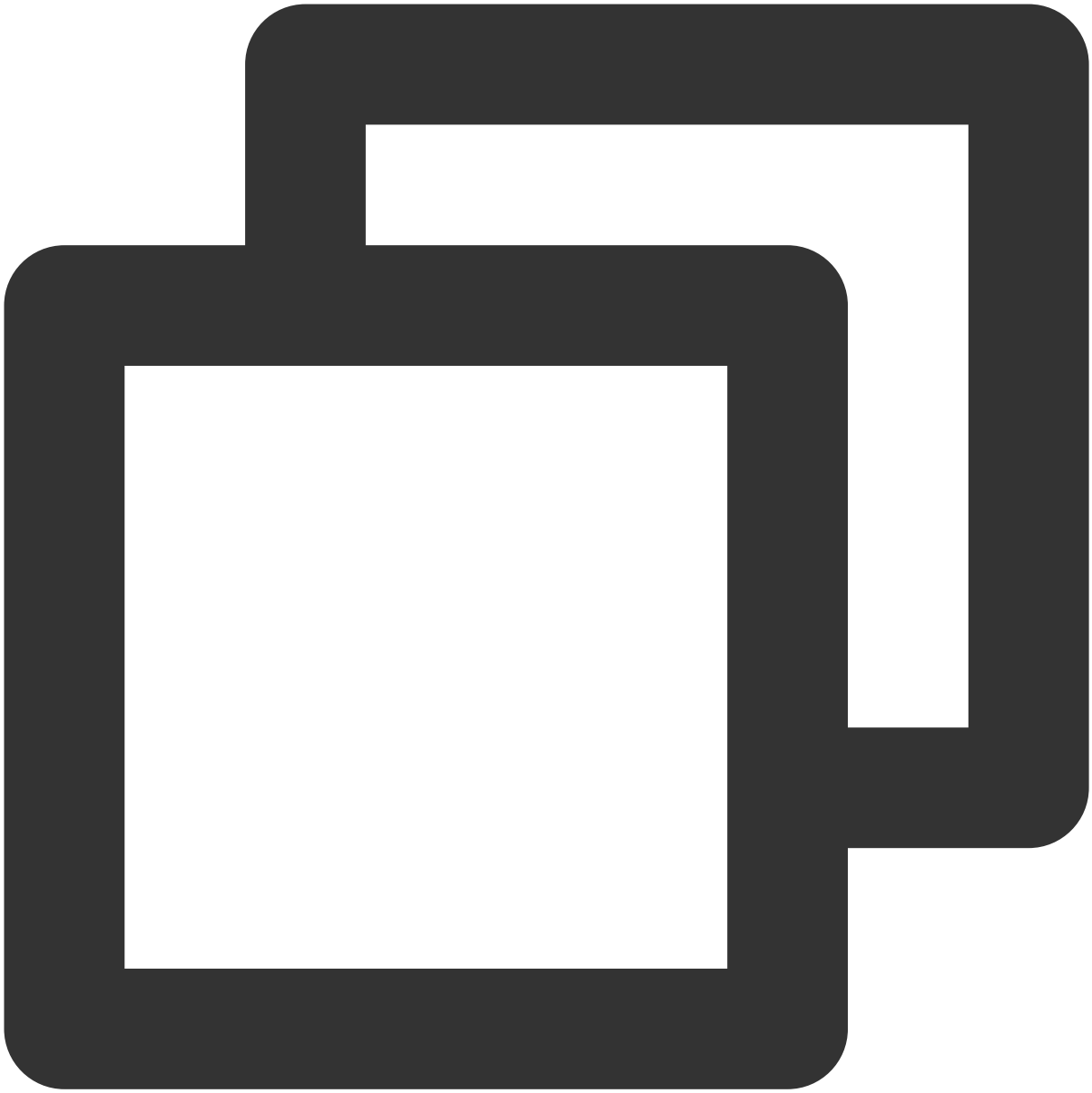


```
./Miniconda3-py39\\_4.11.0-Linux-x86\\_64.sh
```



```
rm Miniconda3-py39\\_4.11.0-Linux-x86\\_64.sh
```

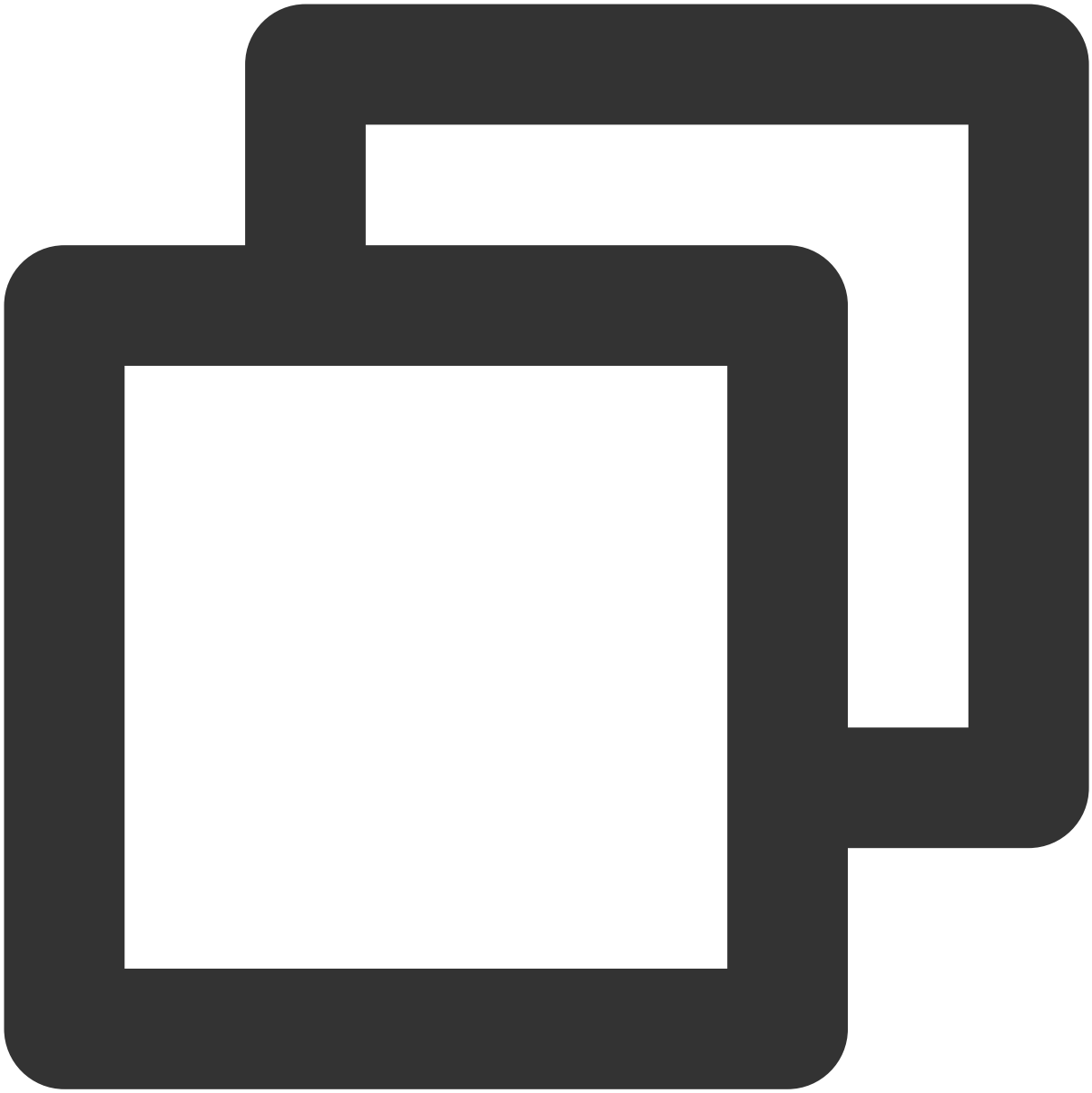
3. 编辑 `~/.condarc` 文件，加入以下软件源信息，将 conda 的软件源替换为清华源。



```
channels:

- defaults

show\\_channel\\_urls: true

default\\_channels:

- https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main

- https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/r
```

```
- https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/msys2

custom\\_channels:

  conda-forge: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud

  msys2: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud

  bioconda: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud

  menpo: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud

  pytorch: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud

  pytorch-lts: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud

  simpleitk: https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud
```

