1. 安装Anaconda3环境

Pip是 python 官方推荐的包下载工具，但是只能安装python包

Conda是一个跨平台（支持linux, mac, win）的通用包和环境管理器，它除了支持python外，还能安装各种其他语言的包，例如 C/C++, R语言等

1. 安装显卡驱动+CUDA Toolkit

1.深度学习离不开显卡，Tensorflow和Pytorch都只支持Nvidia显卡，显卡主要起到一个训练加速的作用

2.检查显卡驱动是否安装->任务管理器->性能选项卡->是否正常显示GPU型号(GPU0)

3.先卸载旧的Nvidia显卡驱动，再在Windows安全模式(win+R->msconfig->引导选项卡)下安装新的Nvidia显卡驱动

（可不选）4.安装显卡驱动相对应的cuda toolkit/cudnn(替代步骤3)

*<https://www.bilibili.com/video/BV1Rz411e7eJ/?t=356>*

CUDA：为“GPU通用计算”构建的运算平台。

NVCC 是CUDA的编译器，只是 CUDA Toolkit 中的一部分

CUDA Toolkit (nvidia)： CUDA完整的工具安装包，其中提供了 Nvidia 驱动程序、开发 CUDA 程序相关的开发工具包等可供安装的选项。包括 CUDA 程序的编译器、IDE、调试器等，CUDA 程序所对应的各式库文件以及它们的头文件。

CUDA Toolkit (pytorch)： CUDA不完整的工具安装包，其主要包含在使用 CUDA 相关的功能时所依赖的动态链接库。不会安装驱动程序。

5.相关命令:

1). nvidia-smi /nvidia-smi -L 显示的是CUDA的驱动版本

2). nvcc -V / nvcc –version 显示的是CUDA Toolkit的版本（CUDA的运行时版本）

3). CUDA Toolkit版本小于等于驱动版本都可以正常使用

4). 常见错误:CUDA driver version is insufficient for CUDA runtime version

1. 配置深度学习环境

1.创建虚环境

1). conda create -n regcn python=3.7

# 构建虚拟环境，名为：tf，对应的python版本号（可以换）

conda remove --name regcn --all

# 删除虚拟环境

conda env list

conda info --envs

# 查看虚拟环境列表

conda clean -p

#删除没有用的包

conda clean -a

# 推荐，直接把所有没用的包全部删掉

2). conda activate tf

# 切换到创建的虚拟环境：tf

3). pip list/conda list

# 查看虚拟环境中已安装的工具包/+显示关联环境下的已安装的包

4). pip install 包名 -i 镜像连接

# 配置其他环境

pip install --upgrade pip

# 升级pip

pip install --upgrade pip setuptools wheel

# 升级pip工具包

5). conda镜像源配置

conda config --add channels 镜像地址

# 添加镜像源地址

conda config --remove channels 镜像源地址

# 删除镜像源地址

conda config --remove-key channels

# 镜像源地址恢复默认设置

conda config --show

# 显示已有配置信息（包括镜像源）

清华镜像源：

*<https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/>*

*<https://pypi.tuna.tsinghua.edu.cn/simple>*

*eg：安装opencv-python（得对应python版本，python3.6.2—opencv-python-4.3.0.38）*

pip install opencv-python -i <https://pypi.tuna.tsinghua.edu.cn/simple>

中科大镜像源

*[https:](https://mirrors.ustc.edu.cn/anaconda/pkgs/free/)*[//mirrors.ustc.edu.cn/anaconda/pkgs/free/](https://mirrors.ustc.edu.cn/anaconda/pkgs/free/)

6).安装CUDA Toolkit (pytorch)

conda install cudatoolkit =对应的版本如10.1 -c conda-forge

#-c conda-forge是指明在库conda-forge中下载cudatoolkit 10.1这个pkg。

2.安装pytorch GPU版本

1).检查Gpu是否支持CUDA

设备管理器->显示适配器->CUDA|支持的GPU|GeForce

nvidia-smi 查看显卡是否支持CUDA

nvidia-smi -L查看显卡型号

2).Nvidia显卡驱动若版本过低，则需要更新驱动

3).安装pytorch GPU

1.离线下载 清华大学开源软件镜像站（下载的tar.bz2文件->conda下的pkgs文件夹）

2.pip install torch==1.3.0+cu92 -f <https://download.pytorch.org/whl/torch_stable.html>

#安装pytorch下载慢可以打开后⾯。链接直接下载cu92对应的是cuda toolkit9.2 -f给了⼀个源，速度会快，+cu表⽰安装torch GPU版本的（torchvision==0.4.1）

4).检查是否安装成功

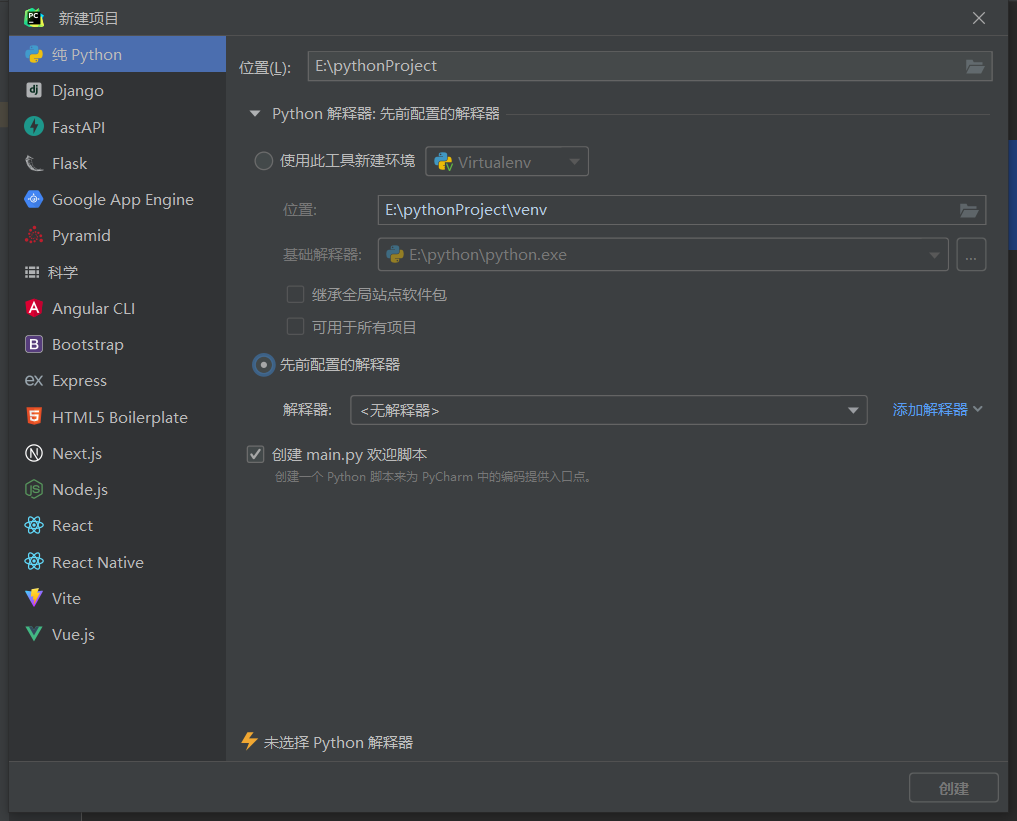
import torch

print(torch.cuda.is\_available()) // True

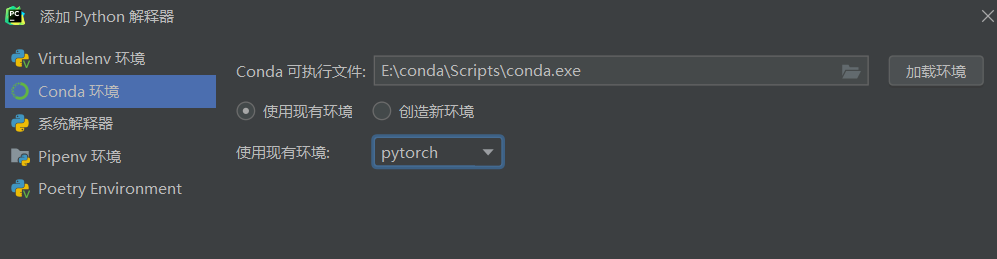
1. pycharm和jupyter交互式环境搭建

1.pycharm新建工程

新建python项目->先前配置的解释器



选择Conda环境->使用现有环境



使用Python控制台进行交互,并在文件->设置->项目->Python解释器可以查看conda软件管理器、pip软件管理器。

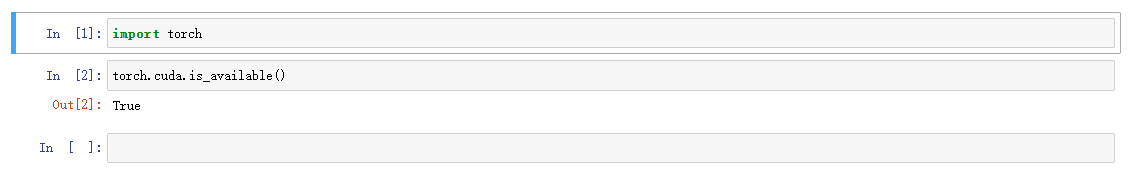
2.jupyter环境搭建

方式1).在base环境中，再安装一次pytorch环境

方式2).在虚拟环境下安装nb\_conda（conda install nb\_conda）

pytorch环境->jupyter notebook->python3->New Python[conda env:pytorch]

->（.ipynb后缀文件)



tip:1.torch.cuda.is\_available()返回false?

gpu是否支持cuda

2.pip 安装从清华镜像源下载到conda下的pkgs文件夹下的安装包?(===三3).1)

conda install --use-local 包名(===三2).1)

3.couldn't call conda: [WinError 2] 系统找不到指定的文件

pip uninstall python-dateutil

pip install python-dateutil -i <https://pypi.tuna.tsinghua.edu.cn/simple>

1. Python学习中的两大法宝函数

dir():打开,看见

dir(pytorch) ======输出:1、2、3、4

dir(pytorch.3)======输出:a、b、c

help():说明书

help(pytorch.3.a)======将此扳手放在特定地方,然后拧动

eg:

1.dir(torch.cuda.is\_available)

#torch.cuda.is\_available是函数地址（相当于指针）

# ['\_\_annotations\_\_','\_\_call\_\_','\_\_class\_\_','\_\_closure\_\_'，…]

#带有两个下划线开头的函数是声明该方法为私有,不能在类地外部被使用或直接访问

PIL.Image的\_size属性

#带有单个下划线开头的属性是声明该属性为私有,却能在类地外部被使用或直接访问

2.help(torch.cuda.is\_available)

# Help on function is\_available in module torch.cuda:is\_available() Returns a bool indicating if CUDA is currently available

1. Pycharm和Jupyter使用及对比

Python文件：代码是以块为一个整体运行的

Python控制台：代码是以每一行为块运行的

Jupyter文件：代码是以任意行为块运行的

1. PyTorch加载数据初认识

常用的数据集形式(train训练数据集、val验证数据集)

A.数据(xxx\_images文件夹)+标签（xxx\_labels文件夹)



B.数据(文件夹中的图片)+标签(文件夹的名称)



1.Dataset提供一种方式去获取数据及其Label

1).如何获取每一个数据及其Label

2).统计总共有多少的数据

from torch.utils.data import Dataset  
from PIL import Image  
import os  
  
  
class MyData(Dataset):  
 def \_\_init\_\_(self, root\_dir, label\_dir):  
 self.root\_dir = root\_dir  
 self.label\_dir = label\_dir  
 self.path = os.path.join(self.root\_dir, self.label\_dir)  
 self.img\_path = os.listdir(self.path)  
  
 def \_\_getitem\_\_(self, idx):  
 img\_name = self.img\_path[idx]  
 img\_item\_path = os.path.join(self.path, 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 = 'dataset/train'  
ants\_label\_dir = 'ants'  
bees\_label\_dir = 'bees'  
ants\_data = MyData(root\_dir, ants\_label\_dir)  
bees\_data = MyData(root\_dir, bees\_label\_dir)  
train\_data = ants\_data + bees\_data

补充：

1.Python的magic方法

img,label=train\_data.\_\_getitem\_\_(123) <===> img,label=train\_data[123]

some\_object.\_\_len\_\_()<===> len(some\_object)

2.DataLoader为后续的网络提供不同的数据形式

1. Tensorboard的使用
2. pip install tensorboard
3. tensorboard --logdir=logs --port=6010会在根目录下的logs文件夹生成日志文件
4. tensorboard绘制损失函数与变量之间的关系

I.add\_scalar()

*Args:  
 tag (string): Data identifier  
 scalar\_value (float or string/blobname): Value to save  
 global\_step (int): Global step value to record*

Eg:

from torch.utils.tensorboard import SummaryWriter  
writer = SummaryWriter('logs')  
# writer.add\_image()  
for i in range(100):  
 writer.add\_scalar('y=2x', 5\*i, i)  
writer.close()

II.add\_image()

*Args:  
 tag (string): Data identifier  
 img\_tensor (torch.Tensor, numpy.array, or string/blobname): Image data  
 global\_step (int): Global step value to record*

Eg:

from torch.utils.tensorboard import SummaryWriter  
from PIL import Image  
import numpy as np  
writer = SummaryWriter('logs')  
img\_path = 'dataset/train/ants/0013035.jpg'  
img\_PIL = Image.open(img\_path)  
img\_array = np.array(img\_PIL)  
writer.add\_image('test', img\_array, 1, dataformats='HWC')

img\_path = 'dataset/train/bees/16838648\_415acd9e3f.jpg'  
img\_PIL = Image.open(img\_path)  
img\_array = np.array(img\_PIL)  
writer.add\_image('test', img\_array, 2, dataformats='HWC')  
writer.close()

tip:

1.add\_image中的dataformats默认是CHW

2.numpy的一些基本操作

np.array:将数据(列表、图片00)转化为矩阵,返回ndarray类型数据

1. torchvision中的transforms

transforms中常见的类：

1).Compose(object)

Composes several transforms together.

