

pytorch 常见包

xbZhong

2025-01-24

Contents

Pytorch 1

[本页 PDF](#)

Pytorch

pytorch 与 **tensorflow** 的区别：

pytorch 支持动态计算图

- 计算图在代码运行时动态生成
- 可根据运行逻辑实时更改计算图

tensorflow 支持静态计算图

- 先定义后执行
- 计算图在编译后无法再修改

常用包

- Dataset(数据集)：提供一种方式去获取数据及其 label，并形成编号
 - 需要完成的功能：
 - * 如何获取每一个数据及其 label
 - * 告诉我们总共有多少的数据
 - 所有的数据集都需要去继承 Dataset 类，所有的子类都要重写 **getitem** 方法（获取每个数据及其 label）与 **len** 方法（获得数据集长度）
- **getitem**(self, index)：这个方法让类的实例对象可以像列表一样通过索引来访问元素。当你调用 `ants_data[3]` 时，本质上是调用了 `ants_data.__getitem__(3)`。在这个方法中，你定义了如何根据索引来获取数据。在你的代码中，**getitem** 方法根据索引获取图像的文件名，并将图像加载到内存中，同时返回图像和标签
- `os` 库中的 `listdir` 方法可以让图片名变成列表，`path.join` 方法可以让前后两个路径用 `'` 拼接起来
- `Image.open(path)` 表示打开以 `path` 为路径的文件，`.show` 表示展示这个图片
- Dataloader(数据装载器)：为后面的网络提供不同的数据形式

导入包

```
from torch.utils.data import Dataset
```

PIL 为图像处理库

```
from PIL import Image
```

os 库是用来操作文件的

```
import os
```

```
class Mydata(Dataset):
```

```

def __init__(self, root_dir, label_dir):
    self.root_dir = root_dir
    self.label_dir = label_dir
    self.path = os.path.join(self.root_dir, self.label_dir)
    self.img_path = os.listdir(self.path)

def __getitem__(self, index):
    img_name = self.img_path[index]
    img_item_path = os.path.join(self.root_dir, self.label_dir, img_name)
    img = Image.open(img_item_path)
    label = self.label_dir
    return img, label

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

root_dir = 'C:\study\pytorch'
label_dir = 'ants'
ants_data = Mydata(root_dir, label_dir)
img, label = ants_data[5]
img.show()

```

Tensorboard

作为 **pytorch** 中的一部分，是一个用于可视化深度学习模型训练过程和结果的工具 #### SummaryWriter 类的使用 * 初始化函数可以输入一个文件夹名称，使得这个文件被 tensorboard 解析, 使用完后要把对象关掉 * writer.add_scalar() * tag: 图表标题 * scalar_value: 要保存的数值，对应的是 y 轴 * global_step: 训练了多少步，对应的是 x 轴

1. 在 pycharm 中配置 conda 环境
2. 设置 pytorch 虚拟环境
3. 编完代码后在终端输入 conda activate pytorch 进入 pytorch 虚拟环境
4. 运行代码并将终端目录调整至生成文件的文件夹的上一级
5. 在终端输入 tensorboard -logdir= 生成文件的文件夹名
6. 打开网址（端口是默认的，也可以自定义）

```

from torch.utils.tensorboard import SummaryWriter
## 创建 SummaryWriter 并指定日志目录

writer = SummaryWriter('C:/study/pytorch/logs')

## 假设在训练过程中记录损失

for i in range(100):
    writer.add_scalar('y=2x', i*2, i)

## 关闭 SummaryWriter

writer.close()

```

- writer.add_image()
 - tag: 图像标题
 - img_tensor: 数据类型要么是 torch.Tensor, numpy.array, 或者 string/blobname
 - * 使用 opencv 库去读取图片，得到的类型是 numpy.array 类型
 - * 也可以利用 numpy.array(), 将 PIL 图片进行转换，但要在 add_image() 中指定 shape 中每一个数字/维表示的含义

