# 2.6.1 从解析解到数值解，从机器学习到深度学习

解析解（analytical solution），又称闭式解，是可以用解析表达式来表达的解（有时也称为公式解）。在数学上，如果一个方程或者方程组存在的某些解，是由有限次常见运算的组合给出的形式，则称该方程存在解析解。二次方程的根就是一个解析解的典型例子。当解析解不存在时，比如五次及更高次的代数方程，则该方程只能用数值分析的方法求解近似值（有限元的方法、数值逼近，差值的方法，大多数深度学习通过优化算法有限次迭代模型参数来尽可能降低损失函数的值），则是数值解（numerical solution） 。例如大多数偏微分方程，尤其是非线性偏微分方程。

在“简单回归，多元回归”部分使用解析解的方法（对真实值与预测值之差的平方和，即残差平方和求微分或偏微分）解得回归系数；在“梯度下降法”部分，则应用了数值解的优化算法梯度下降法，通过配置学习率、精度和随机初始化系数，定义模型，定义损失函数（残差平方和为一种损失函数，又称代价函数），定义梯度下降函数（对损失函数求梯度，即对模型变量微分），迭代训练模型直至损失函数计算的结果小于预设值，解得模型权重值。

机器学习（machine learning）是人工智能的一个分支，机器学习理论主要是设计和分析一些让计算机可以自动“学习”的算法。机器学习算法是一类从数据中自动分析获得规律，并利用规律对未知数据进行预测的算法。常用的机器学习库是scikit-learn，包括数据预处理，及聚类、回归和分类三大方向，其中提供了SGDRegressor随机梯度下降方法。深度学习（deep learning）是机器学习的一个分支，是一种以人工神经网络为结构，对资料数据进行表征学习的算法。深度学习库推荐使用PyTorch，是以Python优先的深度学习框架，其设计符合人类的思维方式（而TensorFlow因为不符合人们的思维习惯，很难应用，但后来结合到Keras框架，确立为Tensorflow高阶API，使得应用TensorFlow的环境得以改善）。基于PyTorch的深度学习资源推荐*Dive into Deep Learning*[1]，中文版动手深度学习。下述的深度学习方法基础解析亦是以其为主要参考。

PyTorch的安装可以查看其官网[Get Started](https://pytorch.org/get-started/locally/)①，根据机器配置情况，包括系统（Linux、Mac和Windows）、安装所用的包管理器（Conda、Pip、LibTorch和Source）、所用语言（Python或者C++/Java）和计算平台（GPU（CUDA）或CPU）等，生成安装代码，例如conda install pytorch torchvision torchaudio pytorch-cuda=11.6 -c pytorch -c nvidia。

### 2.6.1.1 张量(tensor)

最常用的数据格式是使用NumPy库提供的数组（array）形式（也是机器学习库Sklearn数据集，及数据处理的格式），及pandas库提供的DataFrame（常配合地理信息系统中的table/表，及数据库使用）、Series数据格式。在深度学习（[PyTorch](https://pytorch.org/)②库，或者[TensorFlow](https://www.tensorflow.org/)③）中，引入了张量（tensor），可以用于GPU计算，并有自动求梯度等更多功能。张量和数组类似，0维（一个数，rank=0，rank为维度数）张量，即为标量（Scalar），1维张量（rank=1）即为向量/矢量（Vector），2维张量（rank=2）即为矩阵（Matrix），3维张量（rank=3）即为矩阵数组。深度学习中，张量可以看作一个多维数组（multidimentional array）。

import torch  
print("\_"\*50,"A-tensor创建")  
# 01-未初始化，shape=（5，3）的tensor  
t\_a=torch.empty(5,3)  
print("01-创建未初始化，shape=（5，3）的tensor:\nt\_a={},\nshape={}".format(t\_a,t\_a.shape))  
# 02-随机初始化shape=（5，3）的tensor  
t\_b=torch.rand(5,3)  
print("02-随机初始化shape=（5，3）的tensor:\nt\_b={}".format(t\_b))  
# 03-数据类型为long，全0的tensor  
t\_c=torch.zeros(5,3,dtype=torch.long)  
print("03-数据类型为long，全0的tensor:\nt\_c={}".format(t\_c))  
# 04-由给定数据直接创建  
t\_d=torch.tensor([[3.1415,0],[9,2.71828]])  
print("04-由给定数据直接创建:\nt\_d={}".format(t\_d))  
# 05-tensor.new\_ones()方法重用已有tensor，保持数据类型，及torch.device(CPU或GPU)  
t\_e=t\_c.new\_ones(4,3)  
print("05-tensor.new\_ones()方法重用已有tensor，保持数据类型，及torch.device(CPU或GPU):\nt\_e={}\ndtype={},device={}".format(t\_e,t\_e.dtype,t\_e.device))  
# 06-torch.randn\_like()方法重用已有tensor,并可重定义数据类型  
t\_f=torch.randn\_like(t\_c,dtype=torch.float)  
print("#06-torch.randn\_like()方法重用已有tensor,并可重定义数据类型:\nt\_f={},\ndtype={}".format(t\_f,t\_f.dtype))  
# 07-tensor.size()和tensor.shape方法查看tensor形状  
print("07-tensor.size()和tensor.shape方法查看tensor形状:\nt\_f.size()={},t\_f.shape={}".format(t\_f.size(),t\_f.shape))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ A-tensor创建  
01-创建未初始化，shape=（5，3）的tensor:  
t\_a=tensor([[0., 0., 0.],  
 [0., 0., 0.],  
 [0., 0., 0.],  
 [0., 0., 0.],  
 [0., 0., 0.]]),  
shape=torch.Size([5, 3])  
02-随机初始化shape=（5，3）的tensor:  
t\_b=tensor([[0.4364, 0.0622, 0.8621],  
 [0.2628, 0.5856, 0.4788],  
 [0.4085, 0.8417, 0.7704],  
 [0.0316, 0.3081, 0.3179],  
 [0.4681, 0.3444, 0.0639]])  
03-数据类型为long，全0的tensor:  
t\_c=tensor([[0, 0, 0],  
 [0, 0, 0],  
 [0, 0, 0],  
 [0, 0, 0],  
 [0, 0, 0]])  
04-由给定数据直接创建:  
t\_d=tensor([[3.1415, 0.0000],  
 [9.0000, 2.7183]])  
05-tensor.new\_ones()方法重用已有tensor，保持数据类型，及torch.device(CPU或GPU):  
t\_e=tensor([[1, 1, 1],  
 [1, 1, 1],  
 [1, 1, 1],  
 [1, 1, 1]])  
dtype=torch.int64,device=cpu  
#06-torch.randn\_like()方法重用已有tensor,并可重定义数据类型:  
t\_f=tensor([[ 0.9716, 0.5021, -0.9884],  
 [ 0.6704, 0.8553, 1.4779],  
 [ 0.3652, 1.5988, -0.7453],  
 [-0.3100, -1.4392, 0.4762],  
 [-0.0064, -0.3408, 2.0674]]),  
dtype=torch.float32  
07-tensor.size()和tensor.shape方法查看tensor形状:  
t\_f.size()=torch.Size([5, 3]),t\_f.shape=torch.Size([5, 3])

print("\_"\*50,"B-tensor操作")  
# 08-加法形式-1-'+'  
print("08-加法形式-1-'+':\nt\_a+t\_b={}".format(t\_a+t\_b))  
# 09-加法形式-2-torch.add()  
print("09-加法形式-2-torch.add():\ntorch.add(t\_a,t\_b)={}".format(torch.add(t\_a,t\_b)))  
# 10-加法形式-3-原地结果替换inplace/add\_(PyTorch原地操作inplace都有后缀\_)  
print("10-加法形式-3-原地结果替换inplace/add\_:\nt\_a.add\_(t\_b)={}，\nt\_a={}".format(t\_a.add\_(t\_b),t\_a))  
# 11-索引，共享存储地址  
t\_g=t\_a[0,:]  
print("11-索引，共享存储地址:\nt\_a[0,:]={}".format(t\_g))  
t\_g+=1  
print("t\_g+=1后,tg={},t\_a[0,:]={}".format(t\_g,t\_a[0,:]))  
# 12-view()方法改变tensor形状（shape），但为同一存储地址  
print("12-view()方法改变tensor形状（shape），但为同一存储地址:\nt\_a.shape={},t\_a.view(15).shape={},t\_a.view(-1,5).shape={}".format(t\_a.shape,t\_a.view(15).shape,t\_a.view(-1,5).shape))  
# 13-clone()  
print("13-clone():\nid(t\_a)==id(t\_a.clone())={}".format(id(t\_a)==id(t\_a.clone())))  
  
print("\_"\*50,"C-tensor广播机制")  
# 14-广播机制  
t\_h=torch.arange(1,3).view(1,2)  
t\_i=torch.arange(1,4).view(3,1)  
print("14-广播机制:\nt\_h={}\nt\_i={}\nt\_h.shape={},t\_i.shape={}\nt\_h+t\_i={}".format(t\_h,t\_i,t\_h.shape,t\_i.shape,t\_h+t\_i))  
  
print("\_"\*50,"D-inplace原地操作，节约内存")  
# 15-inplace——out参数  
print("15-inplace——out参数:\nid(t\_a)==id(torch.add(t\_a,t\_b,out=t\_a))={}".format(id(t\_a)==id(torch.add(t\_a,t\_b,out=t\_a))))  
# 16-inplace——+=(add\_())  
t\_j=t\_a  
t\_a+=1  
print("16-inplace——+=(add\_()):\nid(t\_a)==id(t\_j)={}".format(id(t\_a)==id(t\_j)))  
# 17-inplace——[:]  
t\_k=t\_a  
t\_a[:]=t\_a+t\_b  
print("17-inplace——[:]:\nid(t\_a)==id(t\_k)={}".format(id(t\_a)==id(t\_k)))  
  