Example:  
transforms.Compose([transforms.CenterCrop(10),transforms.ToTensor()])

#中心裁剪+ToTensor()

2).ToTensor(object)

Convert a ``PIL Image`` or ``numpy.ndarray`` to tensor.

#1.是将输入的数据shape H，W，C(ndarray) ——> C，H，W

#2.将所有数除以255(transforms.ToTensor会进行正则化)，将数据归一化到[0，1]

#3.归一化在训练神经网络时候，可以压缩值域，防止梯度爆炸

3).Normalize(object)

Normalize a tensor image with mean and standard deviation. Given mean: ``(M1,...,Mn)`` and std: ``(S1,..,Sn)`` for ``n`` channels, this transform will normalize each channel of the input ``torch.\*Tensor`` i.e. ``input[channel] = (input[channel] - mean[channel]) / std[channel]

#1.mean是均值,std是标准差

#2.规范化input[channel]=(input[channel]-mean[channel])/std[channel]

4). ToPILImage(object)

Convert a tensor or an ndarray to PIL Image

#mode可以为RGBA,RGB,LA…

5).其他类-Resize、Scale、CenterCrop、Pad

transforms的使用

1).通过PIL.Image获得tensor图像

from PIL import Image  
from torchvision import transforms  
import numpy  
  
img\_path = 'dataset/train/ants/0013035.jpg'  
img = Image.open(img\_path)  
img\_np = numpy.array(img)  
print(type(img\_np))// <class 'numpy.ndarray'>  
tensor\_trans = transforms.ToTensor()  
tensor\_img = tensor\_trans(img)  
print(tensor\_img)//获得一个三维矩阵 RGB三通道

补充:

1. \_\_call\_\_函数：使得类实例对象可以像调用普通函数那样使用。（图片的tensor格式）

2. \_\_init\_\_函数：创建一个类的实例化，每当创建一个类的实例对象python解释器都会主动去调用它（使用此函数第一个参数必须为self）。

2).通过OpenCV获得tensor图像

from torch.utils.tensorboard import SummaryWriter  
from torchvision import transforms  
import cv2  
  
img\_path = 'dataset/train/bees/16838648\_415acd9e3f.jpg'  
img = cv2.imread(img\_path)  
writer = SummaryWriter('logs')  
tensor\_trans = transforms.ToTensor()  
img\_tensor = tensor\_trans(img)  
writer.add\_image('img', img\_tensor, 2)  
writer.close()

Image.open()——PIL Image、ToTensor()——tensor、cv.imread()——numpy.ndarray

3).通过Normalize来规范化区间

from torch.utils.tensorboard import SummaryWriter  
from torchvision import transforms  
import cv2  
  
img\_path = 'images/pytorch.jpg'  
img = cv2.imread(img\_path)  
writer = SummaryWriter('logs')  
# Normalize

trans\_totensor = transforms.ToTensor()  
img\_tensor = trans\_totensor(img)  
print(img\_tensor[0][0][0])  
trans\_norm = transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])  
img\_norm = trans\_norm(img\_tensor)  
writer.add\_image('Normalize', img\_norm)  
print(img\_norm[0][0][0])  
writer.close()

4).通过Resize改变图像尺寸(返回值为PIL.Image.Image)

size类型为int,会进行最小边匹配+等比例放缩(PIL.Image的size为WH,Tensor默认模式为CHW)

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

writer = SummaryWriter('logs')  
img\_path = 'images/pytorch.jpg'  
img = Image.open(img\_path)  
trans\_totensor = transforms.ToTensor()  
# Resize  
print(img.size)// (474, 266)  
trans\_resize = transforms.Resize((512, 512))  
img\_PIL = trans\_resize(img)  
print(type(img\_PIL))// <class 'PIL.Image.Image'>  
img\_resize = trans\_totensor(img\_PIL)  
writer.add\_image('Resize', img\_resize)  
writer.close()

5).通过Compose组合各种transforms操作

from torch.utils.tensorboard import SummaryWriter  
from PIL import Image  
from torchvision import transforms  
  
writer = SummaryWriter('logs')  
  
img\_path = 'images/pytorch.jpg'  
img = Image.open(img\_path)  
# Compose-Normalize  
trans\_compose = transforms.Compose([  
 transforms.ToTensor(),  
 transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])  
])  
img\_norm = trans\_compose(img)  
writer.add\_image('Norm', img\_norm, 1)  
  
# Compose-Resize  
trans\_compose = transforms.Compose([  
 transforms.Resize((100, 100)),  
 transforms.ToTensor()  
])  
img\_resize = trans\_compose(img)  
writer.add\_image('Resize', img\_resize, 2)  
  
writer.close()

6).通过RandomCrop进行随机裁剪

（padding,pad\_if\_needed,padding\_mode,fill）先进行填充再进行随机裁剪

# RandomCrop  
trans\_random = transforms.RandomCrop((200, 50))  
trans\_totensor = transforms.ToTensor()  
trans\_compose = transforms.Compose([  
 trans\_random,  
 trans\_totensor  
])  
for i in range(10):  
 img\_crop = trans\_compose(img)  
 writer.add\_image('RandomCrop', img\_crop, i)  
writer.close()

7).总结使用方法

I.关注输入和输出类型（debug+print+jupyter notebook）

II.多看官方文档

III.关注方法需要的参数（ctrl+p）

1. torchvision中数据集的使用

基本用法:(数据集CIFAR10)

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)  
test\_set = torchvision.datasets.CIFAR10(root='./dataset', train=False, transform=dataset\_transform, download=True)  
  
writer = SummaryWriter(log\_dir='logs')  
for i in range(10):  
 img, target = test\_set[i]  
 writer.add\_image('test', img, i)  
writer.close()

torchvision.datasets.CIFAR10(root='./dataset',train=True,transform=dataset\_transform,download=True)参数的作用:

root:指定数据集存放的路径；

train:取值为True/False,指定数据集为训练集/测试集；

transform:对图片进行各种变换操作,并指定的图片的输出格式；

download:取值为True/False,指定数据集是否从网络上下载；

数据集,可以从迅雷上下载压缩包后放到工程下的datasets文件夹中。

1. DataLoader的使用

基本用法:

import torchvision  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
  
test\_data = torchvision.datasets.CIFAR10(root='dataset', train=False, transform=torchvision.transforms.Compose([  
 torchvision.transforms.ToTensor()  
]), download=True)  
  
test\_loader = DataLoader(dataset=test\_data, batch\_size=64, shuffle=True, num\_workers=0, drop\_last=False)  
  
writer = SummaryWriter('logs')  
for epoch in range(2):  
 step = 0  
 for data in test\_loader:  
 img, tar = data//data是数据类型<class 'list'>  
 writer.add\_images('epoch {}'.format(epoch), img, step)  
 step += 1  
 writer.close()//！！！缩进，避免一次性写入太多的数据

DataLoader(dataset, batch\_size=1, shuffle=None,  num\_workers=0)

dataset:自定义的数据集；

batch\_size:每次取多少个数据；

shuffle:数据集是否打乱；

num\_workers:加载数据采用单个/多个进程；

drop\_last:最后的几个数据是否舍弃；

补充:

1.Python允许使用元组、列表来返回(return)多个值

ls = ['a', 100, 3.1415926]  
a, b, c = ls  
print('a===>', a, 'b===>', b, 'c===>', c)

// a===> a b===> 100 c===> 3.1415926

1. 神经网络的基本骨架-nn.Module的使用

基本用法:

nn.Module是所有神经网络模型的父类;我们在自定义网络时,需要继承nn.Module类,并重新实现构造函数\_\_init\_\_()和方法forward(self,input)

技巧:

1.网络中一般将具有可学习参数的层(全连接层、卷积层等)放在构造函数\_\_init\_\_()中；

2.网络中一般将不具有可学习参数的层(如ReLU、dropout、BatchNormalization)放在非构造函数\_\_init\_\_()中,在forward()方法里面使用nn.functional来代替self；

3.forward()方法是必须要重写,是实现模型功能、实现各个层之间的连接关系的核心；继承父类的\_\_call\_\_()函数会调用子类的forward()方法。

基本模型:

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

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

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

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

super**().**\_\_init\_\_**()**

self**.conv1** **=** **nn.Conv2d(**1**,** 20**,** 5**)**

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

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

**x** **=** **F.relu(**self**.conv1(x))**

**return** **F.relu(**self**.conv2(x))**

import torch  
from torch import nn  
  
  
class Lynn(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 def forward(self, input):  
 output = input + 1  
 return output  
  
  
data\_tensor = torch.tensor(1)//是tensor而不是Tensor  
lynn = Lynn()  
print(lynn(data\_tensor))// tensor(2)

补充:

1.method(\*input: Any,\*\*kwargs: Any)中的input是列表形式的缺省函数,kwargs是字典形式的缺省函数

1. 神经网络的卷积层-Conv2D

基本模型

def conv2d(input: Tensor, weight: Tensor, bias: Optional[Tensor]=None, stride: Union[\_int, \_size]=1, padding: Union[\_int, \_size]=0)

input:输入图像的tensor格式；

weight:卷积核的tensor格式；

bias:偏置值；

stride:步长；

padding:填充

1. 卷积后的输出为单个层

参与卷积操作的input和kernel尺寸有严格要求

input——input tensor of shape(minibatch,in\_channels,iH,iW)

weight——filters of shape (out\_channels,in\_channels/groups,kH,kW)

bias——optional bias tensor of shape(out\_channels)

补充:

1.梯度下降法有着3种不同的形式(epoch的概念),批量梯度下降(BGD)、小批量梯度下降(MGD,减少对内存空间的占用、GPU并行处理）、随机梯度下降（SGD,batch\_size = 1,每次迭代时使用一个样本来对参数进行更新），只要是梯度下降法,就有迭代次数（源码定义）的概念;批量梯度下降稳定、迭代速度慢、可能停留在局部最优解;小批量梯度下降较稳定、迭代速度较快、可跳出局部最优解;随机梯度下降不稳定、迭代速度快、可跳出局部最优解。

2.基本概念:

epoch:当一个完整的数据集经过神经网络一次，并返回一次;

batch:当数据集很大的时候，对于每个epoch，很难将所有的数据集一次读入到内存中，这是需要将数据集分为几次读入，每次称为一个batch;

batch\_size:即batch中样本的数量。

3.神经网络参数个数与输入神经元、输出神经元的个数有关;卷积神经网络参数个数与输入神经元的个数（输入通道数）、输出神经元的个数（输出通道数）、卷积核（权重）的规模有关;

Eg: 假设有一个3×3大小的卷积层，其输入通道为16、输出通道为32,那么标准卷积层一共需要16\*3\*3\*32个参数

4.常见的卷积操作:标准卷积、分组卷积、转置卷积、逐点卷积、深度卷积、深度可分离卷积、空间可分离卷积等

分组卷积:需要指定conv2d中的groups参数(b站Eve的科学频道 演示分组、深度、深度可分离卷积);分组卷积层的参数数量是标准卷积层的参数数量的1/g(通过减少输入和输出之间的连接,而不是连接上的权重)

Eg:假设输入特征图尺寸为H1×W1×c1,卷积核尺寸为h1×w1×c1,输出的特征图尺寸为H2×W2×c2,标准卷积层的参数量为c1×h1×w1×c2;若将输入特征图按照通道数分成g组,则每组输入特征图的尺寸为H1×W1×(c1/g),对应的卷积核尺寸为h1×w1×(c1/g),每组输出特征图尺寸为H2×W2×(c2/g),分组卷积层一共需要(c1/g)×h1×w1×(c2/g)×g个参数

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))  
kernel = torch.reshape(kernel, (1, 1, 3, 3))  
print(input)  
print(kernel)  
  
output1 = F.conv2d(input, kernel, stride=1)  
print(output1)  
output2 = F.conv2d(input, kernel, stride=2)  
print(output2)  
output3 = F.conv2d(input, kernel, stride=1, padding=1)  
print(output3)

