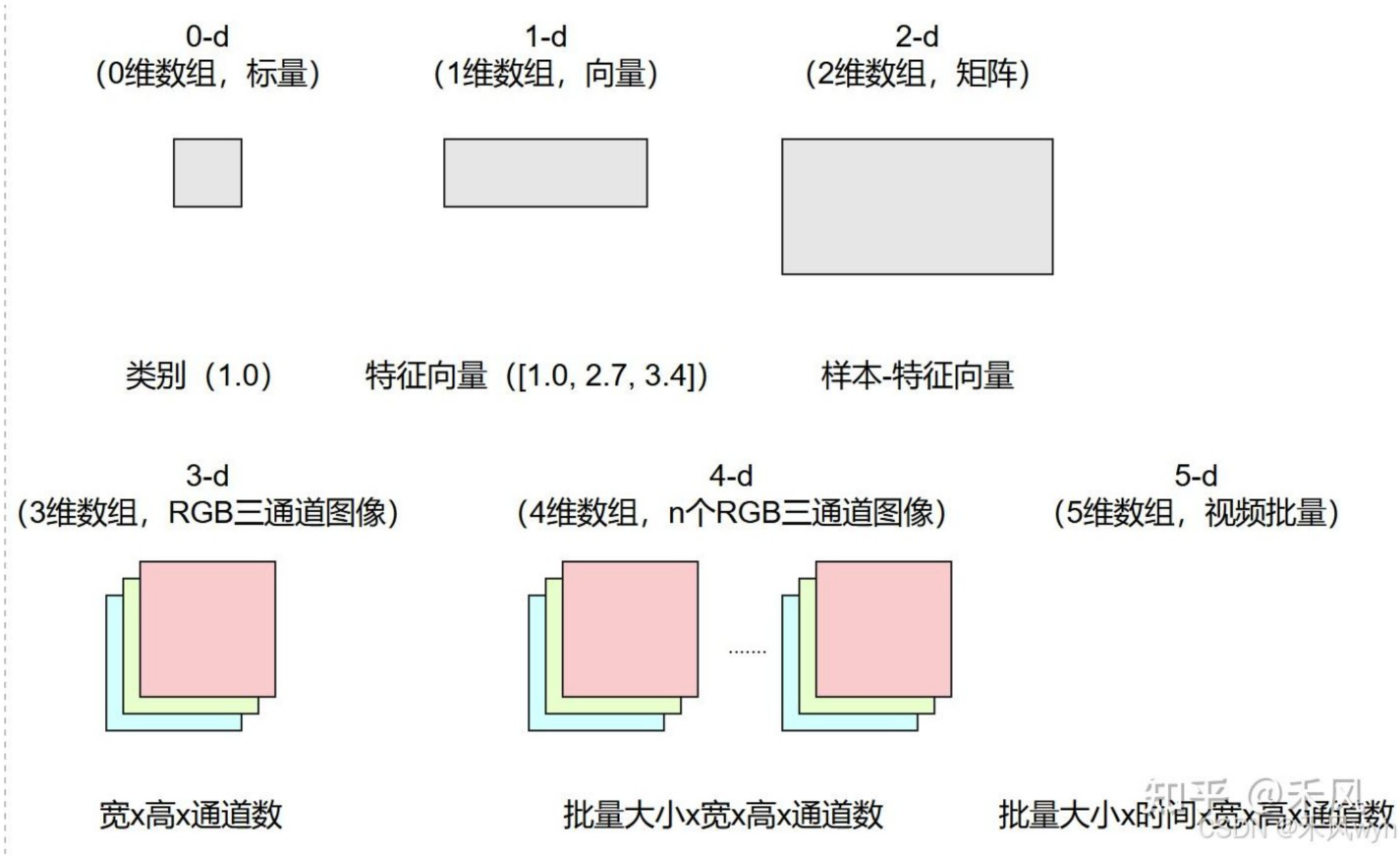
由于数据会存在某些问题，如数据重复（相同的数据）、数据不完整、数据格式不一致、数据冗余等，我们需要进行数据预处理操作。当原始数据往往存在异常值与缺失值等问题时，需要进行数据清洗；当机器学习算法无法直接处理原始格式的数据时，需要进行数据转换；当数据量或数据维度过大、存在冗余，模型无法有效学习，也会影响推理速度，需要进行数据压缩。