print("\_"\*50,"E-tensor<-->array(NumPy)")  
# 18-tensor-->array  
print("18-tensor-->array:\ntype(t\_a)={},type(t\_a.numpy())={}".format(type(t\_a),type(t\_a.numpy())))  
# 19-array-->tenssor  
import numpy as np  
array=np.arange(15).reshape(3,5)  
print("19-array-->tenssor:\ntype(array)={},type(torch.from\_numpy(array))={}".format(type(array),type(torch.from\_numpy(array))))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ B-tensor操作  
08-加法形式-1-'+':  
t\_a+t\_b=tensor([[0.4364, 0.0622, 0.8621],  
 [0.2628, 0.5856, 0.4788],  
 [0.4085, 0.8417, 0.7704],  
 [0.0316, 0.3081, 0.3179],  
 [0.4681, 0.3444, 0.0639]])  
09-加法形式-2-torch.add():  
torch.add(t\_a,t\_b)=tensor([[0.4364, 0.0622, 0.8621],  
 [0.2628, 0.5856, 0.4788],  
 [0.4085, 0.8417, 0.7704],  
 [0.0316, 0.3081, 0.3179],  
 [0.4681, 0.3444, 0.0639]])  
10-加法形式-3-原地结果替换inplace/add\_:  
t\_a.add\_(t\_b)=tensor([[0.4364, 0.0622, 0.8621],  
 [0.2628, 0.5856, 0.4788],  
 [0.4085, 0.8417, 0.7704],  
 [0.0316, 0.3081, 0.3179],  
 [0.4681, 0.3444, 0.0639]])，  
t\_a=tensor([[0.4364, 0.0622, 0.8621],  
 [0.2628, 0.5856, 0.4788],  
 [0.4085, 0.8417, 0.7704],  
 [0.0316, 0.3081, 0.3179],  
 [0.4681, 0.3444, 0.0639]])  
11-索引，共享存储地址:  
t\_a[0,:]=tensor([0.4364, 0.0622, 0.8621])  
t\_g+=1后,tg=tensor([1.4364, 1.0622, 1.8621]),t\_a[0,:]=tensor([1.4364, 1.0622, 1.8621])  
12-view()方法改变tensor形状（shape），但为同一存储地址:  
t\_a.shape=torch.Size([5, 3]),t\_a.view(15).shape=torch.Size([15]),t\_a.view(-1,5).shape=torch.Size([3, 5])  
13-clone():  
id(t\_a)==id(t\_a.clone())=False  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ C-tensor广播机制  
14-广播机制:  
t\_h=tensor([[1, 2]])  
t\_i=tensor([[1],  
 [2],  
 [3]])  
t\_h.shape=torch.Size([1, 2]),t\_i.shape=torch.Size([3, 1])  
t\_h+t\_i=tensor([[2, 3],  
 [3, 4],  
 [4, 5]])  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ D-inplace原地操作，节约内存  
15-inplace——out参数:  
id(t\_a)==id(torch.add(t\_a,t\_b,out=t\_a))=True  
16-inplace——+=(add\_()):  
id(t\_a)==id(t\_j)=True  
17-inplace——[:]:  
id(t\_a)==id(t\_k)=True  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ E-tensor<-->array(NumPy)  
18-tensor-->array:  
type(t\_a)=<class 'torch.Tensor'>,type(t\_a.numpy())=<class 'numpy.ndarray'>  
19-array-->tenssor:  
type(array)=<class 'numpy.ndarray'>,type(torch.from\_numpy(array))=<class 'torch.Tensor'>

if torch.cuda.is\_available():  
 device=torch.device("cuda")  
 print("device={}".format(device))  
 x=torch.tensor([[3.1415,0],[9,2.71828]],device=device)  
 print("x={}".format(x))  
 y=torch.rand(2,2)  
 print("默认CPU,y={}".format(y))  
 y=y.to(device)  
 print(".to(device)，y={}".format(y))   
 print("GPU运算,x+y={}".format(x+y))  
 print(".to('cpu'),(x+y).to('cpu',torch.double)={}".format((x+y).to("cpu",torch.double)))

device=cuda  
x=tensor([[3.1415, 0.0000],  
 [9.0000, 2.7183]], device='cuda:0')  
默认CPU,y=tensor([[0.3476, 0.3809],  
 [0.3052, 0.5888]])  
.to(device)，y=tensor([[0.3476, 0.3809],  
 [0.3052, 0.5888]], device='cuda:0')  
GPU运算,x+y=tensor([[3.4891, 0.3809],  
 [9.3052, 3.3071]], device='cuda:0')  
.to('cpu'),(x+y).to('cpu',torch.double)=tensor([[3.4891, 0.3809],  
 [9.3052, 3.3071]], dtype=torch.float64)

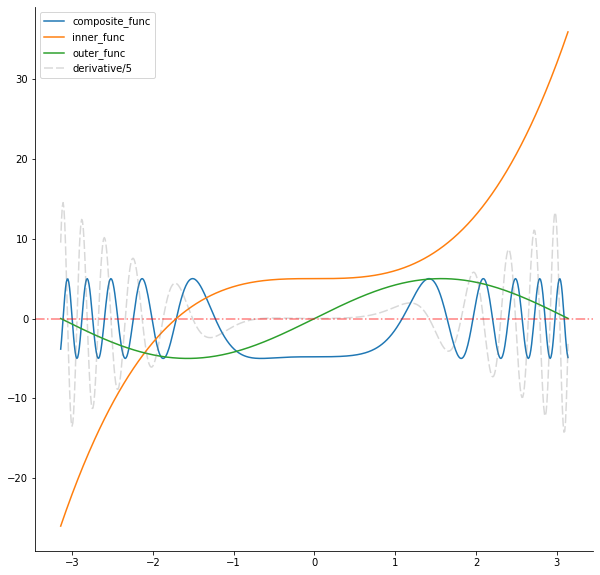
### 2.6.1.2 微积分-链式法则（chain rule）

在“编程微积分”部分，求导的基本公式中给出了复合函数的微分公式为： ，也表示为：。链式法则表述为，两个函数组合起来的复合函数。导数等于内层函数代入外层函数的导函数，乘以内层函数的导函数。下述验证代码中，定义了复合函数为，将其分解为外层函数：和内层函数: 。使用SymPy库diff方法求导，分别对各个分解函数求导，求导后，外层导函数带入内层函数，最后求积即为应用链式法则求复合函数的结果。其结果与直接对复合函数求导的结果保持一致。

链式法则应用于深度学习神经网络的反向传播计算。

import sympy  
from sympy import diff,pprint  
import matplotlib.pyplot as plt  
import numpy as np  
  
x,x\_i,x\_o=sympy.symbols('x x\_i,x\_o')  
composite\_func=5\*sympy.sin(x\*\*3+5)  
composite\_func\_=sympy.lambdify(x,composite\_func,"numpy")  
print("复合函数：")  
pprint(composite\_func)  
  
inner\_func=x\_i\*\*3+5  
inner\_func\_=sympy.lambdify(x\_i,inner\_func,"numpy")  
print("内层函数：")  
pprint(inner\_func)  
  
outer\_func=5\*sympy.sin(x\_o)  
outer\_func\_=sympy.lambdify(x\_o,outer\_func,"numpy")  
print("外层函数：")  
pprint(outer\_func)  
  
# 求导  
d\_composite=diff(composite\_func,x)  
d\_composite\_=sympy.lambdify(x,d\_composite,"numpy")  
print("复合函数-求导：")  
pprint(d\_composite)  
  
d\_chain\_rule\_form\_1=diff(outer\_func,x\_o).subs(x\_o,inner\_func)\*diff(inner\_func,x\_i)  
print("链式法则-求导:")  
pprint(d\_chain\_rule\_form\_1)  
  
t=np.arange(-np.pi,np.pi,0.01)  
y\_composite=composite\_func\_(t)  
y\_inner=inner\_func\_(t)  
y\_outer=outer\_func\_(t)  
  
fig, ax = plt.subplots(figsize=(10,10))  
ax.plot(t,y\_composite,label="composite\_func")  
ax.plot(t,y\_inner,label="inner\_func")  
ax.plot(t,y\_outer,label="outer\_func")  
ax.plot(t,d\_composite\_(t)/10,label="derivative/5",dashes=[6, 2],color='gray',alpha=0.3)  
ax.axhline(y=0,color='r',linestyle='-.',alpha=0.5)  
  
ax.legend(loc='upper left')  
ax.spines['right'].set\_visible(False)  
ax.spines['top'].set\_visible(False)  
plt.show()

复合函数：  
 ⎛ 3 ⎞  
5⋅sin⎝x + 5⎠  
内层函数：  
 3   
xᵢ + 5  
外层函数：  
5⋅sin(xₒ)  
复合函数-求导：  
 2 ⎛ 3 ⎞  
15⋅x ⋅cos⎝x + 5⎠  
链式法则-求导:  
 2 ⎛ 3 ⎞  
15⋅xᵢ ⋅cos⎝xᵢ + 5⎠



### 2.6.1.3 激活函数（activation function）

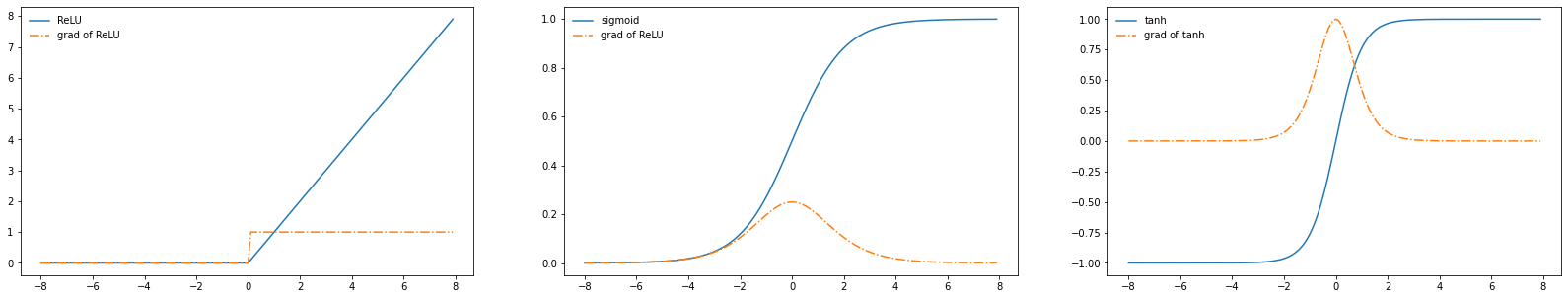
在多层神经网络中（多层感知机，multilayer perception，MLP），如果不使用激活函数，则每一层节点（神经元）的输入都是上一层输出的线性函数，那么无论神经网络有多少层，输出都是输入的线性组合，即为原始的感知机（单层感知机，perception），其网络的逼近能力有限。因此引入了非线性函数作为激活函数，使神经网络可以逼近任意函数。常用的激活函数有ReLU（rectified linear unit），sigmoid，和tanh函数。

给定元素x，ReLU函数的定义为：，可知ReLU函数只保留正数元素，并将负数元素清零，为两段线性函数。ReLU函数的导数对应在负数时，为0，正数时为1。值为0时，ReLU函数不可导，在此处将其导数配置为0。

sigmoid函数可以将元素的值变换到0~1之间，其公式为：。sigmoid函数在早期的神经网络中较为普遍，目前逐渐被更为简单的ReLU函数取代。当输入接近0时，sigmoid函数接近线性变换。sigmoid函数的导数，在输入为0时，导数达到最大值0.25；当输入偏离0时，导数趋近于0.

