指导老师：蓝善祯

4

使用说明介绍

Timm库

目录

[Timm库说明 3](#_Toc9425)

[1 Timm库介绍 3](#_Toc22145)

[1.1什么是Timm库 3](#_Toc4582)

[1.2功能介绍 3](#_Toc17293)

[1.3相关教程 3](#_Toc17288)

[2 pytorch环境配置 5](#_Toc885)

[2.1前置条件 5](#_Toc18429)

[2.2安装pytorch-GPU版本 5](#_Toc25001)

[3 Hugging Face的pytorch-image-models部署 9](#_Toc16974)

[3.1前置工作 9](#_Toc11538)

[3.2克隆项目仓库 9](#_Toc17823)

[3.3进入项目目录 9](#_Toc13474)

[3.4安装依赖 10](#_Toc13319)

[3.5注意事项 10](#_Toc11579)

[4 代码解读 11](#_Toc1386)

[4.1训练 11](#_Toc29613)

[4.2推理 15](#_Toc22726)

[5 MNIST复现练习 17](#_Toc24796)

# Timm库说明

## 1 Timm库介绍

### 1.1什么是Timm库

PyTorch Image Models，简称Timm，是一个巨大的图像分类模型库。通过Timm可以轻松的搭建出各种sota模型（目前内置预训练模型592个，包含densenet系列、efficientnet系列、resnet系列、vit系列、vgg系列、inception系列、mobilenet系列、xcit系列等等），并进行迁移学习。

### 1.2功能介绍

预训练模型集合：Timm库提供了一个广泛的预训练模型集合，包括各种经典的和最新的神经网络架构，如EfficientNet、ResNet、RegNet、ViT（Vision Transformer）等。这些模型在ImageNet和其他数据集上预训练，这些预训练权重可以帮助用户在特定任务上进行快速的模型微调，节省了从头训练的时间和计算资源。（对于某些模型，Timm库提供了不同版本的实现，如ResNet-18、ResNet-50、ResNet-101等，满足不同场景下的需求，用户可以根据任务要求选择合适的模型版本。）

模型加载与使用：Timm库提供了简单易用的API，用户可以通过几行代码快速加载预训练模型。同时，用户可以根据需要自定义模型结构，Timm库支持灵活的模型配置，方便进行模型架构的调整和优化。

训练与优化：Timm库提供了简洁的训练循环实现，用户可以快速搭建训练脚本，包括损失函数、优化器、学习率调度等配置。此外，Timm库也支持多GPU和分布式训练，通过PyTorch的分布式训练功能，用户可以轻松将训练扩展到多个GPU，提升训练速度。

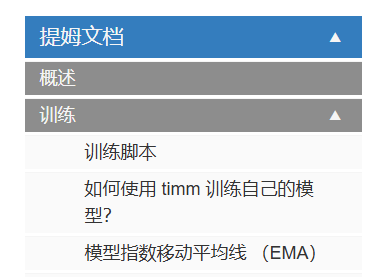
评估与验证工具：Timm库提供了常用的评估指标，如准确率、召回率、F1分数、平均精度（mAP）等，方便用户快速评估模型性能。用户还可以方便地在验证集上评估模型性能，Timm库提供了标准的评估接口，帮助用户快速获取模型在验证集上的表现。

