# 环境安装

## Anaconda

### 下载

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

### 命令行

|  |
| --- |
| 创建环境：conda create -n xxx python=3.10  查看：pip list/conda list  进入环境：conda activate xx  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() |

## tensorboard

### 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 |

## 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 |

## 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 |

## 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)) |

# 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

## 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。 * 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）。 |

### 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  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。  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 |

## 激活层

### 介绍

|  |
| --- |
| 非线性激活层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。 |