tanh（双曲正切）函数可以将元素的值变换到-1~1之间。其公式为：，当输入接近0时，tanh函数接近线性变换。同时，tanh函数在坐标系原点上对称。依据链式法则，tanh函数的导数为：。当输入为0时，tanh的导数达到最大值1；当输入偏离0时，tanh函数的导数趋近于0。

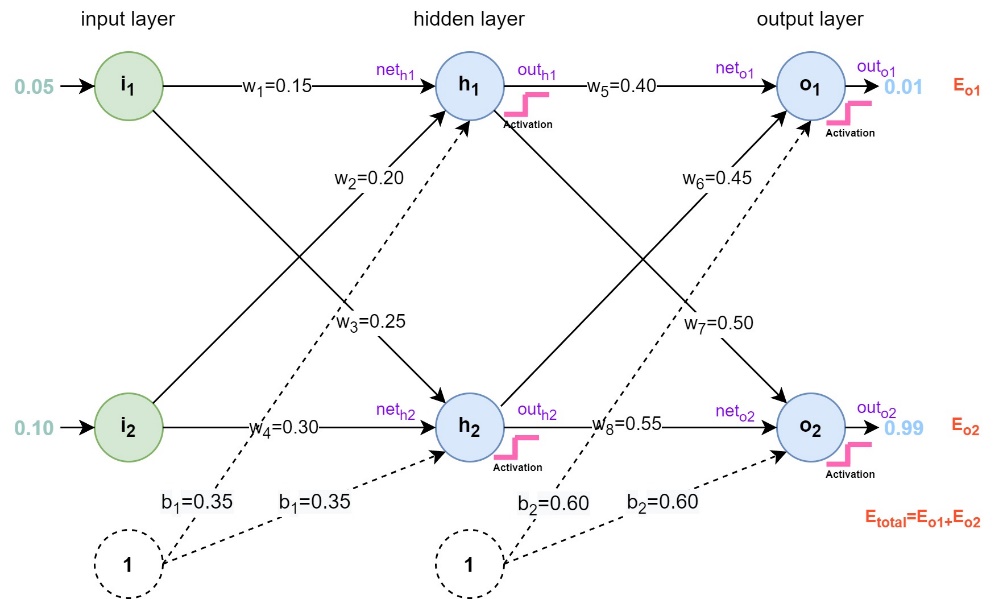
import torch  
import matplotlib.pyplot as plt  
import torch.nn as nn  
fig, axs=plt.subplots(1,3,figsize=(28,5))  
  
# A-ReLU  
x=torch.arange(-8.0,8.0,0.1,requires\_grad=True)  
y\_relu=x.relu()  
axs[0].plot(x.detach().numpy(), y\_relu.detach().numpy(),label="ReLU")  
  
# ReLU函数的导数  
y\_relu.sum().backward()  
axs[0].plot(x.detach().numpy(), x.grad.detach().numpy(),label="grad of ReLU",linestyle='-.')  
  
# B-sigmoid  
y\_sigmoid=x.sigmoid()  
axs[1].plot(x.detach().numpy(), y\_sigmoid.detach().numpy(),label="sigmoid")  
  
# sigmoid函数的导函数  
x.grad.zero\_() # 参数梯度置零  
y\_sigmoid.sum().backward()  
axs[1].plot(x.detach().numpy(), x.grad.detach().numpy(),label="grad of ReLU",linestyle='-.')  
  
# C-tanh  
y\_tanh=x.tanh()  
axs[2].plot(x.detach().numpy(), y\_tanh.detach().numpy(),label="tanh")  
  
# tanh函数的导函数  
x.grad.zero\_()  
y\_tanh.sum().backward()  
axs[2].plot(x.detach().numpy(), x.grad.detach().numpy(),label="grad of tanh",linestyle='-.')  
  
axs[0].legend(loc='upper left', frameon=False)  
axs[1].legend(loc='upper left', frameon=False)  
axs[2].legend(loc='upper left', frameon=False)  
plt.show()



### 2.6.1.4 前向（正向）传播（forward propagation）与后向（反向）传播（back propagation）[2]

构建一个典型的三层神经网络，包括输入层（input layer）包含两个神经元（neuron）i1​,i2​，和一个偏置（偏差/截距，bias）项  
b1；隐含层（hidden layter）包含两个神经元h1​,h2​，和一个偏置  
b2​；及输出层（output layer）o1​,o2​。假设数据集仅包含一个特征向量（feature）并含两个值，即输入数据：i1​=0.05,i2​=0.10；对应类标（label），即输出数据：o1​=0.01,o2​=0.99；同时随机初始化权重值：w1​=0.15,w2​=0.20,w3​=0.25,w4​=0.30,w5​=0.40,w6​=0.45,w7​=0.50,w8​=0.55；偏置值为b1​=0.35,b2​=0.60

通过神经网络，输入值经过与权重及偏置值的计算，使得结果与类标接近，从而最终确定权重值。



**Step-0：初始化**

初始化特征值，类标，及权重值和偏置

i\_1,i\_2=0.05,0.10  
w\_1,w\_2,w\_3,w\_4,w\_5,w\_6,w\_7,w\_8=0.15,0.20,0.25,0.30,0.40,0.45,0.50,0.55  
b\_1,b\_2=0.35,0.60  
o\_1,o\_2=0.01,0.99

**Step-1：前向传播** \* 1. 输入层—>隐含层

计算神经元*h*1的输入加权和

net\_h\_1=w\_1\*i\_1+w\_2\*i\_2+b\_1\*1  
print("net\_h\_1={}".format(net\_h\_1))

net\_h\_1=0.3775

对h1应用激活函数-sigmoid

def sigmoid(x):   
 '''  
 function - sigmoid函数  
 '''  
 import math  
  
 return 1/(1+math.exp(-x))  
out\_h\_1=sigmoid(net\_h\_1)  
print("out\_h\_1={}".format(out\_h\_1))

out\_h\_1=0.5932699921071872

同理，计算神经元h2​的输入加权和，并应用sigmoid函数

net\_h\_2=w\_3\*i\_1+w\_4\*i\_2+b\_1\*1  
out\_h\_2=sigmoid(net\_h\_2)  
print("out\_h\_2={}".format(out\_h\_2))

out\_h\_2=0.596884378259767

* 1. 隐含层—>输出层

计算神经元o1​,o2的值

net\_o\_1=w\_5\*out\_h\_1+w\_6\*out\_h\_2+b\_2\*1  
out\_o\_1=sigmoid(net\_o\_1)  
print("out\_o\_1={}".format(out\_o\_1))  
net\_o\_2=w\_7\*out\_h\_1+w\_8\*out\_h\_2+b\_2\*1  
out\_o\_2=sigmoid(net\_o\_2)  
print("out\_o\_2={}".format(out\_o\_2))

out\_o\_1=0.7513650695523157  
out\_o\_2=0.7729284653214625

随机初始化权重值，逐层计算输入加权和（Summation and Bias）：，并应用激活函数（sigmoid），获得输出值out\_o\_1和out\_o\_2，与实际值o1​=0.01,o2​=0.99相差还很远，现在对误差进行反向传播，更新权重/权值，重新计算输出。

**Step-2：反向传播**

* 1. 计算总误差（残差平方误差，Residual square error）

分别计算各个输出（此时有2个输出）的误差，再求和。

E\_o\_1=1/2\*(out\_o\_1-o\_1)\*\*2  
E\_o\_2=1/2\*(out\_o\_2-o\_2)\*\*2  
E\_total=E\_o\_1+E\_o\_2  
print("E\_total={}".format(E\_total))

E\_total=0.2983711087600027

* 1. 隐含层—>输出层的权值更新

以w\_5权重值参数为例，计算w\_5对整体误差的影响，用整体误差对w\_5求偏导，应用微分链式法则，其公式为：，分别是对总误差计算公式、激活函数和加权和函数求导。只是各个输入值分别对应w\_5计算路径下的各个对应值，可以很容易的从以上神经网络结构图中确定，即函数的输入端值。总误差计算公式导函数对应outo1​即out\_o\_1，激活函数导函数对应neto1​即net\_o\_1，加权和函数导函数对应outh1​即out\_h\_1。

import sympy  
from sympy import diff,pprint  
  
# w\_5-链式法则-01-总误差函数（公式）对out\_o\_1的偏导数  
out\_o\_1\_,o\_1\_,out\_o\_2\_,o\_2\_=sympy.symbols('out\_o\_1\_ o\_1\_,out\_o\_2\_ o\_2\_')  
d\_E\_total\_2\_out\_o\_1=diff(1/2\*(out\_o\_1\_-o\_1\_)\*\*2+1/2\*(out\_o\_2\_-o\_2\_)\*\*2,out\_o\_1\_)  
d\_E\_total\_2\_out\_o\_1\_value=d\_E\_total\_2\_out\_o\_1.subs([(o\_1\_,o\_1),(out\_o\_1\_,out\_o\_1)])  
print("d\_E\_total\_2\_out\_o\_1\_value={}".format(d\_E\_total\_2\_out\_o\_1\_value))  
  
# w\_5-链式法则-02-激活函数对net\_o\_1的偏导数  
from sympy.functions import exp  
x\_sigmoid=sympy.symbols('x\_sigmoid')  
d\_activation=diff(1/(1+exp(-x\_sigmoid)),x\_sigmoid)  
d\_out\_o\_1\_2\_net\_o\_1\_value=d\_activation.subs([(x\_sigmoid,net\_o\_1)])  
print("d\_out\_o\_1\_2\_activation\_value={}".format(d\_out\_o\_1\_2\_net\_o\_1\_value))  
  
# w\_5-链式法则-03-隐藏层h1，加权和函数（公式）对out\_h\_1的偏导数  
w\_5\_,out\_h\_1\_,w\_6\_,out\_h\_2\_,b\_2\_=sympy.symbols('w\_5,out\_h\_1,w\_6,out\_h\_2,b\_2')  
d\_net\_o\_1\_2\_w\_5=diff(w\_5\_\*out\_h\_1\_+w\_6\_\*out\_h\_2\_+b\_2\_\*1,w\_5\_)  
d\_net\_o\_1\_2\_w\_5\_value=d\_net\_o\_1\_2\_w\_5.subs([(out\_h\_1\_,out\_h\_1)])  
print("d\_net\_o\_1\_2\_w\_5\_value={}".format(d\_net\_o\_1\_2\_w\_5\_value))  
  
# w\_5-链式法则-各部分相乘  
d\_E\_total\_2\_w\_5=d\_E\_total\_2\_out\_o\_1\_value\*d\_out\_o\_1\_2\_net\_o\_1\_value\*d\_net\_o\_1\_2\_w\_5\_value  
print("d\_E\_total\_2\_w\_5={}".format(d\_E\_total\_2\_w\_5))  
  
