# 环境安装

## Anaconda

### 下载

<https://www.anaconda.com/download>

### 命令行

|  |
| --- |
| **以管理员身份运行！**  创建环境：conda create -n xxx python=3.10  查看：pip list/conda list  进入环境：conda activate xx  查看环境：conda info --envs  安装torch：在torch的网址生成命令安装  安装jupyter：conda install nb\_conda  启动jupyter：jupyter notebook  查看显卡版本：nvidia-smi |

### 其他

|  |
| --- |
| 查看显卡是否支持CUDA: <https://developer.nvidia.com/zh-cn/cuda-gpus>  查看函数的所有方法：dir(torch) # 注意在python中使用，且先要import  默认环境位置：C:\Users\name\.conda\envs |

## Pytorch

### 网址

<https://pytorch.org/>

### 接口

|  |
| --- |
| 查看是否支持GPU加速：torch.cuda.is\_available() |

# torch

## Dataset

### 重写特殊方法

1. \_\_getitem\_\_

|  |
| --- |
| 允许用户直接用索引访问对象中的数据。  **def** \_\_getitem\_\_(self, item):  item\_path = os.path.join(self.path, self.label\_ls[item])  img = Image.open(item\_path)  **return** img  当对象为obj时，obj[1]则直接调用\_\_getitem\_\_，并把1当入参到getitem，然后返回方法中的返回值。 |

1. \_\_len\_\_

|  |
| --- |
| 允许用户使用python的len函数求对象的长度时。  **def** \_\_len\_\_(self):  **return** len(self.label\_ls)  当求len(obj)时，会直接调用\_\_len\_\_方法，并返回。 |

1. \_\_add\_\_

|  |
| --- |
| 允许用户通过+运算符操作对象。  **def** \_\_add\_\_(self, other: **'Dataset[T\_co]'**) -> **'ConcatDataset[T\_co]'**:  **return** ConcatDataset([self, other])  当用户使用+运算符操作两个对象时，会把后一个Dataset对象当作入参传给\_\_add\_\_方法，默认返回ConcatDataset后的对象。  扩展：  ConcatDataset方法继承Dataset类，调用时入参对象列表，返回\_\_getitem\_\_方法的返回值，其他方法都和Dataset的方法类似，只不过把每个Dataset对象的这些方法返回的结果进行了相加。  （简版，详细看代码接口）  **class** ConcatDataset(Dataset[T\_co]):  @staticmethod  **def** cumsum(sequence):  **return** “所有Dataset对象的长度和”   **def** \_\_init\_\_(self, datasets: Iterable[Dataset]) -> **None**:  ...   **def** \_\_len\_\_(self):  **return** “所有Dataset对象的长度和”   **def** \_\_getitem\_\_(self, idx):  **return** “二分法判断idx属于哪个对象，再取对象的索引” |

### 演示

|  |
| --- |
| 如下demo：  *#!/usr/bin/env python # -\*- coding: utf-8 -\*- # @Time : 2023/6/3 19:00 # @Author : 刘双喜 # @File : 32.dataset.py # @Description : Dataset类演示* **from** torch.utils.data **import** Dataset, ConcatDataset **import** os **from** PIL **import** Image   **class** GetImgData(Dataset):  **def** \_\_init\_\_(self, src\_dir, label\_dir):  self.src\_dir, self.label\_dir = src\_dir, label\_dir  self.path = os.path.join(src\_dir, label\_dir)  self.label\_ls = os.listdir(self.path)   **def** \_\_getitem\_\_(self, item):  item\_path = os.path.join(self.path, self.label\_ls[item])  # 打开对应列表索引的名称的图片  img = Image.open(item\_path)  **return** img   **def** \_\_len\_\_(self):  **return** len(self.label\_ls)   **def** \_\_add\_\_(self, other):  **return** ConcatDataset([self, other])   **if** \_\_name\_\_ == **'\_\_main\_\_'**:  src\_dir =**'..\\..\\dataset\\32.Dataset\\hymenoptera\_data\\train'** ants = GetImgData(src\_dir, **'ants'**)  bees = GetImgData(src\_dir, **'bees'**)  all = ants + bees  all[1].show() |

## SummaryWriter

### 数据显示

|  |
| --- |
| TensorBoard 是一个可视化工具，它可以帮助你更好地理解、调试和优化机器学习模型。它提供了丰富的可视化功能，包括标量图表、图像展示、音频播放、计算图可视化、分布图表等。  要使用 TensorBoard，你需要将数据写入到事件文件中。你可以使用 TensorFlow、PyTorch 或其他机器学习框架提供的工具来完成这个任务。例如，在 PyTorch 中，你可以使用 SummaryWriter 类来将数据写入到事件文件中。  当你准备好了事件文件后，你可以在命令行中使用 tensorboard 命令来启动 TensorBoard 服务器。例如，如果你的事件文件位于 logs 目录下，那么你可以使用以下命令来启动 TensorBoard：  tensorboard --logdir=logs –port=6006 |

### scalar

|  |
| --- |
| 数值类型的数据写入事件文件并展示示例：  **from** torch.utils.tensorboard **import** SummaryWriter **import** os   **with** SummaryWriter(**'..\\data\\33.tensorborad'**) **as** write:  **for** i **in** range(100):  *# 每次给图表取名不同防止多个线条拟合在一起*  write.add\_scalar(**f"chart{**len(os.listdir(**'../data/33.tensorborad'**))**} :y->x"**, i\*i, i)  # add\_scalar入参:图表名称,y,x  # 查看图表，注意要有tensorboard环境，且logdir指定正确  tensorboard --logdir=. --port=6012 |

### image

|  |
| --- |
| 图片类型的数据写入事件文件：  **import** os **from** PIL **import** Image **import** numpy **as** np **from** torch.utils.tensorboard **import** SummaryWriter   dir\_img = **'..\\..\\dataset\\34.tensorboard\_img\_test\\train\\ants\_image' with** SummaryWriter(**'..\\data\\34.tensorborad\_img'**) **as** write:  **for** idx, img\_name **in** enumerate(os.listdir(dir\_img)):  img = Image.open(os.path.join(dir\_img,img\_name))  **if** img **is not None**:  img\_arr = np.array(img)  write.add\_image(**'ants'**, img\_arr, idx, dataformats=**'HWC'**)  # add\_image入参:图片,数据,步数,dataformats为图片格式  # 添加多个图片用add\_images  # 查看图表，注意要有tensorboard环境，且logdir指定正确  tensorboard --logdir=. --port=6012 |

### graph

|  |
| --- |
| add\_graph 方法接受两个必需参数和一个可选参数：  model (nn.Module) - 要可视化的 PyTorch 模型。  input\_to\_model (torch.Tensor or tuple of torch.Tensor) - 用于模型的输入张量。如果模型需要多个输入，则应将它们作为元组传递。  verbose (bool, optional) - 如果为 True，则在添加计算图时打印更多信息。默认值为 False。 |

## DataLoader

### 介绍

|  |
| --- |
| DataLoader是PyTorch中的一个类，用于批量加载数据。它接受一个数据集作为输入，并提供了一个迭代器，用于按批次获取数据。  from torch.utils.data import DataLoader |

### API

|  |
| --- |
| **DataLoader的构造函数接受以下参数：**  dataset：要加载的数据集。  batch\_size：每个批次的数据量。  shuffle：如果为True，则在每次访问时打乱数据，即使只实例化了一次对象。  sampler：定义从数据集中抽取样本的策略。如果指定了sampler，则不能同时指定shuffle。  batch\_sampler：与sampler类似，但一次返回一批索引。如果指定了batch\_sampler，则不能同时指定batch\_size、shuffle、sampler和drop\_last。  num\_workers：用于数据加载的子进程数。默认值为0，表示在主进程中加载数据。  collate\_fn：用于将多个样本合并成一个批次的函数。  pin\_memory：如果为True，则数据加载器会将张量复制到CUDA固定内存中，然后返回它们。  drop\_last：如果为True，则丢弃最后一个不完整的批次。  返回值是n个 一个批次的图像imgs和一个批次的标签labels。  imgs的size是(一个批次处理的图像数量,图像通道,长,宽) |

### 例子

|  |
| --- |
| **import** torchvision **from** torch.utils.data **import** DataLoader **from** torch.utils.tensorboard **import** SummaryWriter  test\_data = torchvision.datasets.CIFAR10(**'../../dataset/36.datasets'**, train=**False**,transform=torchvision.transforms.ToTensor(),  download=**True**)  test\_loader = DataLoader(dataset=test\_data, batch\_size=100, shuffle=**True**, num\_workers=0, drop\_last=**True**)  **with** SummaryWriter(**'..\\data\\37.dataloader'**) **as** write:  **for** epoch **in** range(2):  idx = 0  **for** data **in** test\_loader:  imgs, labels = data  write.add\_images(**f"Epoch:{**epoch**}"**, imgs, idx)  idx += 1 |

