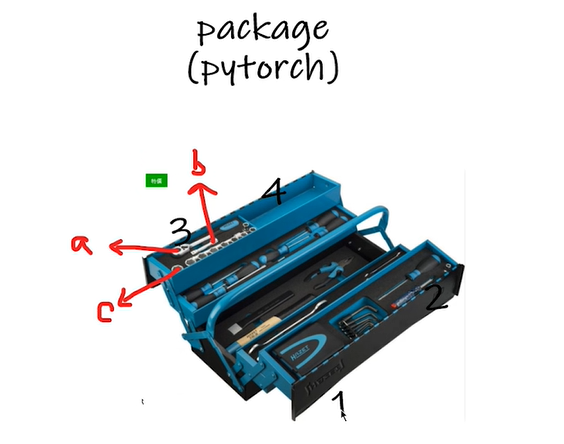
# 1.简介

Package相当于一个工具箱，这个工具箱名为pytorch。假设这个工具箱有四个分区，3号分区又有3个不同的工具



探索工具箱可以有两个工具

dir() 打开，可以让我们知道里面有什么工具

help() 帮助，说明书

例子：

输入：dir(pytorch)

输出：1,2,3,4

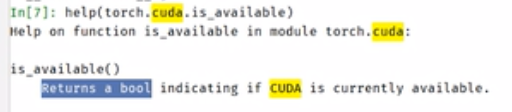
输入：dir(pytorch.3)

输出：a,b,c

输入：help(pytorch.3.a)

输出：将此扳手放在特定地方，然后拧动





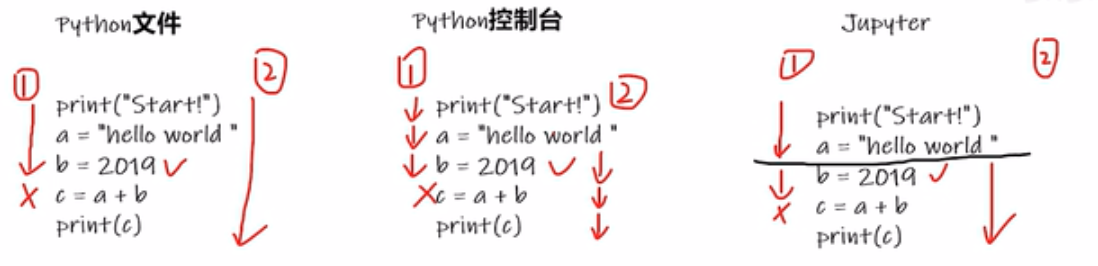
2.不同的编辑方式——python文件，python控制台，jupyter

代码是以块为一个整体运行的话，

python文件： python文件的块是所有行的代码——通用，传播方便，适用于大型项目；但是需要从头运行

python控制台：以每一行（也可以以任意行）为块——显示每个变量属性；不利于代码阅读和修改

jupyter：以任意行为块——利于代码阅读和修改；环境需要配置



# 3.pytorch数据加载初步

dataset提供一种方式去获取数据及其label：如何获取每一个数据及其label，要显示总共有多少数据

dataloader为后面的网络提供不同的数据形式

P7read\_data.py:

from torch.utils.data import Dataset  
from PIL import Image  
import os  
class MyData(Dataset):#\_\_xx\_\_类型的函数是python的类中必有的函数，相当于java中的构造和析构函数  
 def \_\_init\_\_(self,root\_dir,label\_dir):#java里的构造函数  
 self.root\_dir=root\_dir  
 self.label\_dir=label\_dir  
 self.path=os.path.join(self.root\_dir,self.label\_dir)#join的功能是将两个参数的str拼接起来  
 self.img\_path=os.listdir(self.path)#将self.path中的文件名称整合成一个列表list  
 def \_\_getitem\_\_(self, idx):#类数据中单个变量的返回类型，例如实例化mydata=MyData(Dataset),使用mydata[idx]则调用此函数  
 img\_name=self.img\_path[idx]#获取list中的某一个图片的文件名  
 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="D:\\deep\_learning\_dataset\\hymenoptera\_data\\train"  
ants\_label\_dir="ants"  
bees\_label\_dir="bees"  
ants\_dataset=MyData(root\_dir,ants\_label\_dir)  
bees\_dataset=MyData(root\_dir,bees\_label\_dir)  
train\_dataset=ants\_dataset+bees\_dataset#训练集

# 4.tensorboard的使用

transform的使用：对图像进行一种变换（图像统一到同一个尺寸，图像要进行某个类的转换）

tensorboard可以将图像变换结果展示出来

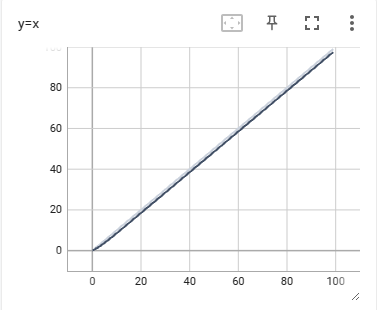
1. add\_scalar(self,tag,scaler\_value,global\_step)——tag是图标的标题，scaler value是纵坐标，global step是横坐标

P8test\_tb.py:

from torch.utils.tensorboard import SummaryWriter  
writer=SummaryWriter("logs")#命名存储tensorboard的日志文件夹名称，会在当前项目文件夹下生成一个名为logs的日志文件夹  
#y=x  
for i in range(100):  
 writer.add\_scalar("y=x",i,i)  
writer.close()#本句一定要有，不然会报错

然后在终端输入指令：tensorboard --logdir=logs #tensorboard –logdir=tensorboard文件所在的日志文件名，这里是设置了名字为logs

或者可以指定一下tensorboard的端口：tensorboard --logdir=logs –ports=6007



（2）add\_image(self,tag,img\_tensor,global\_step)

Img\_tensor是图像的数据类型，torch.tensor，numpy.array，string

注意，此函数对图片的shape也有特定的要求：默认是（3，H，W）——3是指3通道，即图片为rgb三原色，H是高度，W是宽度。如果图片的shape不是这个默认，则需要通过dataformats参数进行修改

实例里是，是shape是（H，W，3）

writer.add\_image("test",img\_array,1)

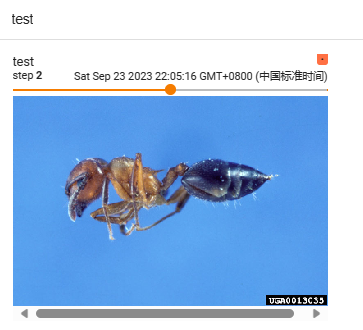
改为

writer.add\_image("test",img\_array,1,dataformats='HWC')

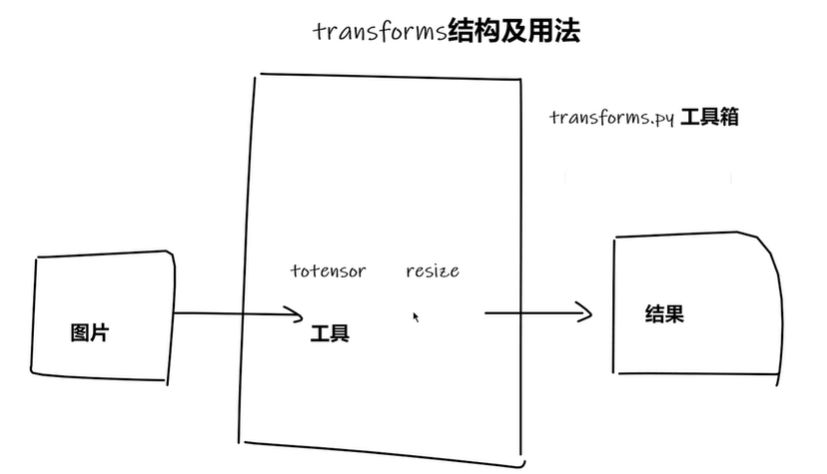
从PIL到numpy需要在add\_image中指定shape每一个数字/维表示的含义

运行P9writer.py：

import numpy as np  
from PIL import Image  
from torch.utils.tensorboard import SummaryWriter  
writer=SummaryWriter("logs")  
image\_path="D:\\deep\_learning\_dataset\\hymenoptera\_data\\train\\ants\\5650366\_e22b7e1065.jpg"  
img\_PIL=Image.open(image\_path)  
img\_array=np.array(img\_PIL)#将PIL图片类型转换为numpy数据类型，这样就可以在add\_image中使用了  
writer.add\_image("test",img\_array,1,dataformats='HWC')  
writer.add\_image("test",img\_array,2,dataformats='HWC')  
writer.add\_image("test",img\_array,6,dataformats='HWC')  
#y=x  
for i in range(100):  
 writer.add\_scalar("y=x",i,i)  
writer.close()

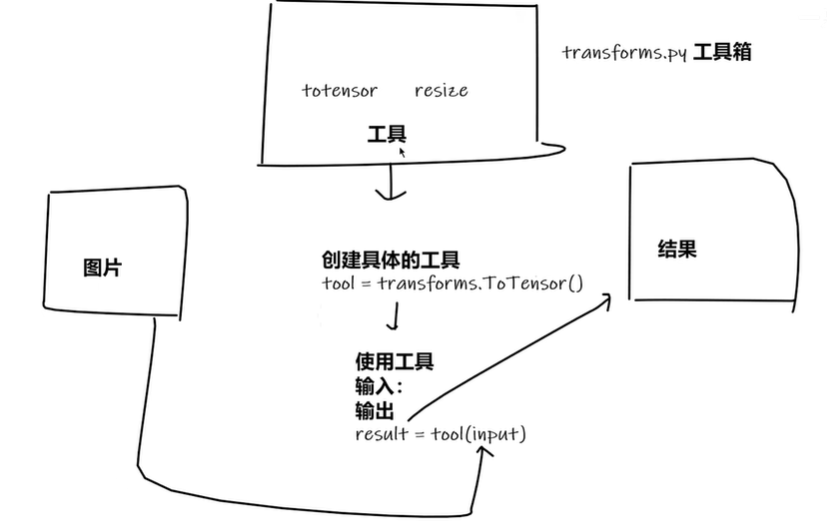


5.torchvision中的transforms



P10transforms.py:

from PIL import Image  
from torchvision import transforms  
#python的用法->tensor数据类型  
#可以将数据类型PIL Image或numpy.ndarray转换为tensor数据类型  
#通过transform.ToTensor去看两个问题：  
# 1.transform该如何使用（python）  
# 2.为什么我们需要Tensor数据类型  
img\_path="D:\\deep\_learning\_dataset\\hymenoptera\_data\\train\\ants\\0013035.jpg"  
img=Image.open(img\_path)  
# 1.transform该如何使用（python）  
tensor\_trans=transforms.ToTensor()#将transforms中的ToTensor类实例化  
tensor\_img=tensor\_trans(img)#将PIL的image类型转换为ToTensor类型  
print(tensor\_img)