### 1.3相关教程

Timm库网址：<https://github.com/huggingface/pytorch-image-models>

具体功能实现教程：https://Timm.fast.ai/





## 2 pytorch环境配置

### 2.1前置条件

已下载Anaconda,并为pytorvh安装创建了一个虚拟环境

`conda create -n <env\_name> python=<version>`

Eg:创建一个名为pytorch,python版本为3.8的虚拟环境命令如下

`conda create -n pytorch python= 3.8 `

虚拟环境激活，关闭命令

`conda activate <env\_name>`

`conda deactivate`

### 2.2安装pytorch-GPU版本

#### 2.2.1查看CUDA显卡驱动版本

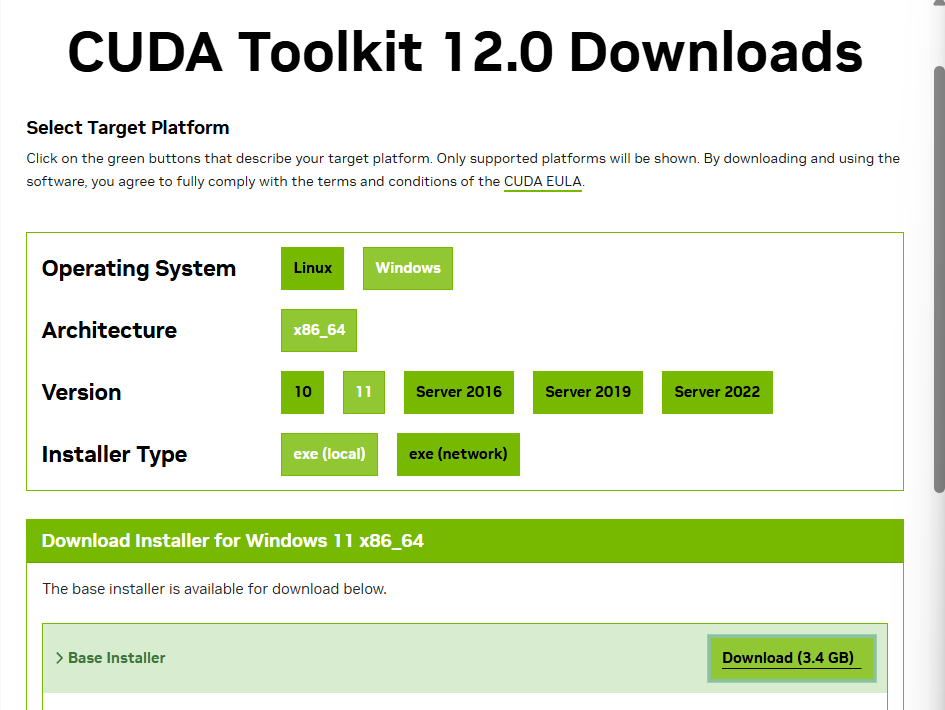
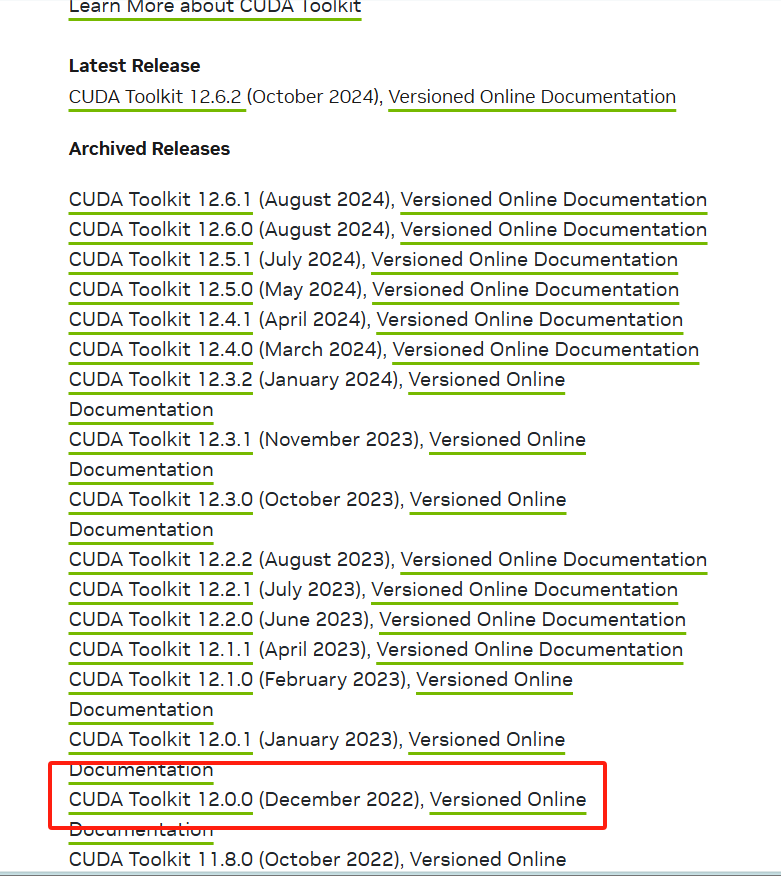
> 在cmd命令行终端输入nvidia-smi,可以查看到版本为12.2

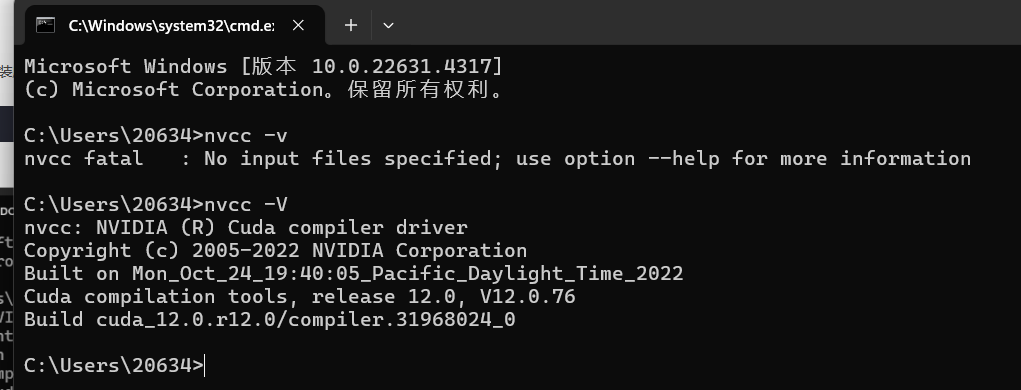
#### 2.2.2安装CUDA

为什么安装pytorch-GPU前一定要安装CUDA和CuDNN呢？

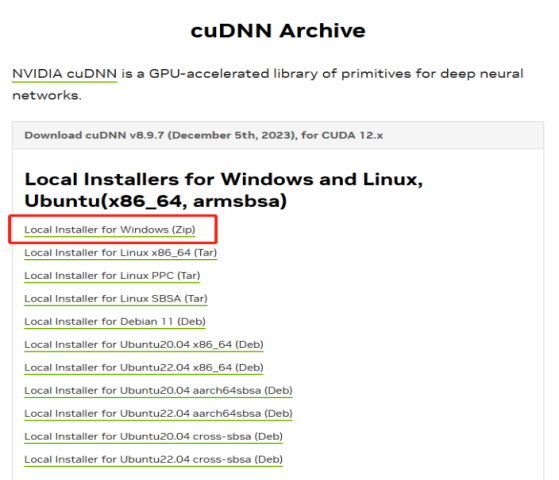
CUDA是NVIDIA推出的并行计算平台和API模型，它使得显卡可以用于图像渲染和计算以外的目的，例如通用并行计算。PyTorch通过CUDA可以充分利用GPU的计算能力，加速深度神经网络的学习和推理过程。

cuDNN是CUDA的扩展库，专门针对深度神经网络中的基础操作提供高度优化的实现方式，例如卷积、池化、规范化以及激活层的前向和后向过程。使用cuDNN可以大大提高深度学习模型在GPU上的运行效率。