## functional

### 介绍

|  |
| --- |
| torch.nn 库中的一个模块，它包含了许多函数，不具有学习参数：损失函数、激活函数、池化函数、进行卷积、线性层等的函数。 |

### API

#### 卷积

|  |
| --- |
| 卷积是一种常用的数学运算，它在不同的领域和应用场景中有不同的计算方法。例如，在信号处理中，卷积可以用来描述线性时不变系统的输入和输出的关系；在图像处理中，卷积可以用来提取图像特征，实现图像模糊、锐化、边缘检测等效果。不同的应用场景对卷积的计算方法和实现方式有不同的要求。  卷积运算本身也可以通过多种算法进行计算，例如直接计算、快速傅立叶转换（FFT）和分段卷积（sectioned convolution）等。这些算法各有优缺点，在不同的应用场景中可以根据需要选择合适的算法。  torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)    **import** torch **from** torch.nn **import** functional **as** F  input = torch.tensor([[1, 2, 0, 3, 1],  [0, 1, 2, 3, 1],  [1, 2, 1, 0, 1],  [5, 2, 3, 1, 1],  [2, 1, 0, 1, 1]])  kernel = torch.tensor([[1, 2, 3],  [0, 1, 0],  [2, 1, 0]])  input = torch.reshape(input, (1, 1, 5, 5)) kernel = torch.reshape(kernel, (1, 1, 3, 3))  output1 = F.conv2d(input, kernel, stride=1) output2 = F.conv2d(input, kernel, stride=2)  print(output1) print(output2) |

## nn

### 介绍

|  |
| --- |
| torch.nn是神经网络的基本骨架，包含了各种用于构建神经网络的类和函数，可以用于训练。如各种卷积层、池化层、激活函数、损失函数等。你可以把 torch.nn 包内的各个类想象成神经网络的一层，它们包括该层的参数以及一些操作，如 forward 和调用参数等。  torch.nn 模块中有许多常用的类和函数，下面是一些常用的类：  nn.Module：所有神经网络模块的基类。  nn.Linear：全连接层。  nn.Conv1d、nn.Conv2d、nn.Conv3d：一维、二维和三维卷积层。  nn.BatchNorm1d、nn.BatchNorm2d、nn.BatchNorm3d：一维、二维和三维批量归一化层。  nn.ReLU、nn.Sigmoid、nn.Tanh 等：各种激活函数。  nn.MSELoss、nn.CrossEntropyLoss 等：各种损失函数。  自己的神经网络需要继承nn.Module并初始化父类的方法，然后重写forward函数。  父类定义了\_\_call\_\_方法，每次用对象(x)时，会自动执行forward方法。 |

### 搭建

|  |
| --- |
| **import** torch **from** torch **import** nn   **class** Demo(nn.Module):  **def** \_\_init\_\_(self):  super(Demo, self).\_\_init\_\_()   **def** forward(self, input):  **return** input + 1  demo1 = Demo() a = torch.tensor(1) print(demo1(a)) |

## 保存和加载

### 介绍

|  |
| --- |
| 模型在内存中是以对象的逻辑结构保存的，但是在硬盘中是以二进制流的方式保存的。  序列化是指将内存中的数据以二进制序列的方式保存到硬盘中。PyTorch 的模型保存就是序列化。  反序列化是指将硬盘中的二进制序列加载到内存中，得到模型的对象。PyTorch 的模型加载就是反序列化。 |

### 方法1

|  |
| --- |
| 保存整个模型，包括模型+参数，比较慢比较大，不推荐。  *# 方法1*  **import** torch **from** torchvision.models **import** VGG16\_Weights, vgg16  vgg16\_model = vgg16(weights=VGG16\_Weights.IMAGENET1K\_V1)  torch.save(vgg16\_model, **'..\\data\\48.module\_test\\vgg16\_method.pth'**) model1 = torch.load(**'..\\data\\48.module\_test\\vgg16\_method.pth'**) |

### 方法2

|  |
| --- |
| 只保存模型参数，推荐。这种方法必须已经有模型的定义（有定义class）。  *# 方法2*  **import** torch **from** torchvision.models **import** VGG16\_Weights, vgg16  vgg16\_model = vgg16(weights=VGG16\_Weights.IMAGENET1K\_V1)  torch.save(vgg16\_model.state\_dict(), **'..\\data\\48.module\_test\\vgg16\_method2.pth'**)  # 先实例化模型 demo = vgg16()  # 再加载 demo.load\_state\_dict(torch.load(**'..\\data\\50.model\\module\_epoch30.pth'**, map\_location=torch.device(**'cpu'**))) |

## other

### 创建

|  |
| --- |
| torch.tensor(data) - 从现有数据（例如列表、数组或其他张量）创建一个张量。  torch.zeros(size) - 创建一个形状为 size 的全零张量。  torch.ones(size) - 创建一个形状为 size 的全一张量。  torch.rand(size) - 创建一个形状为 size 的随机张量，其中元素在 [0, 1] 之间均匀分布。  torch.randn(size) - 创建一个形状为 size 的随机张量，其中元素服从标准正态分布（均值为 0，标准差为 1）。  torch.arange(start, end, step) - 创建一个从 start 到 end（不包括），以 step 为步长的一维张量。  torch.linspace(start, end, steps) - 创建一个从 start 到 end（包括）的一维张量，其中元素在这个范围内均匀分布，共有 steps 个元素。  size为整数或整数元组。 |

### 展平

|  |
| --- |
| torch.flatten 是一个函数，它用于将输入张量展平为一维张量。它接受以下参数：  input（张量）：输入张量。  start\_dim（整数，可选）：要展平的第一个维度。默认值为 0。  end\_dim（整数，可选）：要展平的最后一个维度。默认值为 -1。  如果传递了 start\_dim 或 end\_dim 参数，则只有从 start\_dim 开始到 end\_dim 结束的维度会被展平。输入中元素的顺序不变。 |

### 改变形状

|  |
| --- |
| torch.reshape用于将输入张量重塑为指定形状。它接受以下参数：  input（张量）：输入张量。  shape（整数元组）：新的形状，shape参数中的-1表示该维度的大小由其他维度的大小和输入张量中元素的总数推断得出。 |

### 截取数据集

|  |
| --- |
| from torch.utils.data import Subset  作用：从数据集取固定数量的数据。  **from** torch.utils.data **import** DataLoader, Subset  test\_data = torchvision.datasets.CIFAR10(...)  test\_data = Subset(test\_data, range(1000)) # 取1000个数据 |

### 序列容器

|  |
| --- |
| torch.nn.Sequential 是一个序列容器。模块将按照构造函数中传递的顺序添加到其中。  model = nn.Sequential(  nn.Conv2d(1,20,5),  nn.ReLU(),  nn.Conv2d(20,64,5),  nn.ReLU()  )  或清晰的命名：  model = nn.Sequential(OrderedDict([  ('conv1', nn.Conv2d(1,20,5)),  ('relu1', nn.ReLU()),  ('conv2', nn.Conv2d(20,64,5)),  ('relu2', nn.ReLU())  ])) |

### 求n维度最大值

|  |
| --- |
| a = torch.tensor([[1, 2, 3], [4, 5, 6]])  print(torch.argmax(a, dim=0)) # 返回第0维每个元素里最大值的索引  # a.argmax(dim=0) 或者这样写  # tensor([1, 1, 1])  print(torch.argmax(a, dim=1)) # 返回第1维每个元素里最大值的索引  # tensor([2, 2]) |

# torchvision

## transforms

### 图像预处理

|  |
| --- |
| transforms是PyTorch中的一个模块，它提供了一些常用的图像预处理方法。这些方法可以在将图像输入神经网络之前对其进行预处理，例如缩放、裁剪、翻转、旋转等。transforms模块中的每个方法都可以看作是一个函数，它接收一个图像作为输入，然后对其进行处理，最后返回处理后的图像。  可以使用transforms.Compose函数将多个预处理方法组合在一起，形成一个预处理流水线。 |

