Pytorch基本说明

1.基本使用配置

- 查看GPU是否可用，同时指定程序运行GPU

```
import os
import torch  # 引入Pytorch库
print(torch.cuda.is_available())  # 测试GPU是否可用,为True则表示GPU可用
os.environ["CUDA_VISIBLE_DEVICES"] = "0"  # 指定程序在第一块GPU上运行
```

- 数据操作

```
import torch
torch.from_numpy(numpy_data)  # 与numpy数据转化
torch.tensor(numpt_data)  # 与numpy数据转化
x = torch.rand(2,1)  # 定义一个张量
print(x.shape)  # 打印张量形状，输出:torch.Size([2,1])
print(x.size())  # 打印张量大小，输出:torch.Size([2, 1])
print(x.reshape([1,2]).shape)  # 先将张量的形状重新塑造，然后打印张量的形状，输
出:torch.Size([1, 2])
```

- 随机数操作

```
import torch
torch.manual_seed(2)  # 设置随机数种子
torch.initial_seed()  # 查看随机数种子，输出:2
torch.randn(2,3)  # 随机生成指定形状的随机值张量
torch.arange(1,10,step=2)  # 在1到10之间，按照步长为2进行取值，输
出:tensor([1,3,5,7,9])
torch.linspace(1,9,steps=5)  # 在1到9之间，均匀地取出5个值，输
出:tensor([1.,3.,5.,7.,9.])
torch.empty(1,2)  # 按照指定形状生成未初始化的矩阵
```

- 张量间的计算

```
import torch
a = torch.FloatTensor([4])
b = torch.add(a,a)  # 调用torch.add()进行相加
torch.add(a,a,out=b)  # 调用torch.add()进行相加，并且将结果输出给b
a.add_(b)  # 实现 a+=b
```

2.微分与变量模块

Variable是Pytorch中的一个变量类型，是由Autograd模块对张量进一步封装实现的。一旦张量(Tensor)被转化为Variable对象，便可以实现自动求导的功能。
Autograd，自动微分模块是构成神经网络训练的必要模块。主要是在神经网络的反向传播过程中，基于正向计算的结果对当前参数进行微分计算，从而实现网络权重的更新。
注意在使用requires_grad时，要求张量的类型必须是浮点型。Pytorch不支持整型做梯度运算。

```
import torch
from torch.autograd import Variable
a = torch.FloatTensor([4])  # 定义张量a
print(Variable(a))  # 把张量a转化为Variable对象，输出:tensor([4.])
# 把张量a转化为支持梯度计算的Variable对象，输出:tensor([4.], requires_grad=True)
print(Variable(a, requires_grad=True))
print(a.data)  # Variable对象转化为张量，输出:tensor([4.])
```

no_grad()函数，顾名思义，可以终止局部的梯度计算。

```
import torch
from torch.autograd import Variable
x = torch.ones(2,2,requires_grad=True)
with torch.no_grad():  # 使用本函数限制requires_grad的作用域
    y = x * 2
z = x * 2
print(z.requires_grad)  # 输出:True
print(y.requires_grad)  # 输出:Flase
@torch.no_grad()
def doubler(x):
    return x * 2
z = doubler(x)
print(z.requires_grad)  # 输出:False
torch.set_grad_enabled(True)  # 全局统一打开/关闭梯度计算功能
```

当带有需求梯度计算的张量经过一系列的计算最终生成一个标量，便可以使用该标量的backward()方法进行自动求导。 该方法会自动调用每个需要求导变量的grad_fn()函数，并且将结果放到该变量的grad属性中。
backward()方法一定要在当前变量内容是标量的情况下使用，否则会报错。

```
import torch
x = torch.ones(2,2,requires_grad=True)  # 定义一个Variable对象，并且打开梯度计算
m = x + 2  # 通过计算得到了m变量
f = m.mean()  # 通过m的mean()方法，得到了一个标量
f.backward()  # 调用标量的backward()进行自动求导
print(f,x.grad)
```

需要求梯度的Variable对象无法被直接转化为Numpy对象。 这个时候可以使用detach()方法，可以将Variable从创建它的图中分离出来， 该代码会创建一个新的、从当前图中分离的Variable，并且把它作为叶子节点。 被返

