

pytorch笔记 | (小土堆)

Created @2025年6月8日 16:20

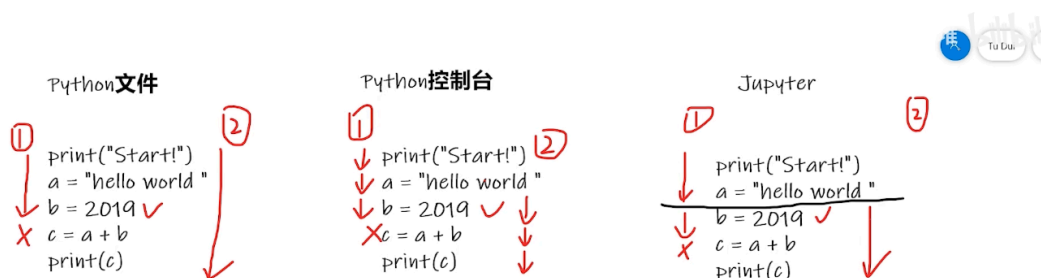
一、torch学习

1、两个工具

用于探索Python或pytorch中的工具包

- `dir ()`: 能让我们知道工具箱中有什么东西
- `help ()`: 告诉我们工具的使用方法

2、pycharm 和 jupyter的区别



代码是以块为一个整体运行的话:

python文件的块是所有行的代码

优: 通用, 传播方便, 适用于大型项目
缺: 需要从头运行

以任意行为块, 运行的

优: 显示每个变量属性
缺点: 不利于代码阅读及修改

以任意行为块运行的

优: 利于代码阅读及修改
缺:

3、pytorch如何加载数据

注: 类中`__init__ (self)`初始化函数

#类首先要初始化, 即根据这个类创建实例的时候自动调用的函数

#作用: 为class提供全局变量, 为后面的函数提供他们所需要的量

主要用到两个库函数:

3.1.1、`Dataset ()`:

3.1.2、`Dataloader ()`:

概念：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
用法：

`test_loader = DataLoader(dataset=test_data, batch_size=4, shuffle=True, num_workers=4)`
如上，`test_loader` 就是我们生成的数据迭代器，
即加载`test_data`数据集，每次打包四个数据，打包成`imgs`和`targets`，
`shuffle`表示每次迭代完之后，下次迭代是否打乱顺序
`drop_last` 表示是否删除非完整页的结尾数据
完整参数参考官方文档。

3.1.3、总结：

`Dataset`是一个包装类，将数据包装为`Dataset`类，然后传入`DataLoader`中，我们再用`DataLoader`这个类来更加快捷的对数据进行操作。

3.2、数据的组织形式

如一个文件夹内存放多个同类的图片：文件夹的名称就是其label，数据和label存放在不同的文件夹内

4、其他工具

4.1、可视化工具Tensorboard—针对模型训练

因为我们编写出来的TensorFlow或pytorch（1.0之后添加了这个模块）程序，建好一个神经网络，

其实我们也不知道神经网络里头具体细节到底做了什么，要人工调试十分困难(就好比无法想象出递归的所有步骤一样)。

有了TensorBoard，可以将TensorFlow程序的执行步骤都显示出来，非常直观。并且，我们可以对训练的参数(比如loss值)进行统计，用图的方式来查看变化的趋势。

- `SummaryWriter`类：'`SummaryWriter`'类提供了一个高级API来创建一个事件文件，在给定的目录中添加摘要和事件。

#命令行中显示事件文件的端口地址：`tensorboard --logdir = 事件文件所在文件夹名`

#修改地址：`tensorboard --logdir=logs --port=6007`

#某些情况可能会有发生事件冲突，造成图像混乱（如修改变量时没有修改标签）：这时可以将事件删除重新运行

• `add_image()`方法：在事件文件中添加图片（本次加载的图片为PIL类型，不符合类型要求，所以要转换，可用OpenCV或numpy直接转换）

4.2、transforms结构及用法

4.2.1、什么是transforms?

常用的图像预处理方法，一般用于转换图片格式，有多个图片处理方法，如：

ToTensor () 对象可传入两种图片格式：

(1) PIL：用PIL的Image工具打开

(2) numpy：用OpenCV打开

4.2.2、transforms该如何使用?