# w\_5-权重值更新  
lr=0.5 # 学习速率  
w\_5\_update=w\_5-lr\*d\_E\_total\_2\_w\_5  
print("w\_5\_update={}".format(w\_5\_update))

d\_E\_total\_2\_out\_o\_1\_value=0.741365069552316  
d\_out\_o\_1\_2\_activation\_value=0.186815601808960  
d\_net\_o\_1\_2\_w\_5\_value=0.593269992107187  
d\_E\_total\_2\_w\_5=0.0821670405642308  
w\_5\_update=0.358916479717885

所有权值的偏导求法基本相同，为了减少重复的代码，可以定义公用的导函数，只是不同的权值根据其在神经网络结构下的路径，调整链式法则中的各个偏导函数，并对应替换值。

# 定义总误差偏导  
def partialD\_E\_total\_prediction(true\_value\_,predicted\_value\_):  
 '''  
 function - 定义总误差偏导  
 '''  
 import sympy  
 from sympy import diff  
  
 true\_value,predicted\_value=sympy.symbols('true\_value predicted\_value')  
 partialD\_E\_total\_prediction=diff(1/2\*(predicted\_value-true\_value)\*\*2,predicted\_value)  
 return partialD\_E\_total\_prediction.subs([(predicted\_value,predicted\_value\_),(true\_value,true\_value\_)])  
  
# 定义激活函数偏导  
def partialD\_activation(x):  
 '''  
 function -定义激活函数偏导  
 '''   
 import sympy  
 from sympy import diff  
 from sympy.functions import exp  
  
 x\_sigmoid=sympy.symbols('x\_sigmoid')  
 partialD\_activation=diff(1/(1+exp(-x\_sigmoid)),x\_sigmoid)  
 return partialD\_activation.subs([(x\_sigmoid,x)])  
  
# 定义加权和偏导  
def partialD\_weightedSUM(w\_):  
 '''  
 fucntion - 定义加权和偏导  
 '''   
 import sympy  
 from sympy import diff  
  
 w,x\_w=sympy.symbols('w x\_w')  
 partialD\_weightedSUM=diff(w\*x\_w,w)  
 return partialD\_weightedSUM.subs([(x\_w,w\_)])

w\_5\_update=w\_5-lr\*partialD\_E\_total\_prediction(o\_1,out\_o\_1)\*partialD\_activation(net\_o\_1)\*partialD\_weightedSUM(out\_h\_1)  
w\_6\_update=w\_6-lr\*partialD\_E\_total\_prediction(o\_1,out\_o\_1)\*partialD\_activation(net\_o\_1)\*partialD\_weightedSUM(out\_h\_2)  
w\_7\_update=w\_7-lr\*partialD\_E\_total\_prediction(o\_2,out\_o\_2)\*partialD\_activation(net\_o\_2)\*partialD\_weightedSUM(out\_h\_1)  
w\_8\_update=w\_8-lr\*partialD\_E\_total\_prediction(o\_2,out\_o\_2)\*partialD\_activation(net\_o\_2)\*partialD\_weightedSUM(out\_h\_2)  
  
print("w\_5\_update={}\nw\_6\_update={}\nw\_7\_update={}\nw\_8\_update={}".format(w\_5\_update,w\_6\_update,w\_7\_update,w\_8\_update))

w\_5\_update=0.358916479717885  
w\_6\_update=0.408666186076233  
w\_7\_update=0.511301270238738  
w\_8\_update=0.561370121107989

* 1. 输入层—>隐含层的权值更新

因为隐含层outh1​会受到Eo1​,Eo2​两个地方传来的误差，因此分别计算并求和，再与其它路径下的导数求积。outh2与之同。

w\_1\_update=w\_1-lr\*(partialD\_E\_total\_prediction(o\_1,out\_o\_1)\*partialD\_activation(net\_o\_1)\*partialD\_weightedSUM(w\_5)+partialD\_E\_total\_prediction(o\_2,out\_o\_2)\*partialD\_activation(net\_o\_2)\*partialD\_weightedSUM(w\_7))\*partialD\_activation(net\_h\_1)\*partialD\_weightedSUM(i\_1)  
w\_2\_update=w\_2-lr\*(partialD\_E\_total\_prediction(o\_1,out\_o\_1)\*partialD\_activation(net\_o\_1)\*partialD\_weightedSUM(w\_5)+partialD\_E\_total\_prediction(o\_2,out\_o\_2)\*partialD\_activation(net\_o\_2)\*partialD\_weightedSUM(w\_7))\*partialD\_activation(net\_h\_1)\*partialD\_weightedSUM(i\_2)  
w\_3\_update=w\_3-lr\*(partialD\_E\_total\_prediction(o\_1,out\_o\_1)\*partialD\_activation(net\_o\_1)\*partialD\_weightedSUM(w\_6)+partialD\_E\_total\_prediction(o\_2,out\_o\_2)\*partialD\_activation(net\_o\_2)\*partialD\_weightedSUM(w\_8))\*partialD\_activation(net\_h\_2)\*partialD\_weightedSUM(i\_1)  
w\_4\_update=w\_4-lr\*(partialD\_E\_total\_prediction(o\_1,out\_o\_1)\*partialD\_activation(net\_o\_1)\*partialD\_weightedSUM(w\_6)+partialD\_E\_total\_prediction(o\_2,out\_o\_2)\*partialD\_activation(net\_o\_2)\*partialD\_weightedSUM(w\_8))\*partialD\_activation(net\_h\_2)\*partialD\_weightedSUM(i\_2)  
  
print("w\_1\_update={}\nw\_2\_update={}\nw\_3\_update={}\nw\_4\_update={}".format(w\_1\_update,w\_2\_update,w\_3\_update,w\_4\_update))

w\_1\_update=0.149780716132763  
w\_2\_update=0.199561432265526  
w\_3\_update=0.249751143632370  
w\_4\_update=0.299502287264739

至此误差的反向传播全部计算完，并更新权重值，重复前向传播。为了方便计算将前向传播定义为一个函数，计算结果显示总误差为0.29102777369359933，较之前次0.2983711087600027有所下降。

def forward\_func(input\_list,weight\_list,bias\_list,output\_list):  
 '''  
 function - 三层神经网络，各层2个神经元的示例，前向传播函数  
 '''  
 def sigmoid(x):  
 '''  
 function - sigmoid函数  
 '''   
 import math  
  
 return 1/(1+math.exp(-x))  
 i\_1,i\_2=input\_list  
 w\_1,w\_2,w\_3,w\_4,w\_5,w\_6,w\_7,w\_8=weight\_list  
 b\_1,b\_2,b\_3,b\_4=bias\_list  
 o\_1,o\_2=output\_list  
   
 net\_h\_1=w\_1\*i\_1+w\_2\*i\_2+b\_1\*1  
 out\_h\_1=sigmoid(net\_h\_1)  
   
 net\_h\_2=w\_3\*i\_1+w\_4\*i\_2+b\_2\*1  
 out\_h\_2=sigmoid(net\_h\_2)   
   
 net\_o\_1=w\_5\*out\_h\_1+w\_6\*out\_h\_2+b\_3\*1  
 out\_o\_1=sigmoid(net\_o\_1)  
  
 net\_o\_2=w\_7\*out\_h\_1+w\_8\*out\_h\_2+b\_4\*1  
 out\_o\_2=sigmoid(net\_o\_2)  
  
 E\_total=1/2\*(out\_o\_1-o\_1)\*\*2+1/2\*(out\_o\_2-o\_2)\*\*2  
 print("out\_o\_1={}\nout\_o\_2={}\nE\_total={}".format(out\_o\_1,out\_o\_2,E\_total))  
 return out\_o\_1,out\_o\_2,E\_total  
out\_o\_1,out\_o\_2,E\_total=forward\_func(input\_list=[0.05,0.10],weight\_list=[w\_1\_update,w\_2\_update,w\_3\_update,w\_4\_update,w\_5\_update,w\_6\_update,w\_7\_update,w\_8\_update],bias\_list=[0.35,0.35,0.60,0.60],output\_list=[0.01,0.99])

out\_o\_1=0.7420881111907824  
out\_o\_2=0.7752849682944595  
E\_total=0.29102777369359933

**多层感知机-代码整合**

将上述的分解过程整合，实现迭代计算求取权重值。代码的结构包括三个类，NEURON类定义神经元前向传播和反向传播的函数；NEURAL\_LAYER类是基于NEURON的层级神经元汇总；NEURAL\_NETWORK类构建神经网络结构，初始化权值并更新权值。此处的输入值保持不变为[0.05,0.1]，输出值修改为[0.01,0.09]，能够更好的观察收敛的过程。如果仍然保持输出值为[0.01,0.99]，因为偏置保持不变，造成损失函数下降的趋势不明显。

class NEURON:  
 '''  
 class - 每一神经元的输入加权和、激活函数，及输出计算  
 '''  
 def \_\_init\_\_(self,bias):  
 '''  
 function - 初始化权重和偏置  
 '''  
 self.bias=bias  
 self.weights=[]  
   
 def activation(self,x\_sigmoid):   
 '''  
 function - 定义sigmoid激活函数  
 '''   
 import math  
  
 return 1/(1+math.exp(-x\_sigmoid))  
   
 def net\_input(self):   
 '''  
 function - 每一神经元的输入加权和，inputs，及weights数组形状为(-1,1)  
 '''  
 import numpy as np  
  
 y=np.array(self.inputs).reshape(-1,1)\*np.array(self.weights).reshape(-1,1)  
 return y.sum()+self.bias  
   
 def net\_output(self,inputs):  
 '''  
 function - 每一神经元，先计算输入加权和，然后应用激活函数获得输出  
 '''  
 self.inputs=inputs  
 self.output=self.activation(self.net\_input())  
 return self.output  
   
 def neuron\_error(self,predicted\_output):  
 '''  
 function - 每一神经元的误差（平方差）  
 '''  
 return 0.5\*(predicted\_output-self.output)\*\*2  
   
 def pd\_activation(self):  
 '''  
 function -定义神经元激活函数偏导  
 '''  
 import sympy  
 from sympy import diff  
 from sympy.functions import exp  
  
 x\_sigmoid=sympy.symbols('x\_sigmoid')  
 partialD\_activation=diff(1/(1+exp(-x\_sigmoid)),x\_sigmoid)  
 return partialD\_activation.subs([(x\_sigmoid,self.output)]) # 由sympy的diff方法获取sigmoid导函数  
 # return self.output \* (1 - self.output) # 推导公式  
   
 def pd\_E\_prediction(self,predicted\_output):  
 '''  
 function - 定义神经元误差偏导  
 '''   
 import sympy  
 from sympy import diff  
  
 true\_value,predicted\_value=sympy.symbols('true\_value predicted\_value')  
 partialD\_E\_total\_prediction=diff(0.5\*(predicted\_value-true\_value)\*\*2,true\_value)  
 return partialD\_E\_total\_prediction.subs([(predicted\_value,predicted\_output),(true\_value,self.output)])  
  
