**Deep leaning cheating sheet**

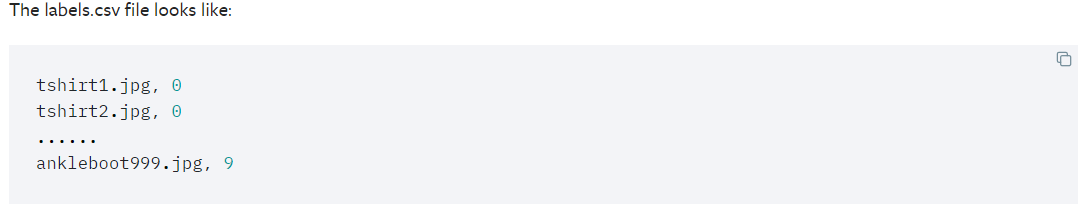
1. **Basic operation**

**os.environ["CUDA\_DEVICE\_ORDER"] = "PCI\_BUS\_ID" ##按照PCI\_BUS\_ID顺序从0开始排列GPU设备**

**os.environ["CUDA\_VISIBLE\_DEVICES"] = "0" ##【0表示由nvidia-smi查得可用的gpu序号】**

1. **Super resolution network （something about pytorch）**

* **CSV FILE: comma separated values 逗号分割值文件，没有统一格式**



* **Torchvision.transforms.CenterCrop(size)**

Crops the given image at the center. If the image is torch Tensor, it is expected to have […, H, W] shape, where … means an arbitrary number of leading dimensions. If image size is smaller than output size along any edge, image is padded with 0 and then center cropped.

* **Torchvision.transforms.totensor()**

把一个取值范围是[0,255]的PIL.Image或者shape为(H,W,C)的numpy.ndarray，转换成形状为[C,H,W]，取值范围是[0,1.0]的torch.FloadTensor

<https://pytorch.org/docs/stable/data.html>

* **TORCH.UTILS.DATA.DataLoader**

(dataset, **batch\_size**=**1, shuffle**=False**, sampler**=None**,batch\_sampler**=None**, num\_workers**=**0,collate\_fn**=None**,pin\_memory**=False**,drop\_last**=False**, timeout**=**0, worker\_init\_fn**=None**,** \***, prefetch\_factor**=**2, persistent\_workers**=False**)**

NOTE：在windows系统下num\_works不支持多进程，只能为0

* **torch.utils.data.dataset**

为一个抽象类，抽象类的概念，如果说类是从一堆对象中抽取相同的内容而来的，那么抽象类就是从一堆类中抽取相同的内容而来的，内容包括数据属性和函数属性。

比如我们有香蕉的类，有苹果的类，有桃子的类，从这些类抽取相同的内容就是水果这个抽象的类，你吃水果时，要么是吃一个具体的香蕉，要么是吃一个具体的桃子。。。。。。你永远无法吃到一个叫做水果的东西。

从设计角度去看，如果类是从现实对象抽象而来的，那么抽象类就是基于类抽象而来的。

从实现角度来看，抽象类与普通类的不同之处在于：抽象类中只能有抽象方法(没有实现功能)，该类不能被实例化，只能被继承，且子类必须实现抽象方法。

Dataloader与dataset相关原文链接

<https://blog.csdn.net/qq_38607066/article/details/98474121>

**一、基本概念**

1. **torch.utils.data.dataset**这样的抽象类可以用来创建数据集。学过面向对象的应该清楚，**抽象类不能实例化**，因此我们需要构造这个抽象类的**子类**来创建数据集，并且我们还可以定义自己的继承和重写方法。
2. 这其中最重要的就是\*\***len和getitem这两个函数，前者给出**数据集的大小\*\*，后者是用于查找**数据和标签**。
3. **torch.utils.data.DataLoader**是一个迭代器，方便我们去**多线程**地读取数据，并且可以实现**batch**以及**shuffle**的读取等。

**二、Dataset的创建和使用**

1. 首先我们需要引入dataset这个抽象类，当然我们还需要引入Numpy：

import torch.utils.data.dataset as Dataset

import numpy as np

1. 我们创建Dataset的一个子类：  
   （1）初始化，定义数据内容和标签：

#初始化，定义数据内容和标签

def \_\_init\_\_(self, Data, Label):

self.Data = Data

self.Label = Label

（2）返回数据集大小：

#返回数据集大小

def \_\_len\_\_(self):

return len(self.Data)

（3）得到数据内容和标签：

#得到数据内容和标签

def \_\_getitem\_\_(self, index):

data = torch.Tensor(self.Data[index])

label = torch.Tensor(self.Label[index])

return data, label

（4）最终这个子类定义为：

import torch

import torch.utils.data.dataset as Dataset

import numpy as np

#创建子类

class subDataset(Dataset.Dataset):