P11transform.py:

from PIL import Image  
from torch.utils.tensorboard import SummaryWriter  
from torchvision import transforms  
#python的用法->tensor数据类型  
#可以将数据类型PIL Image或numpy.ndarray转换为tensor数据类型  
#通过transform.ToTensor去看两个问题：  
# 1.transform该如何使用（python）  
# 2.为什么我们需要Tensor数据类型  
img\_path="D:\\deep\_learning\_dataset\\hymenoptera\_data\\train\\ants\\0013035.jpg"  
img=Image.open(img\_path)  
  
writer=SummaryWriter("logs")  
# 1.transform该如何使用（python）  
tensor\_trans=transforms.ToTensor()#将transforms中的ToTensor类实例化  
tensor\_img=tensor\_trans(img)  
# 2.为什么我们需要Tensor数据类型 可以直接用在add\_image中  
writer.add\_image("Tensor\_img",tensor\_img)  
writer.close()

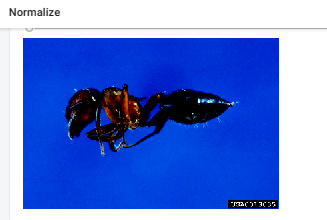
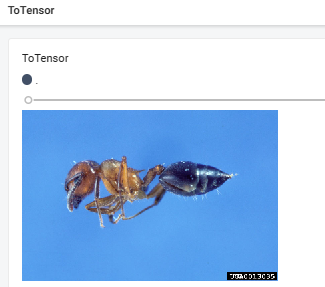
# 5.常见的transforms方法

首先\_\_call\_\_的用法：相当于c++中的重载

class Person:  
 def \_\_call\_\_(self,name):  
 print("\_\_call\_\_"+"Hello"+name)  
 def hello(self,name):  
 print("\_\_call\_\_"+name)  
person=Person()  
person("zhangsan")  
person.hello("lisi")

Totensor使用transforms的Normalize

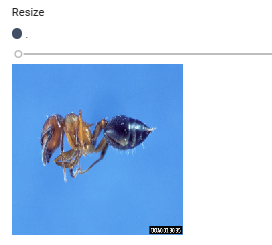
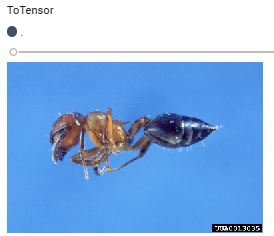
from PIL import Image  
from torch.utils.tensorboard import SummaryWriter  
from torchvision import transforms  
  
writer=SummaryWriter("logs")  
img=Image.open("D:\\deep\_learning\_dataset\\hymenoptera\_data\\train\\ants\\0013035.jpg")  
#ToTensor 这是一个类，下面将PIL的Image类转换为ToTensor类并将其用tensorboard显示出来  
trans\_totensor=transforms.ToTensor()  
img\_tensor=trans\_totensor(img)  
writer.add\_image("ToTensor",img\_tensor)  
#Normalize归一化 output[channel] = (input[channel] - mean[channel]) / std[channel]  
print(img\_tensor[0][0][0])  
trans\_norm=transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])#三个均值和三个标准差，范围自设（0，1）  
img\_norm=trans\_norm(img\_tensor)#归一化output[channel] = (input[channel] - mean[channel]) / std[channel]，则img\_norm=2\*img\_totensor-1  
print(img\_norm[0][0][0])  
writer.add\_image("Normalize",img\_norm)  
  
  
writer.close()



Transform中的Resize

from PIL import Image  
from torch.utils.tensorboard import SummaryWriter  
from torchvision import transforms  
  
writer=SummaryWriter("logs")  
img=Image.open("D:\\deep\_learning\_dataset\\hymenoptera\_data\\train\\ants\\0013035.jpg")  
#ToTensor 这是一个类，下面将PIL的Image类转换为ToTensor类并将其用tensorboard显示出来  
trans\_totensor=transforms.ToTensor()  
img\_tensor=trans\_totensor(img)  
writer.add\_image("ToTensor",img\_tensor)  
#Normalize归一化 output[channel] = (input[channel] - mean[channel]) / std[channel]  
print(img\_tensor[0][0][0])  
trans\_norm=transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])#三个均值和三个标准差，范围自设（0，1）  
img\_norm=trans\_norm(img\_tensor)#归一化output[channel] = (input[channel] - mean[channel]) / std[channel]，则img\_norm=2\*img\_totensor-1  
print(img\_norm[0][0][0])  
writer.add\_image("Normalize",img\_norm)  
  
#Resize  
print(img.size)  
trans\_resize=transforms.Resize((512,512))#将resize类进行实例化，按照h=512，w=512进行输出，或者只有一个int参数的时候则将h和w等比例缩放  
img\_resize=trans\_resize(img)  
print(img\_resize)#此时是PIL的image数据类型，要加入tensorboard需要将去转换为tensor类型  
img\_resize=trans\_totensor(img\_resize)  
writer.add\_image("Resize",img\_resize,0)  
  
writer.close()

输出



Compose和RandomCrop的使用，Compose是将多个transforms操作给结合在一起，RandomCrop就是进行随机裁剪

P12norm\_resize\_compose.py:

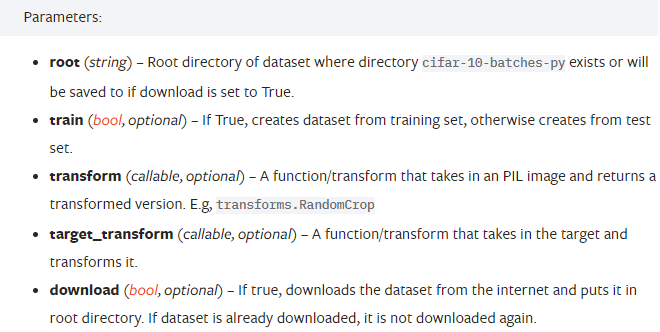
from PIL import Image  
from torch.utils.tensorboard import SummaryWriter  
from torchvision import transforms  
  
writer=SummaryWriter("logs")  
img=Image.open("D:\\deep\_learning\_dataset\\hymenoptera\_data\\train\\ants\\0013035.jpg")  
#ToTensor 这是一个类，下面将PIL的Image类转换为ToTensor类并将其用tensorboard显示出来  
trans\_totensor=transforms.ToTensor()  
img\_tensor=trans\_totensor(img)  
writer.add\_image("ToTensor",img\_tensor)  
#Normalize归一化 output[channel] = (input[channel] - mean[channel]) / std[channel]  
print(img\_tensor[0][0][0])  
trans\_norm=transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])#三个均值和三个标准差，范围自设（0，1）  
img\_norm=trans\_norm(img\_tensor)#归一化output[channel] = (input[channel] - mean[channel]) / std[channel]，则img\_norm=2\*img\_totensor-1  
print(img\_norm[0][0][0])  
writer.add\_image("Normalize",img\_norm)  
  
#Resize 重塑型，可输入序列（H，W）输出为H\*W的图片，或者输入一位int，输出原图的等比例缩减int倍  
print(img.size)  
trans\_resize=transforms.Resize((512,512))#将resize类进行实例化，作用是按照h=512，w=512进行输出  
img\_resize=trans\_resize(img)  
print(img\_resize)#此时是PIL的image数据类型，要加入tensorboard需要将去转换为tensor类型  
img\_resize=trans\_totensor(img\_resize)  
writer.add\_image("Resize",img\_resize,0)  
  
#Compose 可以结合多个transforms，即把一个图片的多个操作结合成一次性完成  
trans\_resize\_2=transforms.Resize(300)  
trans\_compose=transforms.Compose([trans\_resize\_2,trans\_totensor])#将Compose类进行实例化，作用是Resize和ToTensor结合起来  
img\_resize\_2=trans\_compose(img)  
writer.add\_image("Compose",img\_resize\_2,1)  
  
#RandomCrop对图片进行裁剪的函数，如果是一位int，则裁剪为正方形，若是二位序列（H，W），则裁剪为H\*W  
trans\_random=transforms.RandomCrop(512)#将RandomCrop的实例化，裁剪为512\*512  
trans\_compose\_2=transforms.Compose([trans\_random,trans\_totensor])#将Compose实例化，进行裁剪并转为totensor  
for i in range(10):  
 img\_crop=trans\_compose\_2(img)  
 writer.add\_image("RandomCrop",img\_crop,i)  
  
  
writer.close()

总结方法：函数参数和返回值都是什么类型：看官方文档；当不知道返回值的时候，可以通过print()和print(type())来查看

# 6.torchvision的数据集使用

可以查看官方文档来查看数据集如何使用<https://pytorch.org/vision/stable/datasets.html>

以[CIFAR10](https://www.cs.toronto.edu/~kriz/cifar.html)进举例，

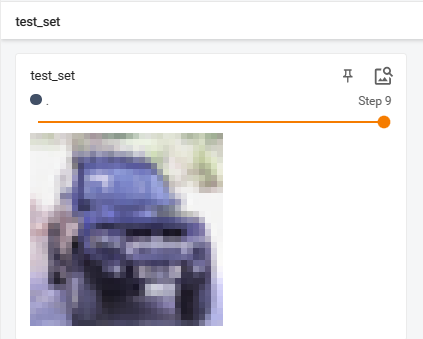


Root代表数据集下载到本地的路径，train表示是否作为训练集，true为训练集，false为测试集，download表示是否下载到root路径

import torchvision  
  
trans\_set=torchvision.datasets.CIFAR10(root="./dataset",train=True,download=True)#./表示当前的untitle文件夹  
test\_set=torchvision.datasets.CIFAR10(root="./dataset",train=False,download=True)  
print(test\_set[0])  
img,target=test\_set[0]#test\_set数据类型是序列(PIL.Image,int)  
print(img)  
print(target)  
print(test\_set.classes[target])#这是一个分类算法的数据集，classes就是分类的类别，即label  
img.show()