1. 卷积后的输出为多个层

class torch.nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding\_mode='zeros', device=None, dtype=None)

in\_channels:输入的通道数；

out\_channels:输出的通道数；

kernel\_size:卷积核尺寸, (*[int](https://docs.python.org/3/library/functions.html" \l "int" \o "(in Python v3.12)) or [tuple](https://docs.python.org/3/library/stdtypes.html" \l "tuple" \o "(in Python v3.12)), optional*)整型或元组；

stride=1:步长;

padding=0:填充；

dilation=1:空洞卷积(不常用)

groups=1:分组；

bias=True:偏置值;

padding\_mode=‘zeros’:填充模式;

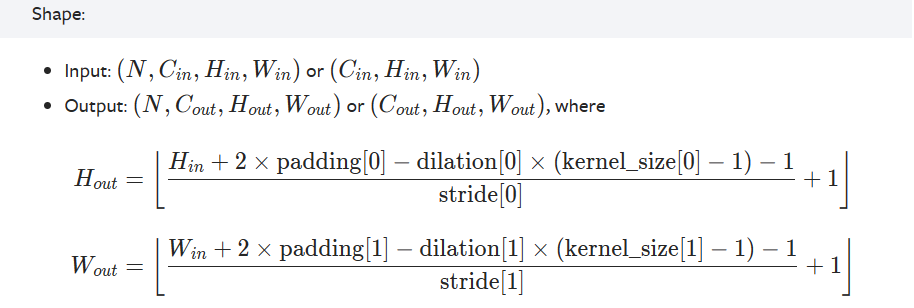
import torch  
from torchvision import transforms  
from torchvision import datasets  
from torch.utils.data import DataLoader  
from torch.utils.tensorboard import SummaryWriter  
import torch.nn as nn  
  
cifar10\_data = datasets.CIFAR10(root='./dataset', train=False, transform=transforms.Compose({  
 transforms.ToTensor()  
}), download=True)

class Lynn(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.conv1 = nn.Conv2d(3, 6, (3, 3))  
  
 def forward(self, input):  
 return self.conv1(input)  
  
lynn = Lynn()  
writer = SummaryWriter('./logs')  
cifar10\_dataloader = DataLoader(dataset=cifar10\_data, batch\_size=64, shuffle=True, drop\_last=False)  
  
step = 0  
for data in cifar10\_dataloader:  
 img, target = data  
 print(img.shape)  
 writer.add\_images(tag='input', img\_tensor=img, global\_step=step)  
 output = lynn(img)  
 print(output.shape)  
 output = torch.reshape(output, [-1, 3, 30, 30])  
 writer.add\_images(tag='output', img\_tensor=output, global\_step=step)  
 step += 1  
 writer.close()

补充:

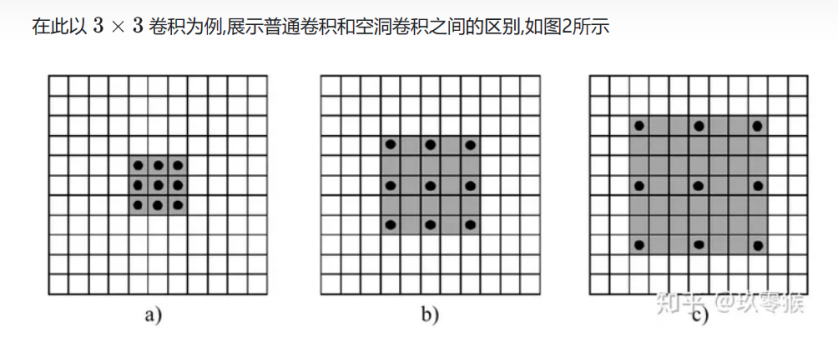
1.tensor的常用属性shape、ndim(维度)、grad(梯度)

2.卷积层输出特征图的宽高



3.感受野:在卷积神经网络中,特征图上某个元素的计算受输入图像(最初的输入图像,不是上一层图像)上某个区域的影响,这个区域即该元素的感受野,该区域的大小即该元素的感受野大小。（池化层的感受野较为特殊 b站 程序员本人 感受野介绍）

4.空洞卷积:dilation（默认值为1）、kernel-size（假设值为k），有两种实现方法,一是卷积核填充0;二是输入等间隔采样，作用是扩大感受野、捕获多尺度上下文信息(目标检测)，空洞卷积实际卷积核大小为k+(dilation-1)×(k-1)。



a是dilation rate=1的普通卷积,卷积后的感受野为3；

b是dilation rate=2的空洞卷积,卷积后的感受野为5；

c是dilation rate=3 的空洞卷积,卷积后的感受野为7

1. 神经网络的池化层-MaxPool2d

基本用法:

class torch.nn.MaxPool2d(kernel\_size, stride=None, padding=0, dilation=1, return\_indices=False, ceil\_mode=False)

kernel\_size:设置取最大值的一个窗口;

stride:步长,默认大小是kernel\_size;

padding:填充,默认是零填充

dilation:空洞窗口(不常用)

return\_indices:布尔类型,若为True,返回窗口中最大值的索引和最大值;默认为False

ceil\_mode: 布尔类型,若为True,会进行向上取整,当kernel窗口的范围超过输入图像的范围，会进行保留

最大池化的作用:是为了尽最大可能保留原有特征的条件下,减少参数的数量,降低模型的复杂度,模型训练的速度会越快。

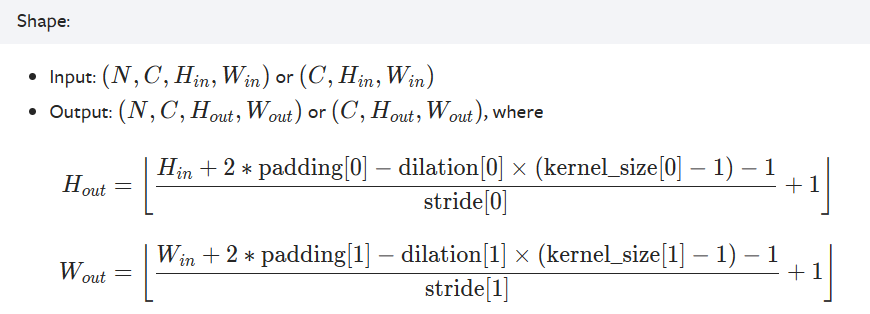
import torch  
import torch.nn as nn  
from torchvision import datasets, transforms  
from torch.utils.data import dataloader  
from torch.utils.tensorboard import SummaryWriter  
  
# 第一段程序  
input = torch.tensor([  
 [1, 2, 0, 3, 1],  
 [0, 1, 2, 3, 1],  
 [1, 2, 1, 0, 0],  
 [5, 2, 3, 1, 1],  
 [2, 1, 0, 1, 1]  
], dtype=torch.float)  
input = torch.reshape(input, (1, 1, 5, 5))  
  
  
class Tudui(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.maxpool1 = nn.MaxPool2d(kernel\_size=3, ceil\_mode=True)  
  
 def forward(self, input):  
 return self.maxpool1(input)  
  
  
tudui = Tudui()  
output = tudui(input)  
print(output)  
//输出

tensor([[[[2., 3.],

[5., 1.]]]])  
  
# 第二段程序  
cifar10\_datasets = datasets.CIFAR10(root='./dataset', train=False, transform=transforms.Compose([  
 transforms.ToTensor()  
]), download=True)  
  
cifar10\_dataloader = dataloader.DataLoader(dataset=cifar10\_datasets, batch\_size=64, shuffle=True, drop\_last=True)  
  
writer = SummaryWriter(log\_dir='./logs')  
  
index = 0  
for data in cifar10\_dataloader:  
 imgs, labels = data  
 writer.add\_images(tag='imgs', img\_tensor=imgs, global\_step=index)  
 output = tudui(imgs)  
 writer.add\_images(tag='output', img\_tensor=output, global\_step=index)  
 writer.close()  
 index += 1

补充:

1.池化层输出特征图的宽高:



2.torch的常用方法:torch.tensor()、torch.reshape()、torch.ones()(创建全1的张量)、tensor.zero()（创建全0的张量）、torch.flatten()、torch.relu()(将多维数据展平为一维数据)

3.最大池化的输入,必须指定dtype=torch.float,否则编译器会误以为为Float类型

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.float)

1. 神经网络的激活函数-非线性激活函数

基本用法:

class torch.nn.ReLU(inplace=False)

inplace:选择是否进行覆盖运算

Eg:1.inplace=True计算可以节省内存,同时还可以省去反复申请和释放内存的时间,但是会对原变量覆盖。

input=-1 input=-1

nn.ReLU(input,inplace=True) output=nn.ReLU(input,inplace=False)

input=0 input=-1

output=0

对从上层网络Conv2d中传递下来的tensor直接进行修改，这样能够节省运算内存，不用多存储其他变量

非线性激活函数ReLU的定义:ReLU(x)=(x)+=max(0,x);非线性激活函数的作用,网络的逼近能力较强,几乎可以逼近任意函数,同时可以缓解梯度消失的问题。（假设激活函数是线性激活函数f(x)=x,该情况下每一层节点的输入都是上层输出的线性函数,无论神经网络有多少层,输出都是输入的线性组合,与没有隐藏层效果相当,该情况就相当于是最原始的感知机,网络的逼近能力就相当有限。）

ReLU函数存在的问题,死亡ReLU(输入小于0时,ReLU函数恒为0,神经元无法学习(ReLU函数输出值为0时,ReLU的导数也为0)),此时可以使用LeakyReLU函数f(x)=max(0,x)+rmin(0,x)来避免。

class torch.nn.Sigmoid(\*args, \*\*kwargs)

非线性激活函数 Sigmoid 的定义:, Sigmoid函数可以将输入值映射到0到1之间，使神经网络的输出具有“饱和”性质，从而增加模型的非线性表达能力,但是会导致在训练过程中权重的微小变化不会太大地改变净输入,在激活后节点的值仍为0-1.0,训练停滞不前或动作非常缓慢。

1.ReLU函数

import torch  
from torch import nn  
from torch.nn import ReLU  
  
input = torch.tensor(data=[  
 [1, -0.5],  
 [-1, 3]  
], dtype=torch.float)  
input = torch.reshape(input, (-1, 1, 2, 2))  
  
  
# 1.ReLU中 inplace=False(默认情况)  
class Tudui2(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.relu = ReLU()  
  
 def forward(self, input):  
 output = self.relu(input)  
 return output  
  
  
tudui = Tudui2()  
output = tudui(input)  
print(output)  
  
  
# 2.ReLU中 inplace=True  
class Tudui1(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.relu = ReLU(inplace=True)  
  
 def forward(self, input):  
 self.relu(input)  
 return input  
  
  
tudui = Tudui1()  
output = tudui(input)  
print(output)

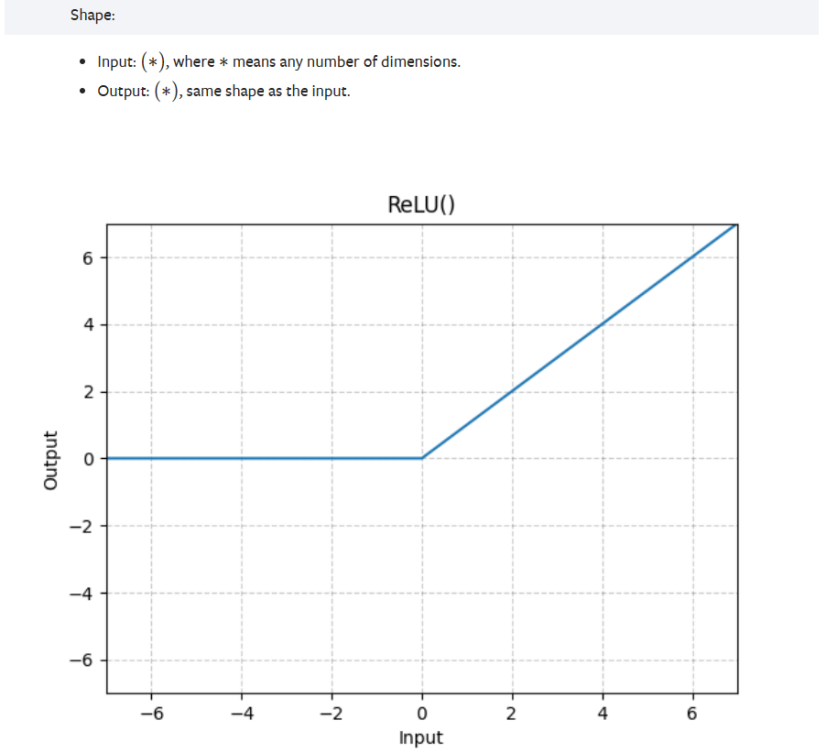
补充:

1.线性映射:是一个函数,而不是方程,该映射要同时满足两个条件:

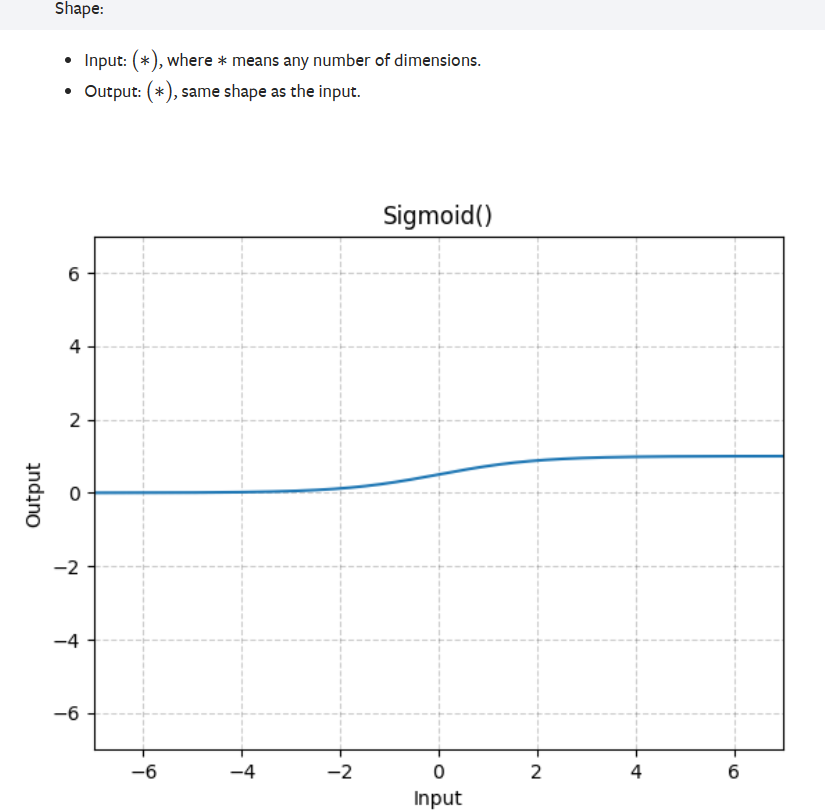
1)可加性 f(x+y)=f(x)+f(y)、2)齐次性f(ax)=af(x)

2.可以通过组合多个ReLU函数来拟合任意一个复杂的非线性函数

3.



4.



1. 神经网络-线性层及其他层

1.归一化层:不常用,在模型训练前,一般会对输入数据进行归一化处理,将数据中不同尺度的特征转换为相同的尺度,从而提升模型的泛化能力,加速模型训练,并防止梯度爆炸等问题,这使得数据在进行ReLU之前不会因为数据过大而导致网络性能的不稳定(BN层一般在Pooling池化层之后,ReLU激活函数之前)。（b站 小黑黑讲AI）

二维批量归一化BatchNorm2d:

BatchNorm2d在计算时,会针对每个通道进行,计算m个输入数据的平均值和方差,进行归一化操作 (是一个非常小的正数,用来保证数值的稳定性,防止计算过程中出现除以0的情况),再对归一化后的数据进行缩放平移BN()=(是BatchNorm2D层中的可训练参数)

class torch.nn.BatchNorm2d(num\_features, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True, device=None, dtype=None)

num\_features:特征数,对应输入张量x中的通道数3；

2.循环层:特定场景下使用,(RNN、LSTM主要用于文字识别)

循环神经网络,不同于普通的神经网络,循环神经网络的隐藏层中的神经元除了将数据发送给输出层,还会将数据重新发回到隐藏层;RNN可以捕捉并存储过去的输入信息,因此RNN在处理序列任务时,有着很大的优势。

class torch.nn.RNN(input\_size, hidden\_size, num\_layers=1, nonlinearity='tanh', bias=True, batch\_first=False, dropout=0.0, bidirectional=False, device=None, dtype=None)

input\_size:输入层神经元的个数；

hidden\_size:隐藏层神经元的个数;

num\_layers:RNN的堆叠层数;

nonlinearity:非线性激活函数,可以选择tanh、relu等;

bias:布尔类型,是否启用偏置值;

batch\_first:布尔类型,是否选择让batch\_size作为输入形状中的第一个参数；

drop\_out:布尔类型,RNN的每一层后面是否加上一个dropout层；

bidirectional:布尔类型,是否启用双向RNN；

3.变换器层:特定场景下使用,(变换器Transformer在特定的网络结构中提出的一种模型,主要用于自然语言处理NLP)。

class torch.nn.Transformer(d\_model=512, nhead=8, num\_encoder\_layers=6, num\_decoder\_layers=6, dim\_feedforward=2048, dropout=0.1, activation=<function relu>, custom\_encoder=None, custom\_decoder=None, layer\_norm\_eps=1e-05, batch\_first=False, norm\_first=False, bias=True, device=None, dtype=None)

d\_model:Encoder和Decoder输入参数的特征维度,也就是词向量的维度;

nhead:多头注意力机制中,head的数量;

num\_encoder\_layers:EncoderLayer的数量,该值越大,网络越深,参数数量越多,计算量越大;

num\_decoder\_layers:DecoderLayer的数量,该值越大,网络越深,参数数量越多,计算量越大;

dim\_feedforward:FeedForward层(Attention后面的全连接网络)的隐藏层的神经元数量,该值越大,网络参数量越多,计算量越大;

dropout:dropout的取值,取值范围为[0.1-0.5](值越大,效果越好,但模型收敛的速度也会越慢);

activation:FeedForward层的激活函数,取值可以是‘relu’或者‘gelu’(str数据类型);

custom\_encoder:自定义编码器;

custom\_decoder:自定义解码器;

layer\_norm\_eps:层归一化组件中的EPS值；

4.线性层:使用得比较多,在神经网络中起到的作用是对输入数据进行线性变换,使用矩阵乘法将输入特征转换为输出特征,一般用来实现全连接层。

class torch.nn.Linear(in\_features, out\_features, bias=True, device=None, dtype=None)

import torch  
from torch.nn import Linear  
from torchvision import datasets  
from torchvision import transforms  
from torch.utils.data import DataLoader  
from torch import nn  
  
cifar10\_datasets = datasets.CIFAR10(root='./dataset', train=False, transform=transforms.Compose([  
 transforms.ToTensor()  
]), download=True)  
cifar10\_dataloader = DataLoader(dataset=cifar10\_datasets, batch\_size=64, drop\_last=True)  
  
  
class Tudui(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.linear = Linear(196608, 10)  
  
 def forward(self, input):  
 output = self.linear(input)  
 return output  
  
  
tudui = Tudui()  
  
for data in cifar10\_dataloader:  
 imgs, target = data  
 print(imgs.shape)  
 imgs\_reshape = torch.reshape(imgs, [-1])  
 # 等价于 imgs\_reshape = torch.flatten(imgs)  
 print(imgs\_reshape.shape)  
 output = tudui(imgs\_reshape)  
 print(output.shape)

5.丢弃层:使用情况不多,主要用于在训练神经网络时进行正则化(在每次前向传播时随机将输入张量的某些元素置0,从而减少模型对特定数据的依赖,提高泛化能力)和防止神经元的共同适应(通过随机关闭神经网络中的某些神经元,迫使网络分散学习特征,防止神经元之间的过度依赖),实质是为了防止模型过拟合。

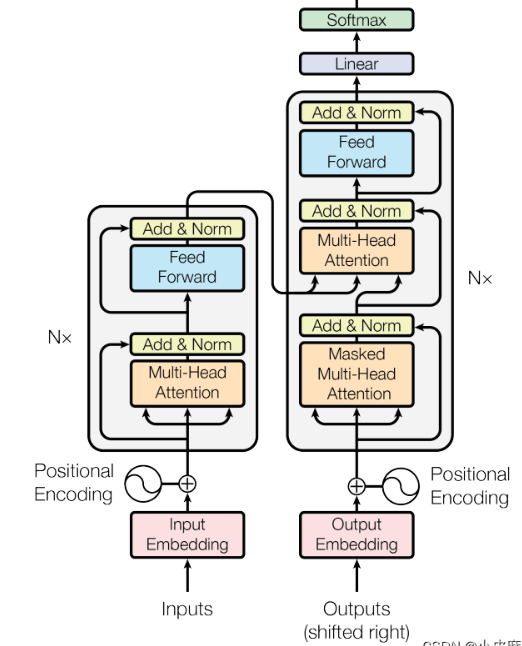
class torch.nn.Dropout(p=0.5, inplace=False)

6.稀疏层: 特定场景下使用,主要用于NLP。

class torch.nn.Embedding(num\_embeddings, embedding\_dim, padding\_idx=None, max\_norm=None, norm\_type=2.0, scale\_grad\_by\_freq=False, sparse=False, \_weight=None, \_freeze=False, device=None, dtype=None)

补充:

1.transformer结构图:



1. 神经网络模型搭建小实战

基本用法：

Sequential的使用:

class torch.nn.Sequential(\*args: [Module](https://pytorch.org/docs/stable/generated/torch.nn.Module.html" \l "torch.nn.Module" \o "torch.nn.modules.module.Module))

class torch.nn.Sequential(arg: OrderedDict[[str](https://docs.python.org/3/library/stdtypes.html" \l "str" \o "(in Python v3.12)), [Module](https://pytorch.org/docs/stable/generated/torch.nn.Module.html" \l "torch.nn.Module" \o "torch.nn.Module)])

Eg:

#方式一

**model** **=** **nn.Sequential(**

**nn.Conv2d(**1**,**20**,**5**),**

**nn.ReLU(),**

**nn.Conv2d(**20**,**64**,**5**),**

**nn.ReLU()**

**)**

#方式二

**model** **=** **nn.Sequential(OrderedDict([**

**(**'conv1'**,** **nn.Conv2d(**1**,**20**,**5**)),**

**(**'relu1'**,** **nn.ReLU()),**

**(**'conv2'**,** **nn.Conv2d(**20**,**64**,**5**)),**

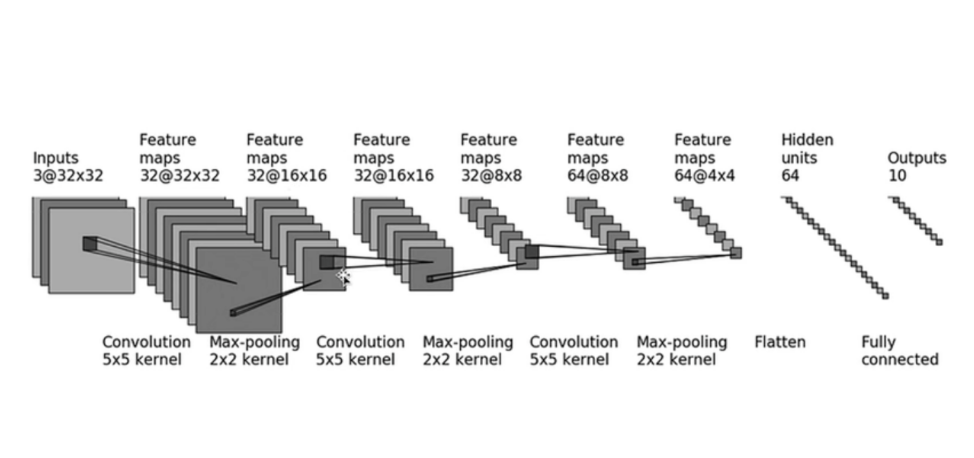
**(**'relu2'**,** **nn.ReLU())**

**]))**

import torch  
from torch.nn import Linear  
from torch.utils.tensorboard import SummaryWriter  
from torchvision import datasets  
from torchvision import transforms  
from torch.utils.data import DataLoader, dataset  
import torch.nn.functional as F  
from torch import nn  
  
  
class Tudui(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.models1 = nn.Sequential(  
 nn.Conv2d(in\_channels=3, out\_channels=32, kernel\_size=(5, 5), padding=2),  
 nn.MaxPool2d(kernel\_size=(2, 2)),  
 nn.Conv2d(in\_channels=32, out\_channels=32, kernel\_size=(5, 5), padding=2),  
 nn.MaxPool2d(kernel\_size=(2, 2)),  
 nn.Conv2d(in\_channels=32, out\_channels=64, kernel\_size=(5, 5), padding=2),  
 nn.MaxPool2d(kernel\_size=(2, 2)),  
 nn.Flatten(),  
 nn.Linear(1024, 64),  
 nn.Linear(64, 10)  
 )  
 # 等价于  
 # self.conv\_1 = nn.Conv2d(in\_channels=3, out\_channels=32, kernel\_size=(5, 5), padding=2)  
 # self.maxpooling\_1 = nn.MaxPool2d(kernel\_size=(2, 2))  
 # self.conv\_2 = nn.Conv2d(in\_channels=32, out\_channels=32, kernel\_size=(5, 5), padding=2)  
 # self.maxpooling\_2 = nn.MaxPool2d(kernel\_size=(2, 2))  
 # self.conv\_3 = nn.Conv2d(in\_channels=32, out\_channels=64, kernel\_size=(5, 5), padding=2)  
 # self.maxpooling\_3 = nn.MaxPool2d(kernel\_size=(2, 2))  
 # self.flatten = nn.Flatten()  
 # self.linear\_1 = nn.Linear(1024, 64)  
 # self.linear\_2 = nn.Linear(64, 10)  
  
 def forward(self, input):  
 input = self.models1(input)  
 # 等价于  
 # input = self.conv\_1(input)  
 # input = self.maxpooling\_1(input)  
 # input = self.conv\_2(input)  
 # input = self.maxpooling\_2(input)  
 # input = self.conv\_3(input)  
 # input = self.maxpooling\_3(input)  
 # input = self.flatten(input)  
 # input = self.linear\_1(input)  
 # input = self.linear\_2(input)  
 return input  
  
  
tudui = Tudui()  
# 1.输出网络结构  
print(tudui)  
# 2.验证网络的正确性(是否报错)  
input = torch.ones((64, 3, 32, 32))  
output = tudui(input)  
print(output.shape)  
# 不常用  
writer = SummaryWriter(log\_dir='./logs')  
writer.add\_graph(tudui, input)  
writer.close()

