# **Pytorch教程六-线性回归的手动实现**

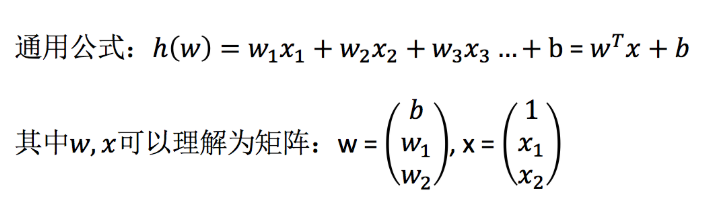
1. **什么是线性回归**

**我这里直接引用《百度百科》中的一段话：**

线性回归方程是利用[数理统计](https://baike.baidu.com/item/%E6%95%B0%E7%90%86%E7%BB%9F%E8%AE%A1/408183" \t "https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E5%9B%9E%E5%BD%92%E6%96%B9%E7%A8%8B/_blank)中的[回归分析](https://baike.baidu.com/item/%E5%9B%9E%E5%BD%92%E5%88%86%E6%9E%90/2625498" \t "https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E5%9B%9E%E5%BD%92%E6%96%B9%E7%A8%8B/_blank)，来确定两种或两种以上变数间相互依赖的[定量关系](https://baike.baidu.com/item/%E5%AE%9A%E9%87%8F%E5%85%B3%E7%B3%BB/4332585" \t "https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E5%9B%9E%E5%BD%92%E6%96%B9%E7%A8%8B/_blank)的一种统计分析方法之一。线性回归也是回归分析中第一种经过严格研究并在实际应用中广泛使用的类型。按自变量个数可分为一元线性回归分析方程和多元线性回归分析方程。

一般来说，在回归方程中当只有自变量的情况称为单变量回归，多于一个自变量情况的叫做多元回归。

线性回归的函数表达式如下：



现在我们打算训练一个房价预测的线性回归模型，即：



上述公式中的和为线性回归模型的权重，权重决定了每个特征对我们预测值即房价的影响。称为偏移量或截距，用于提升模型的表现力。

那么训练上述模型的意思就是在训练数据中学习（或者说寻找到）到一些较好的权重和偏置。较好的量化表达就是依据所构建的模型能够描述房价的真实情况。上述我们的线性表达式只使用了房子的面积和年龄来进行建模。在现实的建模领域中我们的数据维度都是比较高的，即使用很多个特征：



改写成矩阵的形式为：



关于线下回归更深一步的理论我们这里就不在赘述了，大家可以参考公众号后续详细的教程。这里先不赘述了。

1. **数据准备**

我们目标是训练一个线性回归模型，因此我们先创建一些数据用于模型的训练。

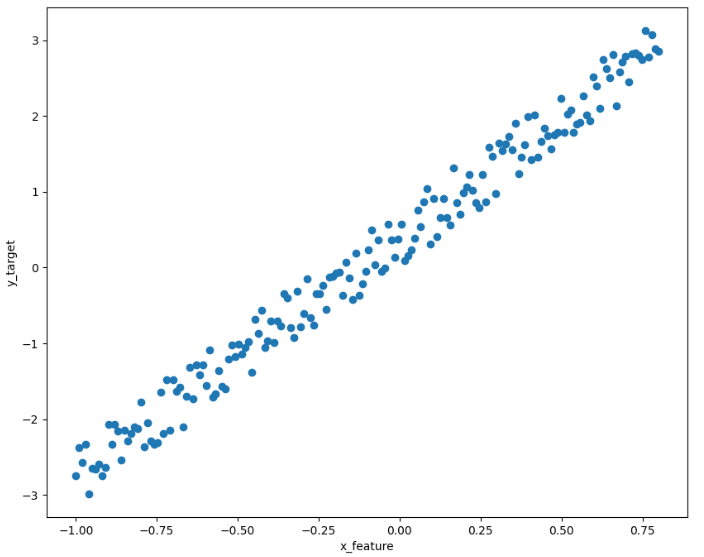
import torch  
from torch.autograd import Variable  
import matplotlib.pyplot as plt

def gen\_data(num):  
 *# 构建num个数据点(X,Y)* X = Variable(torch.linspace(-1,1,num).type(torch.FloatTensor))  
 *# Y = 3.14X + 0.8\*b  
 # 偏差b满足均值为0，方差为取得标准正太分布* Y = 3.14\*X + 0.8 \* torch.rand(X.size())  
 return X,Y

