迭代器：

一个函数返回值的语句若是用yield而非用return，那这个函数可以作为一个生成器(相当于迭代器)。

Python里面有将某个库环境下生成的迭代器转化为通用的迭代器的函数，即为iter(‘’’迭代器’’’)，转化后产生的迭代器如何使用？1.你可以用next(‘’’迭代器’’’)；2.也可以用for item in ‘’’迭代器’’’。

\_\_call\_\_：

一个类若是有\_\_call\_\_方法，则该类实例化的是一个可调用对象(如何函数都是一个可调用1对象)，可调用对象的性质由\_\_call\_\_定义。

Type和object：

在python的世界中，type站在实例化的顶端，object站在继承的顶端。

我发现同一个模块可能能在一个包的不同位置被访问到，例如torch.utils.data.Dataset==torch.utils.data.dataset.Dataset。

Dataset和DataLoader:

(py)torch子库utils表示torch常用工具的库。

Dataset实例化的对象貌似并非一个可调用对象，其具备访问数据集的权限，但访问方式是调用\_\_getitem\_\_方法(我觉得可能也能将其作为迭代器)。

DataLoader实例化的对象貌似也并非一个可调用对象，通过它访问对应数据集的方法是：将它作为一个迭代器使用(iter()+next())。

其中torch.utils.data.Dataset是一个可供访问数据集的抽象类，每当你要使用它时，你都要自己写一个继承这个抽象类的派生类，派生类必须重写Dataset类的抽象方法：\_\_getitem\_\_，用于返回你想要的样本指标+标签。也可以选择性重写\_\_len\_\_方法，用于返回你的数据集的样本数量。也可以选择性重写\_\_init\_\_，从本地或云端读取整个数据集，给\_\_getitem\_\_用。需要注意的是：Dataset对象.\_\_getitem\_\_(index)是Dataset最重要的用法，一般来说，我们调用一次，其应当返回原始数据集中下标为index的样本，这很像是对数组的下标访问。

而DataLoader则比Dataset更高级，DataLoader实例化一个对象需要给其构造函数传入的第一个参数就是Dataset类型的数据，前面说了，这个Dataset本质是提供了一个访问数据集的入口，于是DataLoader的实例化对象就获取了访问数据集的权限。除此之外，DataLoader构造函数传入的参数还有train、batch\_size和shuffle。这里可以说说train参数，train参数表示是从Dataset提供的数据集中取出训练数据还是测试数据，Dataset提供的数据集我们一般指定是pytorch官方提供的数据集，这种数据集中的样本会有标签，表示该样本被视作训练的样本还是测试的样本。

一般对torch.utils.data库的应用，主要就是使用DataLoader类的可调用对象，从指定数据集中取出指定批量的样本。数据集如何指定？可以自定义torch.utils.data.Dataset的派生类的\_\_getitem\_\_，在其中指定，然后传入这个派生类实例化的对象给DataLoader构造函数。还有一种方法，就是使用data\_set=torchvision.dataset.数据集名称(\*arg)，data\_set传给DataLoader的构造函数——但让我疑惑的是，data\_set并非一个Dataset类型的数据，凭什么也能用于构造DataLoader对象？

使用torch.utils.data.TensorDataset(features,labels)也可以将numpy.ndarray类型的数据转换成Dataset。Softmax

数据可视化：

Torch.utils.tensorboard库常用于数据可视化，但要配合tensorboard包，后者tensorboard相对于torch是一个独立的包。

使用这个torch的tensorboard子库，一般情况下只使用其中的SummaryWriter函数，SummaryWriter函数创建一个事件文件event file，并返回一个对象writer，然后调用writer.add\_scalar(\*param)或writer.add\_image(\*param)在这个event file里绘制函数图像，或插入图片。注意绘制或插入完后要关闭event file，即writer.close()。注意：插入图片用的是add\_image，插入神经网络框图用的是add\_graph。

那如何打开这个event file呢？似乎必须在终端里使用tensorbroad包提供的命令。

图像转换：

Pytorch中常用的图像转换工具箱是torchvision.transforms，虽然它叫transforms，但由于是torchvision的子库，所以只用于处理图像问题。它常常用来做图像数据的增广。而且我感觉：1.transforms里的工具基本上都是类，而这些类实例化的都是可调用对象，你在真正对图像进行转化时，使用的是这些可调用对象；2.为了方便对图像的处理嘛，这些可调用对象的输入都可以使PIL.Image.open(path)返回的Image类型的数据，而不一定要是tensor/numpy…。