在了解各种数据预处理方法之前，我们先来了解一下如何进行数据操作。

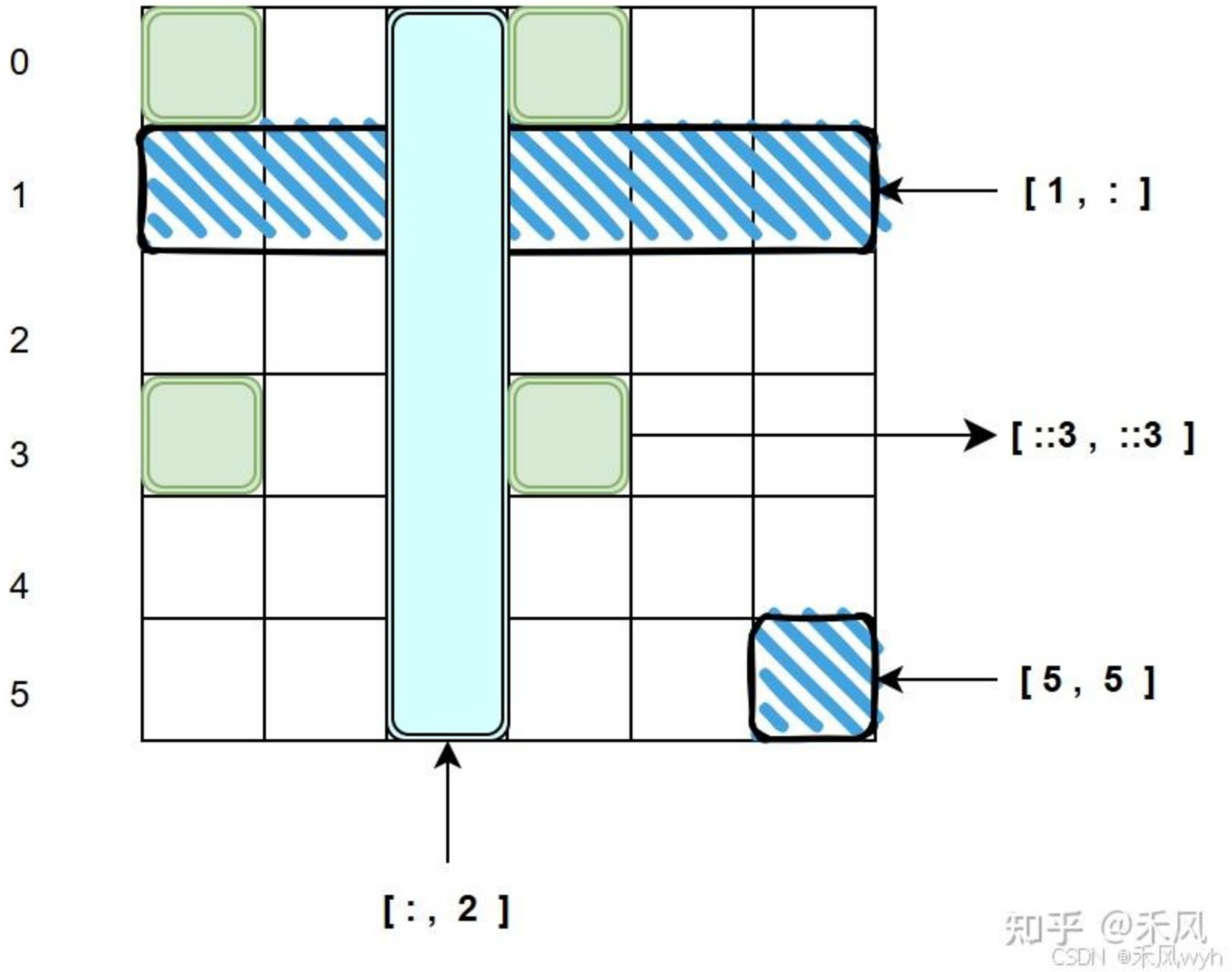
## 数据操作

在机器学习算法中，我们通常采用张量来进行数据的表示，张量指的是数值组成的数组，存在多个维度。在二分类任务中，我们的图像标签就是0维数组（0或1）；而在多分类任务中，我们的图像标签会变成1维数组（特征向量的每个位置会对应一个类别概率）；处理一张图片的效率太低了，通常情况下，我们会选择进行一个批次进行处理（一个批次包含多张图片），这时候便会增加“批次大小”这个维度形成2维数组。




了解张量的概念后，我们需要知道如何进行创建张量、访问张量内的元素以及如何进行张量的基本操作。创建张量可以已知形状、数据类型或者数据内容，访问张量内的元素可以进行读取或者是更改。

0 1 2 3 4 5




**生成张量**

```
# 生成1维张量，大小为12（元素从0-11进行排列）
x = torch.arange(12)
# 生成指定尺寸的全0张量
x = torch.zeros((2, 3, 4))
# 生成指定尺寸的全1张量
x = torch.ones((2, 3, 4))
# 生成指定元素的张量
x = torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

**访问张量元素**