在数据即已经产生的基础上，我们来对数据集进行划分：训练集合测试集

def show\_data(X,Y):  
 x\_train = X[:-20]  
 x\_test = X[-20:]  
 y\_train = Y[:-20]  
 y\_test = Y[-20:]  
 plt.figure(figsize=(10,8))  
 plt.plot(x\_train.data.numpy(),y\_train.data.numpy(),'o')  
 plt.xlabel('x\_feature')  
 plt.ylabel('y\_target')  
 plt.show()  
 return x\_train,y\_train,x\_test,y\_test

训练数据集的可视化如下图所示：



在函数show\_data中我们返回了x\_train,y\_train,x\_test,y\_test用于后期我们进行预测。作图的代码我们也可以选择以下方式：

plt.scatter(x\_train.data.numpy(),y\_train.data.numpy())

1. **构建模型和训练**

接下来的操作是定义模型，后期来进行模型的训练。我们先来看一下我们的代码，然后对代码进行解释：

*# 定义超参数*epoch = 5000  
learning\_rate = 0.001  
  
*# 初始化参数值*W = Variable(torch.rand(1),requires\_grad = True)  
B = Variable(torch.rand(1),requires\_grad = True)  
for i in range(epoch):  
 *# 模型* y\_predictions = W \* x\_train + B  
 *# 计算损失* square\_loss = torch.mean((y\_predictions - y\_train) \*\*2)  
 *# 反向求导* square\_loss.backward()  
 *# 根据上一步的计算结果来更新参数值，大家参考梯度下降的公式  
 # W.data = W.data - learning\_rate \* W.grad.data  
 # B.data = B.data - learning\_rate \* B.grad.data  
 # 或者使用以下代码进行更新* W.data.add\_(-learning\_rate \* W.grad.data)  
 B.data.add\_(-learning\_rate \* B.grad.data)  
 *# 打印loss信息* if epoch % 500 == 0:  
 print(square\_loss)  
 *# 清空自动求导保存的梯度信息，避免在backend的过程中反复的进行累计* W.grad.data.zero\_()  
 B.grad.data.zero\_()

express = str(W.data.numpy()[0]) + 'X + ' + str(B.data.numpy()[0])  
print('W:',W)  
print('B:',B)  
print(express)

plt.figure(figsize=(10,6))  
plt.scatter(x\_train.data.numpy(),y\_train.data.numpy())  
plt.plot(x\_train.data.numpy(),y\_predictions.data.numpy(),'r-',lw = 6)  
plt.xlabel('X')  
plt.ylabel('Y')  
plt.show()

1. 首先我们定义了两个超参数迭代次数epoch和学习率learning\_rate。
2. 之后初始化了我们的参数值权重W和偏置B,并使用随机函数进行初始化。
3. 使用for循环进行迭代，在每一次迭代的过程中都去计算y\_predictions的值，然后计算这个y\_predictions与真实值之间的差即平均误差函数square\_loss.
4. 接着调用.backward()来计算square\_loss对所有计算图中的叶子节点W和B的导数值,backward()函数计算之后，相关的梯度信息就存在于W.grad和B.grad中。
5. 之后使用learning\_rate进行梯度数值的更新。即使用data属性获取数值来进行更新。
6. 最后使用函数data.zero\_()来清空梯度信息，如果不清空梯度信息会在下一次的迭代中进行累计。

上述代码的输出如下：

tensor(1.4648, grad\_fn=<MeanBackward0>)

tensor(0.8666, grad\_fn=<MeanBackward0>)

tensor(0.5277, grad\_fn=<MeanBackward0>)

tensor(0.3312, grad\_fn=<MeanBackward0>)

tensor(0.2166, grad\_fn=<MeanBackward0>)

tensor(0.1497, grad\_fn=<MeanBackward0>)

tensor(0.1107, grad\_fn=<MeanBackward0>)

tensor(0.0879, grad\_fn=<MeanBackward0>)

tensor(0.0746, grad\_fn=<MeanBackward0>)

tensor(0.0668, grad\_fn=<MeanBackward0>)