- * 在传输路径时要在路径前加上 r
- step: 步数
- dataformats: 数据类型, 由通道颜色数, 宽度高度组成, img_np 类型是 HWC, 即高度, 宽度, 颜色通道数 (一般为 3)

```
from torch.utils.tensorboard import SummaryWriter
from PIL import Image
import numpy

## 创建 SummaryWriter 并指定日志目录

writer = SummaryWriter('C:/study/pytorch/logs')
img_path = r'C:\study\pytorch\ants\0013035.jpg'
img_PIL = Image.open(img_path)
img_np = numpy.array(img_PIL)

for i in range(100):
    writer.add_scalar('y=2x', i*2, i)
writer.add_image('test', img_np, 1, dataformats='HWC')

## 关闭 SummaryWriter

writer.close()
```

Transforms(torchvision 库里面)

- 一个工具箱: 里面由许多类 (方法) 组成
 - 关注输入和输出类型
 - 看官方文档
 - 关注需要传入什么参数
- 使用时要先实例化对象: tensor = transforms.ToTensor(), 然后利用对象进行类型转换等操作
- opencv 的 cv2 是把图片变为 numpy 类型 ##### 常见的 Transforms
- PIL: Image.open()
- tensor: ToTensor()
 - torch.tensor() 可以把数据变成 tensor 类型
- numpy: cv.imread() **__call__ 方法可以让实例对象调用方法时像调用函数一样 ** ##### compose 类 把不同的 transforms 结合在一起

ToTensor 类

- 把 PIL 或 numpy 数据类型转换成 tensor 数据类型 ##### ToPILImage 类 把图片类型转换为 PIL 类型 ##### Normalize 类
- 必须要是个 tensor 数据类型
- 参数: 传入为列表
 - 均值: 填三个
 - 标准差: 填三个
 - 计算公式: (输入-均值)/标准差

Resize 类

- 输入为 PIL 类型, 返回值也是 PIL 类型
- 重新定义图片大小, 传入为元组

Compose 类

- Compose() 中的参数需要是一个列表, 且其数据类型需要是 transforms 类型
- 本质上是对图片操作的方法变成列表放在 compose 里, 减少了代码

RandomCrop

- 给定一个 PIL 数据类型，进行随机裁剪
- 传入参数可以是序列也可以是一个整数，整数的话会进行裁剪，裁剪为一个正方形

```
* from torchvision import transforms
from PIL import Image
from torch.utils.tensorboard import SummaryWriter

# totensor

writer = SummaryWriter('logs')
img_path = r'C:\study\pytorch\bees\16838648_415acd9e3f.jpg'
img = Image.open(img_path)
trans_totensor = transforms.ToTensor()
img_tensor = trans_totensor(img)
writer.add_image('text',img_tensor)

# normalize

# 三通道数据：rgb(红绿蓝)，因此要有三个均值，三个标准差

trans_norm = transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])
img_norm = trans_norm(img_tensor)

# resize

trans_resize = transforms.Resize((512,512))
img_resize = trans_resize(img)

# Compose

trans_resize_2 = transforms.Normalize(512)
trans_compose = transforms.Compose([trans.resize_2,trans.totensor])
img_resize_2 = trans_compose(img)

# RandomCrop

trans_random = transforms.RandomCrop(512)
trans_compose_2 = transforms.Compose([trans_random.trans_totensor])
for i in range(10):
    img_crop = trans_compose_2(img)
writer.close()
```

torchvision 中的数据使用

- 提供了许多可用的数据集
- Dataset 参数：
 - root: 下载的数据集要存放的位置
 - train: true 为训练数据集, false 为测试数据集
 - transform: 对数据集进行处理
 - target_transform: 对结果进行处理
 - download: true 则自动为我们下载, false 则不为我们下载

将 PIL 类型转变为 tensor 类型

```
data_transform = torchvision.transforms.Compose(
```

```

        [torchvision.transforms.ToTensor()
        ]
    )
train_set = torchvision.datasets.CIFAR10(root='./dataset',train=True,transform=data_transform,download=True)
test_set = torchvision.datasets.CIFAR10(root='./dataset',train=False,transform=data_transform,download=True)

## 测试集的第一个数据

print(test_set[0])

## 测试集里面的类型

print(test_set.classes)

## 数据包含图片和 label, 且图片为 PIL 类型

img,label = test_set[0]

## img.show()

writer = SummaryWriter('runs')
for i in range(10):
    img,target = test_set[i]
    writer.add_image('test_set',img,i)