   
 def pd\_net\_output(self,predicted\_output):  
 '''  
 function - 每一神经元误差偏导，由链式法则计算，包括神经元激活函数偏导，和神经元总误差偏导  
 '''  
 return self.pd\_activation()\*self.pd\_E\_prediction(predicted\_output)  
   
 def pd\_weightedSUM(self,index):   
 '''  
 fucntion - 定义神经元加权和偏导  
 '''  
 import sympy  
 from sympy import diff  
   
 w,x\_w=sympy.symbols('w x\_w')  
 partialD\_weightedSUM=diff(w\*x\_w,w)  
 return partialD\_weightedSUM.subs([(x\_w,self.inputs[index])])  
   
class NEURAL\_LAYER:  
 '''  
 class - 神经网络每一层的定义  
 '''  
 def \_\_init\_\_(self,num\_neurons,bias):  
 '''  
 fucntion - 初始化每层偏置，及神经元  
 '''  
 self.bias=bias if bias else random.random() # 同一层各个神经元共享同一个偏置   
 self.neurons=[NEURON(self.bias) for i in range(num\_neurons)]  
   
 def layer\_info(self):  
 '''  
 function - 查看每层的神经元信息  
 '''  
 print('neurons:{}'.format(len(self.neurons)))  
 for i in range(len(self.neurons)):  
 print('neuron-{}'.format(i))  
 print('weights:{}'.format([self.neurons[i].weights[j] for j in range(len(self.neurons[i].weights))]))  
 print('bias:{}'.format(self.bias))  
   
 def layer\_output(self,inputs):  
 outputs=[neuron.net\_output(inputs) for neuron in self.neurons]  
 return outputs  
   
class NEURAL\_NETWORK:  
 '''  
 class - 构建多层感知机（神经网络）  
 '''  
 def \_\_init\_\_(self,learning\_rate,num\_inputs,num\_hidden,num\_outputs,hiddenLayer\_weights=None,hiddenLayer\_bias=None,outputLayer\_weights=None,outputLayer\_bias=None):  
 self.lr=learning\_rate  
 self.num\_inputs=num\_inputs  
 self.hidden\_layer=NEURAL\_LAYER(num\_hidden,hiddenLayer\_bias)  
 self.output\_layer=NEURAL\_LAYER(num\_inputs,outputLayer\_bias)  
 self.weights\_init(hiddenLayer\_weights,self.hidden\_layer,self.num\_inputs)  
 self.weights\_init(outputLayer\_weights,self.output\_layer,len(self.hidden\_layer.neurons))   
   
 def weights\_init(self,weights,neu\_layer,num\_previous):  
 '''  
 fucntion - 初始化权值  
 '''  
 weights\_num=0  
 for i in range(len(neu\_layer.neurons)):  
 for j in range(num\_previous):  
 if not weights:  
 neu\_layer.neurons[i].weights.append(random.random())  
 else:  
 neu\_layer.neurons[i].weights.append(weights[weights\_num])  
 weights\_num+=1  
   
 def neural\_info(self):  
 '''  
 function - 查看神经网络结构  
 '''  
 print("\_"\*50)  
 print("inputs number:{}".format(self.num\_inputs))  
 print("\_"\*50)  
 print("hidden layer:{}".format(self.hidden\_layer.layer\_info()))  
 print("\_"\*50)  
 print("output layer:{}".format(self.output\_layer.layer\_info()))  
 print("\_"\*50)  
   
 def forward\_propagation(self,inputs):  
 '''  
 fucntion - 前向传播  
 '''  
 hiddenLayer\_outputs=self.hidden\_layer.layer\_output(inputs)  
 return self.output\_layer.layer\_output(hiddenLayer\_outputs)  
   
 def train(self,training\_inputs,training\_outputs):  
 '''  
 function - 训练，迭代更新权值  
 '''  
 self.forward\_propagation(training\_inputs)   
   
 # 1. 输出层神经元的值  
 pd\_output\_values=[0]\*len(self.output\_layer.neurons)  
 for i in range(len(self.output\_layer.neurons)):  
 # ∂E/∂zⱼ  
 pd\_output\_values[i]=self.output\_layer.neurons[i].pd\_net\_output(training\_outputs[i])  
 # print(pd\_output\_values)  
   
   
 # 2. 隐含层神经元的值  
 hidden\_values=[0]\*len(self.hidden\_layer.neurons)  
 for i in range(len(self.hidden\_layer.neurons)):  
 # dE/dyⱼ = Σ ∂E/∂zⱼ \* ∂z/∂yⱼ = Σ ∂E/∂zⱼ \* wᵢⱼ  
 d\_errors=0  
 for j in range(len(self.output\_layer.neurons)):  
 d\_errors+=pd\_output\_values[j]\*self.output\_layer.neurons[j].weights[i]  
 # ∂E/∂zⱼ = dE/dyⱼ \* ∂zⱼ/∂  
 hidden\_values[i]=d\_errors\*self.hidden\_layer.neurons[i].pd\_activation()  
 # print(hidden\_values)  
   
 # 3. 更新输出层权重系数  
 for i in range(len(self.output\_layer.neurons)):  
 for j in range(len(self.output\_layer.neurons[i].weights)):  
 # ∂Eⱼ/∂wᵢⱼ = ∂E/∂zⱼ \* ∂zⱼ/∂wᵢⱼ  
 weights\_update=pd\_output\_values[i]\*self.output\_layer.neurons[i].pd\_weightedSUM(j)  
 # Δw = α \* ∂Eⱼ/∂wᵢ  
 self.output\_layer.neurons[i].weights[j]-=self.lr\*weights\_update  
   
 # 4. 更新隐含层的权重系数  
 for i in range(len(self.hidden\_layer.neurons)):  
 for j in range(len(self.hidden\_layer.neurons[i].weights)):  
 # ∂Eⱼ/∂wᵢ = ∂E/∂zⱼ \* ∂zⱼ/∂wᵢ  
 weights\_update=hidden\_values[i]\*self.hidden\_layer.neurons[i].pd\_weightedSUM(j)  
 # Δw = α \* ∂Eⱼ/∂wᵢ  
 self.hidden\_layer.neurons[i].weights[j]-=self.lr\*weights\_update  
   
 def total\_error(self,training\_sets):  
 total\_error=0  
 for i in range(len(training\_sets)):   
 training\_inputs, training\_outputs=training\_sets[i]  
 self.forward\_propagation(training\_inputs)  
 for j in range(len(training\_outputs)):  
 total\_error+=self.output\_layer.neurons[j].neuron\_error(training\_outputs[j])  
 return total\_error  
  
   
learning\_rate=0.5   
nn=NEURAL\_NETWORK(learning\_rate,2, 2, 2, hiddenLayer\_weights=[0.15, 0.2, 0.25, 0.3], hiddenLayer\_bias=0.35, outputLayer\_weights=[0.4, 0.45, 0.5, 0.55], outputLayer\_bias=0.6)   
nn.neural\_info()  
  
import numpy as np  
from tqdm import tqdm  
for i in tqdm(range(10000)):  
 nn.train([0.05,0.1],[0.01,0.09])  
 if i%1000==0:  
 print(i,round(nn.total\_error([[[0.05, 0.1], [0.01, 0.09]]]),9))  
hidenLayer\_weights\_=[nn.hidden\_layer.neurons[i].weights for i in range(len(nn.hidden\_layer.neurons))]  
output\_layer\_weights\_=[nn.output\_layer.neurons[i].weights for i in range(len(nn.output\_layer.neurons))]  
print("hidenLayer\_weights\_={}\noutput\_layer\_weights\_={}".format(hidenLayer\_weights\_,output\_layer\_weights\_))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
inputs number:2  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
neurons:2  
neuron-0  
weights:[0.15, 0.2]  
bias:0.35  
neuron-1  
weights:[0.25, 0.3]  
bias:0.35  
hidden layer:None  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
neurons:2  
neuron-0  
weights:[0.4, 0.45]  
bias:0.6  
neuron-1  
weights:[0.5, 0.55]  
bias:0.6  
output layer:None  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
  
 0%| | 14/10000 [00:00<02:28, 67.03it/s]  
  
0 0.493712428  
  
  
 10%|█ | 1015/10000 [00:14<01:57, 76.52it/s]  
  
1000 2.4963e-05  
  
  
 20%|██ | 2016/10000 [00:28<01:48, 73.30it/s]  
  
2000 1.878e-06  
  
  
 30%|███ | 3011/10000 [00:42<01:39, 70.08it/s]  
  
3000 2.28e-07  
  
  
 40%|████ | 4011/10000 [00:55<01:22, 72.39it/s]  
  
4000 3.2e-08  
  
  
 50%|█████ | 5009/10000 [01:08<01:02, 80.29it/s]  
  
5000 5e-09  
  
  
 60%|██████ | 6008/10000 [01:22<00:57, 69.85it/s]  
  
6000 1e-09  
  
  
 70%|███████ | 7011/10000 [01:35<00:40, 73.87it/s]  
  
7000 0.0  
  
  
 80%|████████ | 8014/10000 [01:48<00:25, 78.02it/s]  
  
8000 0.0  
  
  
 90%|█████████ | 9011/10000 [02:01<00:13, 71.49it/s]  
  
9000 0.0  
  
  
100%|██████████| 10000/10000 [02:14<00:00, 74.62it/s]  
  
hidenLayer\_weights\_=[[0.374907950905178, 0.649815901810356], [0.469098467292651, 0.738196934585302]]  
output\_layer\_weights\_=[[-4.28132748279580, -4.25790301529841], [-2.41120108789009, -2.37807889000506]]

​   
​

### 2.6.1.5 自动求梯度（gradient）

在上述前向和反向传播中，反向传播的梯度计算是应用SymPy的diff方法求得导函数计算。因为在深度学习中经常要对函数求梯度，因此PyTorch提供了autograd，可以根据输入和前向传播过程自动构建计算图，并执行反向传播。在定义张量时，如果配置参数requires\_grad=True，将追踪其上的所有操作，因此可以应用链式法则进行梯度传播计算。完成前向传播计算后，可以调用.backward()来完成梯度计算。梯度将累积到grad属性中。如果不想被追踪，可以执行.detach()将其从追踪记录中剥离开来，或者使用with torch.no\_grad()将不想追踪的操作代码包裹起来（例如在评估模型时）。

