pytorch

发现上李沐的课还是有些障碍所以这周先入门了pytorch

两大法宝dir() help()

dir()函数，打开工具箱。用于列出一个对象的所有属性和方法。它返回一个包含对象所有属性和方法名称的列表。如果不传入参数，则返回当前作用域中所有可用的名称

help()函数，查看说明书。用于获取对象、模块、函数、关键字等的帮助信息。当传入对象时，它会显示该对象的帮助文档。如果没有传入任何参数，则会进入交互式帮助模式。

读取数据的两个类 Dataset和DataLoader

Dataset将数据和label进行组织编号0 1 2 3.....，使得可以根据编号读取数据；需获取每一个数据及其label以及数据总数 ， 要实现 len() 方法和 getitem() 方法。
len() 方法返回数据集的样本数量；
getitem() 方法根据给定的索引返回对应的数据样本；

DataLoader对数据进行打包将数据集划分为小批量，按batchsize送入网络模型；可以接收一个 Dataset 对象作为输入，并根据指定的批量大小、是否打乱数据、是否使用多线程等参数，来构建一个用于数据加载的迭代器

总结就是dataset是大数据集,dataloader是取出数据(batch批次)

```
from torch.utils.data import Dataset
from PIL import Image
import os

class MyData(Dataset):
    def __init__(self, root_dir, label_dir):     #根目录加标签目录
        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, idx):     #取出数据集
        img_name = self.img_path[idx]
        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)
```

os库

os库是 Python 的标准库之一，提供了与操作系统进行交互的功能。它可以用于文件和目录的操作、环境变量的管理、进程管理等

- `os.listdir(path)`：列出指定路径下的所有文件和目录。
- `os.mkdir(path)`：创建一个新目录。
- `os.remove(path)`：删除指定文件。
- `os.rmdir(path)`：删除指定空目录。
- `os.rename(src, dst)`：重命名文件或目录。
- `os.path.join(path, *paths)`：连接路径。
- `os.path.exists(path)`：检查路径是否存在。
- `os.path.isfile(path)`：检查路径是否为文件。
- `os.path.isdir(path)`：检查路径是否为目录。
- `os.system(command)`：运行系统命令。
- `os.getpid()`：获取当前进程的 ID。

tensorboard

TensorFlow 提供的一个可视化工具，用于监控和分析机器学习模型的训练过程。它可以帮助你更好地理解模型的性能和训练动态

绘制函数writer.add_scalar()

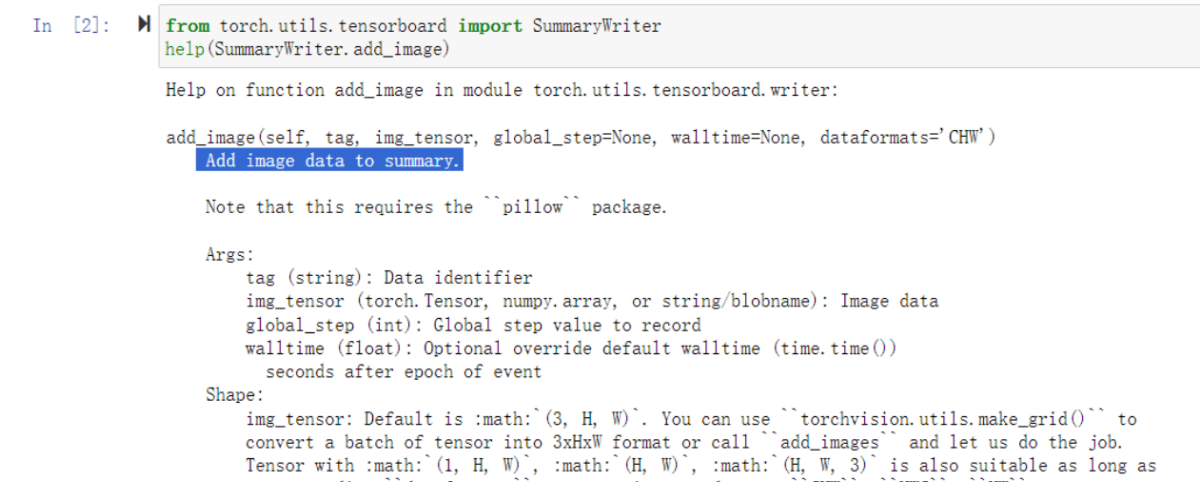
```
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter("logs")    #创建日志文件(一般在脚本目录下)
for i in range(100):
    writer.add_scalar("y = x", i, i)    #前一个i是y 后一个i是横坐标
writer.close()
```

打开日志文件:

```
tensorboard --logdir=logs  #logs是上面指定在writer = SummaryWriter("logs")中指定的文件夹名，日志文件存储在此文件中
```

绘制图片SummaryWriter.add_image()



```
In [2]: from torch.utils.tensorboard import SummaryWriter
        help(SummaryWriter.add_image)
```

```
Help on function add_image in module torch.utils.tensorboard.writer:

add_image(self, tag, img_tensor, global_step=None, walltime=None, dataformats='CHW')
    Add image data to summary.

    Note that this requires the ``pillow`` package.

    Args:
        tag (string): Data identifier
        img_tensor (torch.Tensor, numpy.array, or string/blobname): Image data
        global_step (int): Global step value to record
        walltime (float): Optional override default walltime (time.time())
          seconds after epoch of event
    Shape:
        img_tensor: Default is :math:`(3, H, W)`. You can use ``torchvision.utils.make_grid()`` to
        convert a batch of tensor into 3xHxW format or call ``add_images`` and let us do the job.
        Tensor with :math:`(1, H, W)`, :math:`(H, W)`, :math:`(H, W, 3)` is also suitable as long as
```