#初始化，定义数据内容和标签

def \_\_init\_\_(self, Data, Label):

self.Data = Data

self.Label = Label

#返回数据集大小

def \_\_len\_\_(self):

return len(self.Data)

#得到数据内容和标签

def \_\_getitem\_\_(self, index):

data = torch.Tensor(self.Data[index])

label = torch.Tensor(self.Label[index])

return data, label

值得注意的地方是：

class subDataset(Dataset.Dataset):

如果只写了Dataset而不是Dataset.Dataset，则会报错：module.**init**() takes at most 2 arguments (3 given)

因为Dataset是module模块，不是class类，所以需要调用module里的class才行，因此是Dataset.Dataset！

1. 在类外对Data和Label赋值：

Data = np.asarray([[1, 2], [3, 4],[5, 6], [7, 8]])  
Label = np.asarray([[0], [1], [0], [2]])  
4. 声明主函数，主函数创建一个子类的对象，传入Data和Label参数：

if \_\_name\_\_ == '\_\_main\_\_':

dataset = subDataset(Data, Label)

1. 输出数据集大小和数据：

print(dataset)

print('dataset大小为：', dataset.\_\_len\_\_())

print(dataset.\_\_getitem\_\_(0))

print(dataset[0])

代码变为;

import torch

import torch.utils.data.dataset as Dataset

import numpy as np

Data = np.asarray([[1, 2], [3, 4],[5, 6], [7, 8]])

Label = np.asarray([[0], [1], [0], [2]])

#创建子类

class subDataset(Dataset.Dataset):

#初始化，定义数据内容和标签

def \_\_init\_\_(self, Data, Label):

self.Data = Data

self.Label = Label

#返回数据集大小

def \_\_len\_\_(self):

return len(self.Data)

#得到数据内容和标签

def \_\_getitem\_\_(self, index):

data = torch.Tensor(self.Data[index])

label = torch.IntTensor(self.Label[index])

return data, label

if \_\_name\_\_ == '\_\_main\_\_':

dataset = subDataset(Data, Label)

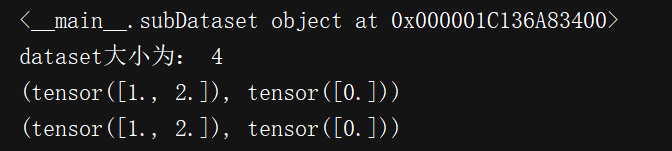
print(dataset)

print('dataset大小为：', dataset.\_\_len\_\_())

print(dataset.\_\_getitem\_\_(0))

print(dataset[0])

结果为：



**三、DataLoader的创建和使用**

1. 引入DataLoader：

import torch.utils.data.dataloader as DataLoader

1. 创建DataLoader，batch\_size设置为2，shuffle=False不打乱数据顺序，num\_workers= 4使用4个子进程：

#创建DataLoader迭代器

dataloader = DataLoader.DataLoader(dataset,batch\_size= 2, shuffle = False, num\_workers= 4)

1. 使用enumerate访问可遍历的数组对象：

{enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 for 循环当中,如果想把dataloader改写成有索引的列表，可以使用 listdataloader=list（enumerate（dataloader实例，start=0））

Dataloader 的item对应的是一个batch batch[0]为data batch[1]为label，如果batchsize为4，那么batch[0]的长度为4}

for step, (data, label) in enumerate(dataloader):

print('step is :', step)

# data, label = item

print('data is {}, label is {}'.format(data, label))

for i, item in enumerate(dataloader):

print('i:', i)

data, label = item

print('data:', data)

print('label:', label)

1. 最终代码如下：

import torch

import torch.utils.data.dataset as Dataset

import torch.utils.data.dataloader as DataLoader

import numpy as np

Data = np.asarray([[1, 2], [3, 4],[5, 6], [7, 8]])

Label = np.asarray([[0], [1], [0], [2]])

#创建子类

class subDataset(Dataset.Dataset):

#初始化，定义数据内容和标签

def \_\_init\_\_(self, Data, Label):

self.Data = Data

self.Label = Label

#返回数据集大小

def \_\_len\_\_(self):

return len(self.Data)

#得到数据内容和标签

def \_\_getitem\_\_(self, index):

data = torch.Tensor(self.Data[index])

label = torch.IntTensor(self.Label[index])

return data, label

if \_\_name\_\_ == '\_\_main\_\_':

dataset = subDataset(Data, Label)

print(dataset)

print('dataset大小为：', dataset.\_\_len\_\_())

print(dataset.\_\_getitem\_\_(0))

print(dataset[0])