数据集用在tensorboard中P14torchvision\_dataset.py：

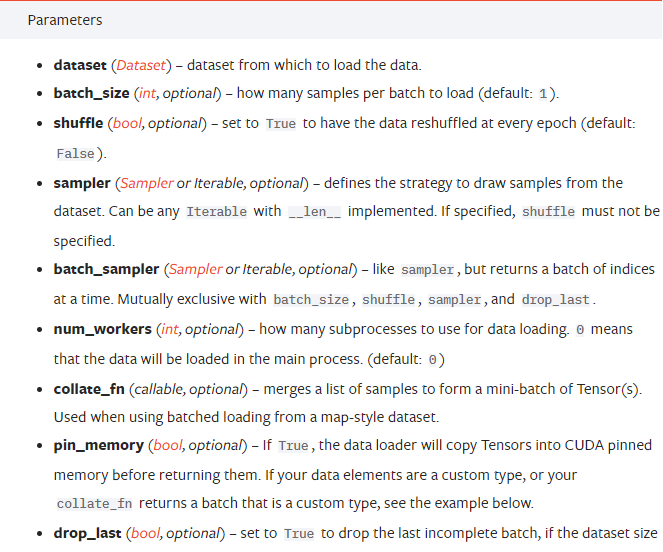
import torchvision  
from torch.utils.tensorboard import SummaryWriter  
  
dataset\_transform=torchvision.transforms.Compose([  
 torchvision.transforms.ToTensor()  
])  
train\_set=torchvision.datasets.CIFAR10(root="./dataset",train=True,transform=dataset\_transform,download=True)#./表示当前的untitle文件夹  
test\_set=torchvision.datasets.CIFAR10(root="./dataset",train=False,transform=dataset\_transform,download=True)  
  
print(test\_set[0])#这里数据类型是序列(ToTensor,int)  
writer=SummaryWriter("p14")#将tensorboard的日志logs名称设置为p14  
for i in range(10):  
 img\_tensor,target=test\_set[i]  
 writer.add\_image("test\_set",img\_tensor,i)  
writer.close()



# 7.Dataloader的使用

数据集加载区别于dataset，其可以实现多个数据打包为一个数据集，类似于抓牌，dataset相当于一次只抓一张牌，而dataloader可以一次抓多个牌查看官方文档如何使用dataloader

<https://pytorch.org/docs/1.8.1/data.html?highlight=dataloader#torch.utils.data.DataLoader>



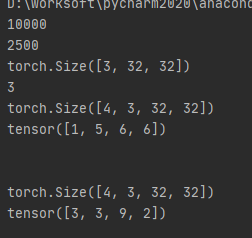
dataset即数据集，batch\_size相当于一次性抓牌数目，shuffle数据集是否进行打乱，num\_workers是否通过多进程加载数据集，drop\_last是当最后剩下的一个分组不到batch\_size的个数是否丢弃

P15dataloader.py：

import torchvision  
from torch.utils.data import DataLoader  
  
#准备测试集  
test\_data=torchvision.datasets.CIFAR10("./dataset",train=False,transform=torchvision.transforms.ToTensor())  
test\_loader=DataLoader(dataset=test\_data,batch\_size=4,shuffle=True,num\_workers=0,drop\_last=False)  
  
  
print(len(test\_data))  
print(len(test\_loader))#test\_loader将test\_data每4个进行打包  
img,target=test\_data[0]  
print(img.shape)  
print(target)  
#imgs,targets=test\_loader[0] 报错，由于test\_loader类型不可根据索引指定  
for data in test\_loader:  
 imgs,targets=data  
 print(imgs.shape)#由于输出均为torch.Size([4, 3, 32, 32])表示4个set打包，3通道，图片大小均为32\*32，则只有一个打包

print(targets)#输出假设为tensor([1, 5, 6, 6])表示这几个dataset的target分别是1，5，6，6  
 print("\n")

输出为：

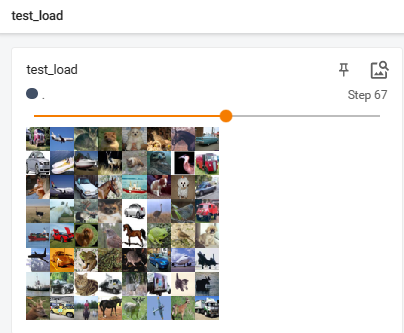




使用tensroboard可视化，将batch\_size改为64张P15\_2.py:

import torchvision  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
#准备测试集  
  
  
test\_data=torchvision.datasets.CIFAR10("./dataset",train=False,transform=torchvision.transforms.ToTensor())  
test\_loader=DataLoader(dataset=test\_data,batch\_size=64,shuffle=True,num\_workers=0,drop\_last=False)#这里改为64  
  
  
img,target=test\_data[0]  
print(img.shape)  
print(target)  
writer=SummaryWriter("dataloader")  
step=0  
#imgs,targets=test\_loader[0] 报错，由于test\_loader类型不可根据索引指定  
for data in test\_loader:  
 imgs,targets=data  
 writer.add\_images("test\_load",imgs,step)#注意这里使用的add\_images而不是add\_image  
 step=step+1  
writer.close()

结果为



# 8.神经网络的基本骨架nn.Module的使用

官方文档<https://pytorch.org/docs/1.8.1/generated/torch.nn.Module.html#torch.nn.Module>

官方实例代码

**import** torch.nn **as** nn

**import** torch.nn.functional **as** F

**class** **Model**(nn**.**Module):

**def** \_\_init\_\_(self):

super(Model, self)**.**\_\_init\_\_()

self**.**conv1 **=** nn**.**Conv2d(1, 20, 5)#构造卷积类

self**.**conv2 **=** nn**.**Conv2d(20, 20, 5)

**def** **forward**(self, x):

x **=** F**.**relu(self**.**conv1(x))#卷积类实例化self.conv1(x)进行第一次卷积操作，F**.**relu()将其非线性化

**return** F**.**relu(self**.**conv2(x))#卷积累实例化self**.**conv2(x)进行第二次卷积操作，F**.**relu()将其非线性化

作为任何神经网络模型model的基础类P16nn\_module.py:

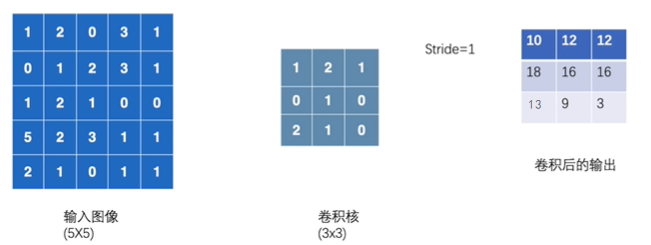
import torch  
from torch import nn  
  
  
class Model(nn.Module):  
 def \_\_init(self):  
 super(Model, self).\_\_init\_\_()  
 def forward(self,input):  
 output=input+1  
 return output  
  
model=Model()  
x=torch.tensor(1.0)  
output=model(x)#这里Model父类中的nn.Module中的魔术方法call函数实现\_call\_impl函数会自动调用forward函数  
print(output)

输出

# 9.神经网络（注意输入数据在程序中的维度变换！）

## 卷积操作

官方文档<https://pytorch.org/docs/1.8.1/nn.functional.html>



P17conv2d.py:

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]])  
kernel=torch.tensor([[1,2,1],  
 [0,1,0],  
 [2,1,0]])  
input=torch.reshape(input,(1,1,5,5))#(batch\_size,channel,H,W)只取一个样本，batch\_size=1，二维张量通道为1  
kernel=torch.reshape(kernel,(1,1,3,3))  
  
output=F.conv2d(input,kernel,stride=1)  
print(output)  
  
output2=F.conv2d(input,kernel,stride=1,padding=1)#padding用来填充卷积的外边部分  
print(output2)

输出

tensor([[[[10, 12, 12],

[18, 16, 16],

[13, 9, 3]]]])

tensor([[[[ 1, 3, 4, 10, 8],

[ 5, 10, 12, 12, 6],

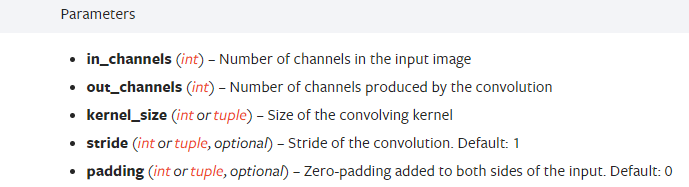
[ 7, 18, 16, 16, 8],

[11, 13, 9, 3, 4],

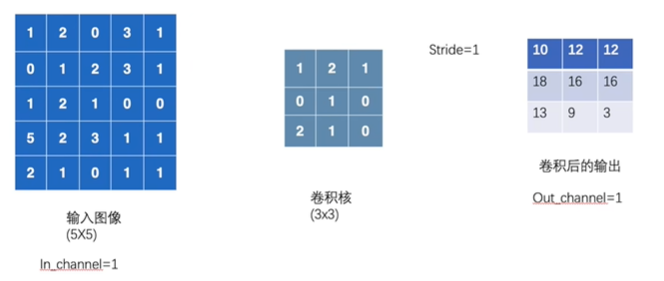
[14, 13, 9, 7, 4]]]])

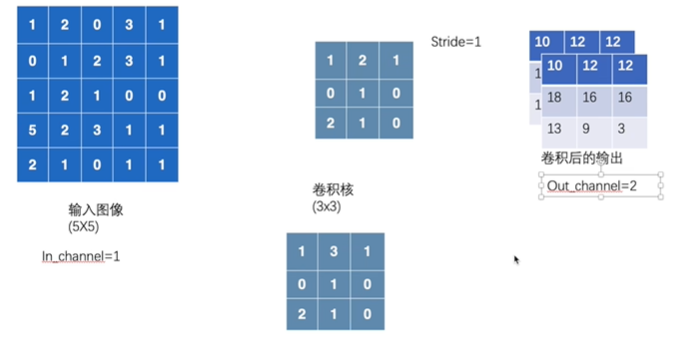
## 卷积层

官方文档<https://pytorch.org/docs/1.8.1/generated/torch.nn.Conv2d.html#torch.nn.Conv2d>



kernel\_size是在进行神经网络训练中不断进行自己调整的参数；对于in\_channel和out\_channel





当out\_channel=2的时候卷积层会生成2个卷积核，对输入图像进行卷积，输出两层（二层重叠形态）

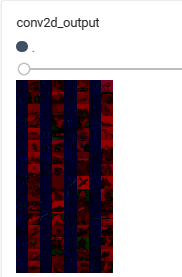
彩色图像的in\_channel=3这是相当于一层的无色图像有代表颜色r、g、b的三层（三层重叠形态）

黑白图像的in\_channel=2这是相当于一层的无色图像有代表颜色黑和白的两层（二层重叠形态）

P18nn\_conv2d.py:

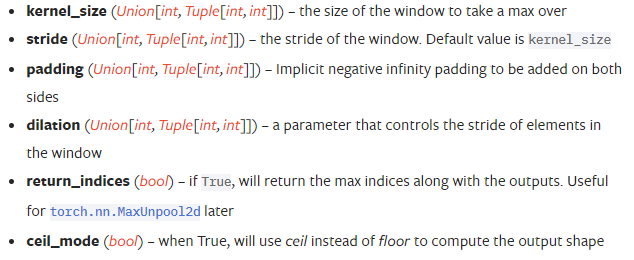
import torch  
import torchvision  
from torch import nn  
from torch.nn import Conv2d  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
  
dataset=torchvision.datasets.CIFAR10("./dataset",train=False,transform=torchvision.transforms.ToTensor(),download=True)  
dataloader=DataLoader(dataset,batch\_size=64)  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model,self).\_\_init\_\_()  
 self.conv1=Conv2d(in\_channels=3,out\_channels=6,kernel\_size=3)#定义一个卷积类  
 def forward(self,x):  
 x=self.conv1(x)#卷积类实例化，进行一次卷积操作  
 return x  
model=Model()  
print(model)  
  
writer=SummaryWriter("con2v\_logs")  
step=0  
for data in dataloader:  
 imgs,targets=data  
 output=model(imgs)  
 print(imgs.shape)  
 print(output.shape)  
 #torch.Size([64, 6, 30, 30])  
 writer.add\_images("conv2d\_input",imgs,step)  
 #torch.Size([64, 6, 30, 30])无法显示出6个channel，直接输出到tensorboard会报错则需要转换成[xxx, 3, 30, 30]  
 output=torch.reshape(output,(-1,3,30,30))#这里(-1,3,30,30)batch\_size=-1表示程序要根据后面的（,3,30,30）来自动计算，计算方式为64\*6\*30\*30/(3\*30\*30)  
 writer.add\_images("conv2d\_output", output, step)  
 step=step+1  
writer.close()

输出



## 最大池化作用

常用maxpool2d<https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html#torch.nn.MaxPool2d>

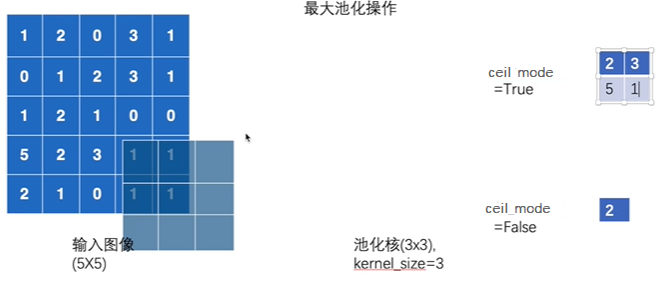


这里的kernel区别与卷积的kernel，卷积的kernel内是有值的，而这里池化的kernel只是一个框，无值，只定义kernel的大小kernek\_size就可；

Stride区别于卷积函数，卷积函数的stride默认是1，而这里默认是kernel\_size；

Dilation在卷积函数中也出现过，进行空洞卷积；

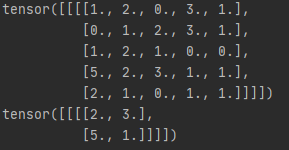
Ceil\_mode主要是用来在池化的过程中是否向上取整，如下图所示，当输入图像的剩余部分不能完全被池化核覆盖时，观察ceil\_mode是否为true，为true则取被池化核覆盖的输入图像最大值；如果为False则不取值



P19nn\_maxpool.py

import torch  
from torch import nn  
from torch.nn import MaxPool2d  
  
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)#最大池化函数max\_pool2d要求input输入的函数值为浮点数，否则会默认为整数long类型而报错  
input=torch.reshape(input,(-1,1,5,5))#将二维的input调整为4维的input以满足maxpool函数的输入shape，  
# 序列的batch\_size=-1表示reshape函数可以根据后面的channel=1，H=5，W=5来计算batch\_size的值，5\*5/(1\*5\*5)  
print(input)  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model,self).\_\_init\_\_()  
 self.maxpool1=MaxPool2d(kernel\_size=3,ceil\_mode=True)  
 def forward(self,input):  
 output=self.maxpool1(input)  
 return output  
model=Model()  
output=model(input)  
print(output)

输出



最大池化的作用：减少数据量的同时保持数据特征。卷积用来提取特征，池化用来压缩特征

在实际应用的体现P19\_2.py：

import torch  
import torchvision  
from torch import nn  
from torch.nn import MaxPool2d  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
  
dataset=torchvision.datasets.CIFAR10("./dataset",train=False,download=True,transform=torchvision.transforms.ToTensor())  
dataloader=DataLoader(dataset,batch\_size=64)  
  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model,self).\_\_init\_\_()  
 self.maxpool1=MaxPool2d(kernel\_size=3,ceil\_mode=True)  
 def forward(self,input):  
 output=self.maxpool1(input)  
 return output  
model=Model()  
writer=SummaryWriter("logs\_maxpool")  
step=0  
for data in dataloader:  
 imgs,targets=data  
 writer.add\_images("maxpool\_input",imgs,step)  
 output=model(imgs)#区别于卷积，池化不改变通道数out\_channel，所以这里不需要像卷积那样对output进行reshape  
 writer.add\_images("output\_maxpool",output,step)  
 step=step+1  
writer.close()

输出



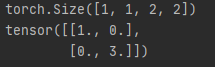
## 非线性激活

ReLu函数和Sigmoid函数

Relu函数P20relu.py

import torch  
from torch import nn  
from torch.nn import ReLU  
  
input=torch.tensor([[1,-0.5],  
 [-1,3]])  
  
output=torch.reshape(input,(-1,1,2,2))#计算batch\_size=2\*2/(1\*2\*2)  
print(output.shape)  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model,self).\_\_init\_\_()  
 self.relu1=ReLU()  
 def forward(self,input):  
 output=self.relu1(input)  
 return output  
model=Model()  
output=model(input)  
print(output)

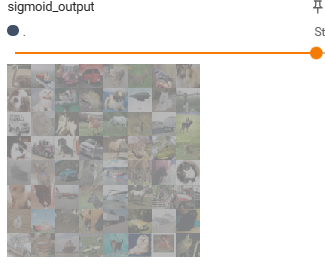
输出



Sigmoid在tensorboard上体现P20sigmoid.py：

import torch  
import torchvision  
from torch import nn  
from torch.nn import ReLU, Sigmoid  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
  
input=torch.tensor([[1,-0.5],  
 [-1,3]])  
dataset=torchvision.datasets.CIFAR10("./dataset",train=False,download=True,transform=torchvision.transforms.ToTensor())  
dataloader=DataLoader(dataset,batch\_size=64)  
output=torch.reshape(input,(-1,1,2,2))#计算batch\_size=2\*2/(1\*2\*2)  
print(output.shape)  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model,self).\_\_init\_\_()  
 self.relu1=ReLU()  
 self.sigmoid1=Sigmoid()  
 def forward(self,input):  
 output=self.sigmoid1(input)  
 return output  
model=Model()  
writer=SummaryWriter("logs\_sigmoid")  
step=0  
for data in dataloader:  
 imgs,targets=data  
 writer.add\_images("sigmoid\_input",imgs,step)  
 output=model(imgs)  
 writer.add\_images("sigmoid\_output",output,step)  
 step=step+1  
writer.close()

输出



## 线性层及其他层介绍

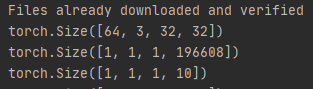
BatchNorm2d：批量标准化，防止过拟合，正则化

Dropout：防止过拟合

线性层P21nn\_linear.py：

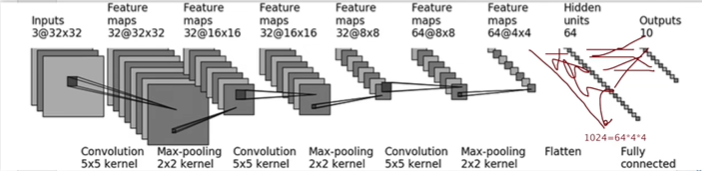
import torch  
import torchvision  
from torch import nn  
from torch.nn import Linear  
from torch.utils.data import DataLoader  
  
dataset=torchvision.datasets.CIFAR10("./dataset",train=False,transform=torchvision.transforms.ToTensor(),download=True)  
dataloader=DataLoader(dataset,batch\_size=64,drop\_last=True)#当batch\_size为64时，最后一项只有16个图像，不足64，要舍去，否则self.linear1=Linear(196608,10)会报错  
  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, self).\_\_init\_\_()  
 self.linear1=Linear(196608,10)  
 def forward(self,input):  
 output=self.linear1(input)  
 return output  
model=Model()  
for data in dataloader:  
 imgs,targets=data  
 print(imgs.shape)  
 output=torch.reshape(imgs,(1,1,1,-1))  
 print(output.shape)  
 output=model(output)  
 print(output.shape)

输出



## 搭建神经网络

数据集依旧使用CIFAR10其模型结构如图

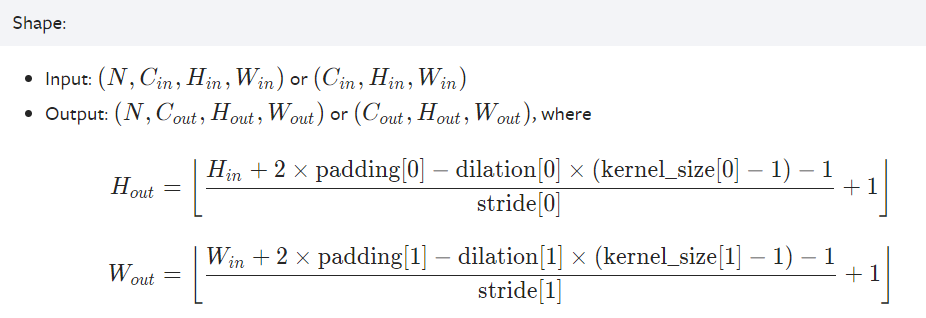


卷积核参数求取（以第一个卷积Convlution 5x5 kernel为例）：

首先打开官方文档<https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d>

其次知道Conv2d函数需要参数in\_channels,out\_channels,kernel\_size,stride,padding, dilation

根据文档的公式



已知输入图像(3,32,32)则in\_channel=3

经过5\*5卷积后输出为（3，32，32）则out\_channel=32

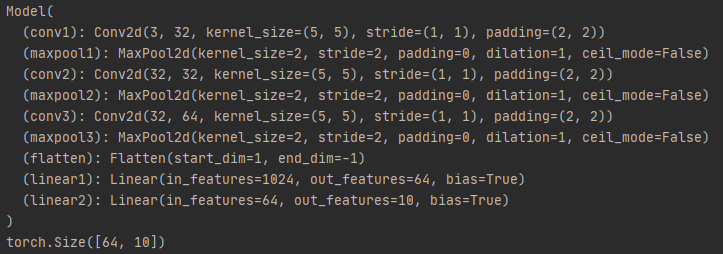
dilation大多情况下默认为1，根据框中的公式可知Hin=32,Hout=32,padding[0]未知,dilation[0]=1,kernel\_size[0]=5.stride[0]未知