### 演示

|  |
| --- |
| 读取图片，并进行预处理，然后写入事件文件。  **import** os **from** torch.utils.tensorboard **import** SummaryWriter **from** torchvision **import** transforms **import** cv2 **from** PIL **import** Image *# 预处理-转换为张量 # preprocess = transforms.ToTensor() # 定义预处理过程集合* preprocess = transforms.Compose([  transforms.Resize(256), *# 缩放到256像素* transforms.CenterCrop(224), *# 在中心裁剪出224\*224像素* transforms.ToTensor() *# 转换为张量* ]) dir\_img = **'..\\..\\dataset\\34.tensorboard\_img\_test\\train\\ants\_image' with** SummaryWriter(**'..\\data\\34.tensorborad\_img'**) **as** write:  **for** idx, img\_name **in** enumerate(os.listdir(dir\_img)):  *# 读取图片  # img = cv2.imread(os.path.join(dir\_img, img\_name)) # 不能读git, 返回数组类型,不能用于tansforms* img = Image.open(os.path.join(dir\_img, img\_name))  **if** img **is not None**:  *# 预处理* tensor\_img = preprocess(img)  *# 写入事件文件* write.add\_image(**'ants'**, tensor\_img, idx, dataformats=**'CHW'**) |

### 功能集

#### 公共

|  |
| --- |
| 组合功能：transforms.Compose([, ,])  入参列表,可以同时进行多个功能的变换。  转换张量：transforms.ToTensor()  转为张量类型。 |

#### 裁剪

|  |
| --- |
| **中心裁剪：transforms.CenterCrop(size)**  作用：裁剪中心区域多少像素.可入参(H,W)  输入：size=int，size=(H,W)  **随机裁剪：transforms.RandomCrop(size, padding=None, pad\_if\_needed=False, fill=0, padding\_mode=‘constant’)**  作用：随机裁剪224\*224像素.  输入：  size：裁剪大小；size=int，size=(H,W)  padding：填充像素值；padding=4:上下左右填充4；=(1,4):左右填充1，上下4；=(1,2,3,4),左上右下填充值1，2，3，4.  pad\_if\_needed：指定当图像小于给定大小时是否添加填充。默认为False  fill：填充的颜色。单通道：整数；多通道：RGB元组(x,x,x)。如果输入图像是torch Tensor，则fill参数仅支持数字。如果输入图像是PIL Image，则fill参数仅支持int或tuple值。  padding\_mode：用于指定填充模式。它可以是以下值之一：  'constant'：使用常数填充。常数值由fill参数指定。  'edge'：使用图像边缘的像素值进行填充。  'reflect'：使用图像边缘的镜像进行填充。最后一个像素不镜像。例如，对于图像[1, 2, 3, 4]，使用'reflect'模式填充两个像素后，结果为[**3, 2**, 1, **2, 3**, 4, **3, 2**]。  'symmetric'：使用图像边缘的镜像进行填充。最后一个像素镜像。例如，对于图像[1, 2, 3, 4]，使用'symmetric'模式填充两个像素后，结果为[**2, 1, 1, 2**, 3, 4, 4, 3]。  **随机长宽比裁剪：transforms.RandomResizedCrop**  作用：随机裁剪并缩放图像。  输入：  size：指定裁剪后的图像大小。  scale：指定裁剪面积的范围。默认为(0.08, 1.0)。  ratio：指定长宽比的范围。默认为(3/4, 4/3)。  interpolation：指定插值(缩放时增加或保留值的方法)方法。默认为PIL.Image.BILINEAR。  **上下左右中心裁剪：transforms.FiveCrop**  作用：从图像的四个角和中心裁剪出给定大小的图像。  输入：size=int，size=(H,W)  输出：返回一个包含五个图像的元组，分别对应左上角、右上角、左下角、右下角和中心的裁剪结果  **上下左右中心裁剪后翻转: transforms.TenCrop**  作用：用于从图像的四个角和中心裁剪出给定大小的图像，并对每个裁剪结果进行水平翻转。  输入：  size：指定裁剪后的图像大小。size=int，size=(H,W)。  vertical\_flip：指定是否进行垂直翻转。默认为False，即进行水平翻转。  输出：返回一个包含十个图像的元组，分别对应左上角、右上角、左下角、右下角和中心的裁剪结果及其翻转结果 |

#### 翻转

|  |
| --- |
| **依概率p水平翻转：transforms.RandomHorizontalFlip**  作用：依据概率p对PIL图片进行水平翻转  输入：p:概率，默认值为0.5.  **依概率p垂直翻转transforms.RandomVerticalFlip**  作用：依据概率p对PIL图片进行水平翻转  输入：p:概率，默认值为0.5.  **随机旋转：transforms.RandomRotation**  作用：用于随机旋转图像**。**  输入：  degrees：int=30，从(-30,30)随机旋转；tuple=(30,60)，则之间随机旋转。  interpolation：期望的枚举类型。  expand：可选扩展标志。如果为 true，则扩展输出以使其足够大以容纳整个旋转图像。如果为 false 或省略，则使输出图像与输入图像大小相同  center：可选旋转中心（x，y）。原点是左上角。默认为图像中心。  fill：指定填充值。对于单通道图像，可以是一个整数；对于多通道图像，可以是一个RGB元组。 |

#### 图像变换

|  |
| --- |
| **转为tensor:transforms.ToTensor**  作用：将PIL Image或者 ndarray转换为tensor，并且归一化至[0-1] 注意事项：归一化至[0-1]是直接除以255，若自己的ndarray数据尺度有变化，则需要自行修改。  **将数据转换为PILImage：transforms.ToPILImage**  作用：用于将张量或ndarray转换为PIL Image，同时保留值范围。此转换不支持torchscript。  输入：  mode：输入数据的颜色空间和像素深度。例如：1, L, P, RGB, RGBA, CMYK, YCbCr, LAB, HSV。  mode为None（默认），其他：  如果输入具有4个通道，则假定模式为RGBA。  如果输入具有3个通道，则假定模式为RGB。  如果输入具有2个通道，则假定模式为LA。  如果输入具有1个通道，则由数据类型（即int，float，short）确定模式1。  **重设分辨率:transforms.Resize(size,interpolation=2)**  作用：转换像素大小为目标大小。  输入：  size: size=int，size=(H,W)  interpolation：期望的插值枚举  max\_size：调整后图像最长边的允许值。仅当size=int时生效。  antialias：是否应用抗锯齿。对于 PIL 图像，在双线性或双三次模式下始终应用抗锯齿；对于其他模式（对于 PIL 图像和张量），抗锯齿没有意义并且此参数被忽略 1。  **正则化:transforms.Normalize(mean, std, inplace=False)**  作用：用于使用均值和标准差对张量图像进行归一化。此转换不支持 PIL Image。只能作用于张量图片。  输入：  mean：均值。各个通道的均值(mean 1，…，mean [n]).  std：标准差。各个通道的标准差(std 1，…，std [n]).  **填充：transforms.Pad**  作用：对图像进行填充。  输入：  padding：填充像素值；padding=4:上下左右填充4；=(1,4):左右填充1，上下4；=(1,2,3,4),左上右下填充值1，2，3，4.  fill：填充的颜色。单通道：整数；多通道：RGB元组(x,x,x)。如果输入图像是torch Tensor，则fill参数仅支持数字。如果输入图像是PIL Image，则fill参数仅支持int或tuple值。  padding\_mode：用于指定填充模式。它可以是以下值之一：  'constant'：使用常数填充。常数值由fill参数指定。  'edge'：使用图像边缘的像素值进行填充。  'reflect'：使用图像边缘的镜像进行填充。最后一个像素不镜像。例如，对于图像[1, 2, 3, 4]，使用'reflect'模式填充两个像素后，结果为[3, 2, 1, 2, 3, 4, 3, 2]。  'symmetric'：使用图像边缘的镜像进行填充。最后一个像素镜像。例如，对于图像[1, 2, 3, 4]，使用'symmetric'模式填充两个像素后，结果为[2, 1, 1, 2, 3, 4, 4, 3]。  **随机更改图像的亮度、对比度、饱和度和色调:transforms.ColorJitter**  作用：随机更改图像的亮度、对比度、饱和度和色调  输入：  brightness：int，亮度调整因子。brightness\_factor从 [max(0, 1 - brightness), 1+brightness]中随机选择。默认为0。  contrast：对比度调整因子。contrast\_factor从 [max(0, 1 - contrast), 1+contrast]中随机选择。默认为0 。  saturation：饱和度调整因子。saturation\_factor从[max(0, 1 - saturation), 1+saturation]中随机选择。默认为0。  hue：色调调整因子。hue\_factor从 [-hue, hue]中随机选择。默认为0。  **转灰度图：transforms.Grayscale**  作用：将图像转为灰度图。  输入：  num\_output\_channels：期望输出图像的通道数（1或3）。为3时，每个通道的值都相等。  **依概率p转为灰度图：transforms.RandomGrayscale**  作用：用于以概率p（默认为0.1）随机将图像转换为灰度图  输入：  p：图像应转换为灰度图的概率。返回输入图像的灰度版本的概率为p，不变的概率为（1-p）。如果输入图像是1通道：灰度版本是1通道。如果输入图像是3通道：灰度版本是3通道，r==g==b。  **仿射变换:transforms.RandomAffine**  作用：随机仿射变换是一种图像变换方法，它可以对图像进行旋转、缩放、剪切和平移等操作。在`transforms.RandomAffine`中，这些操作的参数是随机选择的，因此每次应用该转换时，都会得到不同的结果。这种变换通常用于数据增强，以增加模型的鲁棒性。  输入：  degrees：可从中选择的度数范围。如果degrees是数字而不是序列（如（min，max）），则度数范围将为（-degrees，+degrees）。设置为0以停用旋转1。  translate：水平和垂直平移的最大绝对分数元组。例如translate=（a，b），则水平移位在范围-img\_widtha<dx<img\_widtha内随机采样，垂直移位在范围-img\_heightb<dy<img\_heightb内随机采样。默认不平移1。  scale：缩放因子区间，例如（a，b），则从范围a<=scale<=b内随机采样比例。默认保持原始比例1。  shear：可从中选择的度数范围。如果shear是数字，则将应用范围为（-shear，+shear）的与x轴平行的剪切。否则，如果shear是2个值的序列，则将应用范围为（shear[0]，shear1）的与x轴平行的剪切。否则，如果shear是4个值的序列，则将应用范围为（shear[0]，shear1）的x轴剪切和范围为（shear2，shear3）的y轴剪切。默认不应用剪切1。  interpolation：期望的插值枚举由torchvision.transforms.InterpolationMode定义。默认为InterpolationMode.NEAREST。如果输入是Tensor，则仅支持InterpolationMode.NEAREST和InterpolationMode.BILINEAR1。  fill：转换图像外部区域的像素填充值。默认为0。如果给定一个数字，则该值分别用于所有带宽1。  center：可选旋转中心（x，y）。原点是左上角。默认为图像中心1。  **线性变换：transforms.LinearTransformation()**  作用：使用一个矩阵和一个向量来变换张量图像。可以用于白化变换，白化变换用于去除输入数据的冗余信息。不支持Image。  输入：  transformation\_matrix：一个方阵，用于对图像张量进行线性变换。它的形状为 [D x D]，其中 D = CxHxW，C是图像的通道数，H和W分别是图像的高度和宽度。  mean\_vector：一个向量，用于对图像张量进行平移变换。它的形状为 [D]，其中D=CxHxW。  这两个参数共同定义了一个线性变换，用于对图像张量进行变换。  **自定义函数变换：transforms.Lambda**  作用：自定义函数对图像进行变换。  输入：lambda，自定义的函数。 |