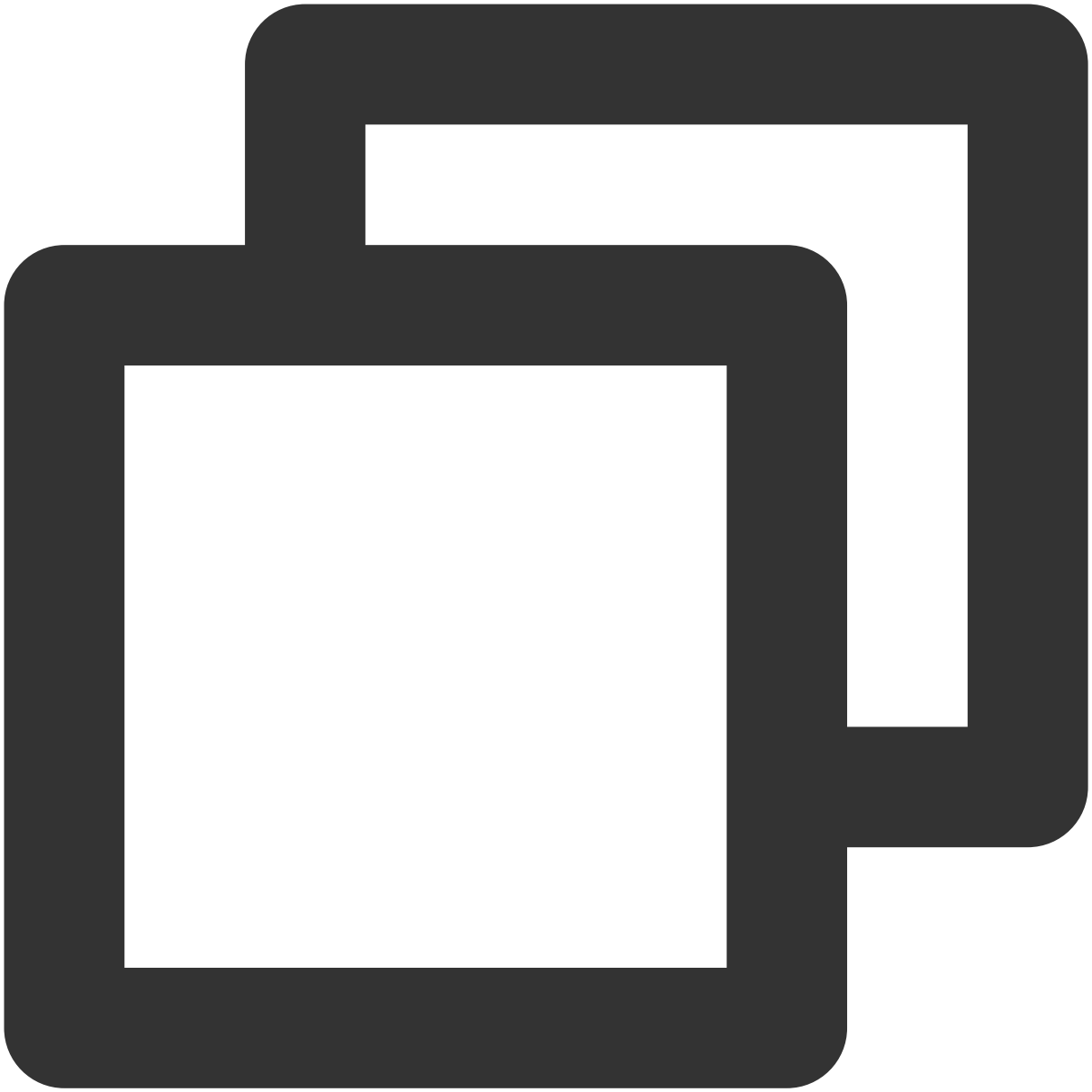
4. 执行以下命令，设置 pip 源为腾讯云镜像源。



```
pip config set global.index-url https://mirrors.cloud.tencent.com/pypi/simple
```

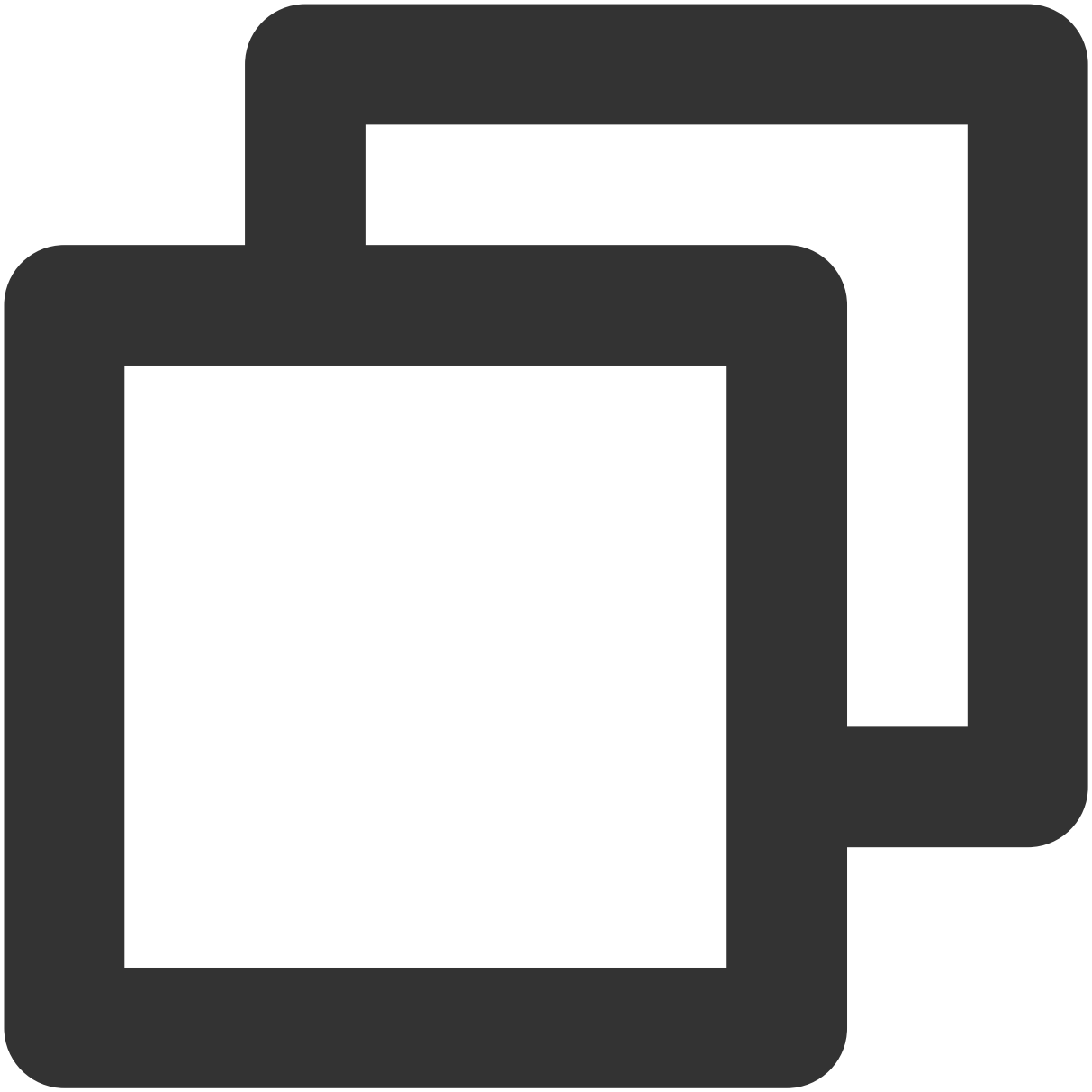
5. 安装 PyTorch。

执行以下命令，安装 PyTorch。

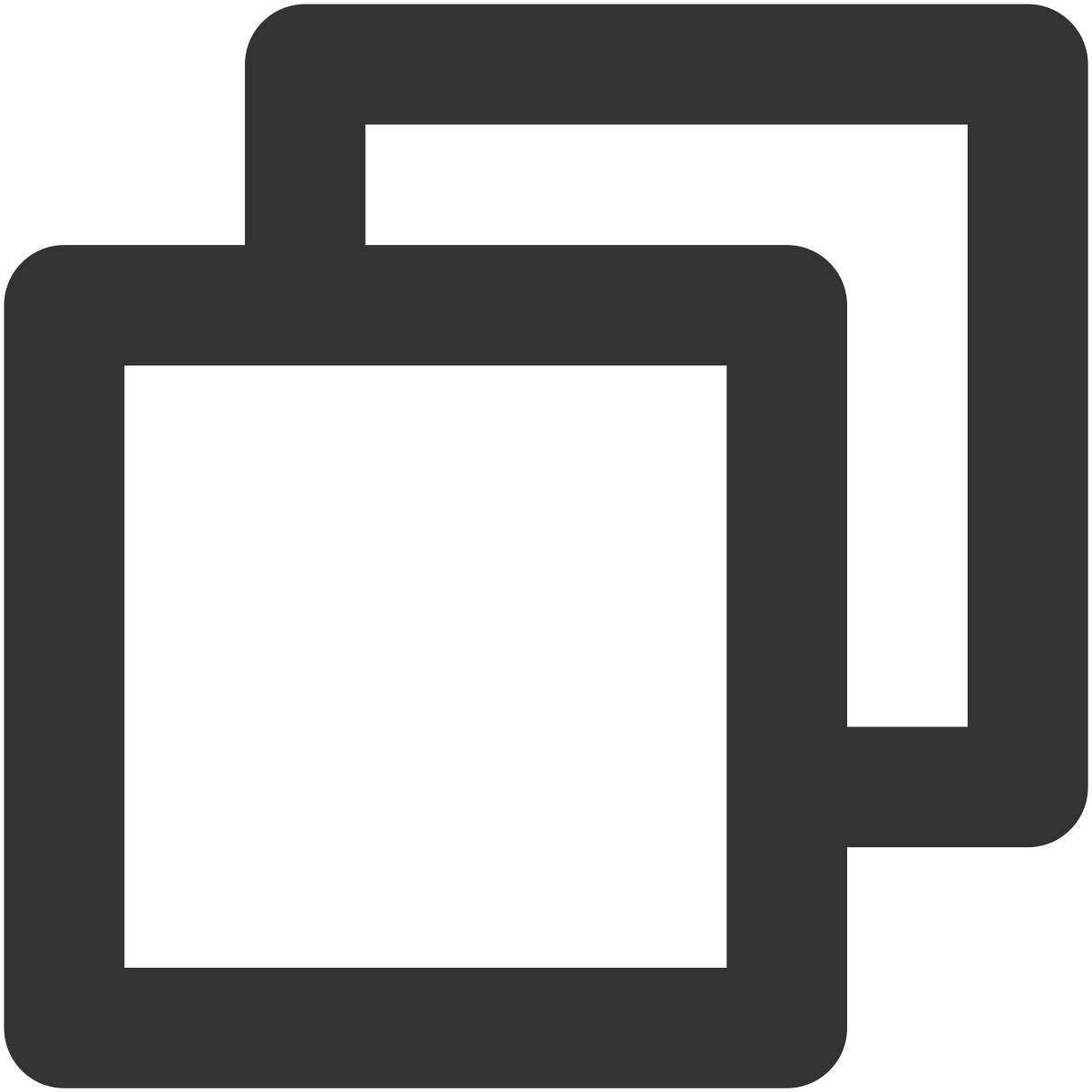


```
conda install pytorch torchvision cudatoolkit=11.4 -c pytorch --yes
```