#创建DataLoader迭代器

dataloader = DataLoader.DataLoader(dataset,batch\_size= 2, shuffle = False, num\_workers= 4)

for i, item in enumerate(dataloader):

print('i:', i)

data, label = item

print('data:', data)

print('label:', label)

结果为：

  
可以看到两个对象，因为对象数\*batch\_size就是数据集的大小\_\_len\_\_

**四、将Dataset数据和标签放在GPU上（代码执行顺序出错则会有bug）**

1. 改写\_\_getitem\_\_函数：

if torch.cuda.is\_available():

data = data.cuda()

label = label.cuda()

代码变为：

#得到数据内容和标签

def \_\_getitem\_\_(self, index):

data = torch.Tensor(self.Data[index])

label = torch.IntTensor(self.Label[index])

if torch.cuda.is\_available():

data = data.cuda()

label = label.cuda()

return data, label

1. 报错：

THCudaCheck FATIHCudaCheck FAIL file=Lc:\n efwile=-builder\_3\win-whce:el\\pnyteorwch-\tborucihl\cdsrec\rge\_3n\weirinc\StorageSharing.cpp-w helienl\epy=t2or3ch1\ toercrhr\cosrrc=\g71e ne:r ioc\pSteorartagieSohanr niotng .cspupppo line=231 error=rt7e1d

: operProcess Process-2:

ation not supportedTraceback (most recent call last):

File "D:\Anaconda3\lib\multiprocessing\process.py", line 258, in \_bootstrap

self.run()

File "D:\Anaconda3\lib\multiprocessing\process.py", line 93, in run

self.\_target(\*self.\_args, \*\*self.\_kwargs)

File "D:\Anaconda3\lib\site-packages\torch\utils\data\dataloader.py", line 110, in \_worker\_loop

data\_queue.put((idx, samples))

Process Process-1:

File "D:\Anaconda3\lib\multiprocessing\queues.py", line 341, in put

obj = \_ForkingPickler.dumps(obj)

File "D:\Anaconda3\lib\multiprocessing\reduction.py", line 51, in dumps

cls(buf, protocol).dump(obj)

File "D:\Anaconda3\lib\site-packages\torch\multiprocessing\reductions.py", line 109, in reduce\_tensor

(device, handle, storage\_size, storage\_offset) = storage.\_share\_cuda\_()

RuntimeError: cuda runtime error (71) : operation not supported at c:\new-builder\_3\win-wheel\pytorch\torch\csrc\generic\StorageSharing.cpp:231

Traceback (most recent call last):

File "D:\Anaconda3\lib\multiprocessing\process.py", line 258, in \_bootstrap

self.run()

File "D:\Anaconda3\lib\multiprocessing\process.py", line 93, in run

self.\_target(\*self.\_args, \*\*self.\_kwargs)

File "D:\Anaconda3\lib\site-packages\torch\utils\data\dataloader.py", line 110, in \_worker\_loop

data\_queue.put((idx, samples))

File "D:\Anaconda3\lib\multiprocessing\queues.py", line 341, in put

obj = \_ForkingPickler.dumps(obj)

File "D:\Anaconda3\lib\multiprocessing\reduction.py", line 51, in dumps

cls(buf, protocol).dump(obj)

File "D:\Anaconda3\lib\site-packages\torch\multiprocessing\reductions.py", line 109, in reduce\_tensor

(device, handle, storage\_size, storage\_offset) = storage.\_share\_cuda\_()

RuntimeError: cuda runtime error (71) : operation not supported at c:\new-builder\_3\win-wheel\pytorch\torch\csrc\generic\StorageSharing.cpp:231

1. 有两种方法解决（对应在dataloader前将数据放入GPU和在dataloader后放入GPU）：

（1）只需要将num\_workers改成0即可：

dataloader = DataLoader.DataLoader(dataset,batch\_size= 2, shuffle = False, num\_workers= 0)

代码变为：

import torch

import torch.utils.data.dataset as Dataset

import torch.utils.data.dataloader as DataLoader

import numpy as np

Data = np.asarray([[1, 2], [3, 4],[5, 6], [7, 8]])

Label = np.asarray([[0], [1], [0], [2]])

#创建子类

class subDataset(Dataset.Dataset):

#初始化，定义数据内容和标签

def \_\_init\_\_(self, Data, Label):

self.Data = Data

self.Label = Label

#返回数据集大小

def \_\_len\_\_(self):

return len(self.Data)