例如ToTensor类，transforms.ToTensor()，它会实例化一个可调用对象，该可调用对象将PIL.image.open(path)读取Image类型的数据转化为tensor类型。

再例如Normalize，也是实例化一个可调用对象，用来给tensor类型变量存储的image做归一化用的。Obj=transforms.normalize(3个信道的means值，3个信道的standard值)。通常情况下，ToTensor实例化的可调用对象将Image类型数据转化为tensor类型数据后，这个tensor类型数据的每个元素值在0到1之间(而且我怀疑，图像在Image数据类型中存储时，每个元素值也是0~1)，而调用transforms.normalize时传入的参数常常是([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])，这样就把image\_tensor中的0~1元素值map到了-1~1。

再例如Resize，改变图像的shape。

再例如**Compose**，很重要，前面讲的那几个类实例化的都是一个可调用对象，这些可调用对象分别对应一种transform(转化)的方法transi，给Compose构造函数传入参数(列表类型)[trans1,trans2,……]，则其将实例化一个可调用对象，该对象能依次做[trans1, trans2，……]里的transform。这个Compose很重要，很精炼。

(图像)数据集的下载：

Pytorch官网有提供图像数据集，这些图像数据集按理来说和torchvision有关，所以获取这些数据集得用torchvision库。

获取方式是：data\_set=torchvision.dataset.数据集名称(\*arg)，它将返回存储这个数据集的对象data\_set。数据集的命名规范我也不知道，反正你从官网下载下来后它已经给你命名好了。参数\*arg一般包括root、train、download……其中download比较重要，它的True或False决定了你是要直接从root目录里本地获取数据集还是先从网上下载再从root里获取数据集。

其实这里的数据集名称也是一个函数，不同数据集的\*arg都不完全相同，你可以在jupyter中通过”数据集名称??”查看。

模型参数初始化：

nn.init里面有大量给模型参数做初始化的函数，例如normal\_()、zeros\_()、constant\_()、xavier\_normal\_、xavier\_uniform\_……像这种函数名字后面带一个下划线的，一般都是直接对参数列表中的模型参数做修改，它不会返回什么值。

一般用nn.init里面的函数搭建其一个大的函数initFun，专门针对模型中的某一种层，例如把一个层做参数传给initFun，initFun内先对个层对象做判断，看看是不是nn.Linear类型，是就进行操作。定义好initFun后，对模型对象net应用.apply(initFun)，.apply会遍历net中的每一层。当然你也可以对不同的层使用不同的initFun，例如net[0].apply(initFunx)。

当然，假如你要遍历层的话，直接for layer in net也可以实现，只是我怀疑这样只能遍历简单的Sequential定义的net。

神经网络的搭建：

神经网络的构成可以由“模块”和“层”解释。在pytorch中，我们通常称一个Sequential里包裹的全部内容为一个模块，该模块中的每个单元则是一个一个层了，但这些层里，有可能有嵌套的Sequential，那这个嵌套的Sequential也可以称作一个模块。还有，对于独立存在于最外层，未被任何Sequential包裹的层，也可以称作一个模块。在\_\_init\_\_函数中往Sequential添加层时，默认层的名字是数字序号，但你也可以用add\_module来自定义名字。总之：被Sequential包裹的是层，未被包裹的是模块。

但实质上，模块和层的定义很模糊，我这里也只是个人理解，方便区分module/layer的数字索引和str索引而已。

需要用到torch.nn，里面有很多现成的神经网络骨架，具体来说，是骨头、积木。但我觉得，完整的untrained神经网络模型它也有——嘶，好像确实没有……但至少torchvision.modle里就有，而且还有已经训练好的。

Torch.nn还有一个子库叫做torch.nn.functional，里面有很多实现上述神经网络骨架的底层函数，但这个库我们一般用不到，了解一下即可。

用法例如，nn里有一个类Conv2d，它能实例化出一个卷积神经网络，这个神经网络也是一个可调用对象。

但我们在自己搭建神经网络骨架时，往往不直接利用nn提供的现成的返回可调用对象的类，而是自定义一个类，这个类必须继承nn.Module。事实上，nn.Module是nn库中的所有神经网络(包括Conv2d，不信你用\_\_mro\_\_方法查一下呗)的父类，它是nn中最重要的神经网络骨架，pytorch人为规定了自定义的神经网络骨架必须继承自nn.Module。