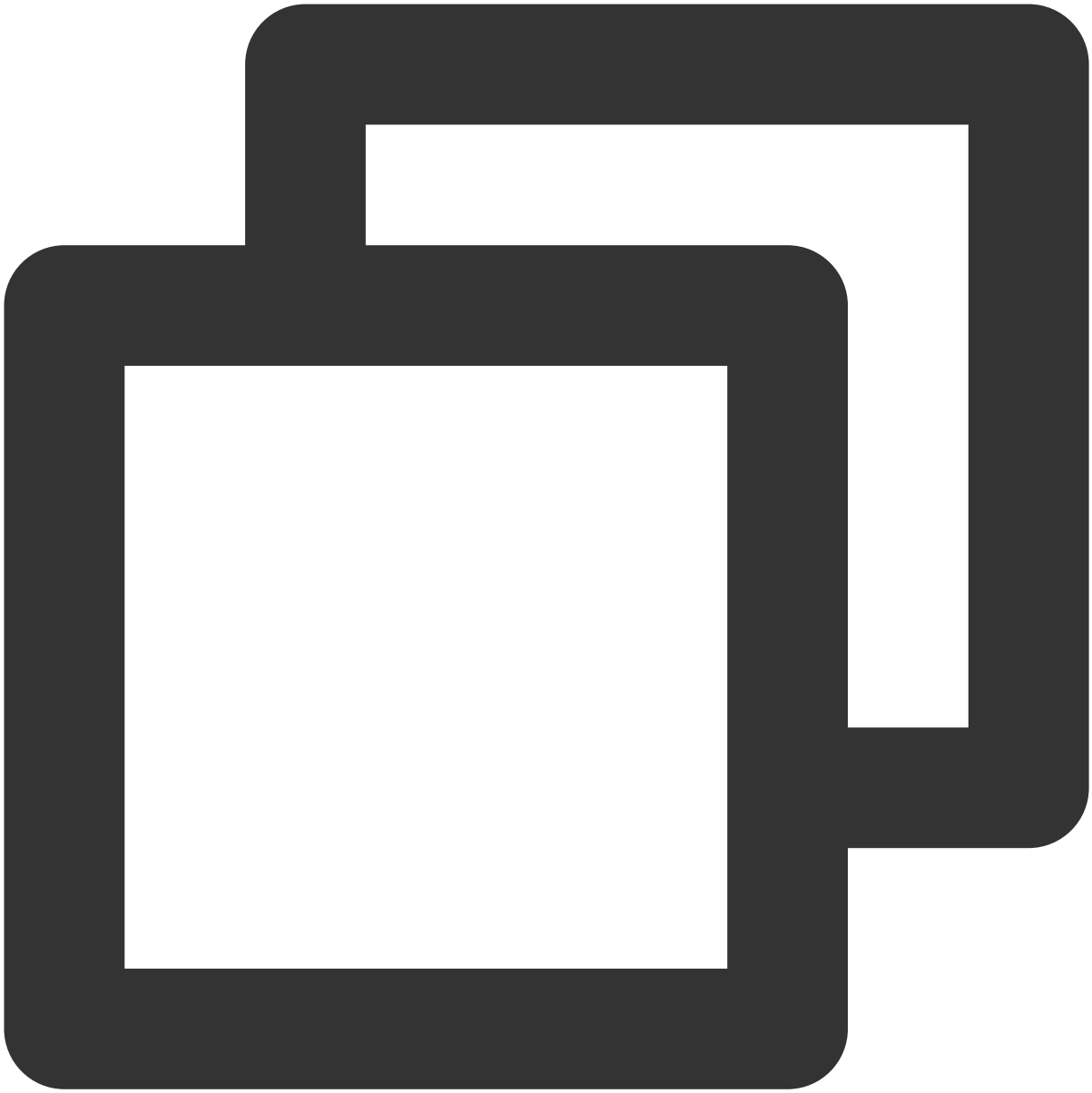
依次执行以下命令，查看 PyTorch 是否安装成功。



```
python
```