注意：grad在反向传播过程中是累加的（accumulated），一般在反向传播之前把应用x.grad.data.zero\_()梯度清零。

梯度，gradient，一种关于多元导数的概况。一元（单变量）函数的导数是标量值函数，而多元函数的梯度是向量值函数。多元可微函数在点上的梯度，是以在上的偏导数为分量的向量。一元函数的导数表示这个函数图形切线的斜率，如果多元函数在点上的梯度不是零向量，则它的方向是这个函数在上最大增长的方向，它的量是在这个方向上的增长率。

import torch  
# 01-定义包含梯度追踪的张量  
x = torch.tensor([3.0, 2.0, 1.0, 4.0], requires\_grad=True) # x.requires\_grad\_(True)的方式可以将未指定requires\_grad = False的张量转换为追踪梯度传播  
print("01-定义包含梯度追踪的张量:")  
print("x={}\nx.grad\_fn={}\nx.requires\_grad={}".format(x,x.grad\_fn,x.requires\_grad))  
print("\_"\*50)  
  
# 02-追踪运算的梯度  
print("02-追踪运算的梯度:")  
y=(x+1)\*\*3+2  
z=y.view(2,2)  
print("z={}\nz.grad\_fn={}".format(z,z.grad\_fn))  
print("x.is\_leaf={},y.is\_leaf={},z.is\_leaf={}".format(x.is\_leaf,y.is\_leaf,z.is\_leaf))  
print("\_"\*50)  
  
# 03-z(张量)关于x的梯度  
print("03-z(张量)关于x的梯度:")  
v=torch.tensor([[1.0, 0.1], [0.01, 0.001]],dtype=torch.float)  
z.backward(v) # 因为z不是标量，在调用backward时需要传入一个和z同形状的权值向量（权值）进行加权求和得到一个标量  
print("dz/dx={}".format(x.grad)) # x.grad是和x同形的张量  
print("\_"\*50)  
  
# 04-中断梯度追踪  
print("04-中断梯度追踪:")  
x\_1=torch.tensor(1.0,requires\_grad=True)  
y\_1=x\_1\*\*2  
with torch.no\_grad():  
 y\_2=x\_1\*\*3  
y\_3=y\_1+y\_2  
print("x.requires\_grad={}\ny\_1={};y\_1.requires\_grad={}\ny\_2={};y\_2.requires\_grad={}\ny\_3={};y\_3.requires\_grad={}\n".format(x.requires\_grad,y\_1,y\_1.requires\_grad,y\_2,y\_2.requires\_grad,y\_3,y\_3.requires\_grad))  
y\_3.backward()  
print("dy\_3/dx={}".format(x\_1.grad)) # y\_2梯度并未被追踪，与y\_2有关的梯度并不会回传  
print("\_"\*50)  
  
# 05-不影响反向传播下修改张量值，tensor.data  
print("05-不影响反向传播下修改张量值，tensor.data:")  
x\_2=torch.ones(1,requires\_grad=True)  
print("x\_2.data={}\nx\_2.data.requires\_grad={}".format(x\_2,x\_2.data.requires\_grad))  
y\_4=2\*x\_2  
x\_2.data\*=100 # 只改变了值，不会记录在计算图中（autograd记录），不影响梯度传播  
y\_4.backward()  
print("x\_2={}\nx\_2.grad={}".format(x\_2,x\_2.grad))

01-定义包含梯度追踪的张量:  
x=tensor([3., 2., 1., 4.], requires\_grad=True)  
x.grad\_fn=None  
x.requires\_grad=True  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
02-追踪运算的梯度:  
z=tensor([[ 66., 29.],  
 [ 10., 127.]], grad\_fn=<ViewBackward0>)  
z.grad\_fn=<ViewBackward0 object at 0x0000024EE6A4B9D0>  
x.is\_leaf=True,y.is\_leaf=False,z.is\_leaf=False  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
03-z(张量)关于x的梯度:  
dz/dx=tensor([48.0000, 2.7000, 0.1200, 0.0750])  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
04-中断梯度追踪:  
x.requires\_grad=True  
y\_1=1.0;y\_1.requires\_grad=True  
y\_2=1.0;y\_2.requires\_grad=False  
y\_3=2.0;y\_3.requires\_grad=True  
  
dy\_3/dx=2.0  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
05-不影响反向传播下修改张量值，tensor.data:  
x\_2.data=tensor([1.], requires\_grad=True)  
x\_2.data.requires\_grad=False  
x\_2=tensor([100.], requires\_grad=True)  
x\_2.grad=tensor([2.])

### 2.6.1.6 用PyTorch构建多层感知机（多层神经网络）

#### 1）自定义激活函数、模型、损失函数及梯度下降法

PyTorch自动求梯度的方式，让神经网络模型的构建变得轻松，可以将更多的注意力放在模型的构建上，而不是梯度的计算上。上述实现了典型三层神经网络的逐步计算、及代码整合，对应输入输出值，权值和偏置、激活函数、损失函数、模型结构及权值更新（优化算法：梯度下降法）保持不变，应用PyTorch分别重新定义。从迭代计算结果来看，因为偏置值也得以更新，损失函数下降的比较快，迅速的收敛。

import torch  
import numpy as np  
  
# A-训练数据  
X=torch.tensor([0.05,0.1])  
y=torch.tensor([0.01,0.9])  
  
# B-定义模型参数  
num\_inputs,num\_outputs,num\_hiddens=2,2,2  
  
W1=torch.tensor([[0.15, 0.2], [0.25, 0.3]]) #hiddenLayer\_weights  
b1=torch.tensor([0.35,0.35]) #hiddenLayer\_bias  
W2=torch.tensor([[0.4, 0.45], [0.5, 0.55]]) #outputLayer\_weights  
b2=torch.tensor([0.6,0.6]) #outputLayer\_bias  
params=[W1, b1, W2, b2]  
for param in params:  
 param.requires\_grad\_(requires\_grad=True)  
  
# C-定义激活函数  
def sigmoid(X):  
 return 1/(1+torch.exp(-X))  
  
# D-定义模型  
def net(X):  
 X=X.view((-1,num\_inputs))  
 H=sigmoid(torch.matmul(X,W1)+b1)  
 return torch.matmul(H,W2)+b2  
  
# F-定义损失函数  
def loss(y\_hat,y):  
 return 0.5\*(y\_hat-y)\*\*2  
  
# G-优化算法  
def sgd(params,lr):  
 for param in params:  
 param.data-=lr\*param.grad  
  
# H-训练模型  
def train(net,X,y,loss,num\_epochs,params=None,lr=None,optimizer=None):  
 train\_l\_sum, train\_acc\_sum, n = 0.0, 0.0, 0  
 for epoch in range(num\_epochs):  
 y\_hat=net(X)  
 l=loss(y\_hat,y).sum()  
 # 梯度清零  
 if params is not None and params[0].grad is not None:  
 for param in params:  
 param.grad.data.zero\_()   
 l.backward()  
 sgd(params,lr)  
 print('epoch %d, loss %.9f'%(epoch+1,l))  
 return net,params  
   
num\_epochs, lr=5, 0.5  
net\_,params\_=train(net,X,y,loss,num\_epochs,params,lr)  
print("应用训练好的模型验证预测值：\nnet\_(x)={}".format(net\_(X)))  
print("参数(权值+偏置)更新结果：\nparams\_={}".format(params\_))

epoch 1, loss 0.677512288  
epoch 2, loss 0.013031594  
epoch 3, loss 0.000361601  
epoch 4, loss 0.000010227  
epoch 5, loss 0.000000290  
应用训练好的模型验证预测值：  
net\_(x)=tensor([[0.0101, 0.9000]], grad\_fn=<AddBackward0>)  
参数(权值+偏置)更新结果：  
params\_=[tensor([[0.1463, 0.1954],  
 [0.2427, 0.2907]], requires\_grad=True), tensor([0.2770, 0.2573], requires\_grad=True), tensor([[0.0102, 0.3532],  
 [0.1094, 0.4529]], requires\_grad=True), tensor([-0.0585, 0.4366], requires\_grad=True)]

#### 2）直接使用PyTorch提供的函数

PyTorch已经内置了多种激活函数、损失函数、优化算法及各类模型算法，可以直接调用参与计算。

import torch  
from torch import nn  
  
# A-训练数据  
X=torch.tensor([0.05,0.1])  
y=torch.tensor([0.01,0.9])  
  
# B-定义模型参数:权值和偏置此次随机初始化  
num\_inputs,num\_outputs,num\_hiddens=2,2,2  
   
# C-定义模型  
net=nn.Sequential(  
 nn.Linear(num\_inputs,num\_hiddens),  
 nn.Sigmoid()  
 )  
  
# D-初始化参数：可以省略，pytorch定义模型时，已经初始化  
# for params in net.parameters():  
 # nn.init.normal\_(params,mean=0,std=0.01)  
  
# E-定义损失函数-均方误差损失  
loss = nn.MSELoss()  
  
# F-优化算法  
optimizer=torch.optim.SGD(net.parameters(),lr=0.5)  
  
# G-训练模型  
def train(net,X,y,loss,num\_epochs,params=None,lr=None,optimizer=None):  
 train\_l\_sum, train\_acc\_sum, n = 0.0, 0.0, 0  
 for epoch in range(num\_epochs):  
 y\_hat=net(X)  
 l=loss(y\_hat,y).sum()  
 # 梯度清零  
 if optimizer is not None:  
 optimizer.zero\_grad()  
 elif params is not None and params[0].grad is not None:  
 for param in params:  
 param.grad.data.zero\_()   
 # 自动求梯度  
 l.backward()  
 optimizer.step()  
   
 if epoch%1000==0:  
 print('epoch %d, loss %.9f'%(epoch+1,l))  
 return net,params  
  
  
num\_epochs=10000  
net,params=train(net,X,y,loss,num\_epochs,params,lr,optimizer)  
print("应用训练好的模型验证预测值：\nnet\_(x)={}".format(net\_(X)))  
print("参数(权值+偏置)更新结果：\nparams\_={}".format(params\_))

epoch 1, loss 0.215000793  
epoch 1001, loss 0.000375652  
epoch 2001, loss 0.000146600  
epoch 3001, loss 0.000081315  
epoch 4001, loss 0.000052157  
epoch 5001, loss 0.000036260  
epoch 6001, loss 0.000026540  
epoch 7001, loss 0.000020135  
epoch 8001, loss 0.000015684  
epoch 9001, loss 0.000012468  
应用训练好的模型验证预测值：  
net\_(x)=tensor([[0.0101, 0.9000]], grad\_fn=<AddBackward0>)  
参数(权值+偏置)更新结果：  
params\_=[tensor([[0.1463, 0.1954],  
 [0.2427, 0.2907]], requires\_grad=True), tensor([0.2770, 0.2573], requires\_grad=True), tensor([[0.0102, 0.3532],  
 [0.1094, 0.4529]], requires\_grad=True), tensor([-0.0585, 0.4366], requires\_grad=True)]