#### 随机变换

|  |
| --- |
| **随机选择图像变换：transforms.RandomChoice**  作用：随机选择一个变换的方法来进行变幻。  输入：  transforms：随机变换的列表。  p：每个变换的概率的列表，如果为None则每个变换概率相等。  **随机概率应用于变换列表：transforms.RandomApply**  作用：对多个变换的列表，同时以相同的随机概率应用于列表的变换。  输入：  transforms：随机变换的列表。  p：所有变换的概率。  **随机顺序应用变换列表：transforms.RandomOrder**  作用：以随机顺序应用一个变换列表。它只有一个参数 transforms，表示要应用的变换列表。 |

## datasets

### 数据集

|  |
| --- |
| Torchvision提供了许多内置数据集，它们都在torchvision.datasets模块中。所有数据集都是torch.utils.data.Dataset的子类，即它们都实现了\_\_getitem\_\_和\_\_len\_\_方法。因此，它们都可以传递给torch.utils.data.DataLoader，它可以使用torch.multiprocessing workers并行加载多个样本。  常见的数据集：  Caltech101: 这是一个包含101个类别的图像数据集，每个类别包含大约40到800张图像。  Caltech256: 这是一个包含256个类别的图像数据集，每个类别至少有80张图像。它被认为是Caltech101数据集的改进版，具有更大的类别大小，新的和更大的杂乱类别，以及整体难度增加。  CelebA: 这是一个大规模的人脸属性数据集，包含超过200,000张名人图像，每张图像都有40个属性注释。这些图像涵盖了大量的姿态变化和背景杂乱。  CIFAR10: 这是一个包含10个类别的图像数据集，每个类别包含6000张32x32彩色图像。共有50,000张训练图像和10,000张测试图像。 |

### API

|  |
| --- |
| 1. CIFAR10  train\_data = torchvision.datasets.CIFAR10(root=**'../../dataset/36.datasets'**, train=**True**, transform=transforms, download=**True**)  root:下载数据集存放的地址  train:下载训练集(True)还是测试集(FALSE)  transform:将图像进行转换  target\_transform:转换标签值  download:是否从互联网下载数据，FALSE则使用本地数据集。  2. CelebA  root：数据集的根目录，其中存储着数据集的文件。  split：要加载的数据集划分，可以是’train’（训练集）、‘valid’（验证集）、‘test’（测试集）或’all’。  target\_type：目标类型，可以是’attr’、‘identity’、‘bbox’或’landmarks’。  attr：加载属性注释。每张图像都有40个二进制属性注释。  identity：加载身份注释。每张图像都有一个身份标签。  bbox：加载边界框注释。每张图像都有一个边界框，表示人脸的位置。  landmarks：加载地标注释。每张图像都有5个地标位置，表示左眼、右眼、鼻子、左嘴和右嘴的位置。  transform：一个可调用对象，用于转换图像。  target\_transform：一个可调用对象，用于转换目标。  download：如果为True，则从互联网下载数据并将其放在根目录下。  返回值：  返回对应的数据集对象obj，可用obj[idx]来获取对象的img和标签值索引。 |

# 模型训练

## 介绍

|  |
| --- |
| 深度学习模型通常由多个层组成，每个层都有其特定的作用。下面是一些常见的层及其作用：   * 输入层：负责接收输入数据。 * 卷积层：通过卷积运算对输入数据进行特征提取。 * 池化层：对输入数据进行降采样，减少数据的空间大小，即特征降维。 * 全连接层：将前一层的所有神经元与当前层的所有神经元相连，用于分类或回归任务。 * 激活函数层：对输入数据进行非线性变换，增强模型的表达能力。 * 线性变换层：对输入数据进行线性变换，增强模型的表达能力。 * 输出层：输出模型的预测结果。   这些层可以按照不同的顺序和组合方式构建深度学习模型，以解决不同类型的问题。 |

## 卷积层

### 介绍

|  |
| --- |
| * 卷积层([Convolution Layers](https://pytorch.org/docs/stable/nn.html#convolution-layers))是卷积神经网络中的一个重要组成部分，它由若干个卷积单元组成，每个卷积单元的参数都是通过反向传播算法最优化得到的。卷积层的功能是对输入数据进行特征提取，其内部包含多个卷积核，组成卷积核的每个元素都对应一个权重系数和一个偏差量（bias vector），类似于一个前馈神经网络的神经元（neuron）。 |

### 卷积计算

|  |
| --- |
|  |

### Conv2d

|  |
| --- |
| [[PyTorch 学习笔记] 3.2 卷积层 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/206427963)  nn.Conv2d(self, in\_channels, out\_channels, kernel\_size, stride=1,  padding=0, dilation=1, groups=1,  bias=True, padding\_mode='zeros')  这个函数的功能是对多个二维信号进行二维卷积，主要参数如下：  in\_channels：输入通道数  out\_channels：输出通道数，等价于卷积核个数  kernel\_size：卷积核尺寸  stride：步长  padding：填充宽度，主要是为了调整输出的特征图大小，一般把 padding 设置合适的值后，保持输入和输出的图像尺寸不变。  dilation：空洞卷积大小，默认为 1，这时是标准卷积，常用于图像分割任务中，主要是为了提升感受野  groups：分组卷积设置，主要是为了模型的轻量化，如在 ShuffleNet、MobileNet、SqueezeNet 中用到  bias：偏置  通过公式计算输出的长宽，或者已知长宽逆推padding参数。    代码示例：  **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   **class** Demo(nn.Module):  **def** \_\_init\_\_(self):  super(Demo, self).\_\_init\_\_()  self.conv = Conv2d(in\_channels=3, out\_channels=6, kernel\_size=3, stride=1, padding=0)   **def** forward(self, x):  **return** self.conv(x)   test\_data = torchvision.datasets.CIFAR10(**'../../dataset/36.datasets'**, train=**False**,  transform=torchvision.transforms.ToTensor(),  download=**True**)  test\_loader = DataLoader(dataset=test\_data, batch\_size=100, shuffle=**False**, num\_workers=0, drop\_last=**True**)  demo = Demo() *# tensorboard --logdir='.\\example\\data\\40.nn\_Conv2d' --port=6012* **with** SummaryWriter(**'..\\data\\40.nn\_Conv2d'**) **as** write:  **for** step, data **in** enumerate(test\_loader):  **if** step > 50:  **continue** imgs, lables = data  *# print(imgs.shape) torch.Size([64, 3, 32, 32])* conv\_imgs = demo(imgs)  *# print(conv\_imgs.shape) torch.Size([64, 6, 30, 30])* conv\_imgs = torch.reshape(conv\_imgs, (-1, 3, 30, 30))  *# print(conv\_imgs.shape) torch.Size([128, 3, 30, 30])* write.add\_images(**'input'**, imgs, step)  write.add\_images(**'output'**, conv\_imgs, step) |