则整理后有 若padding太大会造成图像有太多额外特征而使图像特征降低，因此这里stride=1，padding=2

P22nn\_seq.py

import torch  
from torch import nn  
from torch.nn import Conv2d, MaxPool2d, Flatten, Linear  
  
  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, self).\_\_init\_\_()  
 self.conv1=Conv2d(in\_channels=3,out\_channels=32,kernel\_size=5,stride=1,padding=2)  
 self.maxpool1=MaxPool2d(2)  
 self.conv2=Conv2d(in\_channels=32,out\_channels=32,kernel\_size=5,stride=1,padding=2)  
 self.maxpool2=MaxPool2d(2)  
 self.conv3=Conv2d(in\_channels=32,out\_channels=64,kernel\_size=5,stride=1,padding=2)  
 self.maxpool3 = MaxPool2d(2)  
 self.flatten=Flatten()#将多维向量展开为1维，等同于reshape(1,-1,)  
 self.linear1=Linear(1024,64)  
 self.linear2=Linear(64,10)  
 def forward(self,x):  
 x=self.conv1(x)  
 x=self.maxpool1(x)  
 x=self.conv2(x)  
 x=self.maxpool2(x)  
 x=self.conv3(x)  
 x=self.maxpool3(x)  
 x=self.flatten(x)  
 x=self.linear1(x)  
 x=self.linear2(x)  
 return x  
model=Model()  
print(model)  
input=torch.ones((64,3,32,32))#生成batch\_size=64的3\*32\*32的数值均为1的dataloader数据  
output=model(input)  
print(output.shape)

输出：

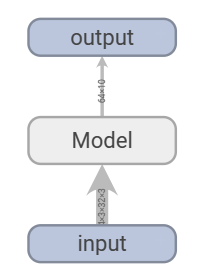
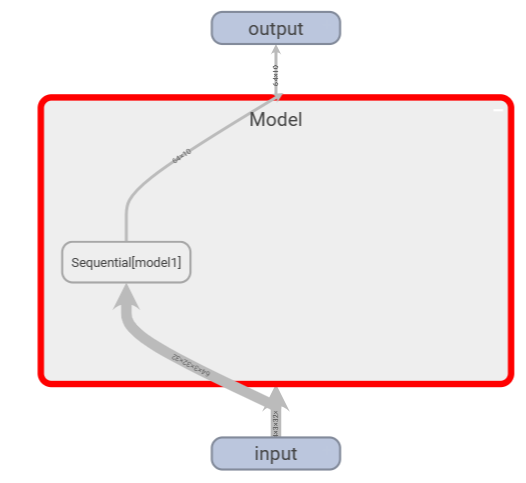


使用sequential函数进行整合（类似于compose函数，可以将重复的卷积、最大池化、线性激活等函数整合在一个模型里）

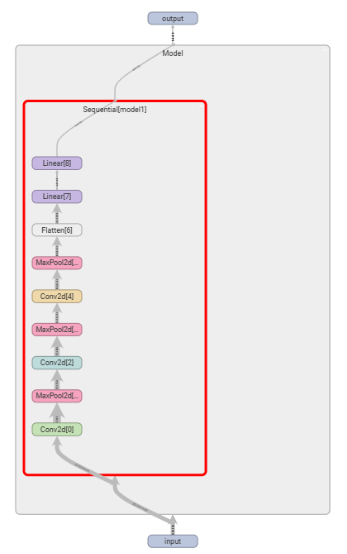
P22\_2.py

import torch  
from torch import nn  
from torch.nn import Conv2d, MaxPool2d, Flatten, Linear, Sequential  
from torch.utils.tensorboard import SummaryWriter  
  
  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, self).\_\_init\_\_()  
 '''self.conv1=Conv2d(in\_channels=3,out\_channels=32,kernel\_size=5,stride=1,padding=2)  
 self.maxpool1=MaxPool2d(2)  
 self.conv2=Conv2d(in\_channels=32,out\_channels=32,kernel\_size=5,padding=2)#stride可省略，默认为1  
 self.maxpool2=MaxPool2d(2)  
 self.conv3=Conv2d(32,64,5,stride=1,padding=2)  
 self.maxpool3 = MaxPool2d(2)  
 self.flatten=Flatten()#将多维向量展开为1维，等同于reshape(1,-1,)  
 self.linear1=Linear(1024,64)  
 self.linear2=Linear(64,10)'''  
 self.model1=Sequential(#Sequentia比较像compose函数，将多个操作整合到一起  
 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.conv1(x)  
 x=self.maxpool1(x)  
 x=self.conv2(x)  
 x=self.maxpool2(x)  
 x=self.conv3(x)  
 x=self.maxpool3(x)  
 x=self.flatten(x)  
 x=self.linear1(x)  
 x=self.linear2(x)'''* x=self.model1(x)  
 return x  
model=Model()  
print(model)  
input=torch.ones((64,3,32,32))#生成batch\_size=64的3\*32\*32的数值均为1的dataloader数据  
output=model(input)  
print(output.shape)  
writer=SummaryWriter("logs\_seq")  
writer.add\_graph(model,input)  
writer.close()

输出

双击Model后

双击Sequential[model1]后



## 损失函数和反向传播

Loss计算预测值和实际值之间的误差；反向传播是通过调整网络过程中的参数来使得loss变小的一个过程，由于和构建神经网络的顺序相反，因此叫做反向传播

L1Loss函数：当*reduction='mean'（默认）求*平均绝对值，；*reduction='sum'求和*

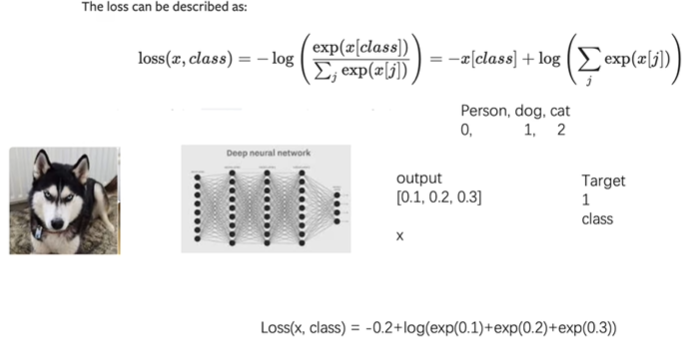
MSE函数：方差求损失——经典用于回归的损失函数

交叉熵：交叉熵，常用于分类的函数。

优化器有一个优化函数，最小二乘法，不同的优化函数对应不同的损失函数

CrossEntropyLoss的输入是（N，C）格式，N是batch\_size，C区别之前的卷积函数（通道），这里指类别数

举例



P23nn\_loss.py

import torch  
from torch import nn  
from torch.nn import L1Loss  
  
inputs=torch.tensor([1,2,3],dtype=torch.float32)#L1Loss函数的输入变量应该是浮点型，否则报错  
targets=torch.tensor([1,2,6],dtype=torch.float32)  
  
inputs=torch.reshape(inputs,(1,1,1,3))  
targets=torch.reshape(targets,(1,1,1,3))  
#使用L1Loss 绝对值损失  
loss=L1Loss(reduction="sum")  
result=loss(inputs,targets)  
print(result)  
#使用MSELoss 方差损失  
loss\_mse=nn.MSELoss()  
result\_mse=loss\_mse(inputs,targets)  
print(result\_mse)  
#交叉熵，常用于分类的函数，优化器有一个优化函数，最小二乘法，不同的优化函数对应不同的损失函数，  
# mse是经典用于回归的损失函数  
x=torch.tensor([0.1,0.2,0.3])  
y=torch.tensor([1])  
x=torch.reshape(x,(1,3))#CrossEntropyLoss的输入是（N，C）格式，N是batch\_size，C区别之前的卷积函数（通道），这里指类别数  
loss\_cross=nn.CrossEntropyLoss()  
result\_cross=loss\_cross(x,y)  
print(result\_cross)

在模型中的实际使用P23\_2.py

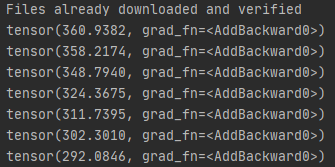
import torch  
import torchvision  
from torch import nn  
from torch.nn import Conv2d, MaxPool2d, Flatten, Linear, Sequential  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
  
dataset=torchvision.datasets.CIFAR10("./dataset",train=False,transform=torchvision.transforms.ToTensor(),download=True)  
dataloader=DataLoader(dataset,64)  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, self).\_\_init\_\_()  
 self.model1=Sequential(#Sequentia比较像compose函数，将多个操作整合到一起  
 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  
model=Model()  
loss=nn.CrossEntropyLoss()  
for data in dataloader:  
 imgs,targets=data  
 output=model(imgs)  
 result\_loss=loss(output,targets)  
 result\_loss.backward()

## 优化器

P24nn\_optim.py

import torch  
import torchvision  
from torch import nn  
from torch.nn import Conv2d, MaxPool2d, Flatten, Linear, Sequential  
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,64)  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, self).\_\_init\_\_()  
 self.model1=Sequential(#Sequentia比较像compose函数，将多个操作整合到一起  
 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  
model=Model()  
loss=nn.CrossEntropyLoss()  
optim=torch.optim.SGD(model.parameters(),lr=0.01)  
for epoch in range(20):  
 running\_loss=0.0  
 for data in dataloader:  
 imgs,targets=data  
 output=model(imgs)  
 result\_loss=loss(output,targets)  
 optim.zero\_grad()#重要！优化器每步都需要清0，否则梯度值就会累加，失去作用  
 result\_loss.backward()  
 optim.step()#优化器对每个参数进行调优  
 running\_loss+=result\_loss  
 print(running\_loss)

输出：



由此可见，优化器会逐步调整使得loss变小

# 10.在已有的神经网络上修改

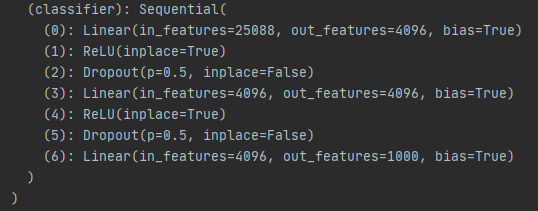
Vgg16模型，一个分类模型，官方文档：<https://pytorch.org/vision/stable/models/vgg.html>