#得到数据内容和标签

def \_\_getitem\_\_(self, index):

data = torch.Tensor(self.Data[index])

label = torch.IntTensor(self.Label[index])

if torch.cuda.is\_available():

data = data.cuda()

label = label.cuda()

return data, label

if \_\_name\_\_ == '\_\_main\_\_':

dataset = subDataset(Data, Label)

print(dataset)

print('dataset大小为：', dataset.\_\_len\_\_())

print(dataset.\_\_getitem\_\_(0))

print(dataset[0][0])

#创建DataLoader迭代器

dataloader = DataLoader.DataLoader(dataset,batch\_size= 2, shuffle = False, num\_workers= 0)

for i, item in enumerate(dataloader):

print('i:', i)

data, label = item

print('data:', data)

print('label:', label)

结果为：

  
可以看到多了一个device=‘cuda:0’

（2）把Tensor放到GPU上的操作放在DataLoader之后，即删除\_\_getitem\_\_函数里的下面内容

if torch.cuda.is\_available():

data = data.cuda()

label = label.cuda()

并在主函数的for循环里添加删除的语句，代码变为

import torch

import torch.utils.data.dataset as Dataset

import torch.utils.data.dataloader as DataLoader

import numpy as np

Data = np.asarray([[1, 2], [3, 4],[5, 6], [7, 8]])

Label = np.asarray([[0], [1], [0], [2]])

#创建子类

class subDataset(Dataset.Dataset):

#初始化，定义数据内容和标签

def \_\_init\_\_(self, Data, Label):

self.Data = Data

self.Label = Label

#返回数据集大小

def \_\_len\_\_(self):

return len(self.Data)

#得到数据内容和标签

def \_\_getitem\_\_(self, index):

data = torch.Tensor(self.Data[index])

label = torch.IntTensor(self.Label[index])

return data, label

if \_\_name\_\_ == '\_\_main\_\_':

dataset = subDataset(Data, Label)

print(dataset)

print('dataset大小为：', dataset.\_\_len\_\_())

print(dataset.\_\_getitem\_\_(0))

print(dataset[0][0])

#创建DataLoader迭代器

dataloader = DataLoader.DataLoader(dataset,batch\_size= 2, shuffle = False, num\_workers= 8)

for i, item in enumerate(dataloader):

print('i:', i)

data, label = item

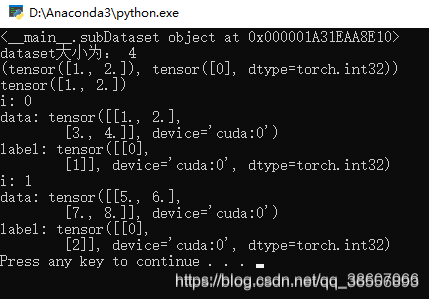
if torch.cuda.is\_available():

data = data.cuda()

label = label.cuda()

print('data:', data)

print('label:', label)

结果为：  


**五、Dataset和DataLoader总结**

1. Dataset是一个抽象类，需要派生一个子类构造数据集，需要改写的方法有\_\_init\_\_，\_\_getitem\_\_等。
2. DataLoader是一个迭代器，方便我们访问Dataset里的对象，值得注意的num\_workers的参数设置：如果放在cpu上跑，可以不管，但是放在GPU上则需要设置为0；或者在DataLoader操作之后将Tensor放在GPU上。
3. 数据和标签是tuple元组的形式，使用Dataloader然后使用enumerate函数访问它们。

类的基本知识：

类的属性：一般分为公有属性和私有属性，默认情况下所有得属性都是公有的，如果属性的名字以两个下划线开始，就表示为私有属性，没有下划线开始的表示公有属性python的属性分为实例属性和静态属性，实例属性是以self为前缀的属性，如果构造函数中定义的属性没有使用self作为前缀声明，则该变量只是普通的局部变量，类中其它方法定义的变量也只是局部变量，而非类的实例属性。

类的方法：类的方法也分为公有方法和私有方法，私有方法不能被模块外的类或者方法调用，也不能被外部的类或函数调用。python利用staticmethon或@staticmethon 修饰器把普通的函数转换为静态方法。

静态方法和实例方法的区别主要体现在两个方面：

1. 在外部调用静态方法时，可以使用"类名.方法名"的方式，也可以使用"对象名.方法名"的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。

2. 静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制。

3. 类方法可以被对象调用，也可以被实例调用；传入的都是类对象，主要用于工厂方法，具体的实现就交给子类处理

4. 静态方法参数没有实例参数 self, 也就不能调用实例参数

1. **Command usually used in vscode**

Vscode tutorial <https://code.visualstudio.com/docs/python/python-tutorial>

VSCODE中，python命令为py

例如:

py main.py

py -m pip list

py -m pip install 等等

windows环境下 VSCODE中的python虚拟环境激活需要先加以下命令：

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process

虚拟环境激活： .venv\scripts\activate

结果：