```

Dataloader 的使用

- 一个加载器，把我们的数据加载到神经网络中，取多少数据。如何取数据都是取决于 Dataloader
- 常见参数：
 - dataset: 数据集
 - batch_size: 每次取的数据个数，然后进行打包
 - shuffle: True 表示每提取一次数据便会打乱数据，False 表示不会打乱数据
 - num_workers: 使用单线程还是多线程
 - drop_last: 当数据集总数除以 batch_size 有余数时，True 表示舍去这剩下的数据，False 表示不舍弃剩下的数据
- 代码中的 imgs 会作为数据传输到神经网络

```

import torchvision
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
test_set=torchvision.datasets.CIFAR10(root='./dataset',train=False,transform=torchvision.transforms.ToTensor)
test_loader = DataLoader(dataset=test_set,batch_size=64,shuffle=True,num_workers=0,drop_last=False)
writer = SummaryWriter('dataloader')
step=0
for data in test_loader:
    imgs,targets = data

    # 要使用 add_images 方法

    writer.add_images('data_test',imgs,step)
    step+=1
writer.close()

```

神经网络的搭建

- 最常用的模块：Module 类
 - 需要继承的父类：nn.Module，需要导入包：from torch import nn
 - forward 方法：input 经过 forward 变成 output ### 代码框架构建步骤
1. 查看官方文档，了解参数
 2. 传入数据集，导入必要的包
 3. 定义类，利用 super() 继承 nn.Module 的属性
 4. 在 forward 方法中定义将 input 转变为 output 的方法
 5. 必要时需要用 reshape 进行 batch_size, channel 的重定义 (可能数据类型不满足要求)
 6. 在 tensorboard 中进行图片的可视化

激活函数

在 torch.nn.Function 下

$$ReLU(x) = \max(0, x)$$

- Relu
 - 输出范围： $[0, +\infty)$
 - 零中心分布，梯度比 Sigmoid 更强。
 - 梯度消失问题仍存在。

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Tanh
 - 输出范围： $(-1, 1)$
 - 计算高效，缓解梯度消失（正区间梯度为 1）。
 - 稀疏激活（负输入直接置 0）。

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Sigmoid
 - 输出范围： $(0, 1)$
 - 梯度消失（输入绝对值较大时梯度接近 0）。
 - 输出非零中心（影响梯度更新效率）。

卷积

- conv2d（二维卷积）
 - 主要参数：
 - * input: 输入
 - 参数：(N,C,H,W) : 分别是 batch_size: 样本数量；通道数：二维张量通道为 1；高度和宽度
 - * weight: 卷积核，本质上是多维数组
 - * bias: 偏置
 - * stride: 步长，可以是单个数：横向移动和纵向移动步数相同。也可以是元组：(纵向移动，横向移动)
 - * padding: 在输入图像左右两边进行填充，给定一个数或元组 (纵向，横向)，空的地方默认为 0
- 卷积：数字一一匹配并相乘，然后相加

```

* from torch import nn
import torch

# input 与 output 都是 tensor 类型

class My(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, input):
        output = input + 1
        return output

my = My()
x = torch.tensor(1.0)

# 也可以是 my(x), 因为 nn.Module 重载了 __call__ 方法, 使其可以直接调用 forward 方法

y = my.forward(input = x)
print(y)
import torch
import torch.nn.functional as F
input = torch.tensor([[1,2,0,3,1],
                      [0,1,2,3,1],
                      [1,2,1,0,0],
                      [5,2,3,1,1],
                      [2,1,0,1,1]])

# 卷积核

kernel = torch.tensor([[1,2,1],
                       [0,1,0],
                       [2,1,0]])

# size 要求四个参数, 因此用 reshape

input = torch.reshape(input, (1,1,5,5))
kernel = torch.reshape(kernel, (1,1,3,3))
output = F.conv2d(input, kernel, stride=1)
print(output)
output_2 = F.conv2d(input, kernel, stride=2)
print(output_2)
output_3 = F.conv2d(input, kernel, stride=1, padding=1)
print(output_3)

