import torch

from torch import nn

import torchvision

环境配置

## GPU

import torch

if torch.cuda.is\_available():

device = torch.device("cuda")

print('There are %d GPU(s) available.' % torch.cuda.device\_count())

print('We will use the GPU')

else:

print('No GPU available, using the CPU instead.')

device = torch.device("cpu")

预处理

**调整tensor轴顺序**

例：x.permute(1, 2, 0) #表示将第0轴换到最内层，1,2轴外置

## 图片相关torchvision.transforms

Pytorch中图片格式默认为 **(channel数,高,宽)**

plt和tf默认为 **(高,宽,channel数)**

**读入图片**

x=torchvision.io.read\_image(path)

**旋转图片**

torchvision.transforms.functional.rotate(x,角度,expand=True) #绕中心旋转

#expand=False则维持旋转前的高\*宽，对空缺填充黑色，对超出部分截断

**图片Resize大小**

resize=torchvision.transforms.Resize((高,宽))

x=resize(x)

**转灰度图**

transform=torchvision.transforms.Grayscale(3) #返回通道数取1或3

x=transform(x)

**Padding**

x=torchvision.transforms.functional.pad(x,padding,\*fill=0,\*padding\_mode=’constant’)

#padding为padding像素值，可以传入1个值或长为2或4的list，对应四条边

#fill为填充颜色，从0(黑)至255(白)

#padding\_mode=[‘constant’,’edge’,’reflect’,’symmetric’],edge为条纹填充, relect和symmetric为利用图片自身的镜像来进行填充

**调整亮度/色差/饱和度/色相**

jitter=torchvision.transforms.ColorJitter(brightness=0,contrast=1,saturation=0.3,hue=0.5)

x=jitter(x)

**区域裁剪**

torchvision.transforms.functional.crop(x,top,left,height,width)

#top和left指定裁剪区域位置；height和width指定裁剪区域大小

**中心裁剪**

crop=torchvision.transforms.CenterCrop(size=(height,width))

x=crop(x)

**图片旋转**

rotate=torchvision.transforms.RandomRotation(degrees=(min,max),fill=0,expand=False,\*center=None)

x=rotate(x)

#fill为填充颜色

#expand为旋转后是否囊括原图全图

#center指定旋转中心

**图层色标准化**

x=x.permute(1,2,0)-torch.mean(x,dim=(1,2),dtype=float)

x=torch.div(x,torch.std(x.double(),dim=(1,2)))

**锐化**

torchvision.transforms.RandomAdjustSharpness()

**反转图像颜色**

torchvision.transforms.RandomInvert(p=0.5)

**转化为PIL(Python Imaging Library)图像**

transform=torchvision.transforms.ToPILImage()

x=transform(x)

x.show()

**Transform Pipeline**

transforms = torch.nn.Sequential(torch.transform1, torch.transform2,…)

scripted\_transforms = torch.jit.script(transforms)

x=scripted\_transforms(x)

**随机图像增强**

transform=torchvision.transforms.AutoAugment(fill=255) #方法1

transform=torchvision.transforms.RandAugment(num\_ops=随机操作数) #方法2

x=transform(x)

**图像加方框**

box=torch.tensor([左上x,左上,右下x,右下y],dtype=torch.int).unsqueeze(0)

x=torchvision.utils.draw\_bounding\_boxes(x,boxes=box,colors="red",width=3)

x=torchvision.transforms.ToPILImage()(x)

plt.imshow(x)

## 视频相关

**读取视频**

video\_frames,\_,metadata=torchvision.io.read\_video("FILE.mp4",pts\_unit,start\_pts,end\_pts)

#pts\_units=’pts’/’sec’，分别代表以帧为基本单位或以秒为基本单位

#start\_pts=开始读取的帧数或秒数

#end\_pts=结束读取的帧数或秒数

#video\_frames: 由视频读取而来的由图片构成的张量[帧数,图高,图宽,channel数]

#metadata：包含了视频帧率和音轨帧率信息

**抽帧**

写个for循环从video\_frames里面抽

Pytorch框架

import torch

from torch import nn

from torchvision.transforms import ToTensor

device='cpu'/’cuda’

#一切自定义的模型架构都以nn.Module为父类创建

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

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

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

self**.**flatten **=** nn**.**Flatten()

#使用nn.Sequential来定义各层

self**.**linear\_relu\_stack **=** nn**.**Sequential(

nn**.**Linear(10000, 16), #各层的两个参数为单样本输入shape和输出shape

nn**.**ReLU(), #激活函数层

nn**.**Linear(16, 16),

nn**.**ReLU(),

nn**.**Linear(16,2),

nn**.**Sigmoid()

)

**def** forward(self, x): #forward方法是nn.Module中前项传播的调用方法

x **=** self**.**flatten(x)

logits **=** self**.**linear\_relu\_stack(x)

**return** logits

model **=** NeuralNetwork()**.**to('cpu') #将模型加载到CPU/cuda

print(model)

张量torch.Tensor

**初始化张量**

tensor = torch.tensor(list或ndarray) #从list或ndarray生成张量

tensor = torch.from\_numpy(ndarray) #从ndarray生成张量

tensor = tensor.numpy() #转为ndarray

**生成张量**

torch.rand(shape) #随机张量

torch.ones(shape) #全1张量

torch.zeros(shape) #全0张量

torch.ones\_like(tensor) #同形状的全1张量

torch.rand\_like(tensor) #同形状的随机张量

**属性**

tensor.shape #张量形状

tensor.dtype #张量数据类型

tensor.device #张量存储设备位置

**计算**

torch.mean(x,dim=计算均值面向的轴元组,dtype=float) #均值

torch.std(x.double(),dim=计算均值面向的轴元组) #标准差

torch.add(x1,x2) #加

torch.sub(x1,x2) #减

torch.mul(x1,x2) #乘

torch.div(x1,x2) #除

进行四则运算前，需要用permute把要动的轴换到-1位，如通道标准化时为**(高,宽,channel数)**

**将张量存储位置及操作转移至GPU**

if torch.cuda.is\_available():

tensor = tensor.to("cuda")

**张量拼接**

tensor=torch.cat([tensor, tensor, tensor], dim=k)

张量的切片、四则运算等各种操作与np.ndarray类似

**auto-grad引擎**

#requires\_grad参数使得张量在计算时可以启用pytorch内的求导引擎

e.g torch.randn(3, requires\_grad=True)

层

**线性模型**

self.model= nn.Sequential(

层1

层2

…

)

**线性层(全连接层)**

layer=nn.Linear(in\_features=输入单样本形状, out\_features=输出单样本形状)

layer.weight #权重

layer.bias #偏置

**二维卷积**

nn.Conv2d(3, 6, 5)

**最大池化**

nn.MaxPool2d(2, 2)

**平化层**

nn.Flatten()

**Relu层**

nn.ReLU()

**Softmax层**

nn.Softmax(dim=1) #dim指示和为1的维度

损失函数

**反向传播**

loss\_function.backward()

**二元交叉熵**

torch.nn.functional.binary\_cross\_entropy\_with\_logits(probs,labels)

优化器

**优化器梯度零初始化**

optimizer.zero\_grad()

**RMSProp**

torch.optim.RMSprop()

操作标示

model.train() #指示模型接下来会进行训练操作，所有计算会修改模型参数

model.eval() #指示模型接下来会进行验证/测试操作，所有计算不会修改模型参数