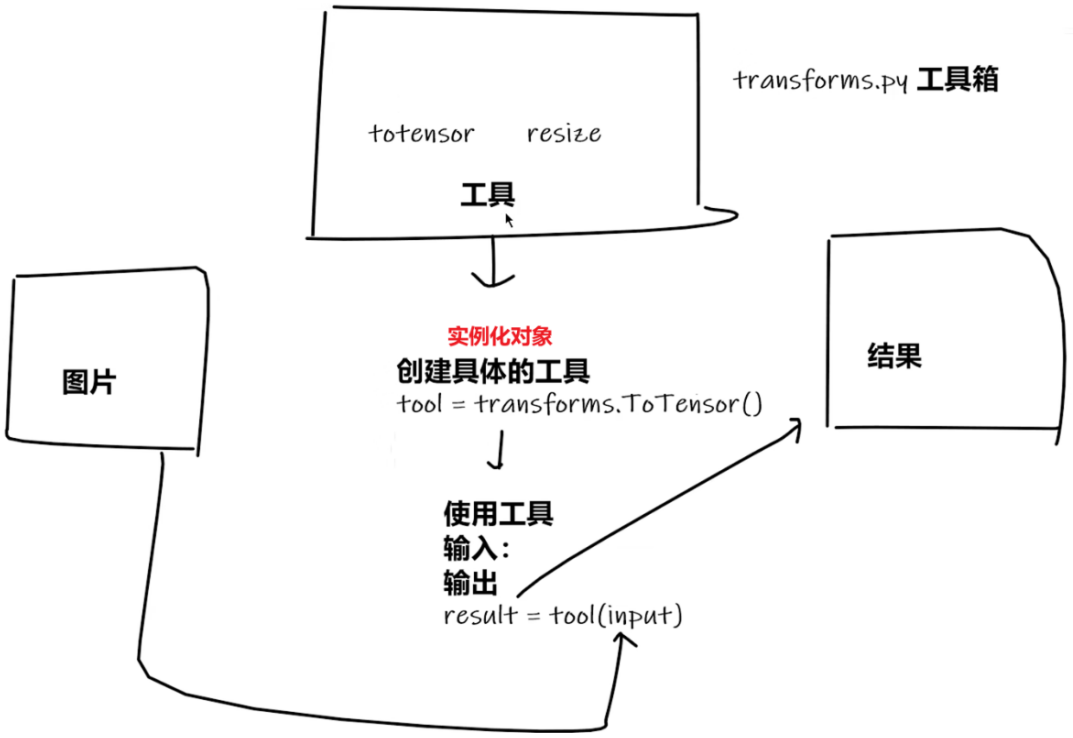
add_image的参数img_tensor类型需为torch.Tensor, **numpy.array**, or string/blobname

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

writer = SummaryWriter("logs") #日志文件存储位置
img_path = "  "         #绝对路径
img = Image.open(img_path)
print(type(img))
img_array = np.array(img)
print(type(img_array))
print(img_array.shape)
writer.add_image("Img Test", img_array, 1, dataformats="HWC")   #注意dataformats,
会改变通道数在格式中的位置
writer.close()
```

Transforms

用于对图像进行预处理和数据增强操作，如调整图像大小、中心裁剪、随机裁剪、随机水平翻转、归一化、将 PIL 图像转换为 Tensor 等等




```
from torchvision import transforms
from PIL import Image
img_path = "  "
img = Image.open(img_path)
print(type(img))
tensor_trans = transforms.ToTensor() # transforms.ToTensor() 这一步相当于实例化 返回totensor的对象
tensor_img = tensor_trans(img)
print(type(tensor_img))
```

输出发现格式改变

快捷键Alt + 7 可于Pycharm中查看包的结构

快捷键 ctrl+P 查看函数需传入的参数

*call*方法

在 Python 中，`__call__` 是一个特殊方法（也称为魔术方法或双下划线方法），用于使对象可以像函数一样被调用。当你在一个对象上调用 obj() 时，Python 解释器会查找该对象的 `__call__` 方法并调用它

隐式调用类中的函数

tensor数据类型

opencv 使用cv2.imread()读取的图像数据类型是numpy.ndarray
可设置tool = totensor() 创建工具改变数据类型
使用add_image()添加tensor类型的图像到日志文件中 通过tensorboard展示

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

writer = SummaryWriter("logs")
img_path = "  "
img = Image.open(img_path)
tensor_trans = transforms.ToTensor()
tensor_img = tensor_trans(img)
writer.add_image("Tensor_img",tensor_img)
writer.close()
```

常见的transforms

使用Normalize对图像进行**归一化**，对每个通道进行归一化：（输入值-均值）/ 标准差
假设三个通道的均值和标准差都为0.5(注意有些图片为四通道)

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

writer = SummaryWriter("logs") # 日志文件存储位置
img_path = "C:\\Users\\86138\\Pictures\\Screenshots\\屏幕截图 2023-12-17
105458.png"
img = Image.open(img_path)
tensor_trans = transforms.ToTensor()
tensor_img = tensor_trans(img)
trans_norm = transforms.Normalize([0.5,0.5,0.5,0.5],[0.5,0.5,0.5,0.5])
#假设某个像素的 RGB 值为 (0.8, 0.4, 0.2) 此处意为(0.8-0.5)/0.5
img_norm = trans_norm(tensor_img)
print(tensor_img[0][0][0])
print(img_norm[0][0][0])
writer.add_image("Normalize",img_norm)
writer.close()
```

transform.Resize()的使用

transform.Resize() 是 PyTorch 中 torchvision 库用于调整图像尺寸的变换方法。它可以通过指定目标大小（宽度和高度）来改变输入图像的尺寸