## 池化层

### 介绍

|  |
| --- |
| * 池化层([Pooling layers](https://pytorch.org/docs/stable/nn.html#pooling-layers))是卷积神经网络中的一种操作，它的作用主要体现在降采样：保留显著特征、降低特征维度，增大 kernel 的感受野。在处理多通道输入数据时，池化层对每个输入通道分别池化，而不是像卷积层那样将各通道的输入按通道相加。 |

### MaxPool2d

|  |
| --- |
| 二维最大池化：  nn.MaxPool2d(kernel\_size, stride=None, padding=0, dilation=1, return\_indices=False, ceil\_mode=False)  kernel\_size：池化核尺寸  stride：步长，通常与 kernel\_size 一致  padding：填充宽度，主要是为了调整输出的特征图大小，一般把 padding 设置合适的值后，保持输入和输出的图像尺寸不变。  dilation：池化间隔大小，默认为 1。常用于图像分割任务中，主要是为了提升感受野  ceil\_mode：默认为 False，尺寸向下取整。为 True 时，尺寸向上取整  return\_indices：为 True 时，返回最大池化所使用的像素的索引，这些记录的索引通常在反最大池化时使用，把小的特征图反池化到大的特征图时，每一个像素放在哪个位置。  池化层的输出大小可以通过以下公式计算：**output\_size = (input\_size - kernel\_size + 2 \* padding) / stride + 1**。其中，input\_size 是输入特征图的尺寸，kernel\_size 是池化核的大小，padding 是填充值，stride 是步幅。  示例：  **class** Demo(nn.Module):  **def** \_\_init\_\_(self):  super(Demo, self).\_\_init\_\_()  self.maxpool = MaxPool2d(kernel\_size=2, ceil\_mode=**False**)   **def** forward(self, x):  **return** self.maxpool(x)   test\_data = torchvision.datasets.CIFAR10(**'../../dataset/36.datasets'**, train=**False**,  transform=torchvision.transforms.ToTensor(),  download=**True**)  test\_loader = DataLoader(dataset=test\_data, batch\_size=64, shuffle=**False**, num\_workers=0, drop\_last=**True**)  demo = Demo() **with** SummaryWriter(**'..\\data\\41.nn\_maxpool'**) **as** write:  **for** step, data **in** enumerate(test\_loader):  imgs, lables = data  write.add\_images(**'input'**, imgs, step)  output = demo(imgs)  write.add\_images(**'output'**, output, step) |

### AvgPool2d

|  |
| --- |
| 二维平均池化：  torch.nn.AvgPool2d(kernel\_size, stride=None, padding=0, ceil\_mode=False, count\_include\_pad=True, divisor\_override=None)  kernel\_size：池化核尺寸  stride：步长，通常与 kernel\_size 一致  padding：填充宽度，主要是为了调整输出的特征图大小，一般把 padding 设置合适的值后，保持输入和输出的图像尺寸不变。  dilation：池化间隔大小，默认为 1。常用于图像分割任务中，主要是为了提升感受野  ceil\_mode：默认为 False，尺寸向下取整。为 True 时，尺寸向上取整  count\_include\_pad：在计算平均值时，是否把填充值考虑在内计算  divisor\_override：除法因子。在计算平均值时，分子是像素值的总和，分母默认是像素值的个数。如果设置了 divisor\_override，把分母改为 divisor\_override。 |

### MaxUnpool2d

|  |
| --- |
| 最大值反池化：  nn.MaxUnpool2d 可以理解为 MaxPool2d 的反向操作。它可以帮助我们把经过 MaxPool2d 操作后的数据还原回去，但是只能还原最大值，其他的值都会被设为零。这样可以帮助我们最大限度地还原信息。  nn.MaxUnpool2d(kernel\_size, stride=None, padding=0)  kernel\_size：池化核尺寸  stride：步长，通常与 kernel\_size 一致  padding：填充宽度 |

## 填充层

### 介绍

|  |
| --- |
| * 填充层([Padding Layers](https://pytorch.org/docs/stable/nn.html#padding-layers)).在神经网络中，填充是指在数据矩阵的边缘添加额外的值（通常为零）的过程。这种方法通常用于卷积神经网络（CNN），它们广泛应用于计算机视觉和图像处理任务。   填充的主要目的是在卷积层中保留空间维度，防止在卷积过程中丢失信息。此外，填充还可以帮助内核（特征提取器）更多次访问图像角落周围的像素，以便更好地提取重要特征。  总之，填充是一种重要的技术，在神经网络中有多种用途。它有助于保持卷积层中的空间维度，防止在卷积过程中丢失信息，并且可以改善模型的性能。 |

### ReflectionPad2d

|  |
| --- |
| torch.nn.ReflectionPad2d类使用输入边界的反射来填充输入张量。  参数padding指定了填充的大小：  如果是整数，则在所有边界中使用相同的填充。如果是4长度的元组，则使用(padding\_left, padding\_right, padding\_top, padding\_bottom)。 |

### ReplicationPad2d

|  |
| --- |
| torch.nn.ReplicationPad2d类使用输入边界的复制来填充输入张量。  参数padding指定了填充的大小：  如果是整数，则在所有边界中使用相同的填充。如果是4长度的元组，则使用(padding\_left, padding\_right, padding\_top, padding\_bottom)。 |

## 非线性激活层

### 介绍

|  |
| --- |
| 非线性激活层（[Non-linear Activations](https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity)）的作用是在神经网络的层与层之间引入非线性变换，使得模型能够拟合复杂的非线性关系。它通过对每个神经元的输出应用非线性激活函数来实现这一点。这些激活函数通常具有一些特殊的性质，例如 ReLU 函数只保留正值并将负值设为零，而 Sigmoid 函数则将输入压缩到 0 和 1 之间。这些函数的使用可以增强模型的表达能力，使其能够更好地拟合数据。 |

### ReLU

|  |
| --- |
| ReLU（Rectified Linear Unit，修正线性单元）是一种常用的非线性激活函数。它的定义非常简单，对于输入 x，它的输出为 max(0, x)。也就是说，如果输入为正数，则输出不变；如果输入为负数，则输出为零。  ReLU 函数的优点在于它的计算非常简单，而且能够有效地缓解梯度消失问题。梯度消失问题是指在训练深度神经网络时，梯度会随着层数的增加而指数级衰减，导致模型难以训练。由于 ReLU 函数在正区间的梯度为 1，因此它能够有效地缓解这个问题。 |

### Sigmoid

|  |
| --- |
| Sigmoid 是一种常用的非线性激活函数，它将输入压缩到 0 和 1 之间。它的数学定义为 f(x) = 1 / (1 + exp(-x))。由于它的输出范围为 (0,1)，因此它常用于表示概率。  Sigmoid 函数具有一些优点，例如它能够将输入压缩到固定范围内，输出具有平滑性，且导数容易计算。然而，它也存在一些问题，例如它容易受到梯度消失问题的影响。这是因为当输入绝对值较大时，Sigmoid 函数的导数接近于零，导致梯度难以传播。  尽管如此，在某些情况下 Sigmoid 函数仍然非常有用。例如，在二元分类问题中，我们可以在输出层使用 Sigmoid 函数来表示正类的概率。 |

## 归一化层

### 介绍

|  |
| --- |
| * 归一化层（[Normalization Layers](https://pytorch.org/docs/stable/nn.html#normalization-layers)）。在torch.nn模块中，有几个类可以用来对模型进行正则化。正则化是一种防止过拟合的技术，它通过在损失函数中添加额外的项来限制模型的复杂度。 |