首先创建一个具体的工具(如ToTensor工具，相当于创建类对象)：tool = transforms.ToTensor ()

然后给工具传入参数（传入图片）：result = tool(input)

最后得到tensor类型的图片

4.2.3、为什么我们需要Tensor数据类型?

tensor类型中的很多属性我们都需要在神经网络中用到，如反向传播、梯度等
所以我们必定要用到transforms将数据转换为tensor类型，然后进行训练

4.3、常见的transforms

可直接去文档里查看其相关方法的及其用法

注：Python类中 ——call—— 方法的用法：

相当于类对象的有参构造

def __call__ (self, 参数列表)

4.3.1、ToTensor 方法的使用：

先把图片打开为PIL类型或者numpy类型的对象

然后将对象传入创建好的ToTensor工具，将图片格式转为Tensor类型

4.3.2、Normalize-归一化的使用：

公式：

$$\text{input}[\text{channel}] = (\text{input}[\text{channel}] - \text{mean}[\text{channel}]) / \text{std}[\text{channel}]$$

目的：改变图像的像素范围，

用法：假设是三通道的图片，且像素范围是 (0, 1)

trans_norm=transforms.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])

先实例化一个Normalize对象，然后传入两个参数，一个均值，一个标准差（都是三通道的），结果就将图片的像素范围变成 (-1,1)

注：只有tensor类型的图片能使用这个方法，所以将上边ToTensor的图片直接使用即可。

4.3.3、Resize-图片缩放：

目的：Resize the input PIL Image to the given size, and return a PIL Image
用法：同样可以输入参数列表或一个参数，一个参数的话代表缩放为参数大小的正方形

```
trans_resize = transforms.Resize((512,700))  
#传入参数列表，实例化resize对象对图片进行缩放  
img_resize = trans_resize(img)  
#传入一个PIL图片最后再将输出的图片转为tensor类型用SummaryWriter进行输出
```

4.3.4、Compose-组合方法的使用

目的：将几个转换组合在一起。此转换不支持torchscript。输入是一个transforms对象的列表
相当于简化操作步骤
用法：还是先实例化对象，然后传参

```
trans_resize_2 = transform.Resize(512)  
#传入transforms对象列表进行实例化  
trans_Compose = transeform.Compose([trans_resize_2, trans_tensor])  
img_resize_2 = trans_compose(img)
```

4.3.5、RandomCrop-随机裁剪

目的：在一个随机的位置裁剪给定的图像。返回的结果也是一个PIL，可以传一个参数或两个通道的列表。用法：与上边的使用类似，详见代码。

4.4、torchvision中的数据使用

去pytorch官方文档找torchvision的datasets模块，里面有很多开源的数据集，可以在代码里直接使用和下载，方法参考官方文档。

下载数据集的时候，download选项可以一直为True，因为已下载的不会重复下载。

下载数据集的同时可以转换整个数据集的数据类型：在每个torchvision.datasets下的数据集都会有transforms选项，即transforms转换的实例对象。

代码如下：

```
dataset_transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),])  
train_set = torchvision.datasets.CIFAR10(root="./datasets", train=True,transform=dataset_transform,  
download=True) #训练集  
test_set = torchvision.datasets.CIFAR10(root="./datasets",  
train=False,transform=dataset_transform,download=True) #测试集
```

5、搭建神经网络

5.1、神经网络的基本骨架—nn.Module的使用

注：Python中类的定义时，类的继承直接在括号里添加class 类名（继承的类）

nn（natural network）下的container是nn的容器，包含了基本骨架

其中container中的nn.Module是所有神经网络模块最基本的一个类，为所有神经网络提供基本骨架，

然后再其中进行填充就能形成一个神经网络

forward是一个前向的神经网络处理器（一般会对其进行重写）

如果神经网络要重写初始方法，则必须要调用父类的初始化函数

一个nn.module可以视为一个块。所有的module包含两个主要函数：

init函数：在里边定义一些需要的类或参数。包括网络层

forward函数：做最终的计算和输出，其形参就是模型（块）的输入。

5.2、卷积操作—torch.nn.function

卷积分为不同的层，如conv1、conv2等

以二层卷积为例，具体的参数可查看官方文档

卷积操作主要是用卷积核（weight）与原始数据进行计算，再加上其他的操作，最后得到一个新的输出

代码示例：

```
output3 = F.conv2d(input, kernel, stride=1, padding=1)
print(output3)
```

input3 就是 卷积神经网络模型计算后的 输出，

input是输入的数据，在此为一个二维数组，代表一张图片

kernel表示卷积核，同input形状，也是一个二维数组，并且两者的形状都要有四个指标，否则要进行reshape

stride表示步长

padding表示周围填充几层，填充的默认值是0

5.3、神经网络-卷积层（torch.nn.conv）

其实就是对nn.function的进一步封装，如nn.Conv2(),最常用的是这五个参数：

in_channels、out_channels、kernel_size、stride=、padding
实例：

```
#在初始化方法中定义进行卷积操作
self.conv1 = Conv2d(in_channels=3, out_channels=6, kernel_size=3, stride=1, padding=0)
in_channels=3: 三通道输入（彩色图片）
out_channels=6: 输出是六通道（6层），即生成6个卷积核
kernel_size=3: 每个卷积核的维度是3*3
stride=1: 步长为1
padding=0: 无填充
```