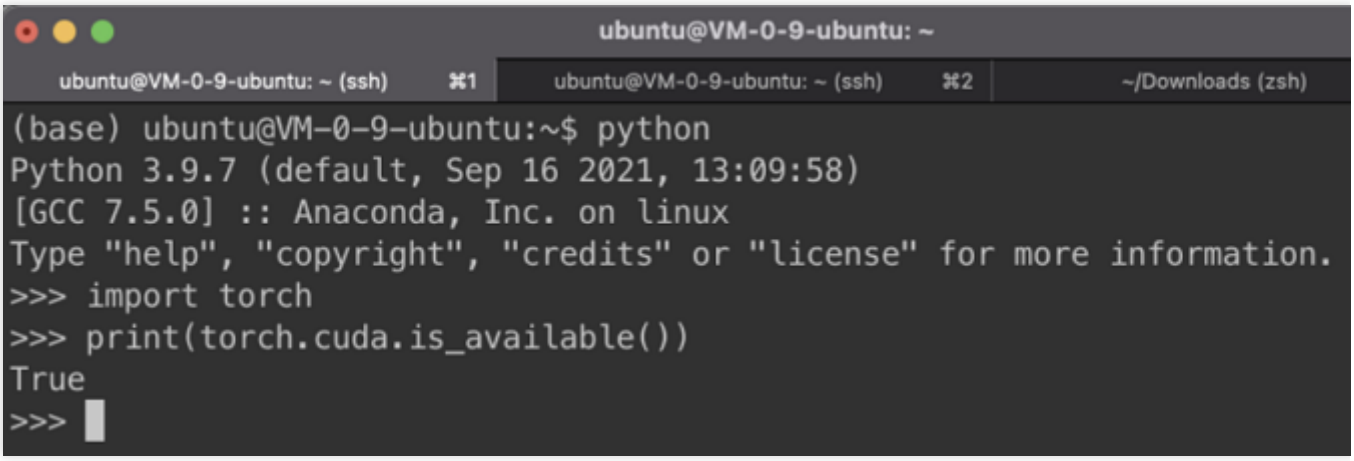


```
import torch
```



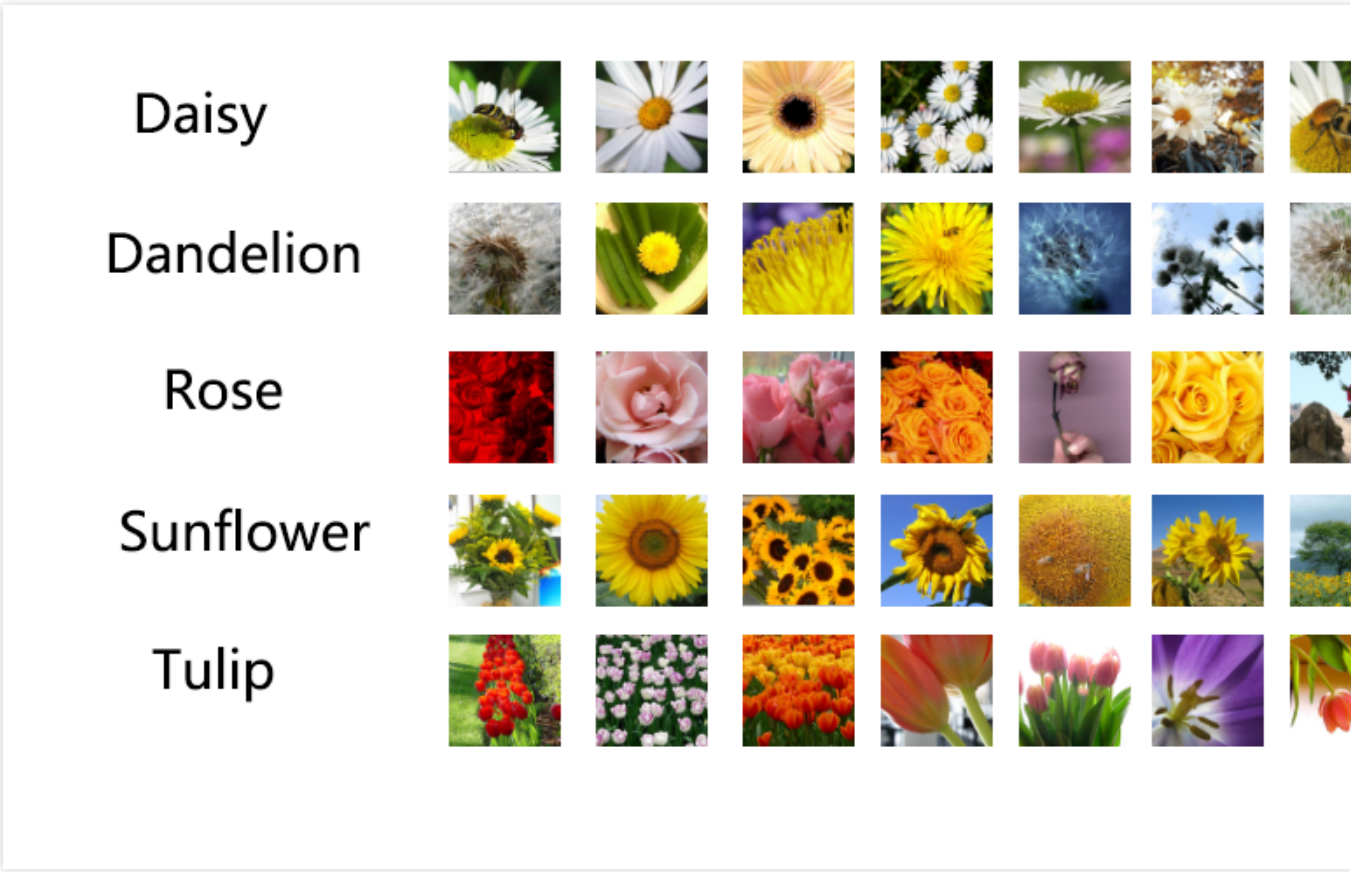
```
print(torch.cuda.is_avaliable())
```

返回结果如下图所示，表示 PyTorch 已安装成功。

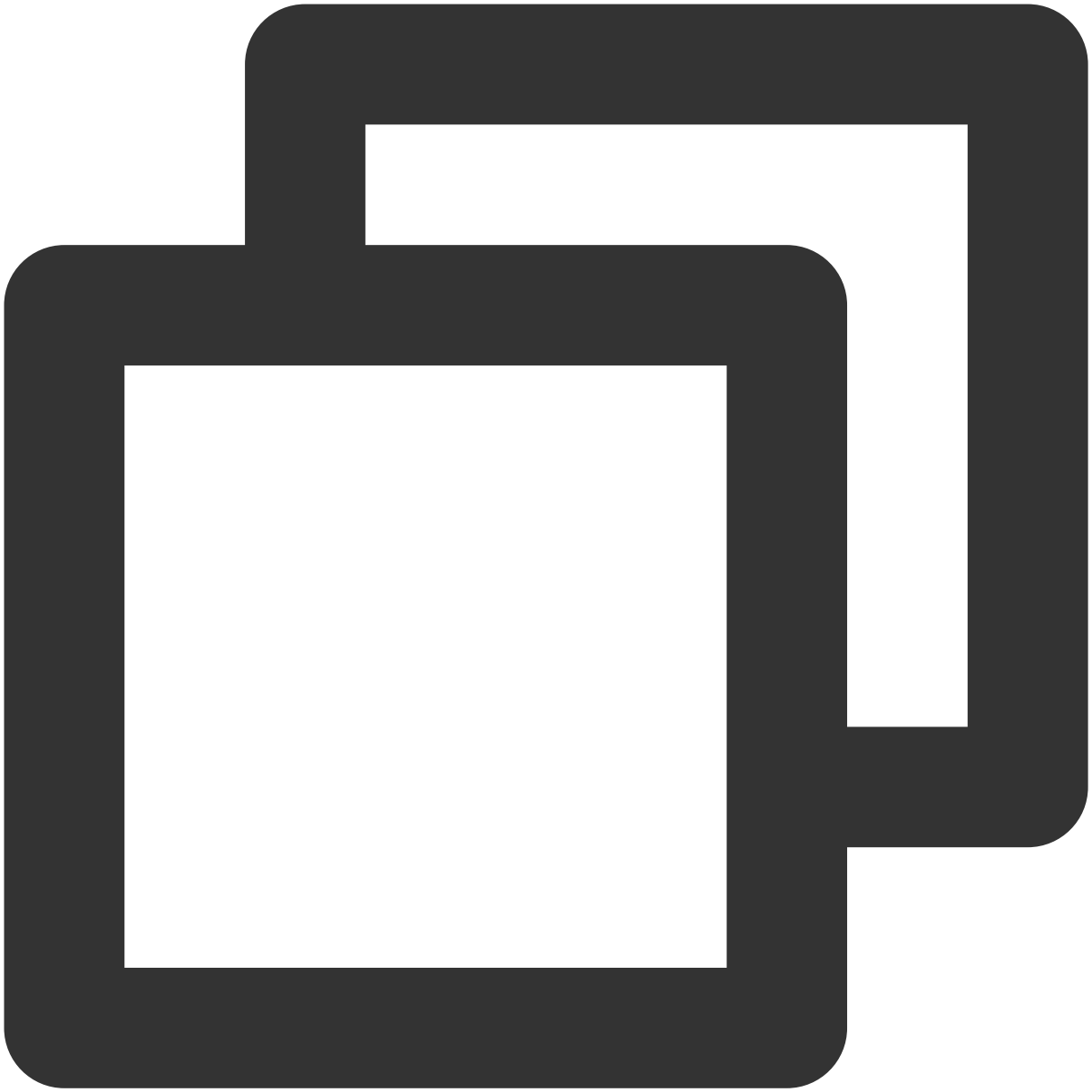


准备实验数据

本次训练的测试任务是图像分类任务，使用了腾讯云在线文档中用到的花朵图像分类数据集。该数据集包含5类花朵，数据大小为218M。数据集抽样展示如下：（各类别下花朵照片示例）