补充:

1.cifar10 模型的数据集:



输出层的数量决定了全连接层神经元的数量;也决定了偏置值的数量。

2.全连接层,在整个卷积神经网络中起到”分类器”的作用。如果说卷积层、池化层和激活函数层等操作是将原始数据映射到隐层特征空间的话,那么全连接层则起到将学到的”分布式特征表示”映射到样本标记空间的作用。全连接层一般通过线性层来实现。

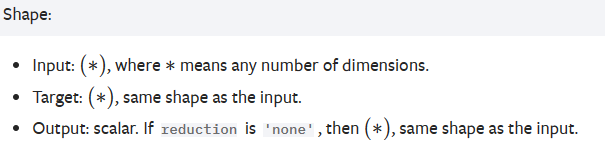
1. 损失函数与反向传播

损失函数,是表示神经网络性能”好坏程度“的指标。神经网络以某个指标(损失函数)为线索寻找最优的权重参数,损失函数一般使用均方误差和交叉熵误差函数。损失函数:1.计算实际输出(预测值)和目标(真实值)之间的差距;2.为更新网络的权重参数、输出提供一定的依据（反向传播-需要调用loss.backward(),使之通过grad来更新参数）

基本用法:

I.和损失误差/均损失误差

class torch.nn.L1Loss(size\_average=None, reduce=None, reduction='mean')



Eg:

I.reduction=‘mean’

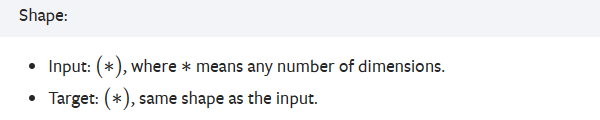
import torch  
from torch.nn import L1Loss  
  
inputs = torch.tensor([1, 2, 3], dtype=torch.float)  
targets = torch.tensor([1, 2, 5], dtype=torch.float)  
  
inputs = torch.reshape(inputs, (1, 1, 1, 3))  
targets = torch.reshape(targets, (1, 1, 1, 3))  
  
loss = L1Loss(reduction='mean')  
result = loss(inputs, targets)  
  
print(result)// tensor(0.6667)

II.reduction=‘sum’

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

II.均方误差

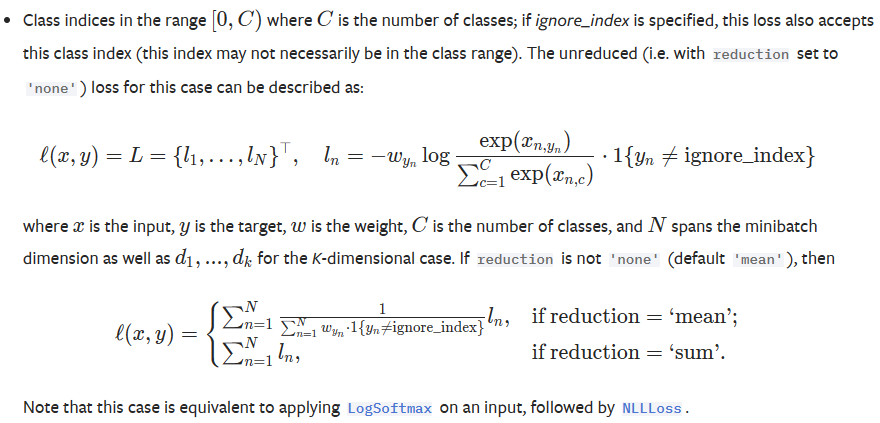
class torch.nn.MSELoss(size\_average=None, reduce=None, reduction='mean')



import torch  
from torch.nn import L1Loss, MSELoss  
  
inputs = torch.tensor([1, 2, 3], dtype=torch.float)  
targets = torch.tensor([1, 2, 5], dtype=torch.float)  
  
inputs = torch.reshape(inputs, (1, 1, 1, 3))  
targets = torch.reshape(targets, (1, 1, 1, 3))  
  
mse\_loss = MSELoss()  
result = mse\_loss(inputs, targets)  
print(result)// tensor(1.3333)

III.交叉熵误差（Softmax+交叉熵损失函数）

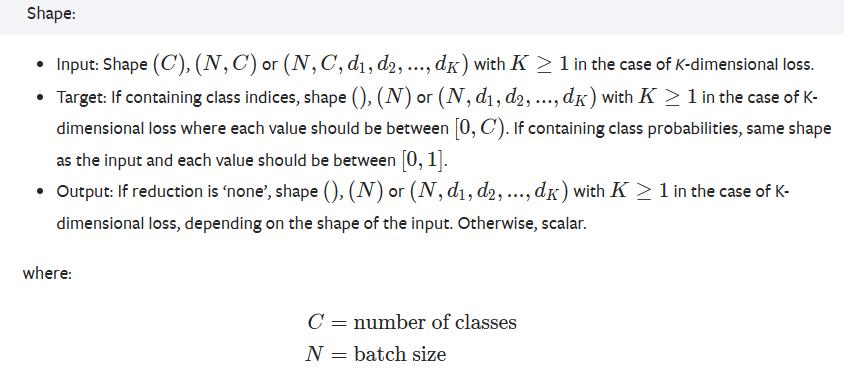
class torch.nn.CrossEntropyLoss(weight=None, size\_average=None, ignore\_index=-100, reduce=None, reduction='mean', label\_smoothing=0.0)



weight:该参数指定每个类别占loss的权重;

ignore\_index:指定忽略哪些类别的loss；

reduction:有三个选项,’none’、‘mean’和‘sum’。’mean’表明对N个样本的loss进行求平均之后返回;‘sum’指对n个样本的loss求和。



import torch  
from torch import nn  
from torch.utils.data import DataLoader  
from torchvision import datasets  
from torchvision import transforms  
  
cifar10\_datasets = datasets.CIFAR10(root='./dataset', train=False, transform=transforms.ToTensor(), download=True)  
  
cifar10\_dataLoader = DataLoader(dataset=cifar10\_datasets, batch\_size=64)  
  
  
class Tudui(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.sequential = 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(1024, 64),  
 nn.Linear(64, 10)  
 )  
  
 def forward(self, x):  
 output = self.sequential(x)  
 return output  
  
  
tudui = Tudui()  
  
loss = nn.CrossEntropyLoss()  
for data in cifar10\_dataLoader:  
 imgs, labels = data  
 output = tudui(imgs)  
 # print(imgs.shape) torch.Size([64, 3, 32, 32])  
 # print(output.shape) torch.Size([64, 10])  
 # print(labels.shape) torch.Size([64])  
 result = loss(output, labels)  
 print(result)  
 result.backward()

补充:

1.Tensor.backward()反向传播的原理:

基于Pytorch的动态图机制,创建一个张量x，并设置其 requires\_grad参数为True,程序将会追踪所有对于该张量的操作,当完成计算后通过调用 .backward(),自动计算所有的梯度,这个张量的所有梯度将会自动积累到 .grad 属性。

2.loss.backward()反向传播,得到并保存每个要更新参数的梯度;loss.backward()会进行梯度叠加,必须检查是否需要optimizer.zero\_grad();optimizer.step()通过loss.backward()得到并保存的梯度来更新参数

3.神经网络训练的三连锁步骤,与batch\_size的不同组合可实现大批量梯度下降法、小批量梯度下降法

optimizer.zero\_grad()

loss.backward()

optimizer.step()

1. 神经网络的优化器

基本用法：

I.构造优化器(Adam、SGD)

Eg:

1.优化器构造

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

optimizer = optim.Adam([var1, var2], lr=0.0001)

2.分别指定神经网络每一层的学习率

optim.SGD([

{'params': model.base.parameters(), 'lr': 1e-2},

{'params': model.classifier.parameters()}

], lr=1e-3, momentum=0.9)

通用参数: params(要传入的模型参数),lr(learning-rate学习率)

II.调用优化器方法

所有的优化器,通过step()方法来更新参数;

Eg:

for input, target in dataset:

optimizer.zero\_grad()

//需要将上一步loss.backward()求出来每个参数所对应的梯度进行清零

output = model(input)

loss = loss\_fn(output, target)

loss.backward()

optimizer.step()

import torch  
from torch import nn, optim  
from torch.nn import CrossEntropyLoss  
from torch.utils.data import DataLoader  
from torchvision import datasets  
from torchvision import transforms  
  
cifar10\_dataset = datasets.CIFAR10(root='./dataset', train=False, transform=transforms.ToTensor(), download=True)  
  
cifar10\_dataLoader = DataLoader(dataset=cifar10\_dataset, batch\_size=64)  
  
  
class Tudui(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.sequential = 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(1024, 64),  
 nn.Linear(64, 10)  
 )  
  
 def forward(self, input):  
 output = self.sequential(input)  
 return output  
  
  
tudui = Tudui()  
  
loss = CrossEntropyLoss()  
  
optimizer = optim.SGD(tudui.parameters(), lr=0.01)  
for data in cifar10\_dataLoader:  
 optimizer.zero\_grad()  
 imgs, targets = data  
 output = tudui(imgs)  
 result = loss(output, targets)  
 result.backward()  
 optimizer.step()

断点调试（optimizer.zero\_grad()、result.backward()、optimizer.step()）,查看

tudui->sequential->\_modules->0->weight

for epoch in range(20):  
 running\_loss = 0.0  
 for data in cifar10\_dataLoader:  
 optimizer.zero\_grad()  
 imgs, targets = data  
 output = tudui(imgs)  
 result\_loss = loss(output, targets)  
 result\_loss.backward()  
 optimizer.step()  
 running\_loss += result\_loss  
 print(running\_loss)

增加epoch的迭代次数,模型的参数训练往往需要成千上万次的训练,才能得到较好的参数



补充:

1.误区——交叉熵损失函数每次小批量梯度下降之后,损失函数的值不一定减少(每批量数据的tensor张量不同);但是交叉熵损失函数在每次epoch迭代后,损失函数的值一定减少。