### API

|  |
| --- |
| 常见的归一化层包括：  torch.nn.BatchNorm1d：对小批量（mini-batch）数据的每个特征进行批量归一化（Batch Normalization）。  torch.nn.BatchNorm2d：对小批量数据的每个通道进行批量归一化。  torch.nn.BatchNorm3d：对小批量数据的每个通道进行批量归一化。  torch.nn.LayerNorm：对小批量数据的最后几个维度进行层归一化（Layer Normalization）。  torch.nn.GroupNorm：将通道分成若干组，然后对每组进行层归一化。  torch.nn.InstanceNorm1d：对小批量数据的每个样本的每个特征进行实例归一化（Instance Normalization）。  torch.nn.InstanceNorm2d：对小批量数据的每个样本的每个通道进行实例归一化。  torch.nn.InstanceNorm3d：对小批量数据的每个样本的每个通道进行实例归一化。  torch.nn.LocalResponseNorm：对局部区域内的输入进行响应归一化（Local Response Normalization）。 |

## 循环层

### 介绍

|  |
| --- |
| * 循环层（[Recurrent Layers](https://pytorch.org/docs/stable/nn.html#recurrent-layers)）是一类神经网络层，它们可以用来处理序列数据。与普通的神经网络层不同，循环层可以在处理每个时间步的输入时保留之前时间步的信息。这使得它们能够捕捉序列数据中的时间依赖性。简而言之就是会记录之前的数据用于预测。   循环层在许多领域都有广泛应用，例如自然语言处理、语音识别和时间序列预测。它们可以用来构建语言模型、机器翻译系统、语音识别系统和股票预测模型等。 |

### API

|  |
| --- |
| 常见的循环层包括：  torch.nn.RNN：实现了一个简单的循环神经网络（RNN）。  torch.nn.LSTM：实现了一个长短时记忆网络（LSTM）。  torch.nn.GRU：实现了一个门控循环单元网络（GRU）。 |

## 线性变换层

### 介绍

|  |
| --- |
| * 线性层（[Linear Layers](https://pytorch.org/docs/stable/nn.html#linear-layers)），如 nn.Linear 和 nn.Bilinear，用于对输入数据应用线性变换。它们可以用于构建神经网络，实现不同类型的线性变换。   线性变换是一种简单但强大的数学工具，它能够对数据进行缩放、旋转、平移等操作。在神经网络中，线性变换可以用来对输入数据进行变换，从而提取特征并将其传递到下一层。  线性变换可以增加神经网络的表达能力，使其能够更好地拟合数据。它也可以与其他类型的层（如激活函数层）结合使用，实现非线性变换，进一步增强神经网络的表达能力。  此外，线性变换还可以用来减少数据的维度，从而降低计算复杂度和过拟合风险。例如，在自然语言处理中，常用线性变换将高维的词嵌入向量映射到低维空间，以便更好地捕捉词语之间的关系。 |

### Linear

|  |
| --- |
| torch.nn.Linear 类的构造函数接受以下参数：  in\_features（整数）：每个输入样本的大小。这个参数指定了输入数据的特征数。  out\_features（整数）：每个输出样本的大小。这个参数指定了输出数据的特征数。  bias（布尔值，可选）：如果设置为 False，则该层不会学习附加偏差。默认值为 True。  例如，如果我们想要创建一个输入样本大小为 20，输出样本大小为 30 的线性层，并且不希望学习附加偏差，我们可以这样实例化一个 Linear 类：  linear = nn.Linear(in\_features=20, out\_features=30, bias=False) |

## 损失函数层

### 介绍

|  |
| --- |
| 损失函数层torch.nn.xxx。  损失函数是衡量模型输出与真实标签之间的差异。我们还经常听到代价函数和目标函数，它们之间差异如下：    知乎笔记：[PyTorch 学习笔记（六）：PyTorch的十八个损失函数 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/61379965) |

### L1loss

|  |
| --- |
| 功能： 计算output和target之差的绝对值，可选返回同维度的tensor或者是一个标量。    参数：  size\_average：用于指定损失是否应除以每个批次中的元素数。如果设置为 True（默认值），则损失将除以每个批次中的元素数；如果设置为 False，则损失将不会除以每个批次中的元素数。  reduction：指定要应用于输出的缩减方式：'none' | 'mean' | 'sum'。'none'：不应用缩减，'mean'：输出的总和将除以输出中的元素数，'sum'：输出将求和。默认值为 'mean'。 |

### MSELoss

|  |
| --- |
| 功能： 计算output和target之差的平方，可选返回同维度的tensor或者是一个标量。    reduction：指定要应用于输出的缩减方式：'none' | 'mean' | 'sum'。'none'：不应用缩减，'mean'：输出的总和将除以输出中的元素数，'sum'：输出将求和。默认值为 'mean'。 |

### CrossEntropyLoss

|  |
| --- |
| nn.CrossEntropyLoss(weight=None,size\_average=None,ignore\_index=-100, reduce=None, reduction='mean')  功能：把nn.LogSoftmax()和nn.NLLLoss()结合，计算交叉熵。nn.LogSoftmax()的作用是把输出值归一化到了 [0,1] 之间。  主要参数：  weight：各类别的 loss 设置权值，如果给定，则必须是大小为C(类别数)的张量。  ignore\_index：忽略某个类别的 loss 计算  reduction：计算模式，可以为 none(逐个元素计算)，sum(所有元素求和，返回标量)，mean(加权平均，返回标量)  label\_smoothing(浮点数, 可选): 在[0.0, 1.0]范围内的浮点数. 指定计算损失时平滑量, 其中0.0表示无平滑. 目标变成原始真实标签和均匀分布的混合体, 如Rethinking the Inception Architecture for Computer Vision所述. 默认: 0.0.  公式： |

### 举例

|  |
| --- |
| **import** torchvision **from** torch **import** nn **from** torch.nn **import** Conv2d, MaxPool2d, Flatten, Linear, Sequential **from** torch.utils.data **import** Subset, DataLoader **from** torch.utils.tensorboard **import** SummaryWriter   **class** Demo(nn.Module):  **def** \_\_init\_\_(self):  super(Demo, self).\_\_init\_\_()  self.module1 = Sequential(  Conv2d(3, 32, 5, padding=2),  MaxPool2d(2),  Conv2d(32, 32, 5, padding=2),  MaxPool2d(2),  Conv2d(32, 64, 5, padding=2),  MaxPool2d(2),  Flatten(),  Linear(1024, 64),  Linear(64, 10),  )   **def** forward(self, x):  **return** self.module1(x)   test\_data = torchvision.datasets.CIFAR10(**'../../dataset/36.datasets'**, train=**False**,  transform=torchvision.transforms.ToTensor(),  download=**True**) test\_data = Subset(test\_data, range(1000))  test\_loader = DataLoader(dataset=test\_data, batch\_size=64, shuffle=**False**, num\_workers=0, drop\_last=**True**)   loss= nn.CrossEntropyLoss() nn.MSELoss() demo = Demo()  **for** data **in** test\_loader:  imgs, targets = data  output = demo(imgs)  res\_loss = loss(output, targets)  res\_loss.backward()  print(res\_loss) |

## 优化器

### 介绍

|  |
| --- |
| PyTorch 中的优化器是用于管理并更新模型中可学习参数的值，使得模型输出更加接近真实标签。即各种机器学习算法。  PyTroch 提供的 10 种优化器：  optim.SGD  optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weight\_decay=0, nesterov=False  随机梯度下降法  主要参数：  params：管理的参数组  lr：初始学习率  momentum：动量系数  weight\_decay：L2 正则化系数  nesterov：是否采用 NAG  optim.Adagrad  自适应学习率梯度下降法  optim.RMSprop  Adagrad 的改进  optim.Adadelta  optim.Adam  RMSProp 集合 Momentum，这个是目前最常用的优化器，因为它可以使用较大的初始学习率。  optim.Adamax  Adam 增加学习率上限  optim.SparseAdam  稀疏版的 Adam  optim.ASGD  随机平均梯度下降  optim.Rprop  弹性反向传播，这种优化器通常是在所有样本都一起训练，也就是 batchsize 为全部样本时使用。  optim.LBFGS  BFGS 在内存上的改进 |