### 2.6.1.7 [runx.logx](https://github.com/NVIDIA/runx#introduction--a-simple-example)④-深度学习实验管理（Deep Learning Experiment Management）与模型构建

runx是NVIDIA开源的一款深度学习实验管理工具。可以帮助深度学习研究者自动执行一些常见的任务。例如参数扫描，输出日志记录，保存训练模型等。该库包括3个模块，runx，logx及sumx。其中最为常用的是logx模块，可以应用logx.metric()保存metrics（以字典格式保存的指定评估参数等，例如损失函数值，学习率等）；应用logx.msg()保存message（任意指定的信息）；应用logx.save\_model()保存模型checkpoint，并可以指定一个模型精度评估指标，保存精度最好的模型； 及初始化时配置tensorboard=True，写入tensorboard文件，可以应用tensorboard打开训练时的一些信息图表，例如损失曲线，学习率，输入输出图片(logx.add\_image())，卷积核的参数分布等。这些信息能够帮助监督网络的训练过程，为参数优化提供帮助。logx保存文件位于指定的文件夹下，包括’best\_checkpoint\_ep1000.pth’，’last\_checkpoint\_ep1000.pth‘网络模型，logging.log日志，metrics.csv评估参数，及’events.out.tfevents.1609820463.LAPTOP-GH6EM1TC’文件。

下述代码也涵盖了模型构建的几种方法，包括继承nn.Module类构造模型，使用nn.Sequential类构造模型（又包括.add\_module方式和OrderedDict方式），使用nn-ModuleList类构造模型，及使用nn.ModuleDict类构造模型。其中优先使用nn.Sequential类构造模型方法，如果需要增加模型构造的灵活性，可以使用继承nn.Module类构造模型。

在生成数据集时，是使用了一个二元一次函数根据随机生成的特征值（特征数为2，即输入值）生成对应的类标（即输出值）。但是在神经网络模型构建时，并未使用单纯的一个线性模型，而是构建了线性模型–>激活函数–>线性模型的神经网络结构，用以说明模型构建的方法。如果希望验证一个层线性回归的参数是否与生成数据集类标的方程权值和偏置趋于一致，可以仅保留第一个线性模型，而移除激活函数及第2个线性模型。

将数据集划分了训练数据集和验证数据集，在训练过程中的每一epoch增加了验证数据集根据已训练的网络预测输出值，并应用损失函数-nn.MSELoss()均方误差的累加和用于网络评估。

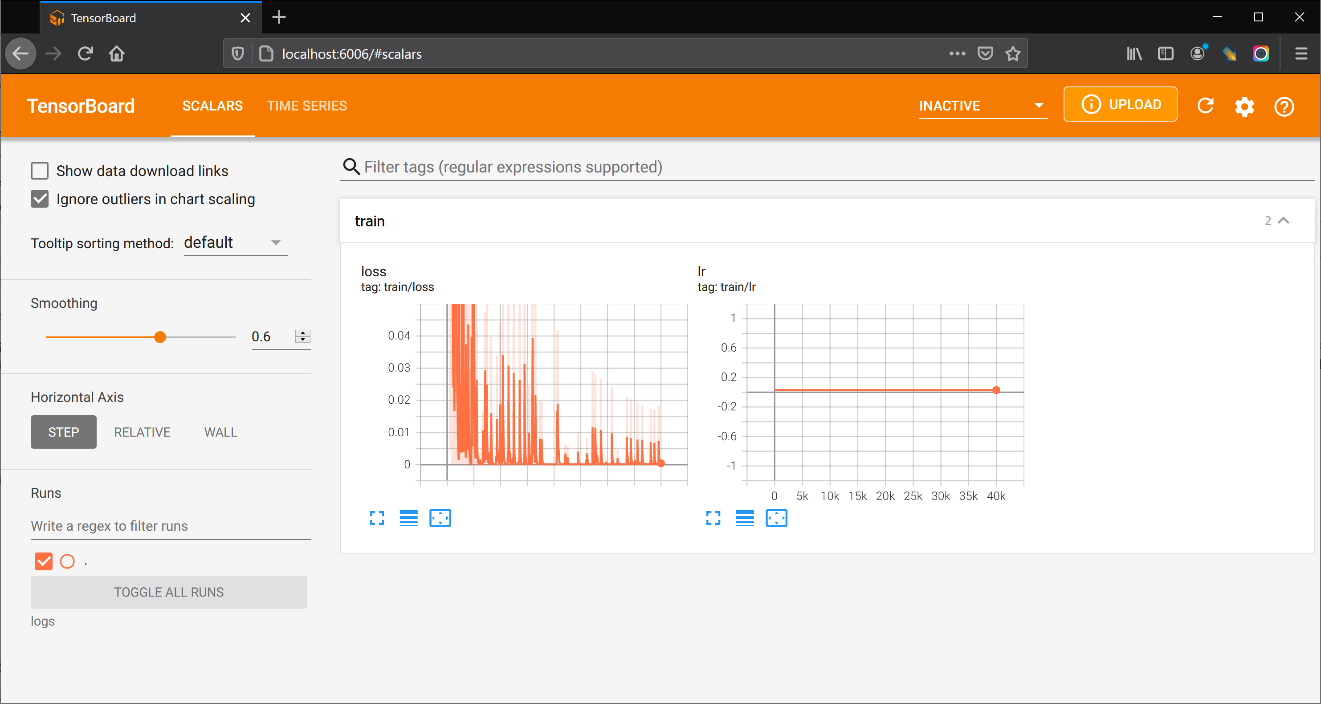
import torch  
from runx.logx import logx  
import torch.utils.data as Data  
from torch import nn  
import numpy as np  
from runx.logx import logx  
  
# 初始化logx  
logx.initialize(logdir="./logs/", # 指定日志文件保持路径（如果不指定，则自动创建）  
 coolname=True, # 是否在logdir下生成一个独有的目录  
 hparams=None, # 配合runx使用，保存超参数  
 tensorboard=True, # 是否自动写入tensorboard文件  
 no\_timestamp=False, # 是否不启用时间戳命名  
 global\_rank=0,   
 eager\_flush=True,   
 )   
  
# A-生成数据集  
num\_inputs=2 # 包含的特征数  
num\_examples=5000  
true\_w=[3,-2.3]  
true\_b=5.9  
features=torch.tensor(np.random.normal(0, 1, (num\_examples, num\_inputs)), dtype=torch.float)  
labels=true\_w[0]\*features[:, 0]+true\_w[1]\*features[:, 1]+true\_b # 由回归方程：y=w1\*x1+-w2\*x2+b，计算输出值  
labels+=torch.tensor(np.random.normal(0, 0.01, size=labels.size()), dtype=torch.float) # 变化输出值  
logx.msg(r"the expression generating labels:w1\*x1+-w2\*x2+b") # 可以将需要查看的对象以文本的形式保存在logging.log文件，此处保存了生成类标的线性方程  
  
# B-读取数据（小批量-随机读取指定数量/batch\_size的样本）  
batch\_size=100  
dataset=Data.TensorDataset(features,labels) # 建立PyTorch格式的数据集（组合输入与输出值）  
train\_size, val\_size=[4000,1000]  
train\_dataset, val\_dataset=torch.utils.data.random\_split(dataset, [train\_size, val\_size])  
train\_data\_iter=Data.DataLoader(train\_dataset,batch\_size,shuffle=True) # 随机读取小批量  
val\_data\_iter=Data.DataLoader(val\_dataset,batch\_size,shuffle=True)  
  
  
# C-定义模型  
# 方法-01-继承nn.Module类构造模型  
class MLP(nn.Module): # 继承nn.Module父类  
 # 声明带有模型参数的层  
 def \_\_init\_\_(self,n\_feature):  
 super(MLP,self).\_\_init\_\_()  
 self.hidden=nn.Linear(n\_feature,2) # 隐藏层,torch.nn.Linear(in\_features: int, out\_features: int, bias: bool = True)，参数指定特征数（输入），及类标数（输出）  
 self.activation=nn.ReLU() # 激活层  
 self.output=nn.Linear(2,1) # 输出层  
   
 # 定义前向传播，根据输入x计算返回所需要的模型输出  
 def forward(self,x):  
 y=self.activation(self.hidden(x))  
 return self.output(y)  
net=MLP(num\_inputs)  
print("nn.Module构造模型：")  
logx.msg("net:{}".format(net)) # 此处保存了神经网络结构  
print("\_"\*50)  
  
# 方法-02-使用nn.Sequential类构造模型。  
net\_sequential=nn.Sequential(  
 nn.Linear(num\_inputs,2),  
 nn.ReLU(),  
 nn.Linear(2,1)  
 )  
print("nn.Sequential构造模型：",net\_sequential)  
  
# nn.Sequential的.add\_module方式  
net\_sequential\_=nn.Sequential()   
net\_sequential\_.add\_module('hidden',nn.Linear(num\_inputs,2))  
net\_sequential\_.add\_module('activation',nn.ReLU())  
net\_sequential\_.add\_module("output",nn.Linear(2,1))  
print("nn.Sequential()-->.add\_module方式：",net\_sequential\_)  
  
# nn.Sequential的OrderedDict方式  
from collections import OrderedDict  
net\_sequential\_orderDict=nn.Sequential(  
 OrderedDict([  
 ('hidden',nn.Linear(num\_inputs,2)),  
 ('activation',nn.ReLU()),  
 ("output",nn.Linear(2,1))  
 ])  
 )  
print("nn.Sequential-->OrderedDict方式：",net\_sequential\_orderDict)  
print("\_"\*50)  
  
# 方法-03-使用nn-ModuleList类构造模型。注意nn.ModuleList仅仅是一个存储各类模块的列表，这些模块之间没有联系，也没有顺序，没有实现forward功能，但是加入到nn.ModuleList中模块的参数会被自动添加到整个网络。  
net\_moduleList=nn.ModuleList([nn.Linear(num\_inputs,2),nn.ReLU()])  
net\_moduleList.append(nn.Linear(2,1)) # 可以像列表已有追加层  
print("nn.ModuleList构造模型：",net\_moduleList)  
print("\_"\*50)  
  
# 方法-04-使用nn.ModuleDict类构造模型。注意nn.ModuleDict与nn.ModuleList类似，模块间没有关联，没有实现forward功能，但是参数会被自动添加到整个网络中。  
net\_moduleDict=nn.ModuleDict({  
 'hidden':nn.Linear(num\_inputs,2),  
 'activation':nn.ReLU()   
 })  
net\_moduleDict['output']=nn.Linear(2,1) # 象字典一样添加模块  
print("nn.ModuleDict构造模型：",net\_moduleDict)  
print("\_"\*50)  
  