```
from PIL import Image
from torchvision import transforms
img_path = "  "
img = Image.open(img_path)
print("图片原始大小，读取为PIL类型",img.size)
trans_resize = transforms.Resize((512,512))
img_resize = trans_resize(img)
print("对PIL类型进行Resize",img_resize)
tensor_trans = transforms.ToTensor()
img_resize = tensor_trans(img_resize)
```

```
print("将PIL Reszie的图片转换为Tensor",img_resize)
trans_resize2 = transforms.Resize((256,256))
img_resize = trans_resize(img_resize)
print("对tensor类型的数据进行Resize",img_resize)
```

利用Compose进行resize

在 PyTorch 的 torchvision.transforms 模块中，transforms.Compose 是一个非常有用的工具，它可以将多个图像变换组合在一起，形成一个复合变换。这使得在数据预处理时，能够方便地应用一系列变换，而不需要逐个调用。
格式Compose([transforms参数1,transforms参数2,...]) //需要传入一个列表类型

```
from PIL import Image
from torchvision import transforms
img_path = "  "
img = Image.open(img_path)
print(img)
trans_resize2 = transforms.Resize(512)
trans_totensor = transforms.ToTensor()
trans_compose = transforms.Compose([trans_resize2,trans_totensor])
img_resize_2 = trans_compose(img)
print(img_resize_2)
```

随机裁剪RandomCrop

RandomCrop 是 PyTorch 中用于图像数据增强（data augmentation）的函数之一，它可以在图像或张量的随机位置裁剪出指定大小的区域
transforms.RandomCrop((128, 128))会随机在输入图像中裁剪出大小为 128x128 的区域，并返回裁剪后的图像对象

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

writer = SummaryWriter("logs")
img_path = "    "
img = Image.open(img_path)
print(img)
trans_random = transforms.RandomCrop(30)    #随机裁剪30*30
trans_totensor = transforms.ToTensor()
trans_compose_2 = transforms.Compose([trans_random,trans_totensor])
for i in range(10):
    img_crop = trans_compose_2(img)
    writer.add_image("RandomCrop",img_crop,i)

writer.close()
```

torchvision.datasets的使用

pytorch官网提供各种类型的工具,数据集

```
import torchvision
from torchvision import transforms
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter("logs") # 日志文件存储位置
dataset_transform = transforms.Compose([
    transforms.ToTensor()
])
train_set = torchvision.datasets.CIFAR10(root="./dataset",
train=True,transform=dataset_transform, download=True)
test_set = torchvision.datasets.CIFAR10(root="./dataset",
train=False,transform=dataset_transform, download=True)
print(test_set[0])
img, target = test_set[0]  # target对应类的编号 对应cat
print(img)
print(target)
print(test_set.classes[target])

for i in range(10):
    img, target = test_set[i]
    writer.add_image("torchvision",img,i)
```

也可用其他方式下载数据集后在pycharm相同模块新建文件夹导入数据集

DataLoader的使用

drop_last=True 表示如果最后一个批次的样本数量小于批次大小，则丢弃该批次；而 drop_last=False 则表示保留最后一个不完整的批次

```
import torchvision
from torchvision import transforms
from torch.utils.tensorboard import SummaryWriter
from torch.utils.data import DataLoader
dataset_transform = transforms.Compose([
    transforms.ToTensor()
])
test_set = torchvision.datasets.CIFAR10(root="文件路径",
train=False,transform=dataset_transform, download=True)
test_loader = DataLoader(dataset=test_set, batch_size=64, shuffle=True,
num_workers=0, drop_last=False)
#batch_size为读取数量 shuffle为是否每次都打乱 drop_last处理最后一批
img, target = test_set[0]
print("单个img:",img.shape)
print("单个target:",target)

writer = SummaryWriter("logs")
step = 0
for data in test_loader:   #每次取64张
    imgs,targets = data
```

```
    writer.add_images("test_data",imgs,step)
    step = step + 1

writer.close()
```

神经网络的基本骨架nn.Module的使用

常用包torch.nn
神经网络的基类Module，定义的模型都需要集成该类nn.Module
自己定义的模型需要实现**init**(实例化)和forward函数(更新)

简单例子:

```
import torch
from torch import nn
class Tudui(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, input):
        output = input +1
        return output
tudui = Tudui()
x = torch.tensor(1.0)
myout = tudui(x)
print(myout)
```