2.卷积神经网络中图像的噪声点,指的是该点的像素和周围像素点的差异很大。

1. 现有网络模型的使用及修改

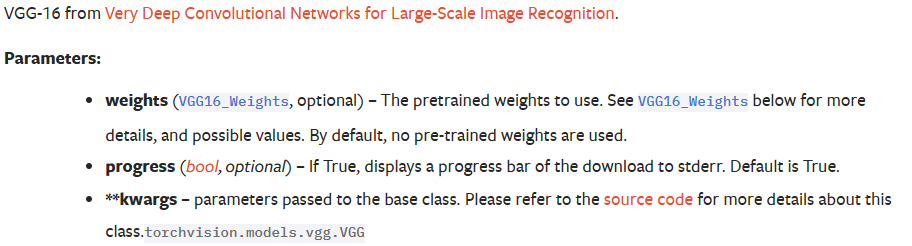
Vgg16网络模型(基于数据集[ImageNet](http://image-net.org/))

torchvision.models.vgg16(\*, weights: [Optional](https://docs.python.org/3/library/typing.html" \l "typing.Optional" \o "(in Python v3.12))[[VGG16\_Weights](https://pytorch.org/vision/stable/models/generated/torchvision.models.vgg16.html" \l "torchvision.models.VGG16_Weights" \o "torchvision.models.vgg.VGG16_Weights)] = None, progress: [bool](https://docs.python.org/3/library/functions.html" \l "bool" \o "(in Python v3.12)) = True, \*\*kwargs: [Any](https://docs.python.org/3/library/typing.html" \l "typing.Any" \o "(in Python v3.12))) → VGG

pretrained:布尔值,若为True,返回基于ImageNet的预训练参数(旧版本)参数为训练得到的;若为False,只返回模型,参数为随机初始化的。

weights:要使用的预训练权重,默认无预训练的权重可以使用,一般选择 [VGG16\_Weights](https://pytorch.org/vision/stable/models/generated/torchvision.models.vgg16.html" \l "torchvision.models.VGG16_Weights" \o "torchvision.models.VGG16_Weights) ;

progress:布尔值,若为True,则显示下载进度条。



[ImageNet](http://image-net.org/)数据集

class torchvision.datasets.ImageNet(root:[Union](https://docs.python.org/3/library/typing.html" \l "typing.Union" \o "(in Python v3.12))[[str](https://docs.python.org/3/library/stdtypes.html" \l "str" \o "(in Python v3.12)), [Path](https://docs.python.org/3/library/pathlib.html" \l "pathlib.Path" \o "(in Python v3.12))], split: [str](https://docs.python.org/3/library/stdtypes.html" \l "str" \o "(in Python v3.12))='train', \*\*kwargs:[Any](https://docs.python.org/3/library/typing.html" \l "typing.Any" \o "(in Python v3.12)))

root:下载的项目根路径；

split:训练集类型,训练集train和验证集val

transform:图像变换

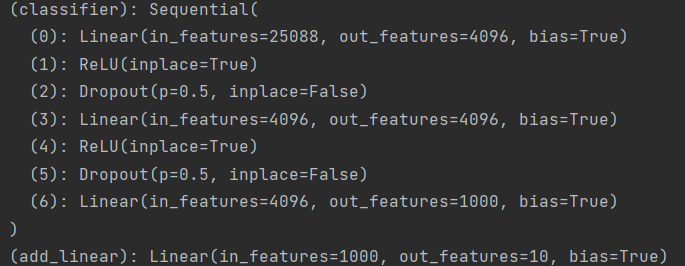
target\_transform:标签变换

loader:加载给定路径的图像的函数,一般采用default\_loader

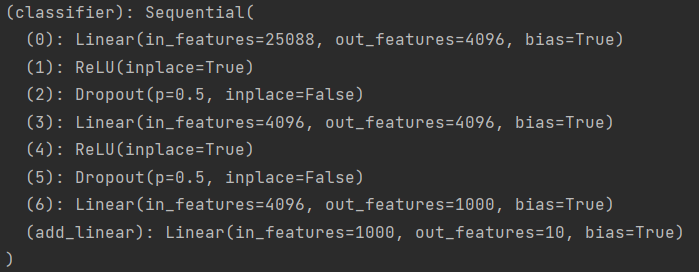
I.在网络模型中添加特定层

import torch  
from torch import nn  
import torchvision  
from torchvision import datasets  
from torchvision import transforms  
  
vgg16\_false = torchvision.models.vgg16(pretrained=False)  
vgg16\_true = torchvision.models.vgg16(pretrained=True)  
  
train\_data = torchvision.datasets.CIFAR10(root='./dataset', train=True, transform=transforms.ToTensor(), download=True)  
  
# vgg16\_true.add\_module('add\_linear', nn.Linear(1000, 10))  
vgg16\_true.classifier.add\_module('add\_linear',nn.Linear(1000, 10))  
print(vgg16\_true)

vgg16\_true.add\_module('add\_linear', nn.Linear(1000, 10))

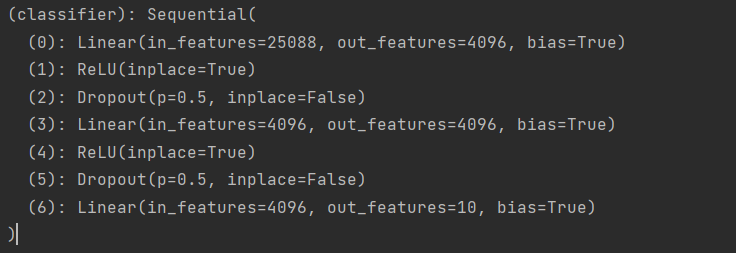


vgg16\_true.classifier.add\_module('add\_linear',nn.Linear(1000, 10))



II.在网络模型中修改特定层

import torch  
from torch import nn  
import torchvision  
from torchvision import datasets  
from torchvision import transforms  
  
vgg16\_false = torchvision.models.vgg16(pretrained=False)  
vgg16\_true = torchvision.models.vgg16(pretrained=True)  
  
train\_data = torchvision.datasets.CIFAR10(root='./dataset', train=True, transform=transforms.ToTensor(), download=True)  
  
vgg16\_true.classifier[6] = nn.Linear(4096, 10)  
print(vgg16\_true)



1. 网络模型的保存与读取

I.模型保存

import torch  
import torchvision  
  
vgg16 = torchvision.models.vgg16(pretrained=True)  
# 保存方式1 模型结构+模型参数  
torch.save(vgg16, 'vgg16\_method1.pth')  
  
# 保存方式2 模型参数(官方推荐)  
torch.save(vgg16.state\_dict(), 'vgg16\_method2.pth')

II.模型加载（通过state\_dict()来验证加载后模型的参数与原模型的参数是否一致）

import torch  
import torchvision  
  
# 保存方式1==>加载模型1  
model = torch.load('vgg16\_method1.pth')  
print(model)  
  
# 保存方式2==>加载模型2  
vgg16 = torchvision.models.vgg16(pretrained=False)  
# 此处的vgg16的state\_dict保存的是随机初始化参数(权重、偏置值等),不是训练得到的参数  
state\_dict = torch.load('vgg16\_method2.pth')  
vgg16.load\_state\_dict(state\_dict)  
# 此处的model的state\_dict保存的是训练得到的参数  
print(vgg16)

补充:

!!!!!1.load\_state\_dict(state\_dict)不能再赋值给其他的变量

2.load\_state\_dict(map\_location=None) map\_location指定如何重新映射存储位置的函数、torch.device、字符串或dict;例如,使用gpu训练的模型,在CPU的加载过程中,要指定map\_location=torch.device(‘cpu’);

1. 完整的模型训练套路(基于CIFAR10数据集)

I.完整的模型训练套路(一)

train.py:

1.准备数据集dataset=>2.数据集长度length=>3.加载数据集dataloader=>

4.创建神经网络模型model=>5.选择损失函数loss=>6.选择优化器optim=>

7.迭代训练(每一轮训练+每一轮测试步骤+每一轮模型保存)

import torch  
from torch.utils.data import dataloader  
from torch import optim  
from torchvision import datasets  
from torchvision import transforms  
from model import \*  
  
# 1.准备数据集dataset  
train\_data = datasets.CIFAR10(root='../dataset', train=True, transform=transforms.ToTensor(), download=True)  
test\_data = datasets.CIFAR10(root='../dataset', train=False, transform=transforms.ToTensor(), download=True)  
  
# 2.数据集长度length  
train\_data\_size = len(train\_data)  
test\_data\_size = len(test\_data)  
print('训练数据集的长度为:%d' % train\_data\_size)  
print('测试数据集的长度为:%d' % test\_data\_size)  
  
# 3.加载数据集dataloader  
train\_dataloader = dataloader.DataLoader(dataset=train\_data, batch\_size=64)  
test\_dataloader = dataloader.DataLoader(dataset=test\_data, batch\_size=64)  
  
# 4.创建神经网络模型model  
tudui = Tudui()  
  
# 5.选择损失函数loss  
loss\_fn = nn.CrossEntropyLoss()  
  
learning\_rate = 1e-2  
# 6.选择优化器optim  
optimizer = optim.SGD(tudui.parameters(), lr=learning\_rate)  
  
total\_train\_step = 0 # 记录训练的次数  
total\_test\_step = 0 # 记录测试的次数  
epoch = 10 # 训练的轮数  
# 7.迭代训练  
for i in range(epoch):  
 print('第{}轮训练开始'.format(i + 1))  
 for data in train\_dataloader:  
 imgs, tars = data  
 outputs = tudui(imgs)  
 loss\_result = loss\_fn(outputs, tars)  
  
 optimizer.zero\_grad()  
 loss\_result.backward()  
 optimizer.step()  
  
 total\_train\_step += 1  
 print('训练次数:{},Loss:{}'.format(total\_train\_step, loss\_result.item()))

补充:

1.loss.item():取一个元素张量里面的具体元素值并返回该位置元素值的高精度,可以将一个零维张量(只有一个元素的张量)转换成int型或者float型,节省内存(不会计入计算图(动态图)中)。

model.py:

1.创建神经网络模型model=>2.验证神经网络的输入/输出

import torch  
from torch import nn  
  
  
class Tudui(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_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(1024, 64),  
 nn.Linear(64, 10)  
 )  
  
 def forward(self, x):  
 y = self.model(x)  
 return y  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 tudui = Tudui()  
 input = torch.ones((64, 3, 32, 32))  
 output = tudui(input)  
 print(output.shape)

II. 完整的模型训练套路(二)

在训练的过程中,评估模型是否训练得当——在每一轮训练中,以测试集数据的损失、正确率来评估模型

1).通过控制台打印每轮测试数据的损失**total\_test\_loss**

import torch  
from torch.utils.data import dataloader  
from torch import optim  
from torchvision import datasets  
from torchvision import transforms  
from model import \*  
  
# 1.准备数据集dataset  
train\_data = datasets.CIFAR10(root='../dataset', train=True, transform=transforms.ToTensor(), download=True)  
test\_data = datasets.CIFAR10(root='../dataset', train=False, transform=transforms.ToTensor(), download=True)  
  
# 2.数据集长度length  
train\_data\_size = len(train\_data)  
test\_data\_size = len(test\_data)  
print('训练数据集的长度为:%d' % train\_data\_size)  
print('测试数据集的长度为:%d' % test\_data\_size)  
  
# 3.加载数据集dataloader  
train\_dataloader = dataloader.DataLoader(dataset=train\_data, batch\_size=64)  
test\_dataloader = dataloader.DataLoader(dataset=test\_data, batch\_size=64)  
  
# 4.创建神经网络模型model  
tudui = Tudui()  
  
# 5.选择损失函数loss  
loss\_fn = nn.CrossEntropyLoss()  
  
learning\_rate = 1e-2  
# 6.选择优化器  
optimizer = optim.SGD(tudui.parameters(), lr=learning\_rate)  
  
total\_train\_step = 0 # 记录训练的次数  
total\_test\_step = 0 # 记录测试的次数  
epoch = 10 # 训练的轮数  
# 7.迭代训练  
for i in range(epoch):  
 print('第{}轮训练开始'.format(i + 1))  
 for data in train\_dataloader:  
 imgs, tars = data  
 outputs = tudui(imgs)  
 loss\_result = loss\_fn(outputs, tars)  
  
 optimizer.zero\_grad()  
 loss\_result.backward()  
 optimizer.step()  
  
 total\_train\_step += 1  
 if total\_train\_step % 100 == 0:  
 print('训练次数:{},Loss:{}'.format(total\_train\_step, loss\_result.item()))  
  
 # 测试步骤开始(保证不会对模型进行调优)  
 with torch.no\_grad():  
 total\_test\_loss = 0  
 for data in test\_dataloader:  
 imgs, tars = data  
 outputs = tudui(imgs)  
 loss = loss\_fn(outputs, tars)  
 total\_test\_loss += loss  
 print('整体测试集上的Loss:{}'.format(total\_test\_loss))

2).通过tensorboard打印每轮测试数据的损失**total\_test\_loss**,并保存每一轮训练后的模型

import torch  
from torch.utils.data import dataloader  
from torch.utils.tensorboard import SummaryWriter  
from torch import optim  
from torchvision import datasets  
from torchvision import transforms  
from model import \*  
  
# 1.准备数据集dataset  
train\_data = datasets.CIFAR10(root='../dataset', train=True, transform=transforms.ToTensor(), download=True)  
test\_data = datasets.CIFAR10(root='../dataset', train=False, transform=transforms.ToTensor(), download=True)  
  
# 2.数据集长度length  
train\_data\_size = len(train\_data)  
test\_data\_size = len(test\_data)  
print('训练数据集的长度为:%d' % train\_data\_size)  
print('测试数据集的长度为:%d' % test\_data\_size)  
  
# 3.加载数据集dataloader  
train\_dataloader = dataloader.DataLoader(dataset=train\_data, batch\_size=64)  
test\_dataloader = dataloader.DataLoader(dataset=test\_data, batch\_size=64)  
  