```

卷积层的使用

- conv2d 的参数:
 - in_channels: 输入通道数
 - out_channels: 输出通道数 (也是卷积核的个数)
 - kernel_size: 卷积核大小, 通常为一个数或元组, 用元组定义不规则的
 - stride: 步长
 - padding: 在输入图像左右两边进行填充, 给定一个数或元组 (纵向, 横向), 空的地方默认为 0

- padding_mode: 空的地方填什么
- dilation:
- groups: 设置为 1
- bias: 设置为 true

```
import torch
import torchvision
from torch import nn
from torch.utils.data import DataLoader

## 这里是卷积层，功能较齐全，和前面的 functional 不一样

from torch.nn import Conv2d
from torch.utils.tensorboard import SummaryWriter

dataset = torchvision.datasets.CIFAR10(root='./dataset', train=False, transform=torchvision.transforms.ToTensor())
dataloader = DataLoader(dataset, batch_size=64)

class My(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = Conv2d(3,6,3,stride=1,padding=0)
    def forward(self,input):
        output = self.conv1(input)
        return output

my = My()
step = 0
writer = SummaryWriter('data1')
for data in dataloader:
    imgs,targets = data
    output = my(imgs)
    writer.add_images('input',imgs,step)
    output = torch.reshape(output,(-1,3,30,30))
    writer.add_images('output',output,step)
    step+=1

writer.close()
```

最大池化的使用（最大池化操作）

- Maxpool2d 主要参数：
 - kernel_size: 池化核 (窗口) 大小，可传入一个整数或元组
 - stride: 步长，默认值是 kernel_size 大小
 - padding: 在输入图像左右两边进行填充，给定一个数或元组 (纵向，横向)，空的地方默认为 0
 - dilation: 空洞卷积，数字匹配时会岔开一定数量格子，格子数量和 dilation 有关
 - ceil_mode: 设置为 true 会使用 ceil 模式 (向上取整，池化核平移至输入图像边界外会进行数的保留)，false 使用 mode 模式 (向下取整，池化核平移至输入图像边界外不会进行数的保留)
- 最大池化操作：取匹配到的数字中的最大值
- 作用：保持数据的特征，减少数据参数，大小，例如 1080p 转到 720p，就是对其进行池化操作

```
* import torch,torchvision
from torch import nn
```



```

from torch.nn import MaxPool2d
from torch.utils.data import DataLoader

from torch.utils.tensorboard import SummaryWriter

# 记得修改数据类型为 32 位浮点数

input = torch.tensor([
    [1,2,0,3,1],
    [0,1,2,3,1],
    [1,2,1,0,0],
    [5,2,3,1,1],
    [2,1,0,1,1]], dtype=torch.float32)

# 输入数据要求是四个参数

input = torch.reshape(input, (-1,1,5,5))

dataset = torchvision.datasets.CIFAR10(root='./dataset', train=False, transform=torchvision.transforms.ToTensor())
dataloader = DataLoader(dataset, batch_size=64)
writer = SummaryWriter('data2')

class My(nn.Module):
    def __init__(self):
        super().__init__()
        self.maxpool = MaxPool2d(3, ceil_mode=True)
    def forward(self, input):
        output = self.maxpool(input)
        return output

my = My()
step = 0
for data in dataloader:
    imgs, targets = data
    output = my(imgs)
    writer.add_images('input', imgs, step)
    writer.add_images('output', output, step)
    step+=1

writer.close()

```

非线性激活

- RELU: input 大于 0, output 等于 input, input 小于 0, output 等于 0
 - inplace 参数: 假设 input 为 -1, 为 true 的话 input 会被替换为 0, 为 false 的话 input 不会被替换, 仍为 -1
- Sigmoid: $y = 1/(1+\exp(x))$

```

* import torch
import torchvision
from torch import nn
from torch.utils.data import DataLoader
from torch.nn import ReLU, Sigmoid
from torch.utils.tensorboard import SummaryWriter
input = torch.tensor([1, -0.5],