关于卷积神经网络原理

【深度学习】一文搞懂卷积神经网络（CNN）的原理（超详细） 卷积神经网络原理-CSDN博客

卷积需要注意哪些问题?
a.步长stride：每次滑动的位置步长。

b. 卷积核的个数：决定输出的depth厚度。同时代表卷积核的个数。

c. 填充值zero-padding：在外围边缘补充若干圈0，方便从初始位置以步长为单位可以刚好滑倒末尾位置，通俗地讲就是为了总长能被步长整除。

卷积神经网络的构造

1 输入层
输入层接收原始图像数据。图像通常由三个颜色通道（红、绿、蓝）组成，形成一个二维矩阵，表示像素的强度值。

2 卷积和激活
卷积层将输入图像与卷积核进行卷积操作。然后，通过应用激活函数（如ReLU）来引入非线性。这一步使网络能够学习复杂的特征。

3 池化层
池化层通过减小特征图的大小来减少计算复杂性。它通过选择池化窗口内的最大值或平均值来实现。这有助于提取最重要的特征。

4 多层堆叠
CNN通常由多个卷积和池化层的堆叠组成，以逐渐提取更高级别的特征。深层次的特征可以表示更复杂的模式。

5 全连接和输出
最后，全连接层将提取的特征映射转化为网络的最终输出。这可以是一个分类标签、回归值或其他任务的结果。

卷积层的使用

输入特征图的通道数 = 卷积核的通道数
输出特征图的通道数 = 卷积核的个数
解释：卷积核的通道数一定和输入的通道数相等，输入对应的每个通道与卷积核对应的每个通道进行计算再求和得到 一个通道的卷积输出；而输出特征图的通道数与卷积核的个数相关，有多少个卷积核最终就有多少个输出通道

卷积核（滤波器）的大小会影响输出特征图的尺寸。例如，如果使用一个 (3 \times 3) 的卷积核，卷积操作会在图像的每个像素周围提取信息，导致输出图像的尺寸比输入图像小。要想维持原状则需要进行填充padding

dilation是空洞卷积(步长)，默认值是1

```
import torch
import torchvision
from torch.utils.data import DataLoader
from torch import nn
from torch.nn import Conv2d
from torch.utils.tensorboard import SummaryWriter

dataset = torchvision.datasets.CIFAR10("./dataset",train= False, transform
=torchvision.transforms.ToTensor(), download=True)
dataloader = DataLoader(dataset, batch_size=64)

#搭建网络
class Tudui(nn.Module):
    def __init__(self):
        super(Tudui, self).__init__()
        self.conv1 = Conv2d(in_channels=3, out_channels=6, kernel_size=3,
stride=1, padding=0)

    def forward(self, x):
        x = self.conv1(x)
        return x

writer = SummaryWriter("logs")
tudui = Tudui()
print(tudui)

step = 0
for data in dataloader:
```

```
    imgs, targets = data
    output = tudui(imgs)
    print("原图像的形状",imgs.shape)
    print("卷积之后图像的形状",output.shape)
    writer.add_images("input",imgs,step)
    # 卷积之后图像的形状 torch.Size([64, 6, 30, 30])是6个通道的 而add_images只能接收3通道的输入
    output = torch.reshape(output, (-1, 3, 30, 30))#不严谨操作 ---对output进行reshape 增大batchsize的数量 减少通道数 -1意为让程序自己拉伸运算
    writer.add_images("Conv_output",output,step)
    step = step + 1
writer.close()
```

关于卷积后大小:

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

池化

池化是卷积神经网络（CNN）中的一种常见操作，主要用于降低特征图的空间尺寸，减少参数数量和计算量，同时保留重要的特征信息。

最大池化层保留输入的特征，同时减少数据量 加快训练速度

池化层默认步长为核大小，卷积层默认步长为1

Parameters

- **kernel_size** (*Union[int, Tuple[int, int]]*) – the size of the window to take a max over
- **stride** (*Union[int, Tuple[int, int]]*) – the stride of the window Default value is `kernel_size`
- **padding** (*Union[int, Tuple[int, int]]*) – Implicit negative infinity padding to be added on both sides
- **dilation** (*Union[int, Tuple[int, int]]*) – a parameter that controls the stride of elements in the window
- **return_indices** (*bool*) – if `True`, will return the max indices along with the outputs. Useful for `torch.nn.MaxUnpool2d` later
- **ceil_mode** (*bool*) – when True, will use *ceil* instead of *floor* to compute the output shape

ceil_mode ceil向上取整，floor向下取整

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

```
from torch.nn import MaxPool2d
dataset = torchvision.datasets.CIFAR10("./dataset",train= False, transform
=torchvision.transforms.ToTensor(), download=True)
dataloader = DataLoader(dataset, batch_size=64)

#搭建网络
class Tudui(nn.Module):
    def __init__(self):
        super(Tudui,self).__init__()
        self.maxpool1 = MaxPool2d(kernel_size=3,ceil_mode=False)

    def forward(self,input):
        output = self.maxpool1(input)
        return output


writer = SummaryWriter("logs")
tudui = Tudui()
print(tudui)

step = 0
for data in dataloader:
    imgs, targets = data
    output = tudui(imgs)
    print("原图像的形状",imgs.shape)
    print("池化之后图像的形状",output.shape)
    writer.add_images("maxpool_input",imgs,step)
    writer.add_images("maxpool_output",output,step)
    step = step + 1
writer.close()
```

非线性激活层

引入非线性的特性，使得神经网络具有更强的表达能力和适应能力

一般为sigmoid和relu

举一个sigmoid的简单例子

```
import torch
import torchvision
from torch.utils.data import DataLoader
from torch import nn
from torch.utils.tensorboard import SummaryWriter
from torch.nn import Sigmoid
dataset = torchvision.datasets.CIFAR10("./dataset",train= False, transform
=torchvision.transforms.ToTensor(), download=True)
dataloader = DataLoader(dataset, batch_size=64)

class Tudui(nn.Module):
    def __init__(self):
        super(Tudui,self).__init__()
        self.sigmoid1 = Sigmoid()
```

```
    def forward(self,input):
        output = self.sigmoid1(input)
        return output

writer = SummaryWriter("logs")
tudui = Tudui()
print(tudui)

step = 0
for data in dataloader:
    imgs, targets = data
    output = tudui(imgs)
    writer.add_images("before_activate",imgs,step)
    writer.add_images("after_activate",output,step)
    step = step + 1
writer.close()
```