### SGD

|  |
| --- |
| optim.SGD 是 PyTorch 中实现随机梯度下降（SGD）的类。它的参数包括：  params：要优化的参数或定义参数组的字典的可迭代对象。  lr：学习率，通常可以从较小的值（例如 1e-3 或 1e-2）开始尝试，然后通过实验来确定最佳值。如果您发现模型收敛速度过慢，可以尝试增大学习率；如果发现模型不稳定或无法收敛，可以尝试减小学习率。。  momentum：动量因子（默认值：0）。较大的 momentum 值可以加快收敛速度，但也可能导致过拟合。较小的 momentum 值可以减少过拟合风险，但可能会减慢收敛速度。  常用的 momentum 值为 0.5、0.9 和 0.99。您可以从这些值中选择一个作为起点，然后通过实验来确定最佳值。  weight\_decay：权重衰减（L2惩罚）（默认值：0）。通常，可以从较小的值（例如 1e-4 或 1e-3）开始尝试，然后通过交叉验证来确定最佳值。如果您发现模型过拟合，可以尝试增大 weight\_decay 值；如果您发现模型欠拟合，可以尝试减小 weight\_decay 值。  dampening：动量阻尼（默认值：0）。  nesterov：启用 Nesterov 动量（默认值：False）。  maximize：如果设置为 True，则优化器将尝试最大化目标函数，而不是最小化它。  foreach：如果设置为 True，则优化器将使用 foreach 实现，这可以在某些情况下提高性能。  differentiable：如果设置为 True，则优化器将使用可微分的版本，这对于某些高阶优化算法可能很有用 |

### 方法

|  |
| --- |
| zero\_grad()：清空所管理参数的梯度。由于 PyTorch 的特性是张量的梯度不自动清零，因此每次反向传播之后都需要清空梯度。  step()：执行一步梯度更新  add\_param\_group()：添加参数组，主要代码如下：  state\_dict()：获取优化器当前状态信息字典  load\_state\_dict()：加载状态信息字典，包括 state 、momentum\_buffer 和 param\_groups。主要用于模型的断点续训练。我们可以在每隔 50 个 epoch 就保存模型的 state\_dict 到硬盘，在意外终止训练时，可以继续加载上次保存的状态，继续训练。 |

### 举例

|  |
| --- |
| **import** torch.optim **import** torchvision **from** torch **import** nn **from** torch.nn **import** Conv2d, MaxPool2d, Flatten, Linear, Sequential **from** torch.utils.data **import** Subset, DataLoader **from** torch.utils.tensorboard **import** SummaryWriter   **class** Demo(nn.Module):  **def** \_\_init\_\_(self):  super(Demo, self).\_\_init\_\_()  self.module1 = Sequential(  Conv2d(3, 32, 5, padding=2),  MaxPool2d(2),  Conv2d(32, 32, 5, padding=2),  MaxPool2d(2),  Conv2d(32, 64, 5, padding=2),  MaxPool2d(2),  Flatten(),  Linear(1024, 64),  Linear(64, 10),  )   **def** forward(self, x):  **return** self.module1(x)   test\_data = torchvision.datasets.CIFAR10(**'../../dataset/36.datasets'**, train=**False**,  transform=torchvision.transforms.ToTensor(),  download=**True**) test\_data = Subset(test\_data, range(1000))  test\_loader = DataLoader(dataset=test\_data, batch\_size=64, shuffle=**False**, num\_workers=0, drop\_last=**True**)   loss= nn.CrossEntropyLoss() demo = Demo() optim = torch.optim.SGD(demo.parameters(), lr=0.01) **for** epoch **in** range(20):  epoch\_loss = 0  **for** data **in** test\_loader:  imgs, targets = data  output = demo(imgs)  res\_loss = loss(output, targets)  optim.zero\_grad()  res\_loss.backward()  optim.step()  epoch\_loss += res\_loss  print(epoch\_loss) |

## 模型训练代码总流程

|  |
| --- |
| 1. 定义模型class 2. 定义全局变量 — 模型对象，训练批次，损失函数、优化器画板等 3. 获取数据集 — 训练集和测试集，加载数据集 4. 开始训练 — 循环批次，循环数据集，卷积层处理特征，计算损失，反向传播 5. 测试集测试 — 循环测试集，预测，计算损失，正确率   *#!/usr/bin/env python # -\*- coding: utf-8 -\*- # @Time : 2023/7/13 22:55 # @Author : 刘双喜 # @File : 49.CIFAR10\_Train.py # @Description : 添加描述* **import** time **import** torch.optim **import** torchvision **from** torch **import** nn **from** torch.utils.data **import** DataLoader **from** torch.utils.tensorboard **import** SummaryWriter **from** torch.nn **import** Conv2d, MaxPool2d, Flatten, Linear, Sequential   *# 模型* **class** Demo(nn.Module):  **def** \_\_init\_\_(self):  super(Demo, self).\_\_init\_\_()  self.module1 = Sequential(  Conv2d(3, 32, 5, padding=2),  MaxPool2d(2),  Conv2d(32, 32, 5, padding=2),  MaxPool2d(2),  Conv2d(32, 64, 5, padding=2),  MaxPool2d(2),  Flatten(),  Linear(64\*4\*4, 64),  Linear(64, 10),  )   **def** forward(self, x):  **return** self.module1(x)  *# =======================================================全局变量=====================================================* cuda\_avaiable = torch.cuda.is\_available() *# GPU是否可用* demo = Demo().cuda() **if** cuda\_avaiable **else** Demo() *# 创建模型* loss\_fn = nn.CrossEntropyLoss().cuda() **if** cuda\_avaiable **else** nn.CrossEntropyLoss() *# 损失函数 # 优化器* learn\_rate = 1e-2 optim = torch.optim.SGD(demo.parameters(), lr=learn\_rate) *# 训练次数* epochs = 2 total\_train\_step = 0 *# 画板 tensorboard --logdir='.\\example\\data\\40.nn\_Conv2d' --port=6012* write = SummaryWriter(**'..\\data\\49.CIFAR10\_Train'**)  *# ======================================================获取数据集合===================================================* train\_data = torchvision.datasets.CIFAR10(**'../../dataset/36.datasets'**, train=**True**,  transform=torchvision.transforms.ToTensor(),  download=**True**) test\_data = torchvision.datasets.CIFAR10(**'../../dataset/36.datasets'**, train=**False**,  transform=torchvision.transforms.ToTensor(),  download=**True**)  train\_loader = DataLoader(dataset=train\_data, batch\_size=64, shuffle=**False**) test\_loader = DataLoader(dataset=test\_data, batch\_size=64, shuffle=**False**) train\_data\_len = len(train\_data) test\_data\_len = len(test\_data)  *# =====================================================开始训练====================================================* **for** epoch **in** range(1, 1 + epochs):  print(**f'第{**epoch**}轮训练'**.center(50, **'-'**))  start\_time = time.time()  epochs\_train\_loss = 0 *# 批次训练集总损失* **for** data **in** train\_loader:  imgs, labels = data  **if** cuda\_avaiable:  imgs = imgs.cuda()  labels = labels.cuda()  *# 卷积层 - 处理特征* outputs = demo(imgs)  *# 损失函数层 - 计算损失* loss = loss\_fn(outputs, labels)  *# 优化器 - 算法优化参数* optim.zero\_grad()  loss.backward()  optim.step()  *# 累加损失* epochs\_train\_loss += loss.item()  total\_train\_step += 1  **if** total\_train\_step % 100 == 0:  write.add\_scalar(**'train\_loss'**, loss.item(), total\_train\_step)  print(**f'第{**epoch**}批次训练，损失：{**loss.item()**},耗时{**time.time()-start\_time**},进度{**round(epoch/epochs, 4) \* 100**}%'**.center(  50, **'-'**))   *# 测试集测试* start\_time = time.time()  total\_test\_loss = 0 *# 批次测试集总损失* right\_num = 0 *# 正确个数* **with** torch.no\_grad():  **for** data **in** test\_loader:  imgs, labels = data  **if** cuda\_avaiable:  imgs = imgs.cuda()  labels = labels.cuda()  outputs = demo(imgs)  *# 计算测试集损失* loss = loss\_fn(outputs, labels)  total\_test\_loss += loss.item()  *# 测试集预测值* predict = outputs.argmax(dim=1)  *# 预测正确的个数 累加* right\_num += (predict == labels).sum()  right\_rate = right\_num / test\_data\_len  print(**f'第{**epoch**}批次测试，损失{**total\_test\_loss**}, 正确率{**right\_rate**},耗时{**time.time()-start\_time**}'**.center(50, **'-'**))  write.add\_scalar(**'test\_loss'**, total\_test\_loss, epochs)   torch.save(demo.state\_dict(), **f'..\\data\\49.CIFAR10\_Train\\module\_epoch{**epoch**}'**) write.close() |