原始数据集中的各个分类数据分别存放在类名对应的文件夹下。首先需将其转化为 imagenet 对应的标准格式。按4：1划分训练和验证集，使用以下代码进行格式转换：

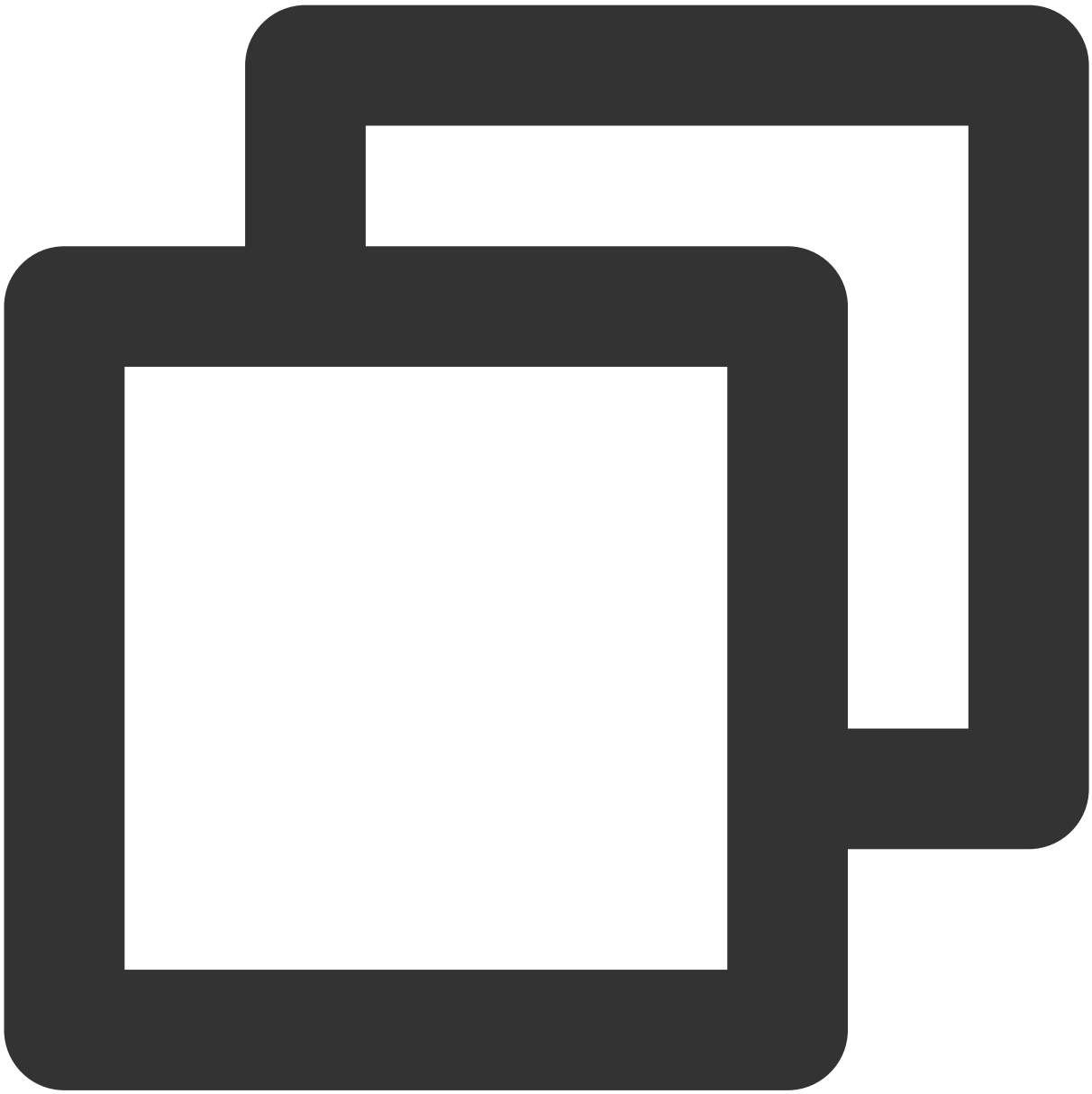


```
# split data into train set and validation set, train:val=scale

import shutil

import os

import math

scale = 4

data\\_path = '../raw'
```

```
data\\_dst = '../train\\_val'

#create imagenet directory structure

os.mkdir(data\\_dst)

os.mkdir(os.path.join(data\\_dst, 'train'))

os.mkdir(os.path.join(data\\_dst, 'validation'))

for item in os.listdir(data\\_path):

    item\\_path = os.path.join(data\\_path, item)

 if os.path.isdir(item\\_path):

        train\\_dst = os.path.join(data\\_dst, 'train', item)

        val\\_dst = os.path.join(data\\_dst, 'validation', item)

        os.mkdir(train\\_dst)

        os.mkdir(val\\_dst)

        files = os.listdir(item\\_path)

 print(f'Class {item}:\\n\\t Total sample count is {len(files)}')

        split\\_idx = math.floor(len(files) \\* scale / ( 1 + scale ))

 print(f'\\t Train sample count is {split\\_idx}')

 print(f'\\t Val sample count is {len(files) - split\\_idx}\\n')

 for idx, file in enumerate(files):

            file\\_path = os.path.join(item\\_path, file)

 if idx <= split\\_idx:

                shutil.copy(file\\_path, train\\_dst)

 else:

                shutil.copy(file\\_path, val\\_dst)
```

```
print(f'Split Complete. File path: {data\\_dst}')
```

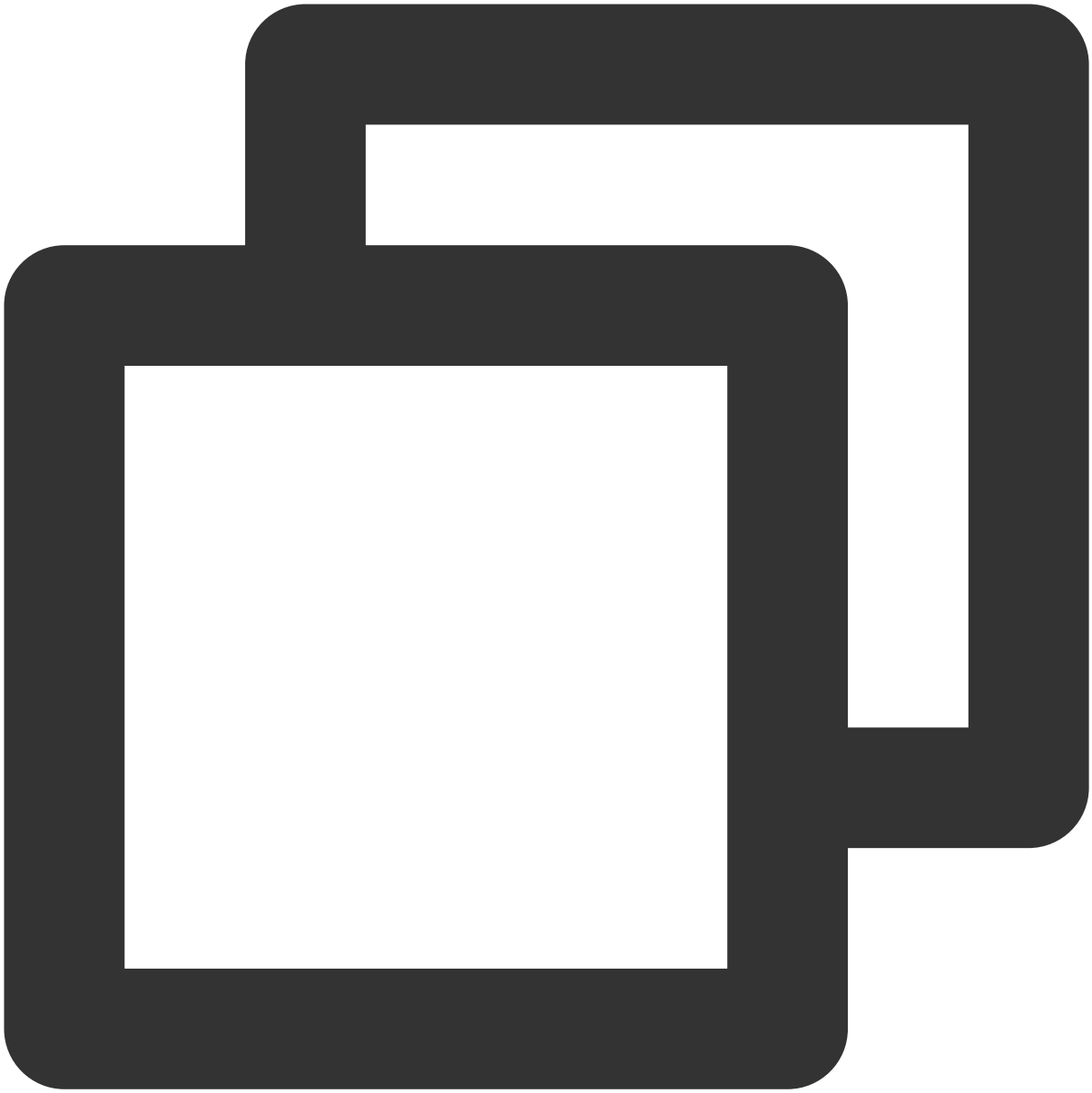
数据集概览如下：



```
Class roses:

     Total sample count is 641

     Train sample count is 512

     Validation sample count is 129
```

```
Class sunflowers:

     Total sample count is 699

     Train sample count is 559

     Validation sample count is 140

Class tulips:

     Total sample count is 799

     Train sample count is 639

     Validation sample count is 160

Class daisy:

     Total sample count is 633

     Train sample count is 506

     Validation sample count is 127

Class dandelion:

     Total sample count is 898

     Train sample count is 718

     Validation sample count is 180
```