得到的结果是变灰的图片,因为像素都处于0-1之间,对比度过小了

线性层（全连接层）

```
import torch
import torchvision
from torch.utils.data import DataLoader
from torch import nn
from torch import flatten
from torch.utils.tensorboard import SummaryWriter
from torch.nn import Linear
dataset = torchvision.datasets.CIFAR10("./dataset",train= False, transform
=torchvision.transforms.ToTensor(), download=True)
dataloader = DataLoader(dataset, batch_size=64, drop_last=True)

#搭建网络
class Tudui(nn.Module):
    def __init__(self):
        super(Tudui,self).__init__()
        self.linear1 = Linear(196608,10)  #接收一个包含 196608 个特征的输入，并将其映射
到一个包含 10 个特征的输出。

    def forward(self,input):
        output = self.linear1(input)
        return output

tudui = Tudui()
print(tudui)

for data in dataloader:
    imgs, targets = data
    print("原尺寸", imgs.shape) #【64,3,32,32】
    output = flatten(imgs)    #flatten函数展平img为一维
    print("flatten后尺寸",output.shape)
    output = tudui(output)
    print("全连接层输出尺寸",output.shape)
```

网络搭建以及Sequential的使用

顺序容器，可以按照添加的顺序依次执行包含的各个模块，torch.nn.Sequential提供了一种简单的方式来构建神经网络模型，代码十分简洁。

使用Sequential并使用tensorboard添加计算图 相当于封装了

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

#使用sequential
class Tudui(nn.Module):
    def __init__(self):
        super(Tudui,self).__init__()
        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, x):
        x = self.model1(x)
        return x

writer = SummaryWriter("logs") # 日志文件存储位置
writer.add_graph(tudui, input)
tudui = Tudui()
print(tudui)

input = torch.ones((64, 3, 32, 32))
output = tudui(input)
print(output.shape)
writer.close()
```

torch.nn中的损失函数

损失函数用于**衡量模型的预测输出与实际标签之间的差异或者误差**，损失越小越好，**根据loss调整参数（反向传播）**更新输出，减小损失

简单损失函数:L1loss MSEloss

交叉熵损失函数 是在分类问题中经常使用的一种损失函数，特别是在多分类问题中。它衡量了模型输出的概率分布与真实标签之间的差异，通过最小化交叉熵损失来调整模型参数，使得模型更好地适应分类任务。

```
loss = nn.CrossEntropyLoss()
tudui = Tudui()
for data in dataloader:
    imgs, targets = data
    outputs = tudui(imgs)
#     print(outputs)
#     print(targets)
    result_loss = loss(outputs, targets)     #实际输出与期望输出
    print(result_loss)
```

反向传播

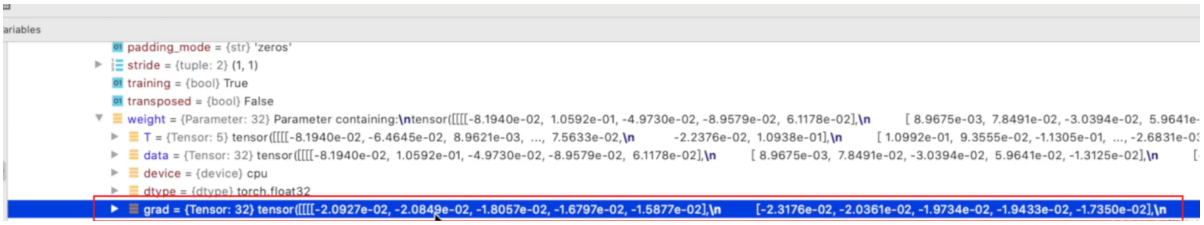
反向传播（Backpropagation）是一种用于训练神经网络的算法，主要用于计算损失函数相对于网络中每个权重的梯度。它是基于链式法则，通过以下步骤实现：

1. **前向传播**：输入数据通过网络进行处理，得到预测输出，并计算损失函数（例如，均方误差或交叉熵）。
2. **计算梯度**：从输出层开始，逐层向后计算损失函数相对于每一层权重的梯度。这通过链式法则实现，将损失对输出的梯度逐层传递回去，直到输入层。
3. **更新权重**：使用计算得到的梯度，结合优化算法（如梯度下降），更新网络的权重和偏置，以最小化损失函数。

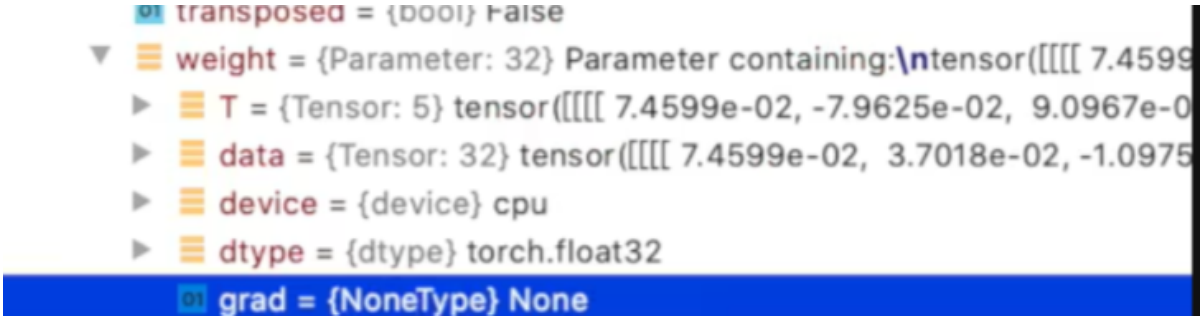
张量梯度的属性：grad
自动求导（Autograd）： 在张量上进行操作时，PyTorch 会自动跟踪操作并构建计算图，可以**使用 .backward() 方法（反向传播）计算梯度**，然后通过 .grad 属性获取梯度值。
在pycharm中对该行打断点，可以看到具体的梯度值