# 4.创建神经网络模型model  
tudui = Tudui()  
  
# 5.选择损失函数loss  
loss\_fn = nn.CrossEntropyLoss()  
  
learning\_rate = 1e-2  
# 6.选择优化器  
optimizer = optim.SGD(tudui.parameters(), lr=learning\_rate)  
  
total\_train\_step = 0 # 记录训练的次数  
total\_test\_step = 0 # 记录测试的次数  
epoch = 10 # 训练的轮数  
writer = SummaryWriter(log\_dir='../logs')  
# 7.迭代训练  
for i in range(epoch):  
 print('----------第{}轮训练开始----------'.format(i + 1))  
 for data in train\_dataloader:  
 imgs, tars = data  
 outputs = tudui(imgs)  
 loss\_result = loss\_fn(outputs, tars)  
  
 optimizer.zero\_grad()  
 loss\_result.backward()  
 optimizer.step()  
  
 total\_train\_step += 1  
 if total\_train\_step % 100 == 0:  
 print('训练次数:{},Loss:{}'.format(total\_train\_step, loss\_result.item()))  
 writer.add\_scalar('train\_loss', loss\_result.item(), total\_train\_step)  
  
 # 测试步骤开始(保证不会对模型进行调优)  
 with torch.no\_grad():  
 total\_test\_loss = 0  
 for data in test\_dataloader:  
 imgs, tars = data  
 outputs = tudui(imgs)  
 loss = loss\_fn(outputs, tars)  
 total\_test\_loss += loss  
 total\_test\_step += 1  
 print('整体测试集上的Loss:{}'.format(total\_test\_loss))  
 writer.add\_scalar('test\_loss', total\_test\_loss, total\_test\_step)  
 writer.close()  
 # 模型保存  
 torch.save(tudui.state\_dict(), 'tudui\_{}.pth'.format(i))  
 print('模型已保存')

3).分类问题的正确率

import torch  
from torch.utils.data import dataloader  
from torch.utils.tensorboard import SummaryWriter  
from torch import optim  
from torchvision import datasets  
from torchvision import transforms  
from model import \*  
  
# 1.准备数据集dataset  
train\_data = datasets.CIFAR10(root='../dataset', train=True, transform=transforms.ToTensor(), download=True)  
test\_data = datasets.CIFAR10(root='../dataset', train=False, transform=transforms.ToTensor(), download=True)  
  
# 2.数据集长度length  
train\_data\_size = len(train\_data)  
test\_data\_size = len(test\_data)  
print('训练数据集的长度为:%d' % train\_data\_size)  
print('测试数据集的长度为:%d' % test\_data\_size)  
  
# 3.加载数据集dataloader  
train\_dataloader = dataloader.DataLoader(dataset=train\_data, batch\_size=64)  
test\_dataloader = dataloader.DataLoader(dataset=test\_data, batch\_size=64)  
  
# 4.创建神经网络模型model  
tudui = Tudui()  
  
# 5.选择损失函数loss  
loss\_fn = nn.CrossEntropyLoss()  
  
learning\_rate = 1e-2  
# 6.选择优化器  
optimizer = optim.SGD(tudui.parameters(), lr=learning\_rate)  
  
total\_train\_step = 0 # 记录训练的次数  
total\_test\_step = 0 # 记录测试的次数  
epoch = 10 # 训练的轮数  
writer = SummaryWriter(log\_dir='../logs')  
# 7.迭代训练  
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, tars = data  
 outputs = tudui(imgs)  
 loss\_result = loss\_fn(outputs, tars)  
  
 optimizer.zero\_grad()  
 loss\_result.backward()  
 optimizer.step()  
  
 total\_train\_step += 1  
 if total\_train\_step % 100 == 0:  
 print('训练次数:{},Loss:{}'.format(total\_train\_step, loss\_result.item()))  
 writer.add\_scalar('train\_loss', loss\_result.item(), total\_train\_step)  
  
 # 测试步骤开始(保证不会对模型进行调优)  
 with torch.no\_grad():  
 total\_test\_loss = 0 # 记录测试的损失值  
 total\_test\_accuracy = 0 # 记录测试的正确率  
 for data in test\_dataloader:  
 imgs, tars = data  
 outputs = tudui(imgs)  
 loss = loss\_fn(outputs, tars)  
 accuracy = (outputs.argmax(1) == tars).sum()  
 total\_test\_loss += loss  
 total\_test\_accuracy += accuracy  
 total\_test\_step += 1  
 print('整体测试集上的Loss:{}'.format(total\_test\_loss))  
 print('整体测试集上的Accuracy:{}'.format(total\_test\_accuracy\*1.0/test\_data\_size))  
 writer.add\_scalar('test\_loss', total\_test\_loss, total\_test\_step)  
 writer.add\_scalar('test\_accuracy', total\_test\_accuracy\*1.0/test\_data\_size, total\_test\_step)  
 writer.close()  
 # 模型保存  
 torch.save(tudui.state\_dict(), 'tudui\_{}.pth'.format(i))  
 print('模型已保存')

III. 完整的模型训练套路(三)

model.train():训练开始前写model.train(),作用是启用batch normalization 和 dropout。

model.eval():测试开始前写model.eval()作用是不启用batch normalization和dropout。

import torch  
from torch.utils.data import dataloader  
from torch.utils.tensorboard import SummaryWriter  
from torch import optim  
from torchvision import datasets  
from torchvision import transforms  
from model import \*  
  
# 1.准备数据集dataset  
train\_data = datasets.CIFAR10(root='../dataset', train=True, transform=transforms.ToTensor(), download=True)  
test\_data = datasets.CIFAR10(root='../dataset', train=False, transform=transforms.ToTensor(), download=True)  
  
# 2.数据集长度length  
train\_data\_size = len(train\_data)  
test\_data\_size = len(test\_data)  
print('训练数据集的长度为:%d' % train\_data\_size)  
print('测试数据集的长度为:%d' % test\_data\_size)  
  
# 3.加载数据集dataloader  
train\_dataloader = dataloader.DataLoader(dataset=train\_data, batch\_size=64)  
test\_dataloader = dataloader.DataLoader(dataset=test\_data, batch\_size=64)  
  
# 4.创建神经网络模型model  
tudui = Tudui()  
  
# 5.选择损失函数loss  
loss\_fn = nn.CrossEntropyLoss()  
  
learning\_rate = 1e-2  
# 6.选择优化器  
optimizer = optim.SGD(tudui.parameters(), lr=learning\_rate)  
  
total\_train\_step = 0 # 记录训练的次数  
total\_test\_step = 0 # 记录测试的次数  
epoch = 10 # 训练的轮数  
writer = SummaryWriter(log\_dir='../logs')  
# 7.迭代训练  
for i in range(epoch):  
 print('----------第{}轮训练开始----------'.format(i + 1))  
 tudui.train()  
 # 训练步骤开始  
 for data in train\_dataloader:  
 imgs, tars = data  
 outputs = tudui(imgs)  
 loss\_result = loss\_fn(outputs, tars)  
  
 optimizer.zero\_grad()  
 loss\_result.backward()  
 optimizer.step()  
  
 total\_train\_step += 1  
 if total\_train\_step % 100 == 0:  
 print('训练次数:{},Loss:{}'.format(total\_train\_step, loss\_result.item()))  
 writer.add\_scalar('train\_loss', loss\_result.item(), total\_train\_step)  
  
 tudui.eval()  
 # 测试步骤开始(保证不会对模型进行调优)  
 with torch.no\_grad():  
 total\_test\_loss = 0 # 记录测试的损失值  
 total\_test\_accuracy = 0 # 记录测试的正确率  
 for data in test\_dataloader:  
 imgs, tars = data  
 outputs = tudui(imgs)  
 loss = loss\_fn(outputs, tars)  
 accuracy = (outputs.argmax(1) == tars).sum()  
 total\_test\_loss += loss  
 total\_test\_accuracy += accuracy  
 total\_test\_step += 1  
 print('整体测试集上的Loss:{}'.format(total\_test\_loss))  
 print('整体测试集上的Accuracy:{}'.format(total\_test\_accuracy \* 1.0 / test\_data\_size))  
 writer.add\_scalar('test\_loss', total\_test\_loss, total\_test\_step)  
 writer.add\_scalar('test\_accuracy', total\_test\_accuracy \* 1.0 / test\_data\_size, total\_test\_step)  
 writer.close()  
 # 模型保存  
 torch.save(tudui.state\_dict(), 'tudui\_{}.pth'.format(i))  
 print('模型已保存')

补充:

1.tensor类型-item()将此张量的值作为标准Python数返回,这仅适用于具有一个元素的张量。

import torch  
  
x = torch.tensor(1)  
print(x)// tensor(1)  
print(x.item())//1

2.tensor类型-argmax()返回指定维度最大值的序号

argmax(0):纵向(跨列)看数据,返回最大值的index

argmax(1):横向(跨行)看数据,返回最大值的index

import torch  
  
output = torch.tensor([  
 [0.1, 0.2],  
 [0.05, 0.4]  
])  
pred = output.argmax(1)  
print(pred)// tensor([1, 1])  
pred = output.argmax(0)  
print(pred)// tensor([0, 1])

import torch  
  
output = torch.tensor([  
 [0.1, 0.2],  
 [0.3, 0.4]  
])  
pred = output.argmax(1)  
print(pred)// tensor([1, 1])  
  
targets = torch.tensor([0, 1])  
  
print(pred == targets)// tensor([False, True])  
accracy\_result = (pred == targets).sum()  
print(accracy\_result)// tensor(1)

3.测试步骤二步走:

1.model.eval() 2.with torch.no\_grad()

1. 神经网络模型的GPU训练0

batch\_size越大(通常取64以上)，越能体现GPU性能（计算速度）。

batch\_size一般取二次幂,需要通过不断调试该参数,使得损失函数达到最小。

I.利用GPU训练(一)

将网络模型、（训练/测试）数据（输入、标注）、损失函数都放到GPU（.cuda（））上

import torch  
from torch.utils.data import dataloader  
from torch.utils.tensorboard import SummaryWriter  
from torch import optim  
from torch import nn  
from torchvision import datasets  
from torchvision import transforms  
import time  
  
# 1.准备数据集dataset  
train\_data = datasets.CIFAR10(root='../dataset', train=True, transform=transforms.ToTensor(), download=True)  
test\_data = datasets.CIFAR10(root='../dataset', train=False, transform=transforms.ToTensor(), download=True)  
  
# 2.数据集长度length  
train\_data\_size = len(train\_data)  
test\_data\_size = len(test\_data)  
print('训练数据集的长度为:%d' % train\_data\_size)  
print('测试数据集的长度为:%d' % test\_data\_size)  
  
# 3.加载数据集dataloader  
train\_dataloader = dataloader.DataLoader(dataset=train\_data, batch\_size=64)  
test\_dataloader = dataloader.DataLoader(dataset=test\_data, batch\_size=64)  
  
  
# 4.创建神经网络模型model  
class Tudui(nn.Module):  
 def \_\_init\_\_(self):  
 super().\_\_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(1024, 64),  
 nn.Linear(64, 10)  
 )  
  
 def forward(self, x):  
 y = self.model(x)  
 return y  
  
  
tudui = Tudui()  
if torch.cuda.is\_available():  
 tudui = tudui.cuda()  
  
# 5.选择损失函数loss  
if torch.cuda.is\_available():  
 loss\_fn = nn.CrossEntropyLoss()  
 loss\_fn = loss\_fn.cuda()  
  
learning\_rate = 1e-2  
# 6.选择优化器  
optimizer = optim.SGD(tudui.parameters(), lr=learning\_rate)  
  
total\_train\_step = 0 # 记录训练的次数  
total\_test\_step = 0 # 记录测试的次数  
epoch = 10 # 训练的轮数  
writer = SummaryWriter(log\_dir='../logs')  
  
start\_time = time.time()  
# 7.迭代训练  
for i in range(epoch):  
 print('----------第{}轮训练开始----------'.format(i + 1))  
 tudui.train()  
 # 训练步骤开始  
 for data in train\_dataloader:  
 imgs, tars = data  
 if torch.cuda.is\_available():  
 imgs = imgs.cuda()  
 tars = tars.cuda()  
 outputs = tudui(imgs)  
 loss\_result = loss\_fn(outputs, tars)  
  
 optimizer.zero\_grad()  
 loss\_result.backward()  
 optimizer.step()  
  
 total\_train\_step += 1  
 if total\_train\_step % 100 == 0:  
 end\_time = time.time()  
 print(end\_time-start\_time)  
 print('训练次数:{},Loss:{}'.format(total\_train\_step, loss\_result.item()))  
 writer.add\_scalar('train\_loss', loss\_result.item(), total\_train\_step)  
  
 tudui.eval()  
 # 测试步骤开始(保证不会对模型进行调优)  
 with torch.no\_grad():  
 total\_test\_loss = 0 # 记录测试的损失值  
 total\_test\_accuracy = 0 # 记录测试的正确率  
 for data in test\_dataloader:  
 imgs, tars = data  
 if torch.cuda.is\_available():  
 imgs = imgs.cuda()  
 tars = tars.cuda()  
 outputs = tudui(imgs)  
 loss = loss\_fn(outputs, tars)  
 accuracy = (outputs.argmax(1) == tars).sum()  
 total\_test\_loss += loss  
 total\_test\_accuracy += accuracy  
 total\_test\_step += 1  
 print('整体测试集上的Loss:{}'.format(total\_test\_loss))  
 print('整体测试集上的Accuracy:{}'.format(total\_test\_accuracy \* 1.0 / test\_data\_size))  
 writer.add\_scalar('test\_loss', total\_test\_loss, total\_test\_step)  
 writer.add\_scalar('test\_accuracy', total\_test\_accuracy \* 1.0 / test\_data\_size, total\_test\_step)  
 writer.close()  
 # 模型保存  
 torch.save(tudui.state\_dict(), 'tudui\_{}.pth'.format(i))  
 print('模型已保存')