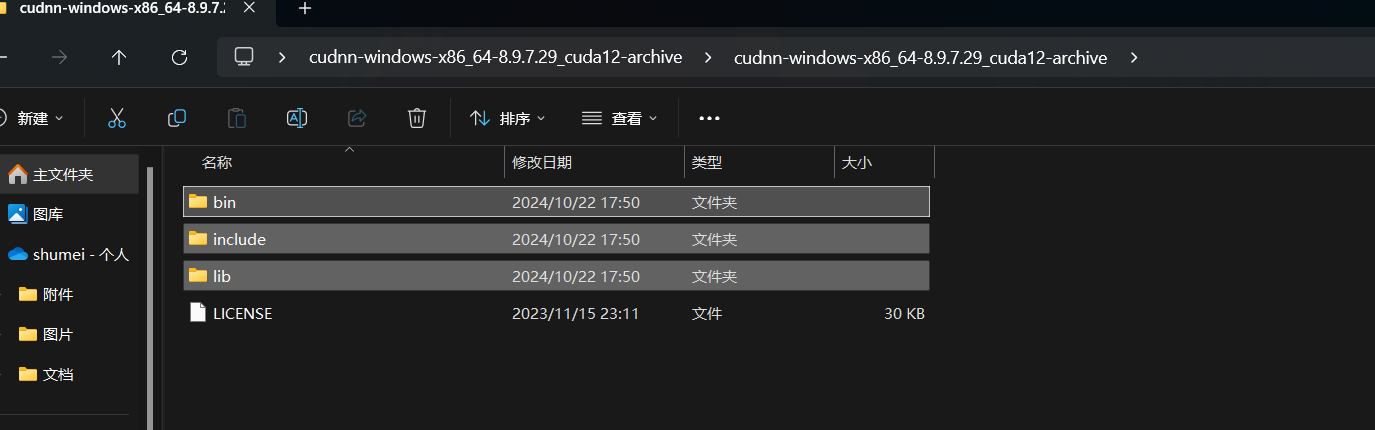
下载CUDA：从官网下载对应的CUDA版本，由于我的显卡版本为12.2，我只需要安装小于或者等于12.2都是可以的，因此这里我安装12.0。 官网地址：[CUDA Toolkit Archive | NVIDIA Developer](https://developer.nvidia.com/cuda-toolkit-archive" \o "CUDA Toolkit Archive | NVIDIA Developer)  


安装CUDA，点击下载的setup.exe,按照所给提示安装即可，安装后在命令行输入nvcc -V，显示如下图则说明安装成功  
  


#### 2.2.3安装CuDNN

下载：通过官网进行安装CuDNN，网址：[CUDA Deep Neural Network (cuDNN) | NVIDIA Developer](https://developer.nvidia.com/cudnn" \o "CUDA Deep Neural Network (cuDNN) | NVIDIA Developer) （**注意：需要注册登录才能进行安装哦）**  


安装：下载后发现其实cudnn不是一个exe文件，而是一个压缩包，解压后，有三个文件夹，把三个文件夹拷贝到cuda的安装目录下。

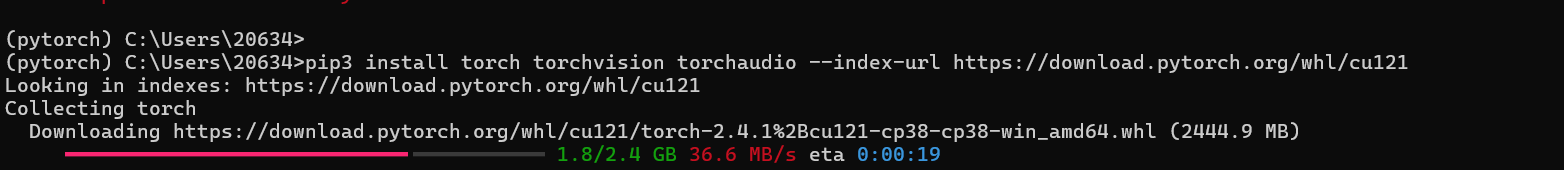
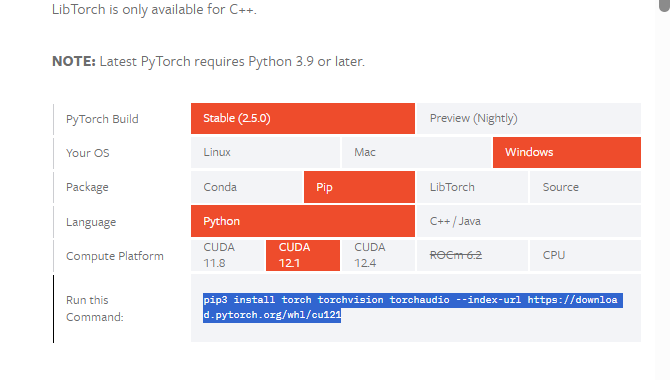