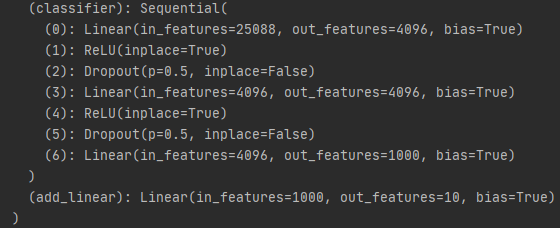
P25pretrained.py

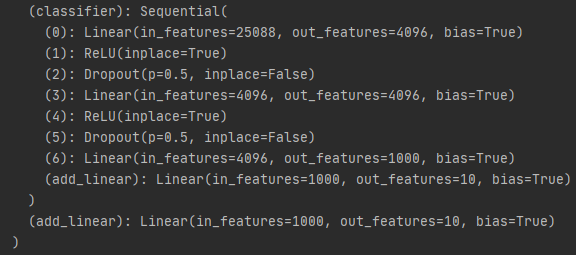
import torchvision  
  
#train\_data=torchvision.datasets.ImageNet("./data\_imageNet",split='train',download=True,transform=torchvision.transforms.ToTensor())  
from torch import nn  
  
vgg16\_false=torchvision.models.vgg16(weights=None)#默认没有预训练   
vgg16\_true=torchvision.models.vgg16(weights='DEFAULT')  
print(vgg16\_true)  
  
vgg16\_true.add\_module('add\_linear',nn.Linear(1000,10))#给原vgg16模型添加一层线性变换  
print(vgg16\_true)

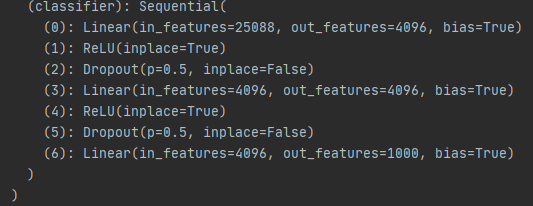
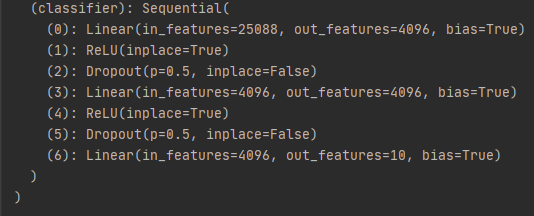
vgg16\_true.classifier.add\_module('add\_linear',nn.Linear(1000,10))#在vgg16模型的classfier里添加线性变换  
print(vgg16\_true)  
  
print(vgg16\_false)  
vgg16\_false.classifier[6]=nn.Linear(4096,10)#修改vgg16false的模型6号层  
print(vgg16\_false)

输出







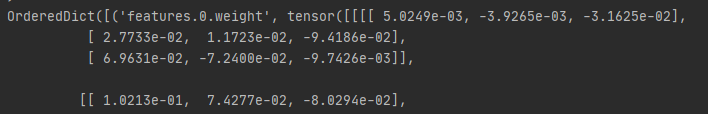
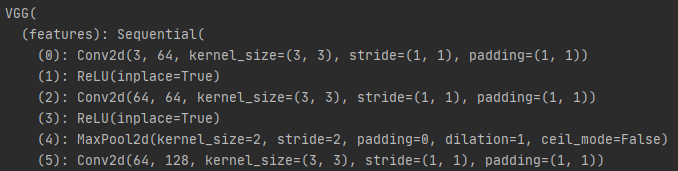


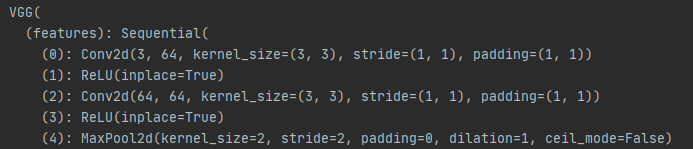
# 11．模型保存与加载

保存与加载vgg16模型，一共两种方法P26model\_saver.py

import torch  
import torchvision  
vgg16=torchvision.models.vgg16()  
  
#保存方式1  
torch.save(vgg16,"vgg16\_methond1.pth")#模型保存  
model=torch.load("vgg16\_methond1.pth")#模型加载  
print(model)  
  
#保存方式2保存为字典格式  
torch.save(vgg16.state\_dict(),"vgg16\_methond2.pth")#模型保存  
model=torch.load("vgg16\_methond2.pth")#模型加载  
print(model)#输出为字典类型  
#将字典类型改为网络模型  
vgg16.load\_state\_dict(model)#通过python中的字典形式加载数据  
print(vgg16)

输出





看上去似乎方法一更简单，但是常用方法二

保存加载自己的模型P26\_2.py

import torch  
from torch import nn  
  
#加载自己的模型  
class Model(nn.Module):#定义模型  
 def \_\_init\_\_(self):  
 super(Model, self).\_\_init\_\_()  
 self.conv1=nn.Conv2d(3,64,kernel\_size=3)  
 def forward(self,x):  
 x=self.conv1(x)  
 return x  
model=Model()  
torch.save(model,"model\_method.pth")#保存模型  
  
model=torch.load("model\_method.pth")#加载模型  
print(model)

# 12.完整的模型训练套路

## 基本的训练模型

准备数据集；搭建神经网络；创建网络模型；定义损失函数；定义优化器；设置训练网络的参数；开始训练（求训练值，求损失，优化器优化）

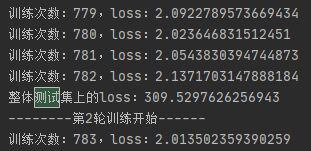
P27train.py

import torch  
import torchvision  
from torch import 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)  
#求数据集长度  
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))  
  
#加载为dataloader数据集  
train\_dataloader=DataLoader(train\_data,batch\_size=64)  
test\_dataloader=DataLoader(test\_data,batch\_size=64)  
  
#搭建神经网络  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, 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(),  
 nn.Linear(64\*4\*4,64),  
 nn.Linear(64,10)  
 )  
 def forward(self,x):  
 x=self.model(x)  
 return x  
  
#创建网络模型  
model=Model()  
#损失函数  
loss\_fn=nn.CrossEntropyLoss()#交叉熵损失  
  
#优化器  
learning\_rate=1e-2 #1\*10^(-2)=0.01  
optimizier=torch.optim.SGD(model.parameters(),lr=learning\_rate)#SGD优化器  
#设置训练网络的一些参数  
total\_train\_step=0#记录训练的次数  
total\_test\_step=0#记录测试的次数  
epoch=10#训练的轮次  
  
for i in range(epoch):  
 print("--------第{}轮训练开始------".format(i+1))  
 #训练步骤开始  
 for data in train\_dataloader:  
 imgs,targets=data  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 #优化器优化模型  
 optimizier.zero\_grad()  
 loss.backward()  
 optimizier.step()  
  
 total\_train\_step+=1  
 print("训练次数：{}，loss：{}".format(total\_train\_step,loss.item()))#item将loss的数据类型转换为只有loss的值的数值型

## 对模型训练后测试

每一epoch的训练后需要用测试集测试当前训练的网络模型效果如何

import torch  
import torchvision  
from torch import 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)  
#求数据集长度  
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))  
  
#加载为dataloader数据集  
train\_dataloader=DataLoader(train\_data,batch\_size=64)  
test\_dataloader=DataLoader(test\_data,batch\_size=64)  
  
#搭建神经网络  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, 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(),  
 nn.Linear(64\*4\*4,64),  
 nn.Linear(64,10)  
 )  
 def forward(self,x):  
 x=self.model(x)  
 return x  
  
#创建网络模型  
model=Model()  
#损失函数  
loss\_fn=nn.CrossEntropyLoss()  
  
#优化器  
learning\_rate=1e-2 #1\*10^(-2)=0.01  
optimizier=torch.optim.SGD(model.parameters(),lr=learning\_rate)  
#设置训练网络的一些参数  
total\_train\_step=0#记录训练的次数  
total\_test\_step=0#记录测试的次数  
epoch=10#训练的轮次  
  
for i in range(epoch):  
 print("--------第{}轮训练开始------".format(i+1))  
 #训练步骤开始  
 for data in train\_dataloader:  
 imgs,targets=data  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 #优化器优化模型  
 optimizier.zero\_grad()  
 loss.backward()  
 optimizier.step()  
  
 total\_train\_step+=1  
 print("训练次数：{}，loss：{}".format(total\_train\_step,loss.item()))#item将loss的数据类型转换为只有loss的值的数值型  
 #测试步骤  
 total\_test\_loss=0#往往训练需要求本次epoch训练的模型在所有测试集上的loss  
 with torch.no\_grad():#在已有的模型基础上设置梯度为0，即不再调优  
 for data in test\_dataloader:  
 imgs,targets=data  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 total\_test\_loss+=loss.item()  
 print("整体测试集上的loss：{}".format(total\_test\_loss))

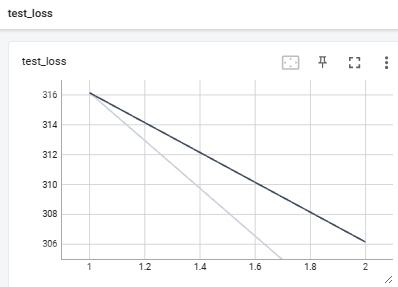
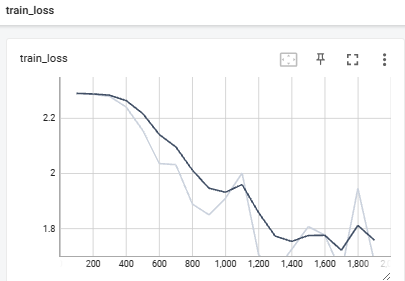


## 使用tensorboard可视化

P28train\_test.py

import torch  
import torchvision  
from torch import nn  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
  
#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)  
#求数据集长度  
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))  
  
#加载为dataloader数据集  
train\_dataloader=DataLoader(train\_data,batch\_size=64)  
test\_dataloader=DataLoader(test\_data,batch\_size=64)  
  
#搭建神经网络  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, 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(),  
 nn.Linear(64\*4\*4,64),  
 nn.Linear(64,10)  
 )  
 def forward(self,x):  
 x=self.model(x)  
 return x  
  
#创建网络模型  
model=Model()  
#损失函数  
loss\_fn=nn.CrossEntropyLoss()  
  
#优化器  
learning\_rate=1e-2 #1\*10^(-2)=0.01  
optimizier=torch.optim.SGD(model.parameters(),lr=learning\_rate)  
#设置训练网络的一些参数  
total\_train\_step=0#记录训练的次数  
total\_test\_step=0#记录测试的次数  
epoch=10#训练的轮次  
  