如何自定义一个神经网络骨架？你这个骨架得是一个类，这个类里至少有两个方法，一个\_\_init\_\_，一个forward，都是改造的nn.Module的方法。改造\_\_init\_\_：它获取一个或多个nn里的神经网络类实例化的对象，例如Conv2d，这些都是可调用对象。改造forward：输入input，然后输出\_\_init\_\_中获取的可调用对象处理的结果。但为什么，net.forward函数负责推理，而我们直接net(X)也可以实现推理呢？这是因为nn.Module实现的\_\_call\_\_函数和forward函数等价了。

你在使用自定义神经网络时，不需要专门调用net.forward(input)，直接使用net(input)即可。

说说sequential的作用。

Sequential也是一个类，它的用法和Conv2d、Linear这些类类似，也是返回一个可调用对象，这个对象也是一个未训练的模型。但不同的是，Sequential类实质上是对Conv2d、Linear这些类实例化的对象的封装，就不需要你在自定义的神经网络模型类中的\_\_init\_\_函数中一个个地去self.Linear1=nn.Linear(1024,10); self.conv1=Conv2d(64,3,3,(3,3))……了——所以，sequential起到一个简化代码的作用。那么，sequential如何对这些神经网络骨架进行封装？就需要你在sequential的构造函数中传入对应参数，参数即是Linear、Conv2d…的实例化对象。

说说self.\_modules的作用：

这是一个数据成员，可以向神经网络中添加层。但很奇怪，由它添加层的神经网络既不能被print，又不能被named\_parameters()、state\_dict()等一般方式访问参数。

这是一个迭代器，可以用[index]访问其中的值：index不是数字，是层的类型，例如”nn.Linear(234,21)”。一般用于\_\_init\_\_中向.\_modules存值，值从\_\_init\_\_的\*arg参数列表来，也就是说用户有了通过构造函数指定神经网络结构的权限。同样道理，我觉得Sequential函数一样能做到。也可以用.values()返回其层对象：方便使用for循环，一般用于forward中访问.\_modules中的层。至于可不可以不调用.values()，直接对self.\_modules用for循环？我没试过不知道。它里面的值都是一个个的神经网络层，它起到与Sequential类似的作用。

说说net.add\_module(name, module/layer)：

向net模型里加入名为name(str)的模块或者层module/layer。

不同层参数共享(share)：

实例化一个层的对象，这个对象在模型的不同地方出现。很简单。

自定义层：

长用到nn.Parameter()函数。传入模型参数w/b的值，这个函数给你加上grad，“名字”(我也不知道这里的名字是什么，或许梯度等数据在tensor中有特定变量名吧，不然哪里来的net.named\_parameters()呢)等属性。

nn.Flatten():

torch.Tensor.flatten()将任意维的张量拉成一维，nn.Flatten()返回的可调用对象却会保留输入的数据的第一个维度，该维度是批量维。

卷积神经网络：

注意：torch.nn.MaxPool2d(\*arg)生成的可调用对象的池化核移动步长stride大小默认为池化核的大小。

Torchvision.modle里的神经网络：

这个包里有现成的搭建好的神经网络，而且貌似这个包里的神经网络不仅仅是用来处理图像分类问题的，还有语义分割。

这里面的神经网络有训练好的和未训练的。

你实例化一个神经网络后，print它，会发现这些神经网络基本都是一个个子层次搭建起来的，例如vgg16由feature、avgpool、classifier构成，下面才细分conv2d、Linear等。假如你想对某个训练好的神经网络进行修改，你可以使用vgg16\_obj.classifier.add\_module(\*arg)、vgg16\_obj.add\_module(\*arg)、vgg16\_obj.classifier[标号]=torch.nn.Linear(1000,10)…

Torch内置的交叉熵损失函数nn.CrossEntropyLoss的输入的pred\_output应当是评估分数，而非softmax后的结果，因为该损失函数会在内部自动帮你做softmax。

访问神经网络模型各参数的方法：

假定神经网络对象名为net。

访问神经网络某一层的方式可以是net[index]，index是数字。但这种访问方式仅限于subscriptable的网络，简单的直接由一个Sequential返回的net是subscriptable，较复杂的网络则不是了。

Net.state\_dict()

.state\_dict()[“2.bias”].data

.bias

.weight

.bias.data

.bias.grad

.named\_parameters()