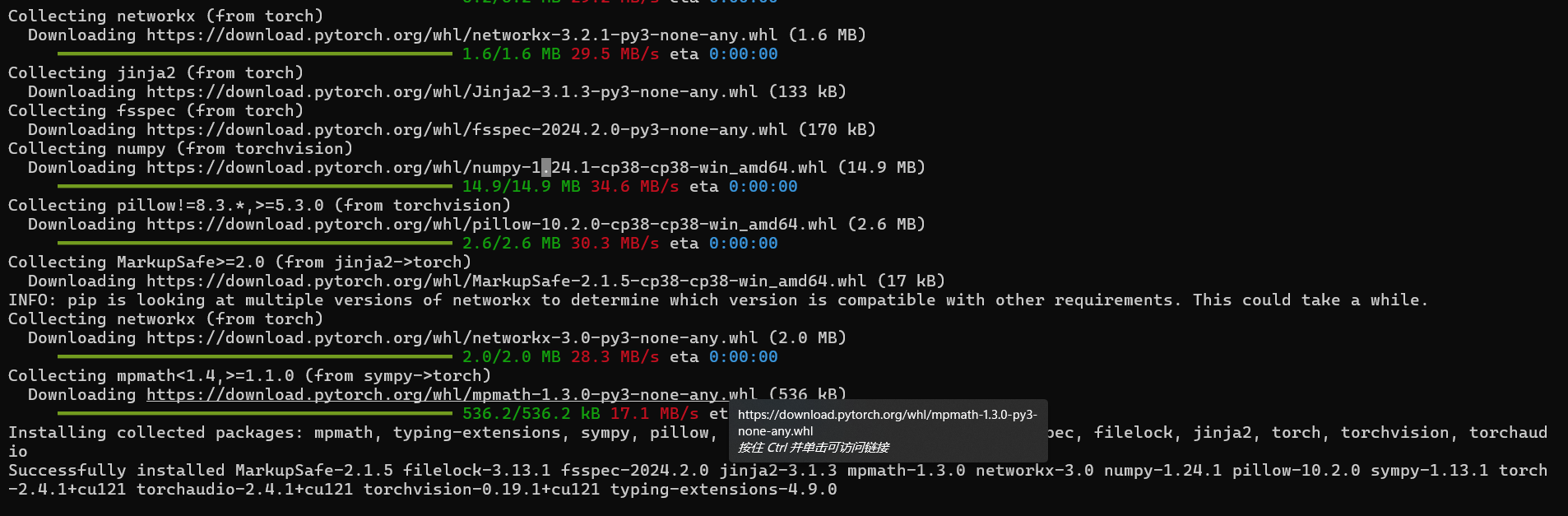
我的安装地址如下：



#### 2.2.4 安装PyTorch（GPU）

先激活前置条件中创建的虚拟环境，打开Anaconda prompt，输入下面的命令：conda activate pytorch

安装pytorch：激活的虚拟环境中，使用pip安装PyTorch。根据你的的需求，选择安装GPU版本。例如：下图所示（网络不佳和必要的时候可以添加国内源） ，注意CUDA版本要与电脑的CUDA显卡驱动版本一致  
**官网链接：**[PyTorch](https://pytorch.org/" \o "PyTorch)  
  
 将指令复制，输入到Anaconda prompt  
  
 安装成功



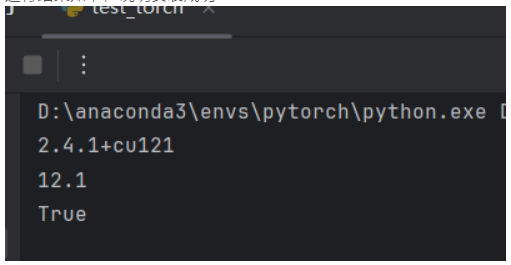
#### 2.2.5 pytorch-GPU安装验证

创建一个python项目，环境选为前置条件中创建的pytorch环境，运行下面这段代码：

import torch print(torch.\_\_version\_\_)

print(torch.version.cuda)

print(torch.cuda.is\_available()) #输出为True，则安装成功

运行结果如下，说明安装成功！  


## 3 Hugging Face的pytorch-image-models部署

### 3.1前置工作

**安装 Git**：若未安装Git，可从[Git官网](https://git-scm.com/downloads)下载并安装。

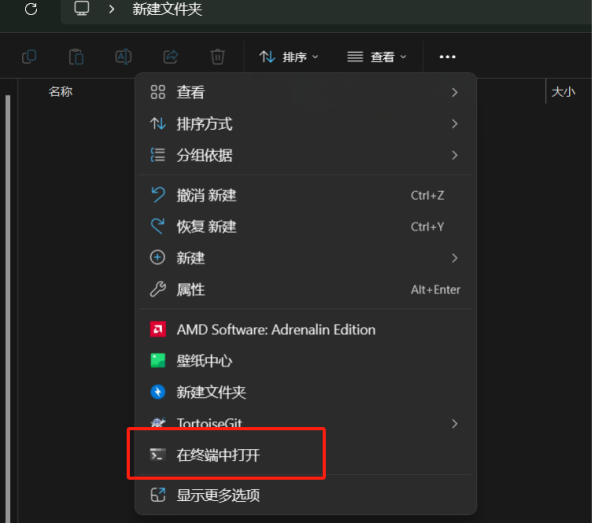
**安装 Python**：建议安装Python 3.7及以上版本，可从[Python 官网](https://www.python.org/downloads/)下载安装。

**安装 pip**：通常Python安装时会自带pip，可通过命令pip --version检查是否已安装及版本信息，若未安装，可参考 [pip 官方安装指南](https://pip.pypa.io/en/stable/installation/) 进行安装。

