线性代数与模型求导

标量，向量，矩阵运算

1. 基础张量操作

函数/属性格式	作用
torch.tensor(data)	创建张量
torch.arange(start,end,dtype=)	创建从start到end-1的序列
torch.zeros(shape)	创建全0张量
torch.ones(shape)	创建全1张量
x.dtype	张量数据类型
x.shape	张量形状
len(x)	张量第一个维度长度
x[...]	索引访问
x.reshape(shape)	改变张量形状
x.clone()	复制张量，分配新内存
x.numel()	张量中元素数量

2. 张量基本运算

函数/属性格式	作用
x.sum()	所有元素求和
x.mean()	所有元素求均值
x.sum(axis=n)	沿着制定维度求和
x.mean(axis=n)	沿着指定维度求均值
x.mean(axis=n，keepdims=True)	保留原维度求和
x.cumsum(axis=n)	沿着指定维度累加求和

函数/属性格式	作用
torch.abs(x)	取绝对值

特定轴求和实现
张量的形状shape为一个list，那么轴axis的值i=0,1,2，...对应的就是list[i]对应的元素

如果对axis为0计算一个sum()，就是在其他不变的情况下改变行进行累加，例如对于5 * 4的一个矩阵A，对axis=0进行求和就是在每一列固定的时候改变不同行 进行求和，即对A[i , 1] i=1,2,3,4求和，最后得到一个shape为[4]的张量，如果对axis=1可以得到shape为[5]的张量，对哪个轴求和该轴对应的shape值就去掉的得到的就是求和后的张量。对于更高维张量也是如此
如果keepdims=True，就是将对应shape列表中的对应值换为1，上述A经过axis=0求和得到的张量的shape为[1,4]，axis=1求和得到的张量的shape为[5,1]
3. 线性代数运算

函数格式	作用
torch.dot(x,y)	一维向量内积
torch.sum(x * y)	一维向量内积
torch.ger(x,y)	一维向量外积
torch.mm(mat1,mat2)	两个二维矩阵相乘
mat1 * mat2	两个二维矩阵相乘
torch.mv(mat,vec)	矩阵和向量相乘
torch.norm(x)	张量的范数
torch.abs(x).sum	L1范数，绝对值和

4. 导数计算
    - 标量导数：基本求导方法。复合函数求导

- 亚导数：对于y=|x|这种在x=0上不存在所谓的导数，可以定其导数在x>0时为1，x<0时为-1，x在0上导数范围可以是[-1,1]
- 梯度计算
  对于因变量y与自变量x之间导数关系
  - 当y是标量，x是标量，符合基本的导数求解
  - 当y是标量，x是列向量，则最后得到的是一个行向量，其组成元素就是y分别对x向量中的元素求偏导。可以理解为y的值沿着梯度方向变化最大
  - 当y是列向量，x是标量，最后求导得到的是一个列向量，组成元素是y向量的元素分别对x求导
  - 当y是一个列向量m，x是一个列向量n，导数为x,y各个元素对应偏导组成的 m * n 雅可比矩阵
  - 链式法则 $$\frac{\partial y}{\partial x}=\frac{\partial y}{\partial u_n}\frac{\partial u_n}{\partial u_{n-1}}\cdots\frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial x}$$
    正向累积：从下往上进行导数的计算
    反向累积：从上往下进行导数的计算，相比正向累积将所有都求出来，反向累积的空间复杂度更低，时间复杂度两者相同

5. 自动求导

函数/属性	作用
`x.requires_grid_(True)`	开启张量x的梯度记录功能，使其参与自动求导的计算
`x.grid`	存储x对最终标量输出的梯度值
`y.backward()`	反向传播，计算y对所有叶子结点的梯度
`x.grad.zero_()`	梯度清零，pytorch默认梯度是累加的，所以多次计算的时候需要清除旧梯度

函数/属性	作用
`tensor.detach()`	创建一个新张量，与原始张量共享数据，但是不再参与梯度计算，相当于将其中的某个变量变为常数

神经网络导数计算机理

神经网络的本质就是求导

1. 正向传播
   正向传播的本质就是将输入的数据按照层进行传递，通过加权激活等操作最后得到输出的预测值
   这里一般就是定义一个net，可以使用Sequential容器输入各层按顺序进行连接，数据就逐层进行计算
2. 反向传播
   反向传播就是根据链式法则，从输出端开始，层层向前进行计算损失函数梯度，最后更新每一层的参数
   使用 `backward()` 函数自动对梯度进行计算，并且可以通过 `.grad` 属性进行查看和输出
   `backward()` 只能对标量张量使用(形状torch.Size为1的单个数值)
   不能对多维张量使用，因为不知道对哪个维度的方向进行求导
   所以需要将多维张量变为标量张量，常用的做法是使用 `.sum().backward()` 或者 `.mean().backward()`
   `.sum().backward()` 的作用是将每个梯度的损失总和求导
   `.mean().backward()` 的作用是对每个梯度损失求平均求导
   先把所有元素的“误差”汇总成一个数字（总损失），再根据这个数字去衡量每个参数的影响，进而计算梯度。
   所以最后优化是通过不断调整参数来减少各个元素对应损失梯度值，最后实现损失梯度值最小
3. 清零梯度
   默认情况下，PyTorch 的 `.grad` 会 **累积梯度（accumulate**

gradients），而不是自动清零。
因此在每次迭代开始前，需要手动清空梯度：

```
optimizer.zero_grad()
```

如果不清零就会导致梯度在每一个batch之间不断累积，导致训练发散