#添加tensorboard  
writer=SummaryWriter("logs\_train\_test")  
#开始训练  
for i in range(epoch):  
 print("--------第{}轮训练开始------".format(i+1))  
 #训练步骤开始  
 for data in train\_dataloader:  
 imgs,targets=data  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 #优化器优化模型  
 optimizier.zero\_grad()  
 loss.backward()  
 optimizier.step()  
  
 total\_train\_step+=1  
 if total\_train\_step%100==0:#每训练100步才打印一次loss，防止输出太多太乱  
 print("训练次数：{}，loss：{}".format(total\_train\_step,loss.item()))#item将loss的数据类型转换为只有loss的值的数值型  
 writer.add\_scalar("train\_loss",loss.item(),total\_train\_step)  
 #测试步骤  
 total\_test\_loss=0#往往训练需要求本次epoch训练的模型在所有测试集上的loss  
 with torch.no\_grad():#在已有的模型基础上设置梯度为0，即不再调优  
 for data in test\_dataloader:  
 imgs,targets=data  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 total\_test\_loss+=loss.item()  
 print("整体测试集上的loss：{}".format(total\_test\_loss))  
 total\_test\_step += 1  
 writer.add\_scalar("test\_loss",total\_test\_loss,total\_test\_step)  
 #保存每epoch轮的模型  
 torch.save(model,"model\_{}.pth".format(i))  
 print("模型已保存")  
writer.close()

输出



## 准确率(常用在分类问题中)

假设2分类(0,1)问题

输出分类的结果为

ouput=[0.1,0.2]

[0.05,0.4]

这个输出说明在第一波数据集中类型为0类的概率为0.1，为1类的概率为0.2；在第二波数据集中类型为0类的概率为0.05，为1类的概率为0.4

而targets=[0][1]说明实际第一波数据集属于0类，第二波数据集属于1类

如何将output的数据类型转换为targets数据类型？通过Argmax方法

Argmax(1)可以将每波数据集预测概率的最大值的下标输出，

即output=[0.1,0.2],[0.05,0.4]->argmax(1)->preds=[1][1] 即预测的结果是第一波数据预测为1类，第二波数据集预测为1类

Preds==targets 输出[false,true] 易知，第一波数据集预测错误，第二波数据集预测正确

则正确率=[false,true].sum()/数据集波数

P28\_argmax.py

import torch  
  
outputs=torch.tensor([[0.1,0.2],  
 [0.05,0.4]])  
print(outputs.argmax(1))#为1表示求横向tensor数据中的最大值的下标，分别是0.1<0.2,0.05<0.4。0.2和0.4下标为1  
print(outputs.argmax(0))#为0表示求纵向tensor数据中的最大值的下标，分别是0.1>0.05,0.2<0.4。  
  
  
preds=outputs.argmax(1)  
targets=torch.tensor([0,1])  
print((preds==targets).sum())  
print((preds==targets).sum()/2)

输出

tensor([1, 1])

tensor([0, 1])

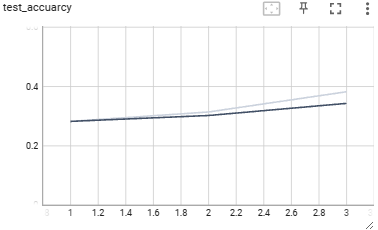
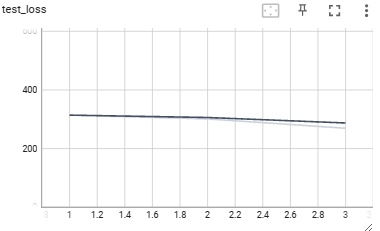
tensor(1)

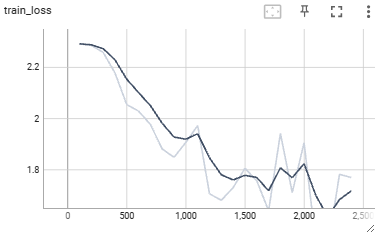
tensor(0.5000) #说明正确率50%

在实际模型中P28accuracy.py

import torch  
import torchvision  
from torch import nn  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
  
#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)  
#求数据集长度  
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))  
  
#加载为dataloader数据集  
train\_dataloader=DataLoader(train\_data,batch\_size=64)  
test\_dataloader=DataLoader(test\_data,batch\_size=64)  
#搭建神经网络  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, 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(),  
 nn.Linear(64\*4\*4,64),  
 nn.Linear(64,10)  
 )  
 def forward(self,x):  
 x=self.model(x)  
 return x  
  
#创建网络模型  
model=Model()  
#损失函数  
loss\_fn=nn.CrossEntropyLoss()  
  
#优化器  
learning\_rate=1e-2 #1\*10^(-2)=0.01  
optimizier=torch.optim.SGD(model.parameters(),lr=learning\_rate)  
#设置训练网络的一些参数  
total\_train\_step=0#记录训练的次数  
total\_test\_step=0#记录测试的次数  
epoch=10#训练的轮次  
  
#添加tensorboard  
writer=SummaryWriter("logs\_train\_test")  
#开始训练  
for i in range(epoch):  
 print("--------第{}轮训练开始------".format(i+1))  
 #训练步骤开始  
 for data in train\_dataloader:  
 imgs,targets=data  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 #优化器优化模型  
 optimizier.zero\_grad()  
 loss.backward()  
 optimizier.step()  
  
 total\_train\_step+=1  
 if total\_train\_step%100==0:#每训练100步才打印一次loss，防止输出太多太乱  
 print("训练次数：{}，loss：{}".format(total\_train\_step,loss.item()))#item将loss的数据类型转换为只有loss的值的数值型  
 writer.add\_scalar("train\_loss",loss.item(),total\_train\_step)  
 #测试步骤  
 total\_test\_loss=0#往往训练需要求本次epoch训练的模型在所有测试集上的loss  
  
 total\_test\_accuracy=0  
 with torch.no\_grad():#在已有的模型基础上设置梯度为0，即不再调优  
 for data in test\_dataloader:  
 imgs,targets=data  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 total\_test\_loss+=loss.item()  
 #求准确率  
 accuracy=(outputs.argmax(1)==targets).sum()  
 total\_test\_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\_test\_accuracy/test\_data\_size))  
 total\_test\_step += 1  
 writer.add\_scalar("test\_loss",total\_test\_loss,total\_test\_step)  
 writer.add\_scalar("test\_accuarcy",total\_test\_accuracy/test\_data\_size,total\_test\_step)  
 #保存每epoch轮的模型  
 torch.save(model,"model\_{}.pth".format(i))  
 print("模型已保存")  
writer.close()

输出



## 增加一些细节

很多算法在模型训练前会调用train()函数；并且在测试之前调用eval()函数

在官方文档<https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module>里，二者作用的训练层有限，会作用在Dropout层, BatchNorm层等，因此当模型中存在这种特殊层时这两个函数才有效

P29.py

import torch  
import torchvision  
from torch import nn  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
  
#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)  
#求数据集长度  
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))  
  
#加载为dataloader数据集  
train\_dataloader=DataLoader(train\_data,batch\_size=64)  
test\_dataloader=DataLoader(test\_data,batch\_size=64)  
#搭建神经网络  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, 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(),  
 nn.Linear(64\*4\*4,64),  
 nn.Linear(64,10)  
 )  
 def forward(self,x):  
 x=self.model(x)  
 return x  
  
#创建网络模型  
model=Model()  
#损失函数  
loss\_fn=nn.CrossEntropyLoss()  
  
#优化器  
learning\_rate=1e-2 #1\*10^(-2)=0.01  
optimizier=torch.optim.SGD(model.parameters(),lr=learning\_rate)  
#设置训练网络的一些参数  
total\_train\_step=0#记录训练的次数  
total\_test\_step=0#记录测试的次数  
epoch=10#训练的轮次  
  
#添加tensorboard  
writer=SummaryWriter("logs\_train\_test")  
#开始训练  
for i in range(epoch):  
 print("--------第{}轮训练开始------".format(i+1))  
 #训练步骤开始  
 model.train()#很多算法中模型在训练之前会调用train函数  
 for data in train\_dataloader:  
 imgs,targets=data  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 #优化器优化模型  
 optimizier.zero\_grad()  
 loss.backward()  
 optimizier.step()  
  
 total\_train\_step+=1  
 if total\_train\_step%100==0:#每训练100步才打印一次loss，防止输出太多太乱  
 print("训练次数：{}，loss：{}".format(total\_train\_step,loss.item()))#item将loss的数据类型转换为只有loss的值的数值型  
 writer.add\_scalar("train\_loss",loss.item(),total\_train\_step)  
 #测试步骤  
 model.eval()#很多算法中模型在训练之前会调用eval函数  
 total\_test\_loss=0#往往训练需要求本次epoch训练的模型在所有测试集上的loss  
 total\_test\_accuracy=0  
 with torch.no\_grad():#在已有的模型基础上设置梯度为0，即不再调优  
 for data in test\_dataloader:  
 imgs,targets=data  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 total\_test\_loss+=loss.item()  
 #求准确率  
 accuracy=(outputs.argmax(1)==targets).sum()  
 total\_test\_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\_test\_accuracy/test\_data\_size))  
 total\_test\_step += 1  
 writer.add\_scalar("test\_loss",total\_test\_loss,total\_test\_step)  
 writer.add\_scalar("test\_accuarcy",total\_test\_accuracy/test\_data\_size,total\_test\_step)  
 #保存每epoch轮的模型  
 torch.save(model,"model\_{}.pth".format(i))#这里是方法一保存的模型

#方式二torch.save(model.state\_dict(),"model\_{}.pth".format(i))  
 print("模型已保存")  
writer.close()

综上是一次深度学习的真正算法流程：

准备数据集，求数据集长度，加载为dataloader数据集，搭建神经网络，创建网络模型，定义损失函数，定义优化器，开始训练（求损失，优化器优化模型，打印数据），测试（求损失，求准确率），保存模型

# 13.使用gpu训练

## .cuda([网络模型|数据（input，target）|损失函数])

P30train\_gpu\_1.py

import torch  
import torchvision  
from torch import nn  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
  
#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)  
#求数据集长度  
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))  
  
#加载为dataloader数据集  
train\_dataloader=DataLoader(train\_data,batch\_size=64)  
test\_dataloader=DataLoader(test\_data,batch\_size=64)  
#搭建神经网络  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, 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(),  
 nn.Linear(64\*4\*4,64),  
 nn.Linear(64,10)  
 )  
 def forward(self,x):  
 x=self.model(x)  
 return x  
  
#创建网络模型  
model=Model()  
if torch.cuda.is\_available():#判断电脑是否可用cuda，即是否有gpu  
 model=model.cuda()#模型加入cuda  
#损失函数  
loss\_fn=nn.CrossEntropyLoss()#交叉熵损失  
if torch.cuda.is\_available():#判断电脑是否可用cuda，即是否有gpu  
 loss\_fn=loss\_fn.cuda()#损失加入cuda  
#优化器  
learning\_rate=1e-2 #1\*10^(-2)=0.01  
optimizier=torch.optim.SGD(model.parameters(),lr=learning\_rate)#SGD优化器  
#设置训练网络的一些参数  
total\_train\_step=0#记录训练的次数  
total\_test\_step=0#记录测试的次数  
epoch=10#训练的轮次  
  