创建好安装好了pytorch的训练环境，详细见 pytorch环境配置文档。

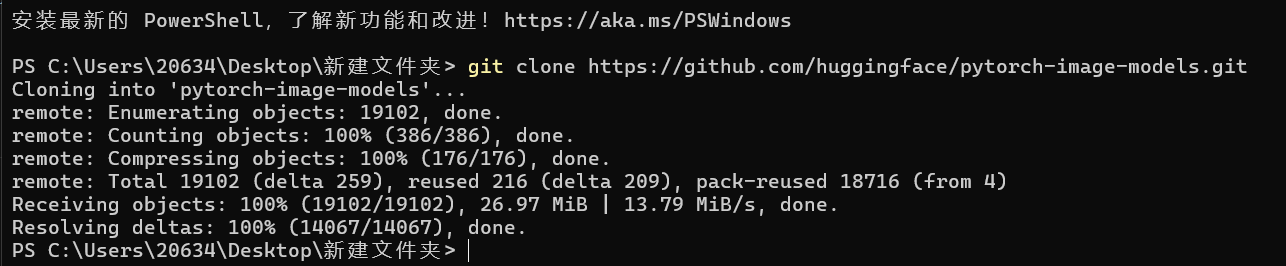
### 3.2克隆项目仓库

创建一个文件夹，打开命令行工具（Windows 系统可使用命令提示符或 PowerShell，Mac/Linux 系统可使用终端）。



输入以下命令克隆 pytorch-image-models 项目仓库到本地：

git clone <https://github.com/huggingface/pytorch-image-models.git>



此命令会将项目仓库克隆到当前目录下的 pytorch-image-models 文件夹中。

### 3.3进入项目目录

克隆完成后，在命令行中输入以下命令进入克隆下来的项目目录：

cd pytorch-image-models

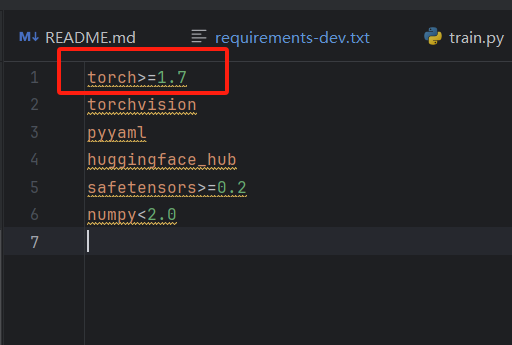
### 3.4安装依赖

安装 PyTorch：根据自己的系统环境和CUDA版本，从 [PyTorch 官网](https://pytorch.org/get-started/locally/) 选择合适的安装命令进行安装。详细见pytorch配置文档。

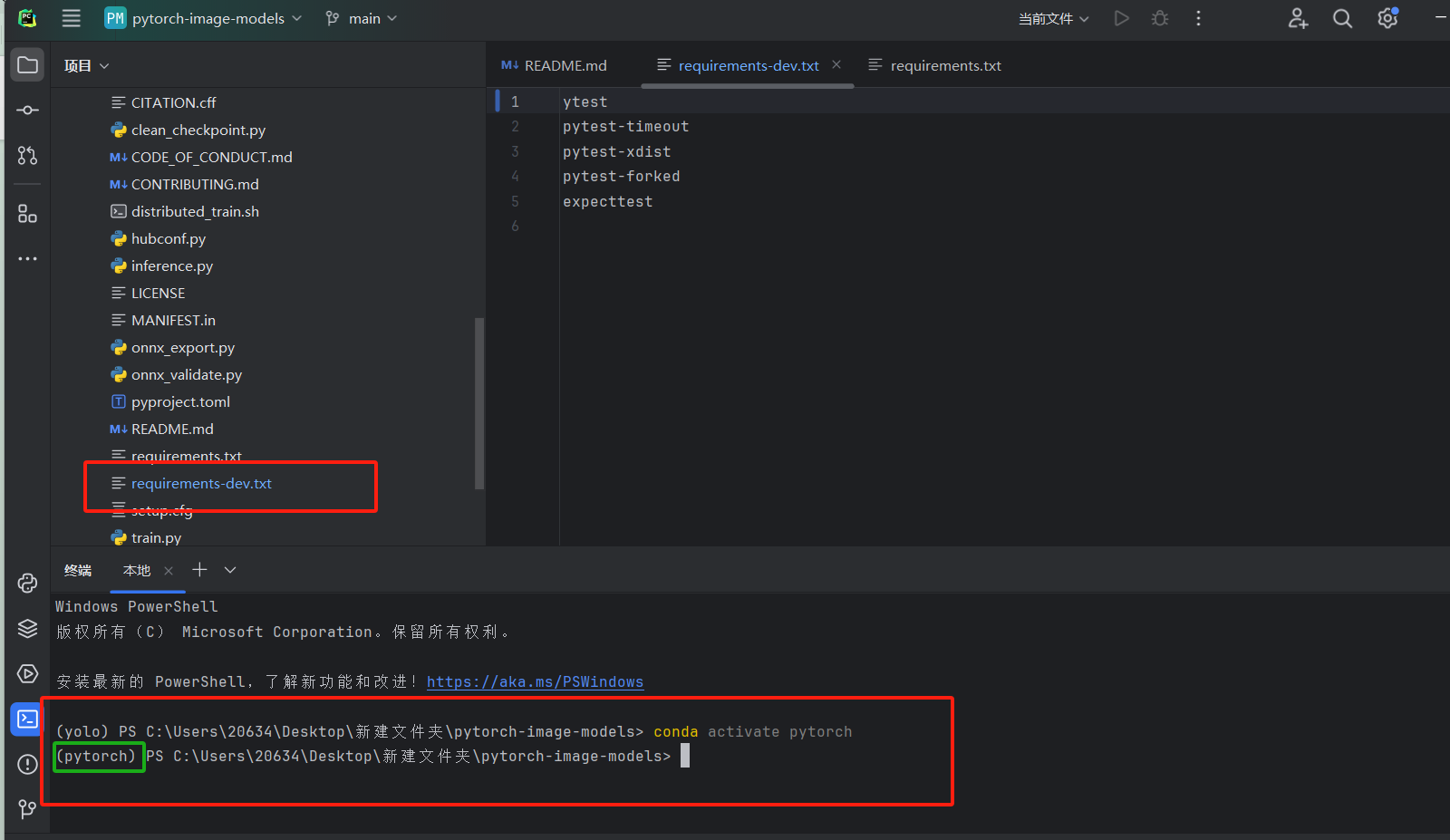
激活虚拟环境：Conda activate pytprch (注意pytorch是虚拟环境的名字)