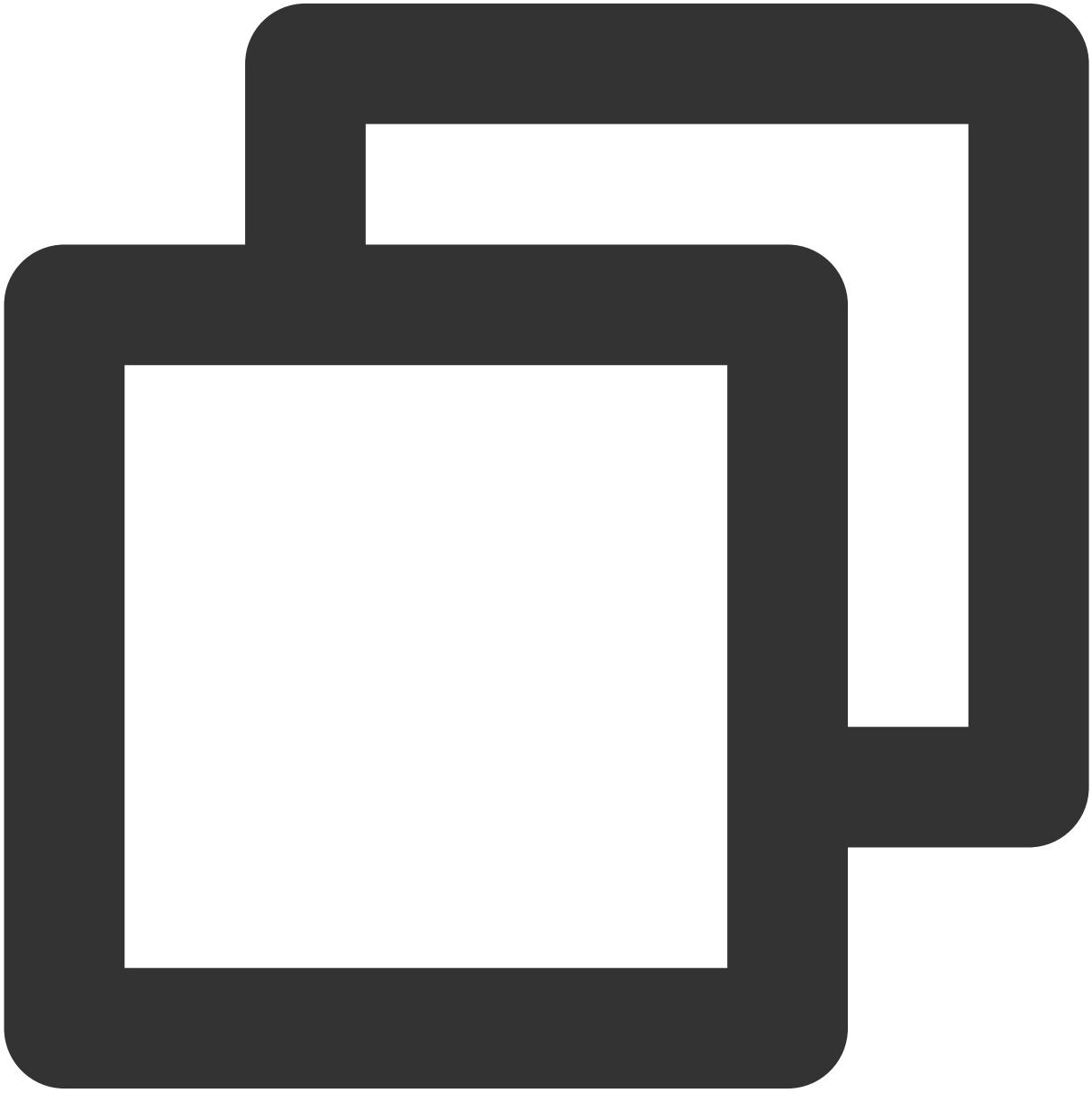
为了加速训练过程，我们进一步将数据集转换为 Nvidia-DALI 这种 GPU 友好的格式。DALI 全称 Data Loading Library，该库可以通过使用 GPU 替代 CPU 来加速数据预处理过程。在已有 imagenet 格式数据的前提下，使用 DALI 只需运行以下命令即可：



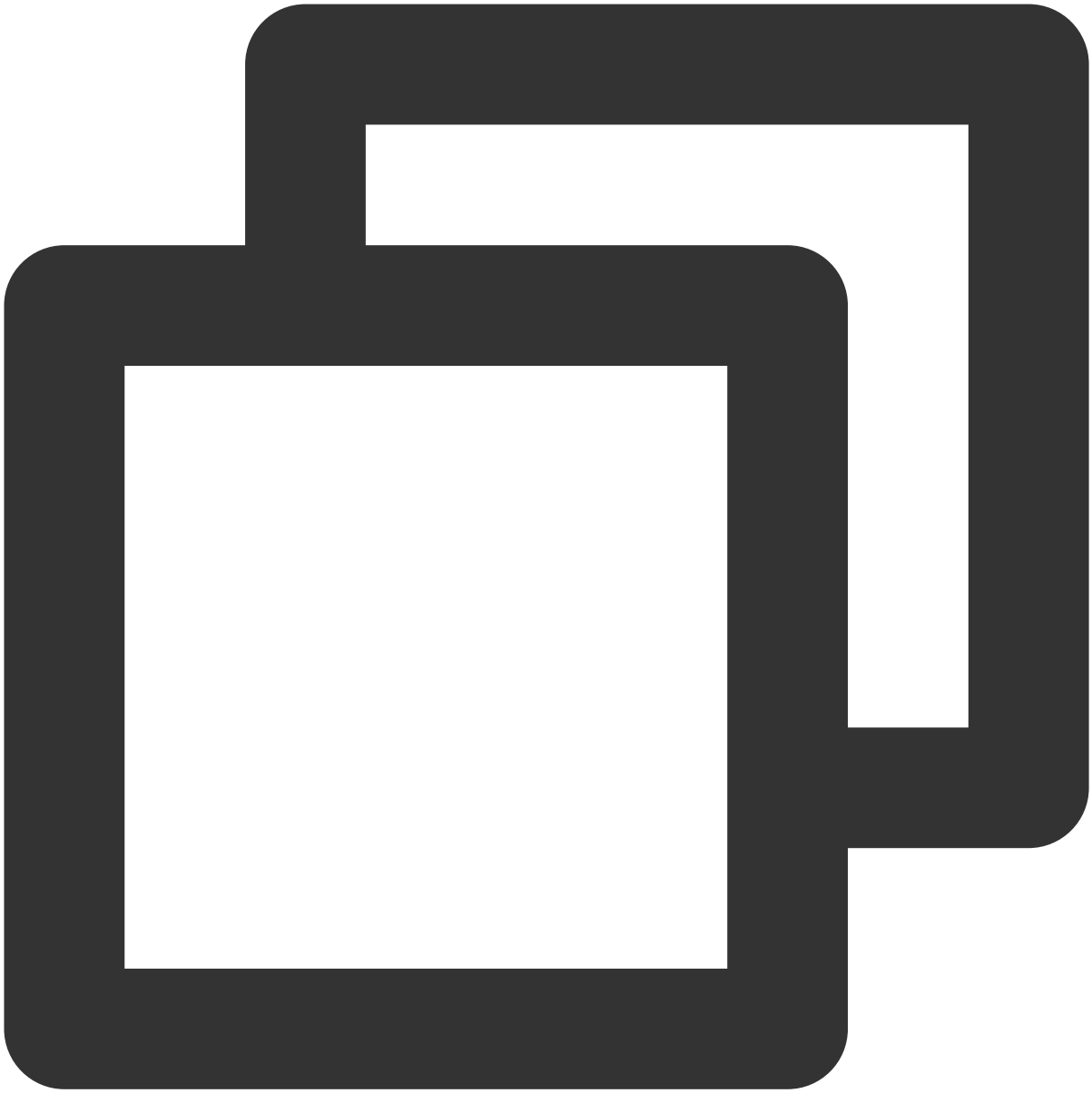
```
git clone https://github.com/ver217/imagenet-tools.git

cd imagenet-tools && python3 make\\_tfrecords.py \\

  --raw\\_data\\_dir="../train\\_val" \\

  --local\\_scratch\\_dir="../train\\_val\\_tfrecord" && \\

python3 make\\_idx.py --tfrecord\\_root="../train\\_val\\_tfrecord"
```

模型训练结果

为了便于后续训练分布式大规模模型，本文在分布式训练框架 Colossal-AI 的基础上进行模型训练和开发。Colossal-AI 提供了一组便捷的接口，通过这组接口能方便地实现数据并行、模型并行、流水线并行或者混合并行。
参考 Colossal-AI 提供的 demo，本文使用 pytorch-image-models 库所集成的 ViT 实现，选择最小的 `vit\\_tiny\\_patch16\\_224` 模型，该模型的分辨率为224*224, 每个样本被划分为16个 `patch` 。
1. 根据版本选择页面通过以下命令，安装 Colossal-AI 和 pytorch-image-models：