```
loss = nn.CrossEntropyLoss()
tudui = Tudui()
for data in dataloader:
    imgs, targets = data
    outputs = tudui(imgs)
    result_loss = loss(outputs, targets)
    result_loss.backward()
    print(result_loss)
```

当未使用.backward()时，grad属性会一直为None



打断点后:



这里还未使用优化器，结合优化器可以对参数进行优化，降低loss

优化器的使用 torch.optim

在深度学习中，optimizer.zero_grad()是一个非常重要的操作，它的含义是将模型参数的梯度清零。在训练神经网络时，通常采用反向传播算法（Backpropagation）来计算损失函数关于模型参数的梯度，并利用优化器（optimizer）来更新模型参数以最小化损失函数。在每次反向传播计算梯度后，梯度信息会被累积在对应的参数张量（tensor）中。如果不清零梯度，在下一次计算梯度时，这些梯度将会被新计算的梯度累加，导致梯度信息错误。
optim.step()是优化器对象（如SGD、Adam等）的一个方法，用于根据计算得到的梯度更新模型的参数

再引入epoch多次批量训练

```
from torch import nn
from torch.nn import Module
from torch.nn import Conv2d
from torch.nn import MaxPool2d,Flatten,Linear, Sequential
import torch
import torchvision
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter

dataset = torchvision.datasets.CIFAR10("./dataset",train= False, transform
=torchvision.transforms.ToTensor(), download=True)
dataloader = DataLoader(dataset, batch_size=1, drop_last=True)

#使用sequential
class Tudui(nn.Module):
    def __init__(self):
        super(Tudui,self).__init__()
        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, x):
        x = self.model1(x)
        return x
```

```
loss = nn.CrossEntropyLoss()
tudui = Tudui()
optim = torch.optim.SGD(tudui.parameters(), lr=0.01, )
for epoch in range(20):
    running_loss = 0.0
    for data in dataloader:
        imgs, targets = data
        outputs = tudui(imgs)
        result_loss = loss(outputs, targets)
        optim.zero_grad()
        result_loss.backward()
        optim.step()
        running_loss = running_loss +result_loss # running_loss相当于扫一遍全部数据的
loss总和
    print(running_loss)
```

使用网络模型VGG16

可通过以下命令`torch.hub.set_dir('新路径')`来设置自定义的缓存文件夹路径用于缓存数据集

```
import torchvision.models as models
# 设置自定义缓存文件夹路径
torch.hub.set_dir('/path/to/custom/cache/dir/')
# 加载预训练的VGG16模型
vgg16 = models.vgg16(pretrained=True)
```

两种方法修改加载的网络模型

1将添加的线性层加在classifier中

```
import torchvision
import torch.nn as nn
vgg16_true = torchvision.models.vgg16(pretrained=True) #pretrained=True加载网络模
型，并从网络中下载在数据集上训练好的参数
dataset = torchvision.datasets.CIFAR10("./dataset",train= False, transform
=torchvision.transforms.ToTensor(), download=True)
vgg16_true.classifier.add_module('add_linear', nn.Linear(1000,10))  #命名为
add_linear添加到classifier
print(vgg16_true)
```

```
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
    (add_linear): Linear(in_features=1000, out_features=10, bias=True)
  )
)
```

2不添加层，仅修改vgg原有网络

```
import torchvision
import torch.nn as nn
vgg16_true = torchvision.models.vgg16(pretrained=True) #pretrained=True加载网络模型，并从网络中下载在数据集上训练好的参数
print("修改前",vgg16_true)
dataset = torchvision.datasets.CIFAR10("./dataset",train= False, transform=torchvision.transforms.ToTensor(), download=True)
vgg16_true.classifier[6] = nn.Linear(4096,10)   #直接修改classifier
print("修改后",vgg16_true)
```

```
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
```

修改前


```
(classifier): Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=10, bias=True)
)
```

修改后

网络模型的保存与读取

两种保存方式：

①`torch.save(vgg16,"vgg16_method1.pth")` 保存模型结构及模型参数

②`torch.save(vgg16.state_dict(),"vgg16_method2.pth")` 仅保存模型参数存为字典，不保存模型结构（官方推荐）(string to tensor)

```
import torchvision
import torch
vgg16 = torchvision.models.vgg16(pretrained=False)
# 保存方式1--保存模型结构及模型参数
torch.save(vgg16,"vgg16_method1.pth")
# 保存方式2--仅保存模型参数存为字典，不保存模型结构（官方推荐）
torch.save(vgg16.state_dict(),"vgg16_method2.pth")
```

两种读取方式：

①`model = torch.load("vgg16_method1.pth")` 加载模型结构 + 参数方式

②`vgg16 = torchvision.models.vgg16(pretrained=False)`

`vgg16.load_state_dict(torch.load("vgg16_method2.pth"))`

为方式2创建模型结构并加载参数的完整写法