5.4池化层-最大池化（torch.nn.maxpool）

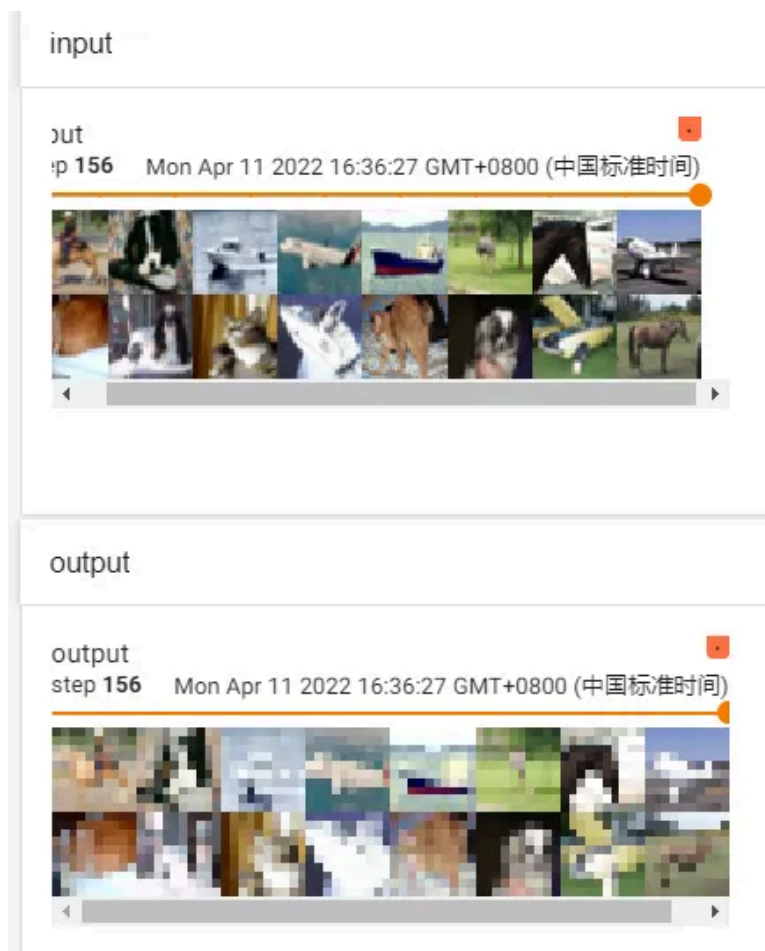
最大池化层（常用的是maxpool2d）的作用：

- 一是对卷积层所提取的信息做更进一步降维，减少计算量
 - 二是加强图像特征的不变性，使之增加图像的偏移、旋转等方面的鲁棒性
- 类似于观看视频时不同的清晰度，实际效果就像给图片打马赛克

maxpool2d：注意输入的图像形状为4维，即形状不对时要先reshape

实例及结果：

```
self.maxpool1 = MaxPool2d(kernel_size=3, ceil_mode=False)
```



5.5、填充层 (torch.nn.padding)