II.利用GPU训练(二)

定义训练的设备device, 将网络模型、（训练/测试）数据（输入、标注）、损失函数都放到GPU（.to(device)）上

device=torch.device(‘cpu’)、device=torch.device(‘cuda’)、device=torch.device(‘cuda:0’)

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

import torch  
from torch.utils.data import dataloader  
from torch.utils.tensorboard import SummaryWriter  
from torch import optim  
from torchvision import datasets  
from torchvision import transforms  
from model import \*  
import time  
  
device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  
print(device)  
  
# 1.准备数据集dataset  
train\_data = datasets.CIFAR10(root='../dataset', train=True, transform=transforms.ToTensor(), download=True)  
test\_data = datasets.CIFAR10(root='../dataset', train=False, transform=transforms.ToTensor(), download=True)  
  
# 2.数据集长度length  
train\_data\_size = len(train\_data)  
test\_data\_size = len(test\_data)  
print('训练数据集的长度为:%d' % train\_data\_size)  
print('测试数据集的长度为:%d' % test\_data\_size)  
  
# 3.加载数据集dataloader  
train\_dataloader = dataloader.DataLoader(dataset=train\_data, batch\_size=64)  
test\_dataloader = dataloader.DataLoader(dataset=test\_data, batch\_size=64)  
  
# 4.创建神经网络模型model  
tudui = Tudui()  
tudui = tudui.to(device)  
  
# 5.选择损失函数loss  
loss\_fn = nn.CrossEntropyLoss()  
loss\_fn = loss\_fn.to(device)  
  
learning\_rate = 1e-2  
# 6.选择优化器  
optimizer = optim.SGD(tudui.parameters(), lr=learning\_rate)  
  
total\_train\_step = 0 # 记录训练的次数  
total\_test\_step = 0 # 记录测试的次数  
epoch = 10 # 训练的轮数  
writer = SummaryWriter(log\_dir='../logs')  
  
start\_time = time.time()  
# 7.迭代训练  
for i in range(epoch):  
 print('----------第{}轮训练开始----------'.format(i + 1))  
 tudui.train()  
 # 训练步骤开始  
 for data in train\_dataloader:  
 imgs, tars = data  
 imgs = imgs.to(device)  
 tars = tars.to(device)  
 outputs = tudui(imgs)  
 loss\_result = loss\_fn(outputs, tars)  
  
 optimizer.zero\_grad()  
 loss\_result.backward()  
 optimizer.step()  
  
 total\_train\_step += 1  
 if total\_train\_step % 100 == 0:  
 end\_time = time.time()  
 print(end\_time - start\_time)  
 print('训练次数:{},Loss:{}'.format(total\_train\_step, loss\_result.item()))  
 writer.add\_scalar('train\_loss', loss\_result.item(), total\_train\_step)  
  
 tudui.eval()  
 # 测试步骤开始(保证不会对模型进行调优)  
 with torch.no\_grad():  
 total\_test\_loss = 0 # 记录测试的损失值  
 total\_test\_accuracy = 0 # 记录测试的正确率  
 for data in test\_dataloader:  
 imgs, tars = data  
 imgs = imgs.to(device)  
 tars = tars.to(device)  
 outputs = tudui(imgs)  
 loss = loss\_fn(outputs, tars)  
 accuracy = (outputs.argmax(1) == tars).sum()  
 total\_test\_loss += loss  
 total\_test\_accuracy += accuracy  
 total\_test\_step += 1  
 print('整体测试集上的Loss:{}'.format(total\_test\_loss))  
 print('整体测试集上的Accuracy:{}'.format(total\_test\_accuracy \* 1.0 / test\_data\_size))  
 writer.add\_scalar('test\_loss', total\_test\_loss, total\_test\_step)  
 writer.add\_scalar('test\_accuracy', total\_test\_accuracy \* 1.0 / test\_data\_size, total\_test\_step)  
 writer.close()  
 # 模型保存  
 torch.save(tudui.state\_dict(), 'tudui\_{}.pth'.format(i))  
 print('模型已保存')

补充:

1.time库

start\_time = time.time()

end\_time = time.time()

1. 完整的模型验证(测试/demo)套路

test.py:利用已经训练好的模型,然后给它提供输入,来进行预测。

from PIL import Image  
from torchvision import transforms  
import torch  
from model import \*  
from torch import nn  
  
dog\_image\_path = '../imgs/dog.jpg'  
airplane\_image\_path = '../imgs/airplane.jpg'  
image = Image.open(dog\_image\_path)  
image = image.convert(mode='RGB')  
print(image)  
  
tran\_compose = transforms.Compose([  
 transforms.Resize((32, 32)),  
 transforms.ToTensor()  
])  
image = tran\_compose(image)  
print(image.shape)  
  
device = torch.device('cuda')  
  
tudui = Tudui()  
model\_state\_dict = torch.load('tudui\_25.pth')  
tudui.load\_state\_dict(model\_state\_dict)  
tudui = tudui.to(device)  
print(tudui)  
  
image = torch.reshape(image, (1, 3, 32, 32)).to(device)  
  
tudui.eval()  
with torch.no\_grad():  
 output = tudui(image)  
print(output)  
print(torch.argmax(output))

补充:

1.image = image.convert(mode='RGB')作用?

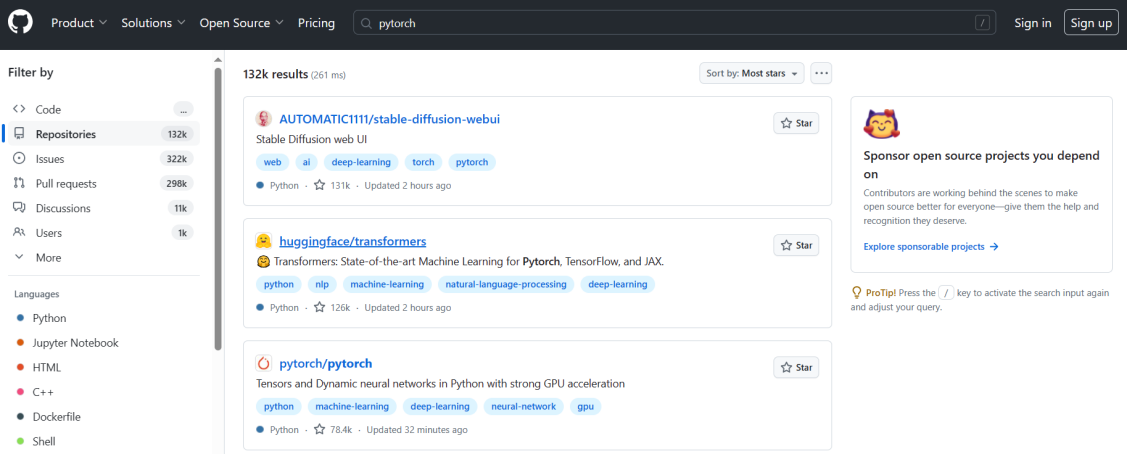
Png格式的图片有4个通道RGBA(3个颜色通道+1个透明度通道),调用该语句能够保留其颜色通道,可以适应png、jpg各种格式的图片。

2.image=torch.reshape(image,(1,3,32,32)).to(device)的作用?

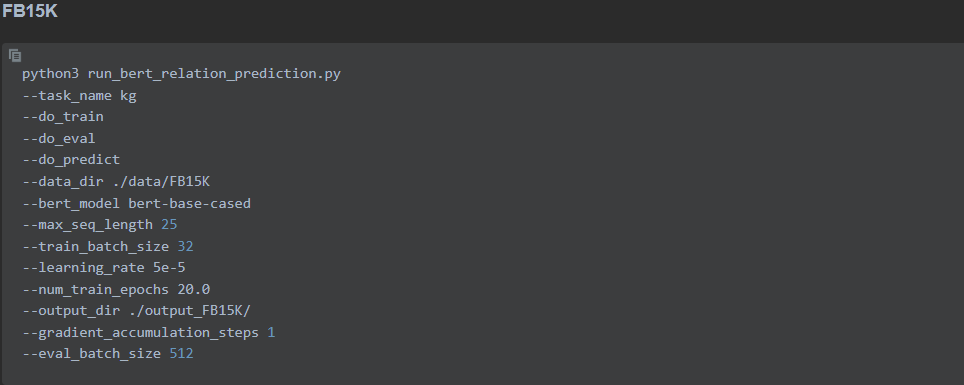
神经网络模型中中的输入,一般为四维向量(batch\_size+RGB)。

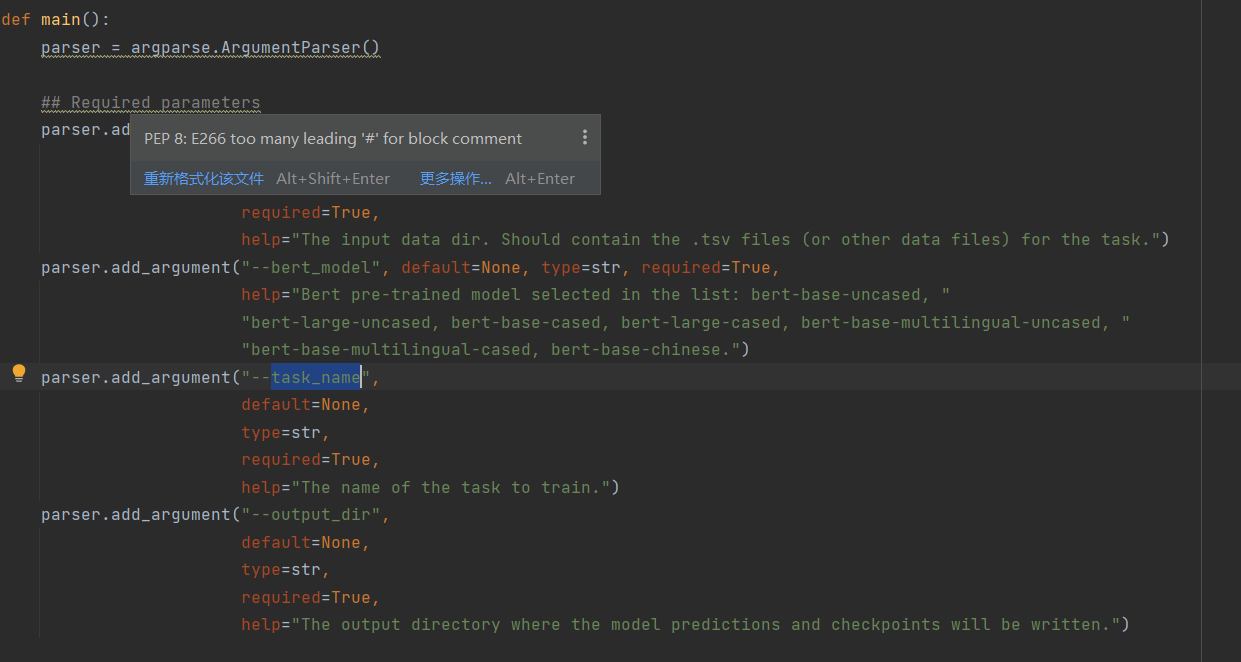
1. github上的开源项目

1.pytorch->Most stars



2.README中的参数设置





python3 run\_bert\_relation\_prediction.py

--task\_name kg

--do\_train

--do\_eval

--do\_predict

--data\_dir ./data/FB15K

--bert\_model bert-base-cased

--max\_seq\_length 25

--train\_batch\_size 32

--learning\_rate 5e-5

--num\_train\_epochs 20.0

--output\_dir ./output\_FB15K/

--gradient\_accumulation\_steps 1

--eval\_batch\_size 512

//运行名为run\_bert\_relation\_prediction的Python文件,并为参数指定其值

parser.add\_argument("--task\_name",  
 default=None,  
 type=str,  
 required=True,  
 help="The name of the task to train.")

required=True 表明必须指定该参数的值

(技巧：删除required,添加default和type)

help="The name of the task to train." 表明参数的意义

parser.add\_argument("--max\_seq\_length",  
 default=128,  
 type=int,  
 help="The maximum total input sequence length after WordPiece tokenization. \n"  
 "Sequences longer than this will be truncated, and sequences shorter \n"  
 "than this will be padded.")

default=128 表明未指定该参数值时,默认值为128(无required=True)

parser.add\_argument("--do\_train",  
 action='store\_true',  
 help="Whether to run training.")

action='store\_true' 表明如果命令行存在--do\_train的参数,代码中

args.do\_train=True