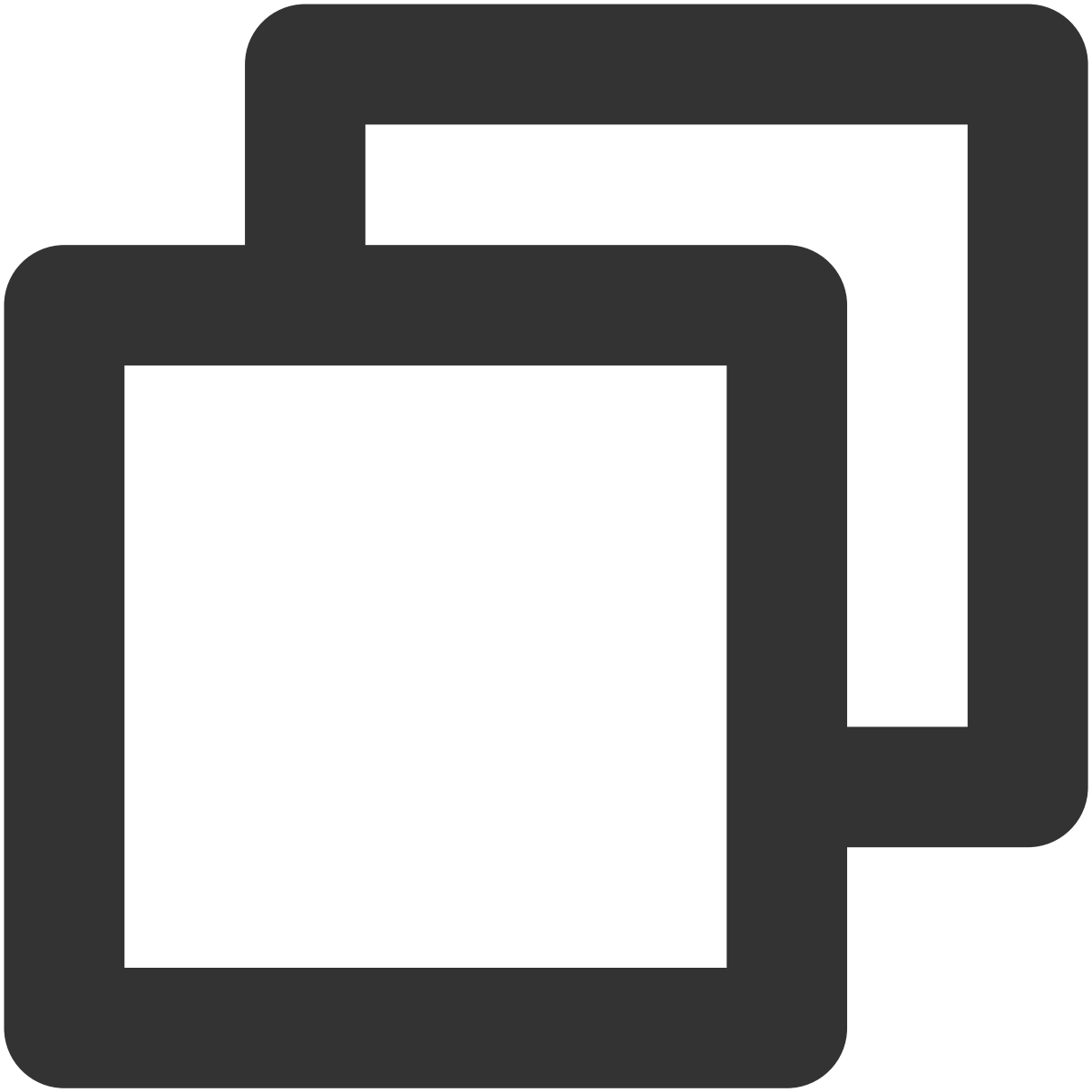


```
pip install colossalai==0.1.5+torch1.11cu11.3 -f https://release.colossalai.org
```



```
pip install timm
```

2. 参考 Colossal-AI 提供的 demo，编写模型训练代码如下：



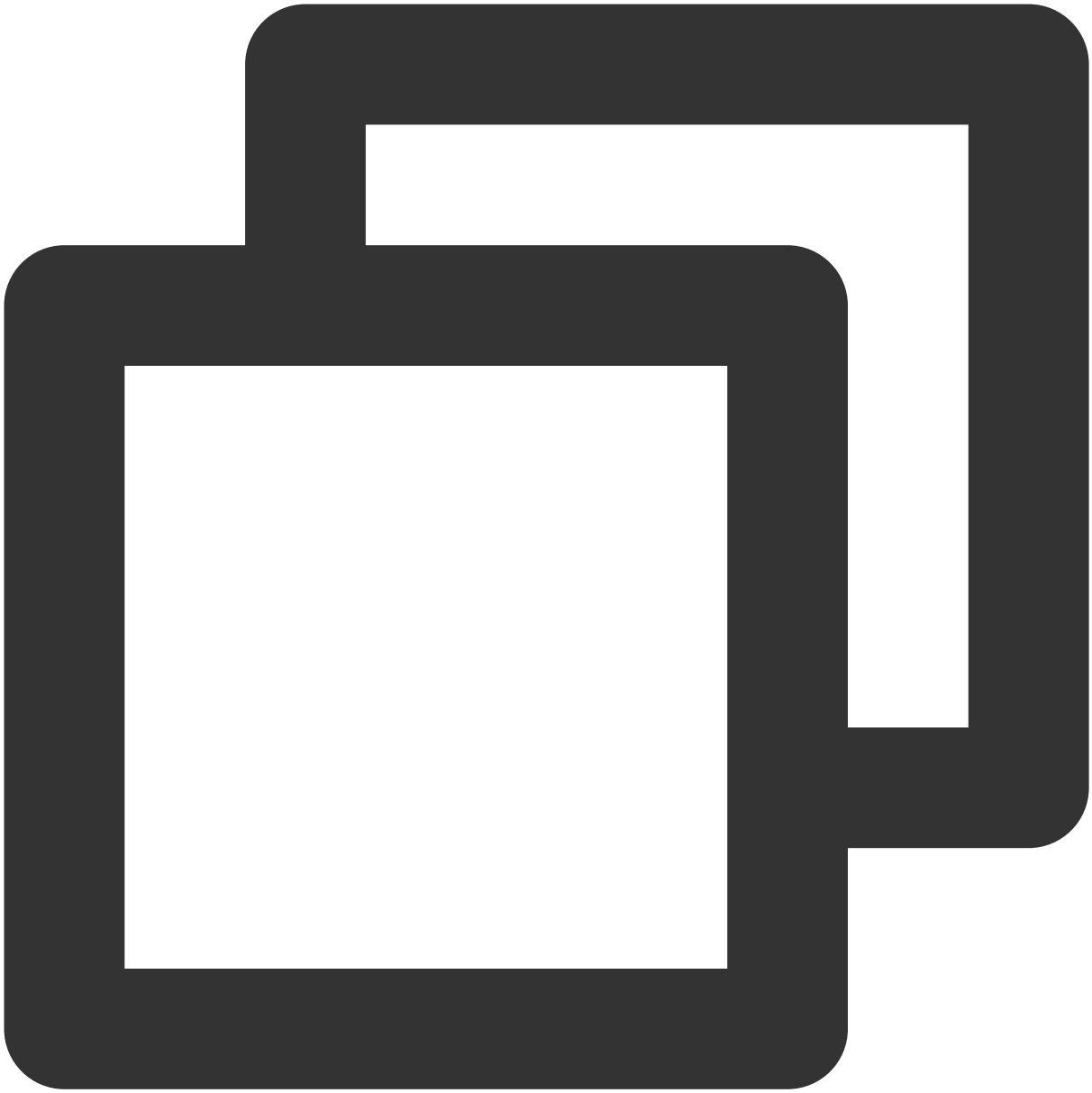
```
from pathlib import Path

from colossalai.logging import get\\_dist\\_logger

import colossalai

import torch

import os

from colossalai.core import global\\_context as gpc
```

```
from colossalai.utils import get\\_dataloader, MultiTimer

from colossalai.trainer import Trainer, hooks

from colossalai.nn.metric import Accuracy

from torchvision import transforms

from colossalai.nn.lr\\_scheduler import CosineAnnealingLR

from tqdm import tqdm

from titans.utils import barrier\\_context

from colossalai.nn.lr\\_scheduler import LinearWarmupLR

from timm.models import vit\\_tiny\\_patch16\\_224

from titans.dataloader.imagenet import build\\_dali\\_imagenet

from mixup import MixupAccuracy, MixupLoss

def main():

 parser = colossalai.get\\_default\\_parser()

 args = parser.parse\\_args()

 colossalai.launch\\_from\\_torch(config='./config.py')

 logger = get\\_dist\\_logger()

 # build model

 model = vit\\_tiny\\_patch16\\_224(num\\_classes=5, drop\\_rate=0.1)

 # build dataloader

 root = os.environ.get('DATA', '../train\\_val\\_tfrecord')

 train\\_dataloader, test\\_dataloader = build\\_dali\\_imagenet(

     root, rand\\_augment=True)

 # build criterion
```

```
criterion = MixupLoss(loss\\_fn\\_cls=torch.nn.CrossEntropyLoss)

# optimizer

optimizer = torch.optim.SGD(

    model.parameters(), lr=0.1, momentum=0.9, weight\\_decay=5e-4)

# lr\\_scheduler

lr\\_scheduler = CosineAnnealingLR(

   optimizer, total\\_steps=gpc.config.NUM\\_EPOCHS)

engine, train\\_dataloader, test\\_dataloader, \\_ = colossalai.initialize(

    model,

    optimizer,

    criterion,

    train\\_dataloader,

    test\\_dataloader,

)

# build a timer to measure time

timer = MultiTimer()

# create a trainer object

trainer = Trainer(engine=engine, timer=timer, logger=logger)

# define the hooks to attach to the trainer

hook\\_list = [

    hooks.LossHook(),

    hooks.LRSchedulerHook(lr\\_scheduler=lr\\_scheduler, by\\_epoch=True),

    hooks.AccuracyHook(accuracy\\_func=MixupAccuracy()),

    hooks.LogMetricByEpochHook(logger),
```

```
        hooks.LogMemoryByEpochHook(logger),
        hooks.LogTimingByEpochHook(timer, logger),
        hooks.TensorboardHook(log\\_dir='./tb\\_logs', ranks=[0]),
        hooks.SaveCheckpointHook(checkpoint\\_dir='./ckpt')
    ]
    # start training
    trainer.fit(train\\_dataloader=train\\_dataloader,
                epochs=gpc.config.NUM\\_EPOCHS,
                test\\_dataloader=test\\_dataloader,
                test\\_interval=1,
                hooks=hook\\_list,
                display\\_progress=True)
if \\_\\_name\\_\\_ == '\\_\\_main\\_\\_':
    main()
```