安装项目所需依赖：在项目目录下，使用 pip 安装项目所需的其他依赖包。可以使用项目提供的 requirements.txt 文件来安装所有依赖：

！！！注意 ，如果自己电脑安装的torch版本小于1.7的话，请将红框框住的这一句注释掉，或者删除。



pip install -r requirements.txt



该文件中列出了项目运行所需的各种库及其版本要求，通过上述命令可一次性安装所有依赖。

### 3.5注意事项

**网络问题**：若在安装过程中遇到网络问题导致下载依赖失败，可尝试配置国内镜像。

例如，在Linux/Mac 系统中输入 export HF\_ENDPOINT=https://hf-mirror.

com，在 Windows 系统中输入 set HF\_ENDPOINT=https://hf-mirror.com。

**兼容性问题**：项目中部分模型可能需要特定版本的 PyTorch 或其他库，若在使用过程中出现兼容性问题，可参考项目文档或报错信息调整相关库的版本。

**环境隔离**：建议在虚拟环境中进行项目部署和开发，可使用 conda 或 venv 创建虚拟环境，避免不同项目之间的依赖冲突。

例如，使用 conda 创建虚拟环境的命令为：

conda create -n pytorch\_image\_models python=3.8

激活创建的名为pytorch的虚拟环境命令：

conda activate pytorch\_image\_models

然后再在该虚拟环境中进行后续的安装和配置操作。

## 4 代码解读

本项目为imagenet模型复现项目，故数据集需要事先处理为imagenet格式（具体参见“MNIST数据集下载”章节）

项目中所有预训练模型（个别标注出的除外）均为在[ImageNet-1k](https://image-net.org/download)数据集（1000类，1,281,167张训练集图片）上预训练得到，并非其对应的原始论文开源出的预训练权重（待做：不确定和论文中开源的权重是否存在关系）

推荐先熟悉代码运行流程，再阅读代码，这样能更好理解代码内容。

更新Timm版本可能导致兼容问题，本文档根据Release v1.0.13（2025.1.10）整理。

### 4.1训练

#### 4.1.1模型创建

> Timm.models.\_factory

>

> def create\_model(model\_name: str,

> pretrained: bool = False,

> pretrained\_cfg: Union[str, Dict[str, Any], PretrainedCfg, None] = None,

> pretrained\_cfg\_overlay: Optional[Dict[str, Any]] = None,

> checkpoint\_path: Union[str, Path, None] = None,

> cache\_dir: Union[str, Path, None] = None,

> scriptable: Optional[bool] = None,

> exportable: Optional[bool] = None,

> no\_jit: Optional[bool] = None,

> \*\*kwargs: Any) -> Any

形参:  
model\_name: 模型名称

pretrained: 如果为`True`则加载于ImageNet-1k上预训练好的权重

pretrained\_cfg: 从外部加载模型的预训练配置

pretrained\_cfg\_overlay: 用于覆盖或更新在 `pretrained\_cfg` 中定义的预训练配置参数，适合在已有配置的基础上做部分修改

checkpoint\_path: 虽然用于加载ckpt相关，但一般在其他地方实现这个任务

cache\_dir: 覆盖模型的缓存路径，指自动下载模型时的存放路径

scriptable: 使模型支持JIT脚本化，提高运算性能（并非所有模型都支持）

exportable: 设置模型层的配置，使模型支持可追踪（traceable）或导出为 ONNX 格式（尚未完全实现）

no\_jit: 用于禁用特定层（目前为激活函数层）的 JIT 脚本化，可能是为了避免与某些不兼容的组件冲突，或用于调试

关键字实参:  
drop\_rate: 训练时分类头（Classifier）的dropout rate

drop\_path\_rate – 随机深度（Stochastic Depth）的dropout比例

global\_pool: 分类头的global pooling类型.

官方给出的模型创建示例

The variety of training args is large and not all combinations of options (or even options) have been fully tested. For the training dataset folder, specify the folder to the base that contains a train and validation folder.