## 其他

### train()和eval()

#### model.train()

model.train()的作用是启用 Batch Normalization 和 Dropout。

如果模型中有BN层(Batch Normalization）和Dropout，需要在训练时添加model.train()。model.train()是保证BN层能够用到每一批数据的均值和方差。对于Dropout，model.train()是随机取一部分网络连接来训练更新参数。

#### model.eval()

model.eval()的作用是不启用 Batch Normalization 和 Dropout。

如果模型中有BN层(Batch Normalization）和Dropout，在测试时添加model.eval()。model.eval()是保证BN层能够用全部训练数据的均值和方差，即测试过程中要保证BN层的均值和方差不变。对于Dropout，model.eval()是利用到了所有网络连接，即不进行随机舍弃神经元。

训练完train样本后，生成的模型model要用来测试样本。在model(test)之前，需要加上model.eval()，否则的话，有输入数据，即使不训练，它也会改变权值。这是model中含有BN层和Dropout所带来的的性质。

在做one classification的时候，训练集和测试集的样本分布是不一样的，尤其需要注意这一点。

#### 分析原因

使用PyTorch进行训练和测试时一定注意要把实例化的model指定train/eval。model.eval()时，框架会自动把BN和Dropout固定住，不会取平均，而是用训练好的值，不然的话，一旦test的batch\_size过小，很容易就会被BN层导致生成图片颜色失真极大

### GPU训练

#### 方式一

|  |
| --- |
| cuda\_avaiable = torch.cuda.is\_available()  demo = Demo().cuda() **if** cuda\_avaiable **else** Demo() *# 创建模型* |

#### 方式二

|  |
| --- |
| cuda\_avaiable = torch.cuda.is\_available() *# GPU是否可用* device = torch.device(**'cuda' if** cuda\_avaiable **else 'cpu'**) demo = Demo() demo.to(device) *# 创建模型* |

### 测试

|  |
| --- |
| 在没有梯度下测试  **with** torch.no\_grad(): *# 测试的时候进入没有梯度计算模式，节省内存消耗并加快计算速度* **for** data **in** test\_loader:  imgs, labels = data  imgs = imgs.to(device)  labels = labels.to(device)  outputs = demo(imgs)  *# 计算测试集损失* loss = loss\_fn(outputs, labels)  total\_test\_loss += loss.item()  *# 测试集预测值* predict = outputs.argmax(dim=1)  *# 预测正确的个数 累加* right\_num += (predict == labels).sum() |

|  |
| --- |
| 完整的测试流程：  **import** time **import** torch.optim **from** torch **import** nn **from** torch.utils.data **import** DataLoader **from** torch.utils.tensorboard **import** SummaryWriter **from** torch.nn **import** Conv2d, MaxPool2d, Flatten, Linear, Sequential **import** torchvision **from** PIL **import** Image **import** numpy **as** np   img\_path = **'..\\data\\50.model\\person1.png'** img = Image.open(img\_path) img = img.convert(**"RGB"**) transform = torchvision.transforms.Compose([torchvision.transforms.Resize((32,32)),  torchvision.transforms.ToTensor()]) img = transform(img)  **class** Demo(nn.Module):  **def** \_\_init\_\_(self):  super(Demo, self).\_\_init\_\_()  self.module1 = Sequential(  Conv2d(3, 32, 5, padding=2),  MaxPool2d(2),  Conv2d(32, 32, 5, padding=2),  MaxPool2d(2),  Conv2d(32, 64, 5, padding=2),  MaxPool2d(2),  Flatten(),  Linear(64\*4\*4, 64),  Linear(64, 10),  )   **def** forward(self, x):  **return** self.module1(x)   demo = Demo() demo.load\_state\_dict(torch.load(**'..\\data\\50.model\\module\_epoch30.pth'**, map\_location=torch.device(**'cpu'**))) demo.eval()  img = torch.reshape(img, (1, 3, 32, 32)) class\_to\_idx = {**'airplane'**: 0, **'automobile'**: 1, **'bird'**: 2, **'cat'**: 3, **'deer'**: 4, **'dog'**: 5, **'frog'**: 6, **'horse'**: 7, **'ship'**: 8, **'truck'**: 9} idx\_to\_class = {v: k **for** k, v **in** class\_to\_idx.items()} **with** torch.no\_grad():  output = demo(img)  predict\_label\_idx = output.argmax(1)  print(idx\_to\_class.get(predict\_label\_idx.item())) |

### 参数调整优化

|  |
| --- |
| * 调整卷积核大小：增加卷积核大小可以增加感受野，有助于捕捉更大范围内的上下文信息。但是，过大的卷积核可能会导致计算量增加和过拟合。 * 调整步幅和填充：增加步幅可以减小输出大小，有助于减少计算量。增加填充可以保持输出大小不变，有助于保留边缘信息。 * 增加滤波器数量：增加滤波器数量可以增加模型的表达能力，有助于提高模型的性能。但是，过多的滤波器可能会导致计算量增加和过拟合。 * 添加批量归一化层：在卷积层后添加批量归一化层可以加速模型收敛，有助于提高模型的性能。 * 调整激活函数：尝试使用不同的激活函数（如ReLU、LeakyReLU、ELU等），以找到最适合当前任务的激活函数。 |

# 现有模型

## torchvision

### 介绍

|  |
| --- |
| torchvision.models 包含了许多用于解决不同任务的预训练模型，包括图像分类、像素级语义分割、目标检测、实例分割、人体关键点检测、视频分类和光流等。  在 torchvision.models 中，有一些模型在计算机视觉领域非常常用。这些模型包括：  AlexNet：这是一个经典的卷积神经网络，它在 2012 年的 ImageNet 挑战赛中取得了巨大的成功。  VGG：这是一个深度卷积神经网络，它在 2014 年的 ImageNet 挑战赛中取得了优异的成绩。  ResNet：这是一个深度残差网络，它通过引入残差连接来解决深度网络中的梯度消失问题。  Inception v3：这是一个深度卷积神经网络，它通过使用多尺度卷积来提高性能。  用model\_names = torchvision.models.list\_models()可以查看有哪些模型 |

### 公用方法

|  |
| --- |
| **import** torchvision **from** torch **import** nn   vgg16 = torchvision.models.vgg16() vgg16.classifier[6] = nn.Linear(4096, 100) *# 修改在classifier层的第6个模块* print(vgg16) vgg16.classifier.add\_module(**'7'**, nn.Linear(100, 10)) *# 在classifier层增加一个模块，将线性回归1000个类别变成10个* print(vgg16\_model) *# 打印模型的结构* print(vgg16\_model.state\_dict()) *# 打印模型的权重和偏置* print(vgg16\_model.state\_dict()[**'classifier.0.weight'**]) *# 打印模型特定层的权重偏置* |

### vgg16

|  |
| --- |
| torchvision.models.vgg16 是一个预训练的卷积神经网络模型，它用于图像识别。它包含三个部分：features，avgpool 和 classifier。  features: 一个卷积层序列，用于提取图像的特征。  avgpool: 一个自适应平均池化层，用于减少特征图的空间大小。  classifier: 一个全连接层序列，用于对图像进行分类。  torchvision.models.vgg16()  weights (可选的 VGG16\_Weights)：预训练的权重。默认情况下，不使用预训练的权重。  None：不使用预训练权重。  'imagenet' 或 VGG16\_Weights.IMAGENET1K\_V1：使用在 ImageNet 数据集上预训练的权重。  'imagenet\_features' 或 VGG16\_Weights.IMAGENET1K\_FEATURES：使用在 ImageNet 数据集上预训练的权重，但仅用于特征提取，不能用于分类。这时候classifier 模块中的权重将不会被加载。  progress (可选的 bool)：如果为 True，则在下载过程中在标准错误上显示进度条。默认为 True。  \*\*kwargs：传递给 torchvision.models.vgg.VGG 基类的参数。请参考源代码了解有关此类的更多详细信息。  例：  vgg16\_model = vgg16(weights=VGG16\_Weights.IMAGENET1K\_V1) |

# CycleGAN

## 原理

CycleGAN是一种生成对抗网络，它能够在两个不同的图像域之间进行图像转换，而不需要成对的训练数据。CycleGAN本质上是两个镜像对称的GAN，构成了一个环形网络。它由两个生成器和两个判别器组成，其中一个生成器将一个域中的图像转换为另一个域中的图像，另一个生成器则执行相反的操作。判别器用于判断生成的图像是否与目标域中的真实图像相似。CycleGAN通过最小化对抗损失和循环一致性损失来训练模型。

## 图解

|  |
| --- |
|  |