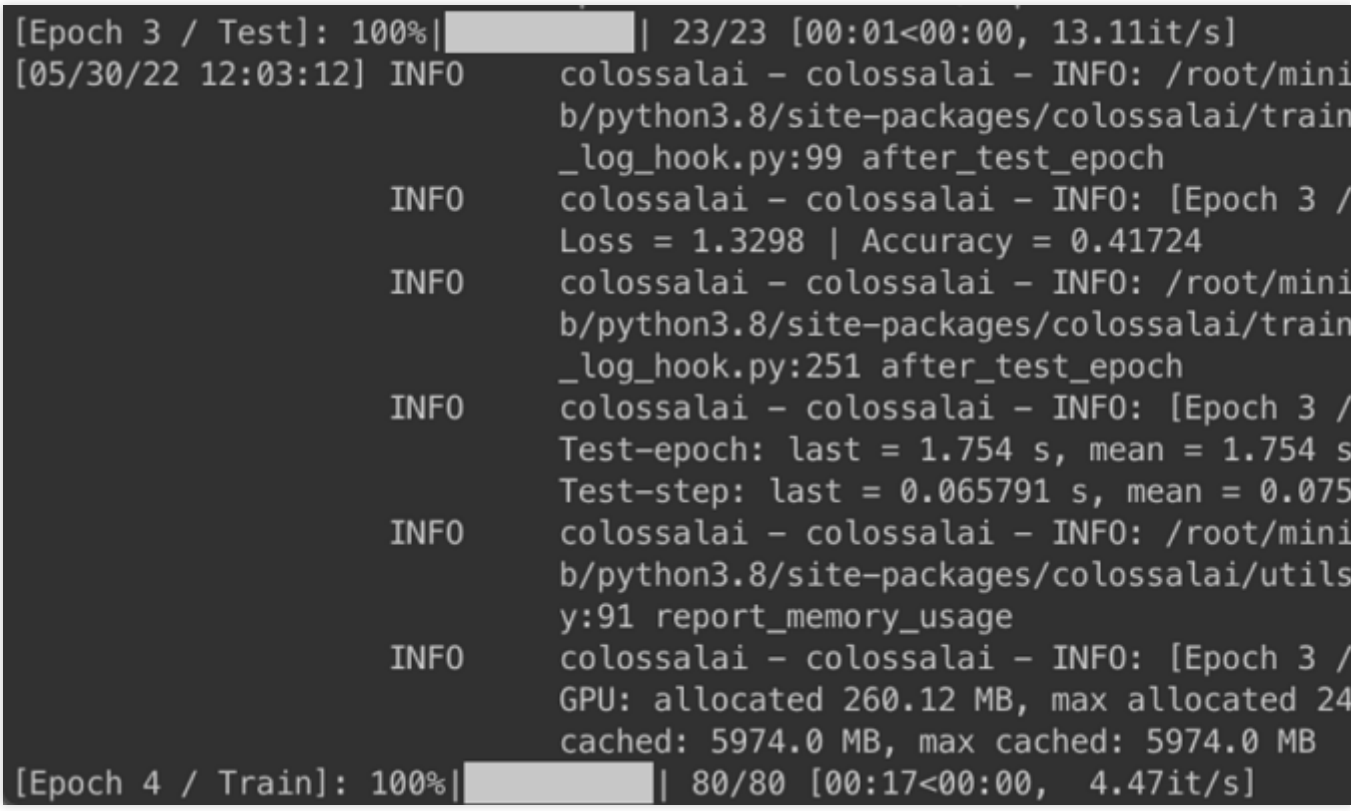
模型的具体配置如下所示：



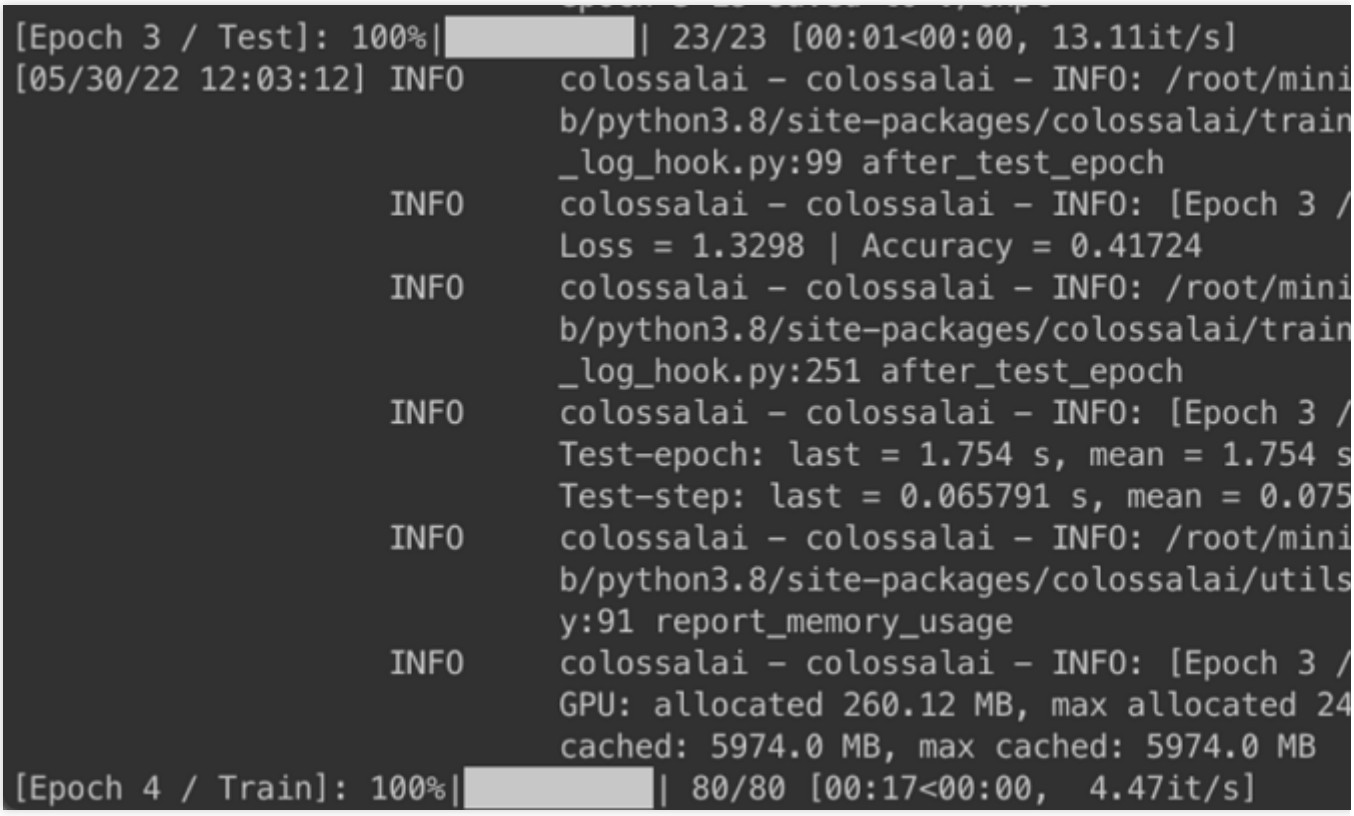
```
from colossalai.amp import AMP\\_TYPE

BATCH\\_SIZE = 128

DROP\\_RATE = 0.1

NUM\\_EPOCHS = 200

CONFIG = dict(fp16=dict(mode=AMP\\_TYPE.TORCH))

gradient\\_accumulation = 16
```

```
clip\\_grad\\_norm = 1.0

dali = dict(

 gpu\\_aug=True,

 mixup\\_alpha=0.2

)
```

模型运行过程如下图所示， 单个 epoch 的时间在20s以内：



结果显示模型在验证集上达到的最佳准确率为66.62%。我们也可以通过增加模型的参数量， 例如修改模型为 v。



总结

本次使用过程中遇到的最大的问题是从 GitHub 克隆非常缓慢，为了解决该问题，尝试使用了 tunnel 和 proxychains 工具进行提速。但该行为违反了云服务器使用规则，导致了一段时间的云服务器不可用，最终通过删除代理并提交工单的方式才得以解决。
借此也提醒其他用户，进行外网代理不符合云服务器使用规范，为了保证您服务的稳定运行，切勿违反规定。

参考

[1] Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." arXiv preprint arXiv:2010.11929 (2020).
[2] NVIDIA/DALI
[3] Bian, Zhengda, et al. "Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training." arXiv preprint arXiv:2110.14883 (2021).