To train an SE-ResNet34 on ImageNet, locally distributed, 4 GPUs, one process per GPU w/ cosine schedule, random-erasing prob of 50% and per-pixel random value:

./distributed\_train.sh 4   
 --data-dir /data/imagenet   
 --model seresnet34   
 --sched cosine   
 --epochs 150   
 --warmup-epochs 5   
 --lr 0.4   
 --reprob 0.5   
 --remode pixel   
 --batch-size 256   
 --amp -j 4

（原命令行无换行）

代码自动从huggingface下载模型（需配置代理），模型存放路径为C:\Users\用户名\.cache\huggingface\hub，可手动下载模型后放入对应路径。

读取模型时，代码会根据模型类型的不同自动使用不同的解析方法。

下载的模型均为.safetensors格式，而训练得到的checkpoint为.pth.tar格式

代码中，直接对创建好的模型进行print可以得到模型的结构信息。关于获取模型信息相关的代码已经写好。（try\_mnist.py）

可以选取一个经典模型的经典版本（如ViT，选取论文中列举出具体结构的版本），结合代码来学习模型架构。

imagenet-1k的数据集中，每个图片的尺寸并不固定，通常训练时会将其处理成相同大小（裁剪、缩放或填充）。

待做：Timm是如何根据模型的需要对数据集进行大小调整的；Timm是否会把所有模型都应用为相同尺寸的输入

#### 4.1.2可调参数

**·num\_classes（分类类别数量）**

例：num\_classes: 1000

1000类的分类头

**·checkpoint\_hist（保存几个checkpoint）**

**·eval\_metric（选取最佳模型的依据）**

例：eval\_metric: loss

实际效果：选取验证集上loss最低的若干个checkpoint进行保存

**·aa（数据增强）**

例：aa: rand-m9-mstd0.5

rand：random augmentation随机增强（代码中给出了论文）

m9：幅度为9，意为操作（旋转、调色等）的强度

mstd0.5：幅度的标准差为0.5，意为幅度的波动程度

例：rand-m9-n3-mstd0.5

RandAugment 变换，强度为 9，层数为 3，强度标准差为 0.5；

例：rand-mstd1-tweights

强度标准差为 1.0，使用加权变换，默认强度为 10，层数为 2

其他未指定参数：

n：每张图片应用的增强操作个数（默认为2）

p：每张图片每种增强操作的应用概率（默认为0.5）

mmax：强度上限的自定义值，默认为 \_LEVEL\_DENOM (10)

inc：整数（布尔值），决定是否使用强度随增强级别增加的变换（默认值 0）

t：字符串，指定使用的变换集的名称

输入参数的格式为 “参数a+值-参数b+值”，第一部分固定为'rand'，剩余部分顺序不限

**·freeze\_layer**

ibiohash的配置文件中出现了此参数，但暂未在项目中搜索到，疑似为无效参数

**AMP混合精度训练（减小模型显存消耗）**

# line 448

有两种实现方法：APEX 或 PyTorch 原生 AMP，实现时会先确认数值类型（float16 或 bfloat16）的合法性。

APEX 是 NVIDIA 提供的一个库，用于加速混合精度训练，适用于支持 CUDA 的硬件。PyTorch 也提供了原生的 AMP 支持，但要求使用较新的 PyTorch 版本。

参数配置：

amp: True（启用AMP）

amp\_impl: 'apex'（通过APEX实现，否则通过pytorch的native实现）

amp\_dtype: 'float16'（本条疑似可以省略，程序包含默认值）

具体实现细节：

根据 args.amp\_impl 的值，选择不同的 AMP 实现：

如果 args.amp\_impl == 'apex'，使用 APEX 实现。代码会检查 APEX 是否安装，如果没有安装则报错。并且确定 AMP 数据类型args.amp\_dtype必须是 float16。

如果 args.amp\_impl != 'apex'，则使用 PyTorch 原生 AMP（Native AMP）。代码会检查当前 PyTorch 版本是否支持原生 AMP（通过 has\_native\_amp 标志）。并且确定 AMP 数据类型必须是 float16 或 bfloat16。

如果 args.amp\_dtype 是 'bfloat16'，则将 amp\_dtype 设置为 torch.bfloat16，这是另一种混合精度格式，通常用于一些特定的硬件（如 TPUs）。

待做：什么样的模型适用AMP？

**EMA权重平均**

# line 615

基本的假设是，模型权重在最后的 步内会在实际的最优点处抖动。对最后 步权重取平均，可以一定程度上修正数据偏差，从而使得比直接使用最新的模型权重效果更好

EMA（Exponential Moving Average，指数移动平均） 是一种平滑技术，用于对模型权重进行加权平均，从而提高模型的稳健性和泛化能力。其更新公式为：