回的Variable和被分离的Variable指向同一个张量，并且永远不会需要梯度。 实际使用中，可以使用detach()方法实现对网络中的部分参数求梯度的功能。

3.Dataset与DataLoader

PyTorch为我们提供的两个Dataset和DataLoader类分别负责可被Pytorch使用的数据集的创建以及向训练传递数据的任务。 如果想个性化自己的数据集或者数据传递方式，也可以自己重写子类。 一般来说PyTorch中深度学习训练的流程是这样的：

1. 创建Dateset
2. Dataset传递给DataLoader
3. DataLoader迭代产生训练数据提供给模型

```
# 创建Dateset(可以自定义)
    dataset = face_dataset # Dataset部分自定义过的face_dataset
# Dataset传递给DataLoader
    dataloader =
torch.utils.data.DataLoader(dataset,batch_size=64,shuffle=False,num_workers=8)
# DataLoader迭代产生训练数据提供给模型
    for i in range(epoch):
        for index,(img,label) in enumerate(dataloader):
            pass
```

Dataset负责建立索引到样本的映射，DataLoader负责以特定的方式从数据集中迭代的产生 一个个batch的样本集合。
在enumerate过程中实际上是dataloader按照其参数sampler规定的策略调用了其dataset的getitem方法。

- 制作自定义Dataset torch.utils.data.Dataset 是一个表示数据集的抽象类。任何自定义的数据集都需要继承这个类并覆写相关方法。 所谓数据集，其实就是一个负责处理索引(index)到样本(sample)映射的一个类(class)。 Pytorch提供两种数据集： Map式数据集和Iterable式数据集。

Map式数据集

一个Map式的数据集必须要重写getitem(self, index),len(self) 两个内建方法，用来表示从索引到样本的映射（Map）.

这样一个数据集dataset，举个例子，当使用dataset[idx]命令时，可以在你的硬盘中读取你的数据集中第idx张图片以及其标签（如果有的话）;len(dataset)则会返回这个数据集的容量。

```
class CustomDataset(data.Dataset):#需要继承data.Dataset
    def __init__(self):
        # TODO
        # 一般在这里将数据读入DataSet的这个子类。
        pass
    def __getitem__(self, index):
        # TODO
        # 1. Read one data from file (e.g. using numpy.fromfile, PIL.Image.open).
        # 2. Preprocess the data (e.g. torchvision.Transform).
        # 3. Return a data pair (e.g. image and label).
```

```
        #这里需要注意的是，第一步：read one data，是一个data
        pass
    def __len__(self):
        # You should change 0 to the total size of your dataset.
        return 0
```

举一个实例： 图片文件储存在"./data/faces/"文件夹下，图片的名字并不是从1开始，而是从final_train_tag_dict.txt这个文件保存的字典中读取，label信息也是用这个文件中读取。 大家可以照着上面的注释阅读这段代码：

```
from torch.utils import data
import numpy as np
from PIL import Image


class face_dataset(data.Dataset):
    def __init__(self):
        self.file_path = './data/faces/'
        f=open("final_train_tag_dict.txt","r")
        self.label_dict=eval(f.read())
        f.close()

    def __getitem__(self,index):
        label = list(self.label_dict.values())[index-1]
        img_id = list(self.label_dict.keys())[index-1]
        img_path = self.file_path+str(img_id)+".jpg"
        img = np.array(Image.open(img_path))
        return img,label

    def __len__(self):
        return len(self.label_dict)
```

- 制作自定义DataLoader

```
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)
```

- dataset (Dataset) – 定义好的Map式或者Iterable式数据集。
- batch_size (python:int, optional) – 一个batch含有多少样本 (default: 1)。
- shuffle (bool, optional) – 每一个epoch的batch样本是相同还是随机 (default: False)。
- sampler (Sampler, optional) – 决定数据集中采样的方法. 如果有，则shuffle参数必须为False。
- batch_sampler (Sampler, optional) – 和 sampler 类似，但是一次返回的是一个batch内所有样本的index。和 batch_size, shuffle, sampler, and drop_last 三个参数互斥。
- num_workers (python:int, optional) – 多少个子程序同时工作来获取数据，多线程。 (default: 0)
- collate_fn (callable, optional) – 合并样本列表以形成小批量。