```
x = torch.arange(12)
# 访问张量的形状
x.shape
# 访问张量中元素的总数
x.numel()
# 修改尺寸
x.reshape(4,3)

X[1, 2] = 9 #赋值
X[0:2, :] = 12
X[-1]
X[1:3]
```

**对张量进行操作**

```
# 算术运算（+ - * /）,对张量内所有元素求和
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x.sum()

# 将张量连接在一起
X = torch.arange(12, dtype = torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3],[1, 2, 3, 4],[4, 3, 2, 1]])
Z = torch.cat((X, Y), dim = 0)
Z = torch.cat((X, Y), dim = 1)

# 广播机制对张量进行操作
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a + b

# 转换为Numpy张量，将大小为1的张量转换为Python标量
A = X.numpy()
B = torch.tensor(A)
a = torch.tensor([3.5])
type(A), type(B),a, a.item(), float(a), int(a)
```

**对张量进行运算**

```
# 创建一个形状为mxn的矩阵，矩阵的转置，对称矩阵
A = torch.arange(20).reshape(5,4)
B = torch.tensor([[1, 2, 3],[2, 0, 4],[3, 4, 5]]) #对称矩阵
A, A.T, B, B == B.T

# 向量是标量的推广，矩阵是向量的推广
X = torch.arange(24).reshape(2, 3, 4)
# 给定具有相同形状的任何两个张量，任何按元素二元计算的结果都将是相同形状的张量
A = torch.arange(20, dtype = torch.float32).reshape(5, 4)
B = A.clone()
X, A, A + B, A * B # 两个矩阵按元素乘法称为哈达玛积

# 张量与标量计算(广播机制)
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, a * X, (a * X).shape

#计算元素的和
x = torch.arange(4, dtype=torch.float32)
# 表示任意形状张量的元素和
A = torch.arange(20*2).reshape(2, 5, 4)
#按批次求和
A_sum_axis0 = A.sum(axis=0)
#按列求和
A_sum_axis1 = A.sum(axis=1)
 #按批次求和,然后按行求和
A_sum_axis01 = A.sum(axis=[0, 1])
x, x.sum(),A,A.shape, A.sum(), A_sum_axis0, A_sum_axis0.shape,A_sum_axis1, A_su

# 与求和相关的量,平均值
A = torch.arange(20, dtype = torch.float32).reshape(5, 4)
#将A进行按行求和,然后求平均
A, A.mean(), A.sum() / A.numel(),A.mean(axis=0), A.sum(axis=0) / A.shape[0]

# 计算总和或均值时保持轴数不表
sum_A = A.sum(axis=1, keepdims=True)
sum_A_ = A.sum(axis=1)
A, sum_A, sum_A_,A / sum_A #通过广播机制进行求均值

# 某个轴计算A元素的累计总和
A, A.cumsum(axis=0)

# 矩阵的点积、向量积运算、矩阵乘法
y = torch.ones(4, dtype = torch.float32) # 点积是相同位置的按元素乘积的和
x, y, torch.dot(x, y),torch.sum(x * y) # 通过执行按元素乘法,然后进行求和来表示两个向量
```

```
A.shape, x.shape, torch.mv(A,x) # 矩阵向量积Ax是一个长度为m的列向量,第i个元素是点积aix

B = torch.ones(4, 3)# 矩阵矩阵乘法
torch.mm(A, B)

# L2范数(元素平方求和然后求根号),L1范数(向量元素的绝对值之和),F范数(矩阵元素的平方和的平方根
u  = torch.tensor([3.0, -4.0]
torch.norm(u),torch.abs(u).sum(),torch.norm(torch.ones(4,9))
```

## 数据预处理

下面的代码定义了两个用于加载 CIFAR-100 数据集的 PyTorch 数据集类 CIFAR100Train 和 CIFAR100Test，它们分别用于训练数据和测试数据的加载和预处理。其中，需要使用到Dataset类，它是一个存储图像数据、图像索引、图像标签的大容器。

首先，需要进行__init__()方法的定义。__init__()方法主要负责初始化数据集对象，使用pickle读取指定路径中的train文件，加载训练数据，将数据保存在self.data中。self.transform 用于保存传入的变换方法（数据增强或预处理），如果存在则在 __getitem__ 方法中应用。