```

```

        [-1,3]])
input = torch.reshape(input,(-1,1,2,2))
dataset = torchvision.datasets.CIFAR10('./dataset',train=False,transform=torchvision.transforms.ToTensor())
dataloader = DataLoader(dataset,batch_size=64)
class My(nn.Module):
    def __init__(self):
        super().__init__()
        self.relu = ReLU()
        self.sigmoid = Sigmoid()
    def forward(self, input):
        output = self.sigmoid(input)
        return output
writer = SummaryWriter('data3')
my = My()
step = 0
for data in dataloader:
    imgs,targets = data
    writer.add_images('input',imgs,step)
    output = my(imgs)
    writer.add_images('output',output,step)
    step+=1

writer.close()

```

神经网络-线性层及其它层介绍

- 线性层参数：
 - in_feature: 输入数据大小
 - out_feature: 输出数据大小
 - bias: 偏置
 - 输入维度要和前一层输出维度的最后一个维度相等，线性层只对前一层的**最后一个维度**做变换 ### Sequential(类似 transforms 的 compose)

```

import torch
from torch import nn
from torch.utils.tensorboard import SummaryWriter
from torch.nn import Conv2d,MaxPool2d,Flatten,Linear,Sequential

class My(nn.Module):
    def __init__(self):
        super().__init__()
        # self.conv1 = Conv2d(3,32,5,padding=2)
        # self.maxpool1 = MaxPool2d(2)
        # self.conv2= Conv2d(32,32,5,padding=2)
        # self.maxpool2 = MaxPool2d(2)
        # self.conv3 = Conv2d(32,64,5,padding=2)
        # self.maxpool3 = MaxPool2d(2)
        # self.flatten = Flatten()
        # self.linear1 = Linear(1020,64)
        # self.linear2 = Linear(64,10)
        self.model1 = Sequential(
            Conv2d(3, 32, 5, padding=2),
            MaxPool2d(2),
            Conv2d(32, 32, 5, padding=2),
            MaxPool2d(2),
            Conv2d(32, 64, 5, padding=2),

```

```

        MaxPool2d(2),
        Flatten(),
        Linear(1024, 64),
        Linear(64, 10)
    )

    def forward(self, input):
        input = self.model1(input)
        return input

my = My()
input = torch.ones((64,3,32,32))
output = my(input)
writer = SummaryWriter('data4')
writer.add_graph(my, input)
writer.close()

```

损失函数与反向传播

- 损失函数：计算实际输出与目标之间的差距
- 反向传播：为我们更新输出提供一定的依据 #### L1loss()
- 计算各个位置之间的差，将差累加并除以维度
- 参数：
 - input：可以是任意维度
 - output：大小要和输入相同
 - reduction：为 sum 表示相加，不除以维度

Mseloss() 平方差：先作差再平方 * 参数为 input 和 target

Crossentropyloss 交叉熵： $-x[class] + \log(\exp(x[j]))$ 求和 * 参数：要求 input 有 (N,C)，target 有 (N)

```

import torch
from torch.nn import L1Loss
inputs = torch.tensor([1,2,3], dtype=torch.float32)
targets = torch.tensor([1,2,5], dtype=torch.float32)
inputs = torch.reshape(inputs, (1,1,1,3))
targets = torch.reshape(targets, (1,1,1,3))
loss = L1Loss()
result = loss(inputs, targets)
print(result)

```

优化器

- 所在库：torch.optim
- 要放入模型参数，lr(学习速率)
- 工作流程：
 1. 输入经过模型得到输出
 2. 根据真实的 target 得到 loss
 3. 调用误差的反向传播得到每个参数对应的梯度
 4. 利用优化器进行优化
 5. 对梯度清零 ### VGG(分类模型) 模型参数：
- pretrained：为 true 说明模型已经训练好
- progress：为 true 会显示下载进度条

数据集参数：和前面的差不多