一般在其它层里就可以实现，所以就不过多介绍padding层。

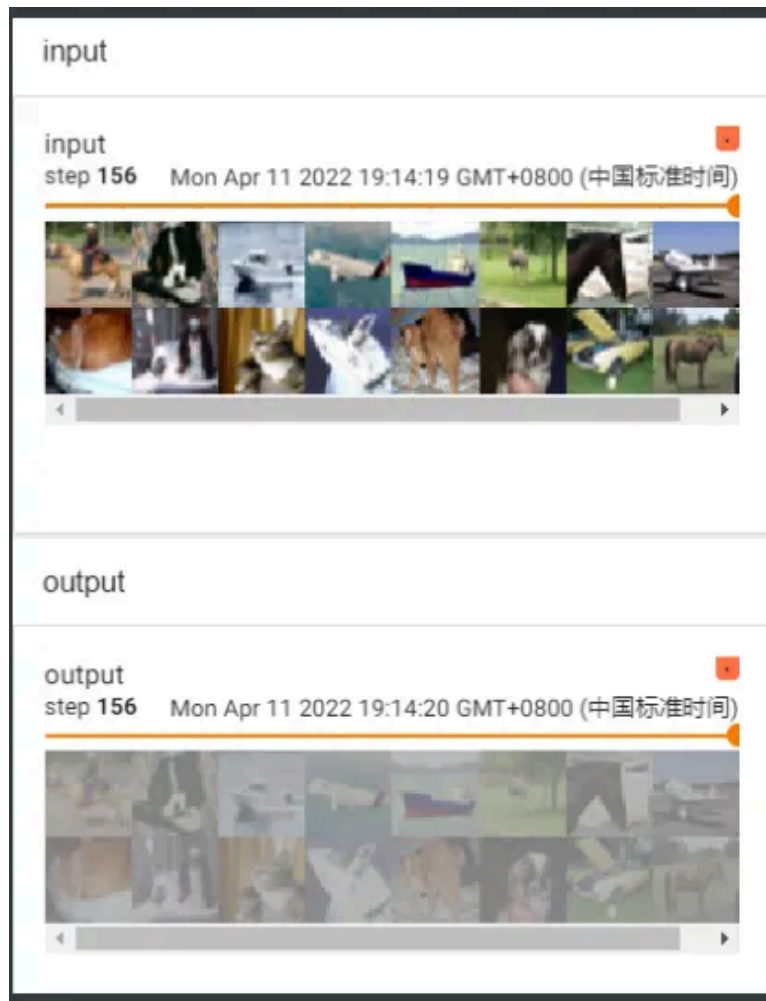
5.6非线性激活 (Non-linear Activations)

非线性变换的主要目的就是给网中加入一些非线性特征，

非线性越多才能训练出符合各种特征的模型。常见的非线性激活：

ReLU：主要是对小于0的进行截断（将小于0的变为0），图像变换效果不明显
主要参数是inplace：inplace为真时，将处理后的结果赋值给原来的参数；为假时，原值不会改变。

SIGMOID：归一化处理，效果没有ReLU好，但对于多元分类问题，必须采用sigmoid
处理后结果：



5.7、线性层（torch.nn.linear）

- 线性层又叫全连接层，其中每个神经元与上一层所有神经元相连

线性函数为：`torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)`，其中重要的3个参数 `in_features`、`out_features`、`bias`说明如下：

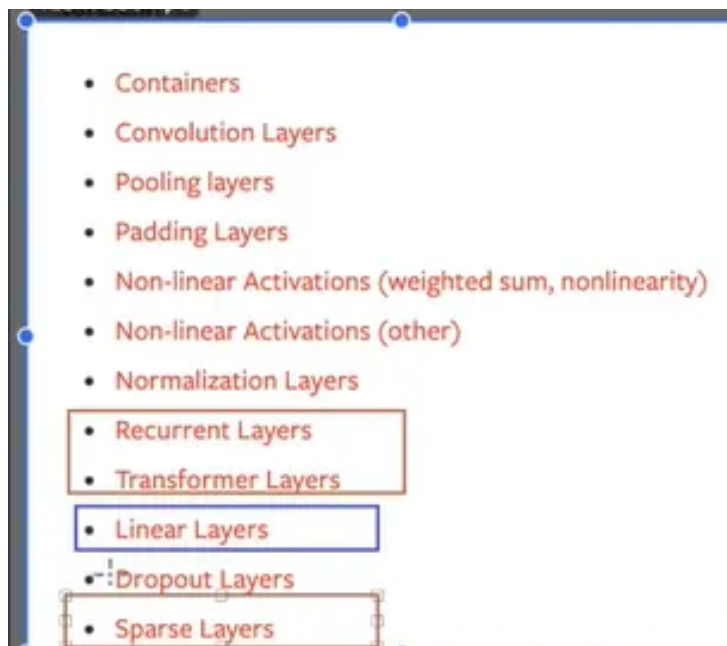
`in_features`：每个输入（x）样本的特征的大小

`out_features`：每个输出（y）样本的特征的大小

`bias`：如果设置为`False`，则图层不会学习附加偏差。默认值是`True`，表示增加学习偏置。

在上图中，`in_features=d`，`out_features=L`。

5.8、其它层



除了前面学的和图中标出来的，其它层用到的一般较少

5.9、SEQUENTIAL的使用 (torch.nn.Sequential)

- 主要是方便代码的编写，使代码更加简洁
- 根据下图搭建神经网络：判断一个图的类别（最后输出为十个类别，最后进行判断）
 - 根据公式计算其他参数的值
- 实例：注意，每个层后要加逗号，相当于传递参数
- 使用tensorboard中的add_graph 查看神经网络的流程图