#添加tensorboard  
writer=SummaryWriter("logs\_train\_test")  
#开始训练  
for i in range(epoch):  
 print("--------第{}轮训练开始------".format(i+1))  
 #训练步骤开始  
 model.train()#很多算法中模型在训练之前会调用train函数  
 for data in train\_dataloader:  
 imgs,targets=data  
 if torch.cuda.is\_available(): # 判断电脑是否可用cuda，即是否有gpu  
 imgs=imgs.cuda()#input加入cuda  
 targets=targets.cuda()#target加入cuda  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 #优化器优化模型  
 optimizier.zero\_grad()  
 loss.backward()  
 optimizier.step()  
  
 total\_train\_step+=1  
 if total\_train\_step%100==0:#每训练100步才打印一次loss，防止输出太多太乱  
 print("训练次数：{}，loss：{}".format(total\_train\_step,loss.item()))#item将loss的数据类型转换为只有loss的值的数值型  
 writer.add\_scalar("train\_loss",loss.item(),total\_train\_step)  
 #测试步骤  
 model.eval()#很多算法中模型在训练之前会调用eval函数  
 total\_test\_loss=0#往往训练需要求本次epoch训练的模型在所有测试集上的loss  
 total\_test\_accuracy=0  
 with torch.no\_grad():#在已有的模型基础上设置梯度为0，即不再调优  
 for data in test\_dataloader:  
 imgs,targets=data  
 if torch.cuda.is\_available(): # 判断电脑是否可用cuda，即是否有gpu  
 imgs=imgs.cuda()#input加入cuda  
 targets=targets.cuda()#target加入cuda  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 total\_test\_loss+=loss.item()  
 #求准确率  
 accuracy=(outputs.argmax(1)==targets).sum()  
 total\_test\_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\_test\_accuracy/test\_data\_size))  
 total\_test\_step += 1  
 writer.add\_scalar("test\_loss",total\_test\_loss,total\_test\_step)  
 writer.add\_scalar("test\_accuarcy",total\_test\_accuracy/test\_data\_size,total\_test\_step)  
 #保存每epoch轮的模型  
 torch.save(model,"model\_{}.pth".format(i))#这里是方式一保存模型  
 #方式二torch.save(model.state\_dict(),"model\_{}.pth".format(i))  
 print("模型已保存")  
writer.close()

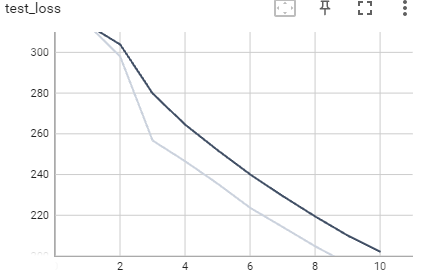
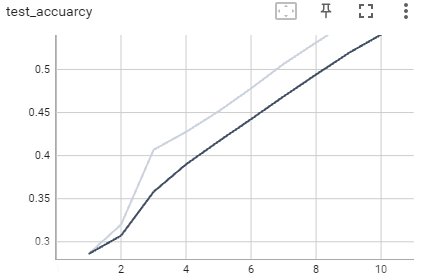
使用colab运行，但是由于colab不能定位到本地tensorboard，因此需要在代码前加入一句

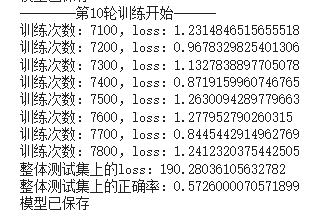
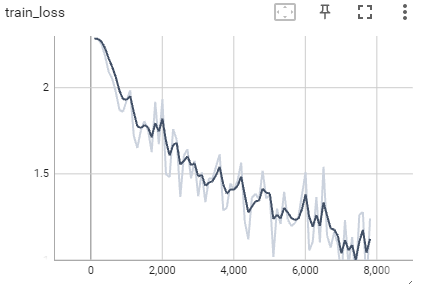
%load\_ext tensorboard

在代码执行后加入一句

%tensorboard --logdir logs\_train\_test

即可在编辑页面里直接输出tensorboard





## 使用.to(decive)调用gpu

Device=torch.device(“cpu”) |torch.device(“cuda”)

当有多个显卡时 torch.device(“cuda:0”)| torch.device(“cuda:1”)

然后 [网络模型|数据（input，target）|损失函数].to(device)

P31.py

import torch  
import torchvision  
from torch import nn  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
  
#定义训练的设备  
device=torch.device("cpu")  
#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)  
#求数据集长度  
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))  
  
#加载为dataloader数据集  
train\_dataloader=DataLoader(train\_data,batch\_size=64)  
test\_dataloader=DataLoader(test\_data,batch\_size=64)  
#搭建神经网络  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, 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(),  
 nn.Linear(64\*4\*4,64),  
 nn.Linear(64,10)  
 )  
 def forward(self,x):  
 x=self.model(x)  
 return x  
  
#创建网络模型  
model=Model()  
model.to(device)  
#损失函数  
loss\_fn=nn.CrossEntropyLoss()#交叉熵损失  
loss\_fn.to(device)  
#优化器  
learning\_rate=1e-2 #1\*10^(-2)=0.01  
optimizier=torch.optim.SGD(model.parameters(),lr=learning\_rate)#SGD优化器  
#设置训练网络的一些参数  
total\_train\_step=0#记录训练的次数  
total\_test\_step=0#记录测试的次数  
epoch=10#训练的轮次  
  
#添加tensorboard  
writer=SummaryWriter("logs\_train\_test")  
#开始训练  
for i in range(epoch):  
 print("--------第{}轮训练开始------".format(i+1))  
 #训练步骤开始  
 model.train()#很多算法中模型在训练之前会调用train函数  
 for data in train\_dataloader:  
 imgs,targets=data  
 imgs=imgs.to(device)  
 targets=targets.to(device)  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 #优化器优化模型  
 optimizier.zero\_grad()  
 loss.backward()  
 optimizier.step()  
  
 total\_train\_step+=1  
 if total\_train\_step%100==0:#每训练100步才打印一次loss，防止输出太多太乱  
 print("训练次数：{}，loss：{}".format(total\_train\_step,loss.item()))#item将loss的数据类型转换为只有loss的值的数值型  
 writer.add\_scalar("train\_loss",loss.item(),total\_train\_step)  
 #测试步骤  
 model.eval()#很多算法中模型在训练之前会调用eval函数  
 total\_test\_loss=0#往往训练需要求本次epoch训练的模型在所有测试集上的loss  
 total\_test\_accuracy=0  
 with torch.no\_grad():#在已有的模型基础上设置梯度为0，即不再调优  
 for data in test\_dataloader:  
 imgs,targets=data  
 imgs=imgs.to(device)  
 targets=targets.to(device)  
 outputs=model(imgs)  
 loss=loss\_fn(outputs,targets)  
 total\_test\_loss+=loss.item()  
 #求准确率  
 accuracy=(outputs.argmax(1)==targets).sum()  
 total\_test\_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\_test\_accuracy/test\_data\_size))  
 total\_test\_step += 1  
 writer.add\_scalar("test\_loss",total\_test\_loss,total\_test\_step)  
 writer.add\_scalar("test\_accuarcy",total\_test\_accuracy/test\_data\_size,total\_test\_step)  
 #保存每epoch轮的模型  
 torch.save(model,"model\_{}.pth".format(i))#这里是方式一保存模型  
 #方式二torch.save(model.state\_dict(),"model\_{}.pth".format(i))  
 print("模型已保存")  
writer.close()

# 14.完整的模型验证套路

测试，demo：利用已经训练好的模型为他提供输入

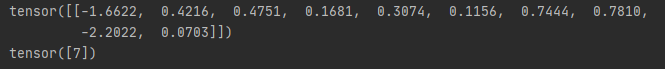
本例使用之前用数据集CIFAR10训练得到的分类模型去测试随便在网络上找到的一张修狗的照片，用来验证分类是否成功

使用第一次训练得到的模型model\_1，有

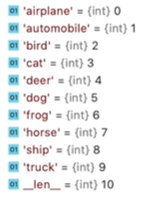
P32test.py

import torch  
import torchvision  
from PIL import Image  
from torch import nn  
  
image\_path="./img/55.png"  
image=Image.open(image\_path)  
image=image.convert('RGB')#png图像是4通道，多了一个透明通道，需要转换为RGB的三通道  
print(image)  
  
transform=torchvision.transforms.Compose([torchvision.transforms.Resize((32,32)),#resize图像大小  
 torchvision.transforms.ToTensor()])#将PIL的image格式转换为tensor格式  
image=transform(image)  
print(image.shape)  
  
#拷贝网络模型  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, 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(),  
 nn.Linear(64\*4\*4,64),  
 nn.Linear(64,10)  
 )  
 def forward(self,x):  
 x=self.model(x)  
 return x  
model=torch.load("model\_0.pth")#加载训练得到的网络模型  
print(model)  
image=torch.reshape(image,(1,3,32,32))#模型输入需要是4维，加入一个batch\_size  
model.eval()  
with torch.no\_grad():  
 output=model(image)  
print(output)#输出结果是0.7810比较大  
print(output.argmax(1))

输出：



说明下标7是模型分类的结果，在CIFAR10的分类中，下标代表如下

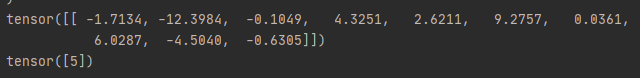
有图知，model\_1的训练模型将dog预测为hourse（🐎？），预测错误

当使用gpu训练得到的model\_29的模型进行预测

P32\_2.py

import torch  
import torchvision  
from PIL import Image  
from torch import nn  
  
image\_path="./img/55.png"  
image=Image.open(image\_path)  
image=image.convert('RGB')#png图像是4通道，多了一个透明通道，需要转换为RGB的三通道  
print(image)  
  
transform=torchvision.transforms.Compose([torchvision.transforms.Resize((32,32)),#resize图像大小  
 torchvision.transforms.ToTensor()])#将PIL的image格式转换为tensor格式  
image=transform(image)  
print(image.shape)  
  
#拷贝网络模型  
class Model(nn.Module):  
 def \_\_init\_\_(self):  
 super(Model, 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(),  
 nn.Linear(64\*4\*4,64),  
 nn.Linear(64,10)  
 )  
 def forward(self,x):  
 x=self.model(x)  
 return x  
model=torch.load("model\_29\_gpu.pth",map\_location=torch.device('cpu'))#使用gpu训练得到的数据或模型想在cpu上运行需要进行设一次映射  
print(model)  
image=torch.reshape(image,(1,3,32,32))#模型输入需要是4维，加入一个batch\_size  
model.eval()  
with torch.no\_grad():  
 output=model(image)  
print(output)#输出结果是9.2757比较大  
print(output.argmax(1))

输出：



发现结果预测正确，说明获得的模型准确率较高