- pin_memory (bool, optional) – 如果为True，数据加载器在返回前将张量复制到CUDA固定内存中。
- drop_last (bool, optional) – 如果数据集大小不能被batch_size整除，设置为True可删除最后一个不完整的批处理。如果设为False并且数据集的大小不能被batch_size整除，则最后一个batch将更小。(default: False)
- timeout (numeric, optional) – 如果是正数，表明等待从worker进程中收集一个batch等待的时间，若超出设定的时间还没有收集到，那就不收集这个内容了。这个numeric应总是大于等于0。 (default: 0)
- worker_init_fn (callable, optional*) – 每个worker初始化函数 (default: None)

4.定义网络模型

定义神经网络模型可以分为以下几个步骤：

- 定义网络模型类，使其继承与Module类；
- 在网络模型类的初始化接口中定义网络层；
- 在网络模型类的正向传播接口中，将网络层连接起来，并且添加激活函数，搭建网络结构。

举例:多层全连接神经网络的模型构建

首先是需要生成模型，可以通过继承nn.Module类来生成我们的模型。 其中，初始函数(init)是用来初始化网络的结构，该函数中定义了两个全连接层和一个交叉熵函数；forward函数规定了本模型的正向传播算法；predict函数规定了本模型的预测算法；getloss函数实现了本模型的loss值的计算。

```
import torch.nn as nn
import torch
import numpy as np
import matplotlib.pyplot as plt
class LogicNet(nn.Module):  #继承nn.Module类，构建网络模型
    def __init__(self,inputdim,hiddendim,outputdim):  # 初始化函数中确定网络结构
        super(LogicNet,self).__init__()
        self.Linear1 = nn.Linear(inputdim,hiddendim)  # 定义全连接层
        self.Linear2 = nn.Linear(hiddendim,outputdim)  # 定义全连接层
        self.criterion = nn.CrossEntropyLoss()  # 定义交叉熵函数

    def forward(self,x):  # 搭建网络模型
        x = self.Linear1(x)  # 将输入数据传入第1搁全连接层
        x = torch.tanh(x)  # 对第1个连接层的结果进行非线性变换（使用tanh函数）
        x = self.Linear2(x)  # 将完成了非线性变换后的数据传入第二个连接层
        return x  # 输出第二个连接层的结果

    def predict(self,x):  # 实现LogicNet类的预测接口
        # 调用自身网络模型，并且对结果进行softmax处理，分别得出预测数据中属于每一类的概
率
        x = self.forward(x)  # 调用自身网络模型，将结果赋到x中
        pred = torch.softmax(x,dim=1)  # 返回每组预测概率中最大值的索引

    def getloss(self,x,y):  # 实现LogicNet类的损失值接口
        y_pred = self.forward(x)
        loss = self.criterion(y_pred,y)  # 计算损失值的交叉熵
        return loss
```

只需要将定义好的网络模型类进行实例化，即可真正地完成网络模型的搭建。同时，需要定义训练模型所需要的优化器，优化器会在训练模型的反向传播过程中使用。

```
model = LogicNet(inputdim=2,hiddendim=3,outputdim=2)  # 实例化模型
optimizer = torch.optim.Adam(model.parameters(),lr=0.01)  # 定义优化器
```

在完成实例化模型和定义好优化器后，此时应该开始训练模型，神经网路的训练过程是一步步进行的，每一步的详细操作如下：
(1)每次把数据传入到网络中，通过正向结构得到预测值。
(2)把预测结果与目标间的误差作为损失。
(3)利用反向求导的链式法则，求出神经网络中每一层的损失。
(4)根据损失值对其当前网络层的权重参数进行求导，计算出每个参数的修正值，并且对该层网络中的参数进行更新。

```
x = torch.from_numpy(feature_data).type(torch.FloatTensor)  # 将numpy数据转化为张量
y = torch.from_numpy(target_data).type(torch.LongTensor)
epochs = 270  # 定义迭代次数
losses = list()  # 定义列表，用于接受每一步的损失值
for i in range(epochs):
    loss = model.getloss(x,y)  # 将数据带入模型进行计算，得到损失值
    losses.append(loss.item())  # 保留这一轮训练的损失值到losses中
    optimizer.zero_grad()  # 清空之前的梯度
    loss.backward()  # 反向传播损失值
    optimizer.step()  # 更新参数
```