```
    def __init__(self, path, transform=None):
        #if transform is given, we transoform data using
        # 使用 os.path.join 拼接路径以提高代码兼容性
        with open(os.path.join(path, 'train'), 'rb') as cifar100:
            # 读取文件时指定 encoding='bytes'，确保兼容 CIFAR-100 的二进制编码格式
            self.data = pickle.load(cifar100, encoding='bytes')
        # 提供 transform 参数，便于用户灵活地定义数据增强操作
        self.transform = transform
```

然后，进行__len__()方法的实现，返回数据集的样本数。'fine_labels'是该字典中的一个键，表示包含每个图像标签（fine-grained label）的数据。'fine_labels'.encode()会将该字符串转换为字节字符串(b'fine_labels')，因为在加载 CIFAR-100 数据时，它是以字节编码格式存储的。所以，self.data['fine_labels'.encode()]获取到的是数据集中每个图像的标签，即一个数组，表示该图像的分类标签。

```
    def __len__(self):
        return len(self.data['fine_labels'.encode()])
```

最后，进行__getitem__()方法实现。__getitem()__方法获取指定索引index的数据样本，包括标签和图像；CIFAR-100 图像存储在data中，每张图像的像素值按通道（R、G、B）顺序存储，fine_labels提供标签。图像通道按 1024 个像素分隔：R，G， B分别提取红、绿、蓝通道的像素数据并重塑为 32x32 的二维数组。numpy.dstack() 将 3 个二维数组组合为 32x32x3 的 RGB 图像。如果定义了transform，则对图像进行变换。我们的数据增强或预处理操作可以写在tranform中。

```
    def __getitem__(self, index):
        # 处理索引获取的单个样本，提供图像和对应标签
        label = self.data['fine_labels'.encode()][index]
        r = self.data['data'.encode()][index, :1024].reshape(32, 32)
        g = self.data['data'.encode()][index, 1024:2048].reshape(32, 32)
        b = self.data['data'.encode()][index, 2048:].reshape(32, 32)
        # 使用分块方式解码像素数据，重构为 RGB 图像
        image = numpy.dstack((r, g, b))

        if self.transform:
            image = self.transform(image)
        return label, image
```

下面是整合后的完整代码：

```
import os
import sys
import pickle

from skimage import io
import matplotlib.pyplot as plt
import numpy
import torch
from torch.utils.data import Dataset

class CIFAR100Train(Dataset):
    def __init__(self, path, transform=None):
        #if transform is given, we transoform data using
        with open(os.path.join(path, 'train'), 'rb') as cifar100:
            self.data = pickle.load(cifar100, encoding='bytes')
        self.transform = transform

    def __len__(self):
        return len(self.data['fine_labels'.encode()])

    def __getitem__(self, index):
        label = self.data['fine_labels'.encode()][index]
        r = self.data['data'.encode()][index, :1024].reshape(32, 32)
        g = self.data['data'.encode()][index, 1024:2048].reshape(32, 32)
        b = self.data['data'.encode()][index, 2048:].reshape(32, 32)
        image = numpy.dstack((r, g, b))

        if self.transform:
            image = self.transform(image)
        return label, image
```

```
class CIFAR100Test(Dataset):

    def __init__(self, path, transform=None):
        with open(os.path.join(path, 'test'), 'rb') as cifar100:
            self.data = pickle.load(cifar100, encoding='bytes')
        self.transform = transform

    def __len__(self):
        return len(self.data['data'.encode()])

    def __getitem__(self, index):
        label = self.data['fine_labels'.encode()][index]
        r = self.data['data'.encode()][index, :1024].reshape(32, 32)
        g = self.data['data'.encode()][index, 1024:2048].reshape(32, 32)
        b = self.data['data'.encode()][index, 2048:].reshape(32, 32)
        image = numpy.dstack((r, g, b))

        if self.transform:
            image = self.transform(image)
        return label, image
```

利用DataLoader进行数据集调用操作：

```
def get_training_dataloader(mean, std, batch_size=16, num_workers=2, shuffle=Tr
    # 数据增强和预处理操作
    transform_train = transforms.Compose([
        #transforms.ToPILImage(),
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(15),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ])

    cifar100_training = torchvision.datasets.CIFAR100(root='./data', train=True
    cifar100_training_loader = DataLoader(
        cifar100_training, shuffle=shuffle, num_workers=num_workers, batch_size

    return cifar100_training_loader

def get_test_dataloader(mean, std, batch_size=16, num_workers=2, shuffle=True):
    # 数据预处理操作
    transform_test = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ])
```

```
    cifar100_test = torchvision.datasets.CIFAR100(root='./data', train=False, d
    cifar100_test_loader = DataLoader(
        cifar100_test, shuffle=shuffle, num_workers=num_workers, batch_size=bat

    return cifar100_test_loader
```