```
import torch
import torchvision
# 保存方式1对应的加载模型结构 + 参数方式
model = torch.load("vgg16_method1.pth")
print(model)
# 保存方式2对应的加载模型参数方式
model2 = torch.load("vgg16_method2.pth") #加载的是字典
print(model2)
vgg16 = torchvision.models.vgg16(pretrained=False) #为方式2创建模型结构并加载参数的完整写法
vgg16.load_stat
```

完整训练简单套路

```
from torch.utils.tensorboard import SummaryWriter
from model import *
import torchvision
import torch.nn as nn
from torch.utils.data import DataLoader
# 准备数据集
train_data = torchvision.datasets.CIFAR10("../dataset",train=True,
transform=torchvision.transforms.ToTensor(),
                                          download=True)
test_data = torchvision.datasets.CIFAR10("../dataset",train=False,
transform=torchvision.transforms.ToTensor(),
                                          download=True)
# len()获取数据集长度
```

```
train_data_size = len(train_data)
test_data_size = len(test_data)
print("训练数据集的长度为：{}".format(train_data_size))
print("测试数据集的长度为：{}".format(test_data_size))
# 利用dataloader加载数据集
train_dataloader = DataLoader(train_data, batch_size=64, drop_last=True)
test_dataloader = DataLoader(test_data, batch_size=64, drop_last=True)
#创建网络模型
tudui = Tudui()
# 损失函数
loss_fn = nn.CrossEntropyLoss()
# 优化器
learning_rate = 1e-2
optimizer = torch.optim.SGD(tudui.parameters(), lr=learning_rate)
# 设置训练网络的一些参数
# 记录训练的次数
total_train_step = 0
# 记录测试的次数
total_test_step = 0
# 训练的轮数
epoch = 10
# 添加tensorboard
writer = SummaryWriter("../logs_train")

for i in range(epoch):
    print("-------------第 {} 轮训练开始------------".format(i+1))
    #训练步骤开始
    tudui.train()
    for data in train_dataloader:
        imgs, targets = data
        output = tudui(imgs)
        loss = loss_fn(output, targets)

        #优化器优化模型
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_train_step = total_train_step + 1
        if total_train_step % 100 ==0:
            print("训练次数：{}， Loss：{}".format(total_train_step, loss.item()))
            writer.add_scalar("train_loss",loss.item(), total_train_step)

    # 测试步骤开始
    tudui.eval()
    total_test_loss = 0
    total_accuracy = 0
    with torch.no_grad():
        for data in test_dataloader:
            imgs, targets = data
            outputs = tudui(imgs)
            loss = loss_fn(outputs, targets)
            total_test_loss = total_test_loss + loss.item()  #item()将张量转为常数
            accuracy = (outputs.argmax(1) == targets).sum()  #argmax 是一个数学和编
程中常用的术语，它表示找到一个函数或数组中最大值的索引或位置。在 PyTorch 中，torch.argmax 是
一个函数，用于返回输入张量（Tensor）中最大值的索引。
            total_accuracy = total_accuracy + accuracy
```

```
    print("整体测试集上的Loss：{}".format(total_test_loss))
    print("整体测试集上的正确率：{}".format(total_accuracy/test_data_size))
    writer.add_scalar("test_loss",total_test_loss, total_test_step)
    writer.add_scalar("test_accuracy", total_accuracy/test_data_size,
total_test_step)
    total_test_step = total_test_step + 1

    torch.save(tudui, "tudui_{}.pth".format(i))
    print("模型已保存")
writer.close()
```

model.train()和model.eval()

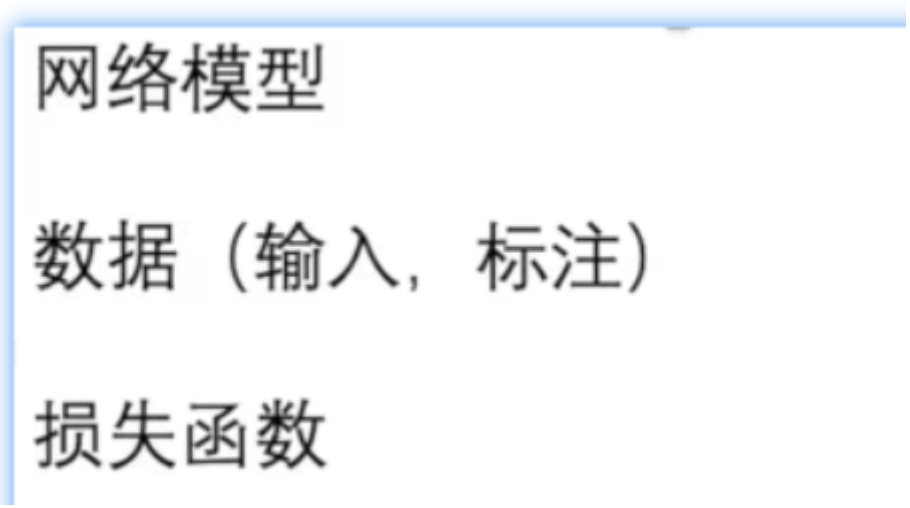
model.train()开启训练模式，模型会跟踪所有层的梯度，以便在优化器（如 torch.optim.SGD 或 torch.optim.Adam）进行梯度下降时更新模型的权重。此外，train() 方法还会将模型中的某些层（如 BatchNorm 和 Dropout）设置为训练行为。

BatchNorm 层：对于包含 BatchNorm（批量归一化）层的模型，model.train() 确保在训练过程中使用每一批数据来计算层的运行均值和方差。这些运行统计量用于归一化网络的激活值，有助于提高训练的稳定性和性能。

Dropout 层：对于包含 Dropout 层的模型，model.train() 在训练过程中随机选择一部分网络连接进行训练，即“丢弃”一部分神经元的输出。这样做可以防止网络过拟合，因为每次训练时只有一部分神经元被激活，从而增加了模型的泛化能力。

model.eval()：开启评估模式，在评估模式下，模型不会跟踪梯度，这有助于减少内存消耗并提高计算效率。此外，eval() 方法还会将模型中的某些层（如 BatchNorm 和 Dropout）设置为评估行为，这意味着它们的行为会根据固定的参数进行调整，而不是根据训练数据。
在评估模式（model.eval()）下，BatchNorm 层会使用在训练过程中学习到的均值和方差，而不是使用当前批次的数据。
在评估模式下，Dropout 层会被禁用，所有的神经元都会保留其输出，确保评估时的确定性。

GPU训练



1代码后加入.cuda()即可

2 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")　再在代码后加入.to(device)

前面讲的两种在GPU训练方法，其实只有数据和标签(imgs和targets)需要进行 数据 = 数据.cuda() 或者 数据 = 数据.to(device)
模型和损失函数可以直接model.to() ,model.cuda() ,loss.to(),loss.cuda()而无需赋值

最后总结一版