5.10、损失函数与反向传播

- 注意：inputs和targets的格式一定要符合要求，一般要对其进行reshape和dtype

损失函数举例：

均方差损失 (MSELoss)：

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = \frac{1}{2} (x_n - y_n)^2,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean'}; \\ \text{sum}(L), & \text{if reduction = 'sum'}. \end{cases}$$

L1Loss(): 如图所示，计算的结果

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

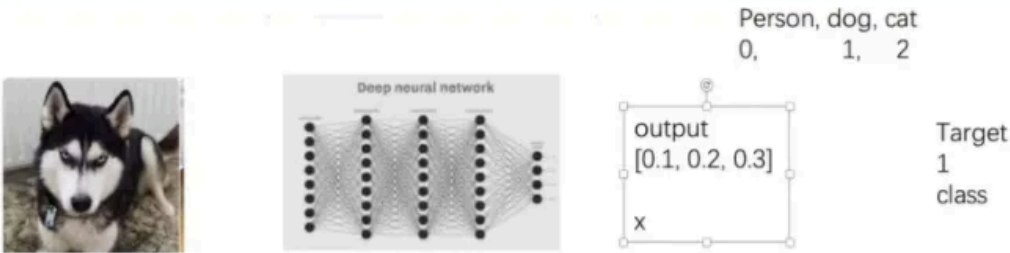
$$\ell(x, y) = L = \{l_1, \dots, l_N\}^T, \quad l_n = |x_n - y_n|,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

交叉熵损失 (CrossEntropyLoss) :x是网络输出的数组，class是类别的下标

$$\text{loss}(x, \text{class}) = -\log \left(\frac{\exp(x[\text{class}])}{\sum_j \exp(x[j])} \right) = -x[\text{class}] + \log \left(\sum_j \exp(x[j]) \right)$$



反向传播：

1. 反向传播是一个更新参数的过程。

(1) 前向传播：将训练集数据输入到ANN的输入层，经过隐藏层，最后到达输出层并输出结果。【输入层-隐藏层-输出层】

(2) 反向传播：由于ANN的输入结果与输出结果有误差，则计算估计值与实际值之间的误差，并将该误差从输出层向隐藏层反向传播，直至传播到输入层。【输出层-隐藏层-输入层】

(3) 权重更新：在反向传播的过程中，根据误差调整各种参数的值；不断迭代上述过程，直至收敛。

5.11、优化器

1、过程描述

继上节的计算损失函数和反向传播，之后便是根据损失值，利用优化器进行梯度更新，然后不断降低loss的过程，一般要对数据集扫描多遍，进行参数的多次更新，才能得到一个较好的效果。

注意，每次更新后要将梯度置0，然后重新计算梯度注意

2、常用优化器：

- 优化器的种类比较多，常用的就是随机梯度下降（SGD）等
- 不同的优化器的参数列表一般不同，但都会有 params(模型的参数列表)和lr(学习率)参数，
- 一般设置这两个参数，其他的可用默认值

二、模型训练

1、现有网络模型的使用及修改

1.1、VGG16模型：

```
vgg16_false = torchvision.models.vgg16(pretrained=False)
vgg16_true = torchvision.models.vgg16(pretrained=True)
```

上面两行代码的区别：

第一个是模型初始化的参数，第二个是经过训练的参数

第一个相当于一个单纯的神经网络结构，第二个是经过训练的神经网络结构（需要
因为VGG16最终的输出是1000个分类，加入我们需要10个分类的话，就需要改动：
修改的方法有两种：

添加一层（线性层），改变最后的输出类别数

或者直接改变原有模型的输出

2、网络模型的保存与读取

2.1、概述

因为有些较大的网络模型（无论是加载初始参数还是预训练过的参数）都需要花费一定的时间，特别是预训练的模型，要花很长时间下载参数，所以我们可以将反复用到的模型保存下来，直接读取使用即可

2.2、保存和读取方法

一般训练好的模型都需要进行保存，否则每次使用都要重新训练。

方式一

保存：保存模型结构及其参数。`torch.save(model, path)`

读取：获取一个完整的模型。`torch.load("模型名")`

方式二

保存：只保存模型的参数。`torch.save(model.state_dict(), path)`

读取：只能加载出模型的参数，要先新建网络模型，然后再装载参数（一般用于加载预训练的参数）。

陷阱：自定义的网络如果保存后再加载的话，需要再重新定义一遍网络结构。

3、完整的模型训练套路

- 准备数据集
- 创建迭代器
- 创建网络模型
- 创建损失函数
- 添加优化器
- 设置一些训练的参数（训练次数、训练轮数等），
- 开始模型训练
 - 优化器不断优化模型
- 开始测试
- 打印测试结果（准确度）