：前t个时刻的 的EMA平均值，称为 时刻的影子权重。在梯度下降的过程中会一直维护这个影子权重，但是这个影子权重并不会参与训练。

通常情况下 ，在步数较少时存在一定偏差，需要引入修正系数

：当前时间步的模型权重

：衰减系数，通常设为接近于 1 的值（0.9 - 0.999）

SWA（Stochastic Weight Averaging，随机权重平均） 对多个独立的模型权重（通常来自不同训练阶段）进行简单的平均：

：模型在第 i 时间步的权重

SWA 在训练过程中定期保存模型权重（通常在训练后期的多个epoch），最终使用这些权重的平均值作为模型的最终权重，计算效率较低。

参数配置：

model\_ema: True （启用EMA）

model\_ema\_decay: 0.999 （默认0.9999）

model\_ema\_force\_cpu: True （在CPU上计算EMA以节约显存，默认关闭）

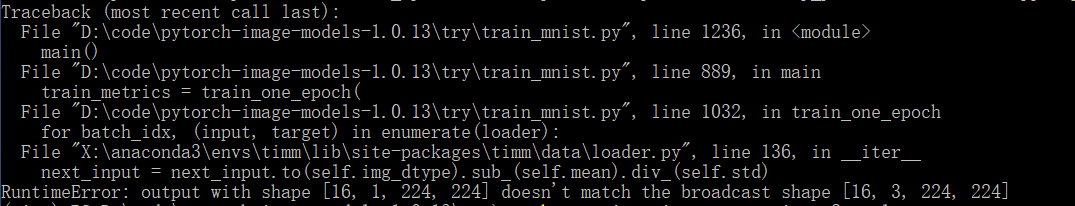
warmup相关 默认关闭，参见EMA实现脚本

待做：使用EMA后，最终输出的模型是什么样子？

**input\_img\_mode（数据集图像色彩模式）**

数据集相关的设定。如果没有传入参数，代码会根据数据集的输入通道数自动设定格式为 'RGB' 或者 'L' （Luminance，即亮度，指黑白模式）。

目前存在问题：尝试将mnist数据集以黑白模式输入，但报了格式不匹配的错误，可能是因为和VIT模型的接口不兼容



待看：模型建立，数据集建立，优化器建立（区分）  
整理可调参数时，还是要再从头捋一遍代码

### 4.2推理

#### 4.2.1推理参数

模型导入

model：模型名称

checkpoint：需要推理的模型ckpt的路径

可一次性推理同一模型的多个ckpt，通过将--checkpoint设定为ckpt的存放\*\*文件夹\*\*

可一次性推理多个模型，通过将--model设定为列举出模型名称的\*\*txt文件的路径\*\*

推理配置

和训练阶段类似的参数：amp相关、num\_classes、data\_dir、input\_img\_mode

split：设为测试集的分割目录名称

num\_gpu：GPU数目，大于1时启用分布式训练

device：默认为cuda，可更改为npu

retry：默认为False，设为True时启用函数\_try\_run（见下方注释）以实现自动调整batchsize推理

```

函数 \_try\_run(args, initial\_batch\_size):

# 尝试调用validate函数进行推理。当推理出错时，如果错误类型在允许范围

（如outofmemory），则自动降低batchsize重新推理

# 仅在--retry参数为True、或者存在多个模型需要推理时才会启用这个函数

# 分类模型推理过程一般不受batchsize影响，如此可以减少调试的工作量

```

use\_ema：使用EMA版本的权重

待研究：

EMA在推理上的作用？

参数名称中，短横线和下划线是否可以混用？从代码来看，作者经常混用。参数解析声明处均使用 - ，但有时会通过dest关键字让其存储在 \_ 中，但有时又不会进行这个操作且在下文中直接使用 \_ 。之前的训练配置文件也都是混用的，并没有报过错，具体原理待研究。

crop（裁剪）待学习

crop\_pct：crop裁剪操作的百分比

crop\_mode：

crop\_border\_pixels：

推理开始时，先进行了一次warmup，具体功能待研究

```

model.eval()

with torch.no\_grad():

# warmup, reduce variability of first batch time, especially for comparing torchscript vs non

input = torch.randn((args.batch\_size,) + tuple(data\_config['input\_size'])).to(device=device, dtype=model\_dtype)

if args.channels\_last:

input = input.contiguous(memory\_format=torch.channels\_last)

with amp\_autocast():

model(input)

```

结果输出

通过--results\_file参数指定输出文件路径

如果没有输入本参数，会以标准json格式输出至默认路径（否则输出两份，指定路径和默认路径）

经测试，不设定本参数时并不存在默认输出（或者不清楚所谓的“默认路径”是哪里）

通过--results\_format指定输出文件的格式（格式默认为\*\*csv\*\*，可更改为json）

#### 4.2.2分类别推理

指获取不同类别的推理结果、获取每一张图片的推理结果

修改了validate.py代码文件，使其可以在推理的过程中直接存储下每张图片的推理结果，并导出到本地，得以在后续代码中进行处理。

存储结果时顺序是固定的（图片的输入顺序固定，图片和标签的对应关系也固定），故可以直接将其用做分图片推理的结果

经测试，推理时的图片输入顺序就是 `类别→名称` 排序（MNIST排序规则：0类＜1类，118.jpg < 787.jpg < 1489.jpg）

以上数据集拓展到一般情况时做的假设：图片名称、类别名称均为纯数字。如果不符合上述假设，顺序可能会被打乱。

分类别推理、分图片推理的代码：

category-wise\_inference.py、picture-wise\_inference.py，均放在文件夹“category-wise\_inference”中。修改代码文件中output的目录后直接运行即可

**其他**：发现还存在一个inference.py，应该也是推理相关，后续研读一下这个代码

## 5 MNIST复现练习

**MNIST数据集下载**

原始数据集的格式为非常规格式，直接从官方下载需要转换格式。可通过PyTorch的torchvision.datasets.MNIST接口下载，并将数据集保存为JPEG图片保存到本地。

目前已经完成了本环节的代码，可以完成MNIST数据集的下载、转换、保存、转为imagenet数据集格式。（download\_mnist.py，dataset\_splitting.py）

**开始训练**

python train.py -c config.yaml

config.yaml为训练的配置文件

代码会自动下载模型文件，需要保证终端能够连接到hugging face。也可事先手动从hugging face下载，放到模型缓存目录内，代码可直接识别。

**推理**

python validate.py -c config.yaml

config.yaml为推理的配置文件

validate.py原本不支持传入配置文件，只能在命令行逐个传入参数（或使用其他方式启动）。为了保持一致性方便操作，将代码文件作了一些修改，目前可以支持传入配置文件了。