```
from torch.utils.tensorboard import SummaryWriter

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

# 定义训练的设备
device =  torch.device("cuda")
# 准备数据集
train_data = torchvision.datasets.CIFAR10("./dataset",train=True,
transform=torchvision.transforms.ToTensor(),
                                          download=True)
test_data = torchvision.datasets.CIFAR10("./dataset",train=False,
transform=torchvision.transforms.ToTensor(),
                                          download=True)
# len()获取数据集长度
train_data_size = len(train_data)
test_data_size = len(test_data)
print("训练数据集的长度为：{}".format(train_data_size))
print("测试数据集的长度为：{}".format(test_data_size))

# 利用dataloader加载数据集
train_dataloader = DataLoader(train_data, batch_size=64, drop_last=True)
test_dataloader = DataLoader(test_data, batch_size=64, drop_last=True)

#创建网络模型
class Tudui(nn.Module):
    def __init__(self):
        super(Tudui, self).__init__()
        self.model = nn.Sequential(
            nn.Conv2d(3, 32, 5, 1, 2),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 32, 5, 1, 2),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 5, 1, 2),
            nn.MaxPool2d(2),
            nn.Flatten(),  # 展平后的序列长度为 64*4*4=1024
            nn.Linear(1024, 64),
            nn.Linear(64, 10)

        )

    def forward(self, x):
        x = self.model(x)
        return x

tudui = Tudui()
```

```
tudui = tudui.to(device)
# 损失函数
loss_fn = nn.CrossEntropyLoss()
loss_fn = loss_fn.to(device)
# 优化器
learning_rate = 1e-2
optimizer = torch.optim.SGD(tudui.parameters(), lr=learning_rate)
# 设置训练网络的一些参数
# 记录训练的次数
total_train_step = 0
# 记录测试的次数
total_test_step = 0
# 训练的轮数
epoch = 10

# 添加tensorboard
writer = SummaryWriter("../logs_train")

for i in range(epoch):
    print("-------------第 {} 轮训练开始------------".format(i+1))
    #训练步骤开始
    tudui.train()
    for data in train_dataloader:
        imgs, targets = data

        imgs = imgs.to(device)
        targets = targets.to(device)
        output = tudui(imgs)
        loss = loss_fn(output, targets)

        #优化器优化模型
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_train_step = total_train_step + 1
        if total_train_step % 100 ==0:
            print("训练次数：{}， Loss：{}".format(total_train_step, loss.item()))
            writer.add_scalar("train_loss",loss.item(), total_train_step)

    # 测试步骤开始
    tudui.eval()
    total_test_loss = 0
    total_accuracy = 0
    with torch.no_grad():
        for data in test_dataloader:
            imgs, targets = data
            imgs = imgs.to(device)
            targets = targets.to(device)
            outputs = tudui(imgs)
            loss = loss_fn(outputs, targets)
            total_test_loss = total_test_loss + loss.item()
            accuracy = (outputs.argmax(1) == targets).sum()
            total_accuracy = total_accuracy + accuracy
    print("整体测试集上的Loss：{}".format(total_test_loss))
    print("整体测试集上的正确率：{}".format(total_accuracy/test_data_size))
    writer.add_scalar("test_loss",total_test_loss, total_test_step)
```

```
    writer.add_scalar("test_accuracy", total_accuracy/test_data_size,
total_test_step)
    total_test_step = total_test_step + 1

    torch.save(tudui, "tudui_{}.pth".format(i))
    print("模型已保存")
writer.close()
```

使用gpu训练保存的模型在cpu上使用

`model = torch.load("XXXX.pth",map_location= torch.device("cpu"))`

map_location=torch.device("cpu") 是在使用 PyTorch 的 torch.load 函数加载模型或张量时的一个参数，它用于指定加载数据的目标设备。当你使用这个参数时，你告诉 PyTorch 将加载的数据映射到 CPU 上，而不是默认的 CUDA 设备（如果你的系统上有 GPU）