# D-查看参数  
for param in net.parameters():  
 print(param)  
print("\_"\*50)  
  
# E-初始化模型参数  
from torch.nn import init  
init.normal\_(net.hidden.weight,mean=0,std=0.01) # 权值初始化为均值为0，标准差为0.01的正态分布。如果是用net\_sequential,则可以使用net[0].weight指定权值，如果层自定义了名称，则可以用.layerName来指定  
init.constant\_(net.hidden.bias,val=0) # 偏置初始化为0  
  
print("初始化参数后：")  
for param in net.parameters():  
 print(param)  
print("\_"\*50)  
  
# F-定义损失函数  
loss=nn.MSELoss()  
  
# J-定义优化算法  
import torch.optim as optim  
optimizer=optim.SGD(net.parameters(),lr=0.03)  
print("optimizer:",optimizer)  
  
# 配置不同的学习率-方法-01  
optimizer\_=optim.SGD([   
 {'params': net.hidden.parameters(),'lr':0.03},   
 {'params': net.output.parameters(), 'lr':0.01}  
 ], lr=0.02) # 如果对某个参数不指定学习率，就使用最外层的默认学习率  
print("配置不同的学习率-optimizer\_:",optimizer\_)  
  
# 配置不同的学习率-方法-02-新建优化器  
for param\_group in optimizer\_.param\_groups:  
 param\_group['lr'] \*= 0.1 # 学习率为之前的0.1倍  
print("新建优化器调整学习率-optimizer\_:",optimizer\_)  
print("\_"\*50)  
  
# H-训练模型  
num\_epochs=1000  
best\_loss=np.inf  
for epoch in range(1,num\_epochs+1):  
 loss\_acc\_val=0  
 for X,y in train\_data\_iter:  
 output=net(X)  
 l=loss(output,y.view(-1,1))  
 optimizer.zero\_grad() # 梯度清零，  
 l.backward()  
 optimizer.step()   
   
 # 验证数据集，计算预测值与真实值之间的均方误差累加和（用MSELoss损失函数）  
 with torch.no\_grad():  
 for X\_,y\_ in val\_data\_iter:  
 y\_pred=net(X\_)  
 valid\_loss=loss(y\_pred,y\_.view(-1,1))  
 loss\_acc\_val+=valid\_loss   
   
 # print('epoch %d, loss: %f' % (epoch, l.item()))   
 if epoch%100==0:  
 logx.msg('epoch %d, loss: %f' % (epoch, l.item())) # logx.msg也会打印待保存的结果，因此注释掉上行的print  
 print("验证数据集-精度-loss：{}".format(loss\_acc\_val))  
   
 metrics={'loss':l.item() ,  
 'lr':optimizer.param\_groups[-1]['lr']}  
 curr\_iter=epoch\*len(train\_data\_iter)  
 # print("+"\*50)  
 # print(metrics,curr\_iter)  
 logx.metric(phase="train",metrics=metrics,epoch=curr\_iter) # 对传入字典中的数据进行保存记录，参数phase可以选择'train'和'val'；参数metrics为传入的字典；参数epoch表示全局轮次。若开启了tensorboard则自动增加。  
   
 if loss\_acc\_val<best\_loss:  
 best\_loss=loss\_acc\_val  
 save\_dict={"state\_dict":net.state\_dict(),  
 'epoch':epoch+1,  
 'optimizer':optimizer.state\_dict()  
 }  
 ''' logx.save\_model在JupyterLab环境下运行目前会出错，可以在Spyder下运行  
 logx.save\_model(  
 save\_dict=save\_dict, # checkpoint字典形式保存，包括epoch，state\_dict等信息  
 metric=best\_loss, # 保存评估指标  
 epoch=epoch, # 当前轮次  
 higher\_better=False, # 是否更高更好，例如准确率  
 delete\_old=True # 是否删除旧模型  
 )  
 '''

the expression generating labels:w1\*x1+-w2\*x2+b  
nn.Module构造模型：  
net:MLP(  
 (hidden): Linear(in\_features=2, out\_features=2, bias=True)  
 (activation): ReLU()  
 (output): Linear(in\_features=2, out\_features=1, bias=True)  
)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
nn.Sequential构造模型： Sequential(  
 (0): Linear(in\_features=2, out\_features=2, bias=True)  
 (1): ReLU()  
 (2): Linear(in\_features=2, out\_features=1, bias=True)  
)  
nn.Sequential()-->.add\_module方式： Sequential(  
 (hidden): Linear(in\_features=2, out\_features=2, bias=True)  
 (activation): ReLU()  
 (output): Linear(in\_features=2, out\_features=1, bias=True)  
)  
nn.Sequential-->OrderedDict方式： Sequential(  
 (hidden): Linear(in\_features=2, out\_features=2, bias=True)  
 (activation): ReLU()  
 (output): Linear(in\_features=2, out\_features=1, bias=True)  
)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
nn.ModuleList构造模型： ModuleList(  
 (0): Linear(in\_features=2, out\_features=2, bias=True)  
 (1): ReLU()  
 (2): Linear(in\_features=2, out\_features=1, bias=True)  
)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
nn.ModuleDict构造模型： ModuleDict(  
 (hidden): Linear(in\_features=2, out\_features=2, bias=True)  
 (activation): ReLU()  
 (output): Linear(in\_features=2, out\_features=1, bias=True)  
)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Parameter containing:  
tensor([[0.3069, 0.2901],  
 [0.0576, 0.5021]], requires\_grad=True)  
Parameter containing:  
tensor([-0.6169, -0.1993], requires\_grad=True)  
Parameter containing:  
tensor([[ 0.0393, -0.1843]], requires\_grad=True)  
Parameter containing:  
tensor([-0.4607], requires\_grad=True)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
初始化参数后：  
Parameter containing:  
tensor([[-0.0077, -0.0019],  
 [-0.0037, -0.0100]], requires\_grad=True)  
Parameter containing:  
tensor([0., 0.], requires\_grad=True)  
Parameter containing:  
tensor([[ 0.0393, -0.1843]], requires\_grad=True)  
Parameter containing:  
tensor([-0.4607], requires\_grad=True)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
optimizer: SGD (  
Parameter Group 0  
 dampening: 0  
 lr: 0.03  
 momentum: 0  
 nesterov: False  
 weight\_decay: 0  
)  
配置不同的学习率-optimizer\_: SGD (  
Parameter Group 0  
 dampening: 0  
 lr: 0.03  
 momentum: 0  
 nesterov: False  
 weight\_decay: 0  
  
Parameter Group 1  
 dampening: 0  
 lr: 0.01  
 momentum: 0  
 nesterov: False  
 weight\_decay: 0  
)  
新建优化器调整学习率-optimizer\_: SGD (  
Parameter Group 0  
 dampening: 0  
 lr: 0.003  
 momentum: 0  
 nesterov: False  
 weight\_decay: 0  
  
Parameter Group 1  
 dampening: 0  
 lr: 0.001  
 momentum: 0  
 nesterov: False  
 weight\_decay: 0  
)  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
epoch 100, loss: 0.000083  
验证数据集-精度-loss：0.001036209985613823  
epoch 200, loss: 0.000092  
验证数据集-精度-loss：0.0011525173904374242  
epoch 300, loss: 0.000131  
验证数据集-精度-loss：0.0010611915495246649  
epoch 400, loss: 0.000086  
验证数据集-精度-loss：0.000993600464425981  
epoch 500, loss: 0.000108  
验证数据集-精度-loss：0.000992958783172071  
epoch 600, loss: 0.000086  
验证数据集-精度-loss：0.0009836057433858514  
epoch 700, loss: 0.000087  
验证数据集-精度-loss：0.0009949144441634417  
epoch 800, loss: 0.000090  
验证数据集-精度-loss：0.0009794483194127679  
epoch 900, loss: 0.000112  
验证数据集-精度-loss：0.000968275882769376  
epoch 1000, loss: 0.000092  
验证数据集-精度-loss：0.000963839003816247

print("params\_hidden:{}\nbias:{}\nparams\_output:{}\nbias:{}".format(net.hidden.weight,net.hidden.bias,net.output.weight,net.output.bias))

params\_hidden:Parameter containing:  
tensor([[ 0.6233, -0.4065],  
 [-0.4955, 0.4709]], requires\_grad=True)  
bias:Parameter containing:  
tensor([2.8779, 2.2247], requires\_grad=True)  
params\_output:Parameter containing:  
tensor([[ 2.9630, -2.3273]], requires\_grad=True)  
bias:Parameter containing:  
tensor([2.5503], requires\_grad=True)

在cmd命令行中输入tensorboard --logdir=logs可以根据提示在http://localhost:6006/页面下打开tensorboard，并显示以下损失曲线和学习率的图表。



本章是从机器学习（以[scikit-learn](https://scikit-learn.org/stable/)⑤库为主）各类分析数据规律的算法（聚类、回归和分类），到深度学习（以[pytorch](https://PyTorch.org/）%3Csup%3E②%3C/sup%3E库为主)人工神经网络（多层感知机））的解读。从典型三层神经网络的逐步构建，代码整合中可以深度的理解前向传播和反向传播，及梯度计算是如何实现权值（参数）更新，并应用梯度下降法（一种优化算法）实现迭代，降低损失函数（代价函数，例如平方误差），最终确定权值的过程。进一步应用同一神经网络结构，用PyTorch库实现，包括自定义函数实现，和直接使用PyTorch的内置算法实现。通过同一神经网络结构的逐步构建、代码整合和PyTorch实现，能够清晰的理解深度学习的核心思想，有益于在各个专业领域更准确的构建和应用神经网络。

国内的深度学习框架探索也在发展，例如[飞桨PaddlePaddle](https://www.paddlepaddle.org.cn/)⑥，是集深度学习核心框架、工具组件和服务平台为一体的开源深度学习平台。

注释（Notes）：

① PyTorch的安装方法 （Get Started），（<https://pytorch.org/get-started/locally>）。

② 深度学习库PyTorch官网首页，（<https://pytorch.org>）。

③ 深度学习库TensorFlow]官网首页，（<https://www.tensorflow.or>）。

④ runx.logx，NVIDIA开源的一款深度学习实验管理工具（<https://github.com/NVIDIA/runx#introduction--a-simple-example>）。

⑤ scikit-learn，机器学习库，（<https://scikit-learn.org/stable>）。

⑥ 飞桨PaddlePaddle，国内深度学习框架，（<https://www.paddlepaddle.org.cn>）。

参考文献（References）:

[1] Aston Zhang, Zack C. Lipton, Mu Li,etc.Dive into Deep Learning[M]. <https://d2l.ai/>；中文版-阿斯顿.张,李沐,扎卡里.C. 立顿,亚历山大.J. 斯莫拉.动手深度学习[M].人民邮电出版社,北京,2019-06-01. <https://zh.d2l.ai/>

[2] 一文弄懂神经网络中的反向传播法——BackPropagation，<https://www.cnblogs.com/charlotte77/p/5629865.html>.