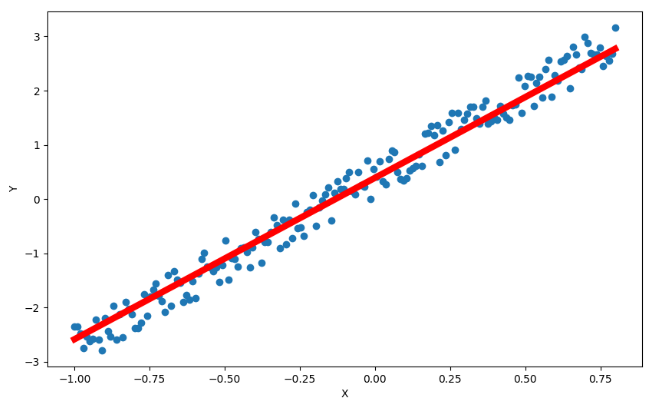
W: tensor([2.9768], requires\_grad=True)

B: tensor([0.3918], requires\_grad=True)

2.9768481X + 0.39175728

即我们得到的函数表达式为：**Y = 2.9768481X + 0.39175728**

图像输出为：



我们这里将epoch的大小更改为10000后我们的输出为：

tensor(2.4309, grad\_fn=<MeanBackward0>)

tensor(1.1678, grad\_fn=<MeanBackward0>)

tensor(0.6682, grad\_fn=<MeanBackward0>)

tensor(0.4077, grad\_fn=<MeanBackward0>)

tensor(0.2597, grad\_fn=<MeanBackward0>)

tensor(0.1738, grad\_fn=<MeanBackward0>)

tensor(0.1238, grad\_fn=<MeanBackward0>)

tensor(0.0946, grad\_fn=<MeanBackward0>)

tensor(0.0775, grad\_fn=<MeanBackward0>)

tensor(0.0676, grad\_fn=<MeanBackward0>)

tensor(0.0617, grad\_fn=<MeanBackward0>)

tensor(0.0583, grad\_fn=<MeanBackward0>)

tensor(0.0564, grad\_fn=<MeanBackward0>)

tensor(0.0552, grad\_fn=<MeanBackward0>)

tensor(0.0545, grad\_fn=<MeanBackward0>)

tensor(0.0541, grad\_fn=<MeanBackward0>)

tensor(0.0539, grad\_fn=<MeanBackward0>)

tensor(0.0538, grad\_fn=<MeanBackward0>)

tensor(0.0537, grad\_fn=<MeanBackward0>)

tensor(0.0536, grad\_fn=<MeanBackward0>)

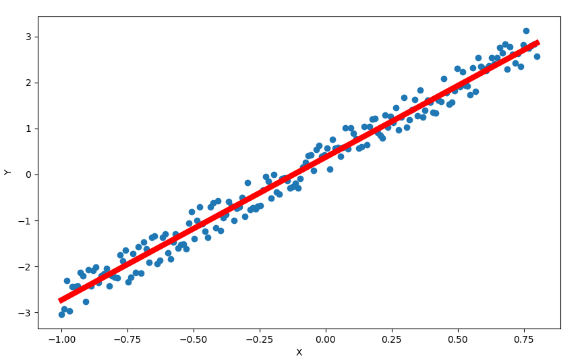
W: tensor([3.1052], requires\_grad=True)

B: tensor([0.3757], requires\_grad=True)

3.105226X + 0.37572587

更改**epoch=10000**后的得到的函数表达式为：**Y = 3.105226X + 0.37572587**

图像输出为：



Epoch迭代步数增加的会提升最终的精度，但是训练的时间也会变得漫长。

1. **预测**

接下来我们对x\_test来进行预测：

y\_pred = W \* x\_test + B  
print('x\_test 预测结果:',y\_pred)  
print('真实的y\_test:',y\_test)

# 输出为

x\_test 预测结果: tensor([2.9262, 2.9579, 2.9895, 3.0212, 3.0529, 3.0846, 3.1162, 3.1479, 3.1796,3.2112, 3.2429, 3.2746, 3.3062, 3.3379, 3.3696, 3.4013, 3.4329, 3.4646, 3.4963, 3.5279], grad\_fn=<AddBackward0>)

真实的y\_test: tensor([2.9822, 3.1645, 3.2761, 2.7792, 3.4197, 3.4546, 2.8431, 2.9376, 3.1887,2.9724, 3.4938, 3.5429, 2.9767, 3.0686, 3.1811, 3.8107, 3.7920, 3.4827,

3.8138, 3.8148])

1. **torch.nn构建模型**

之前我们基本上是手动来实现的线性模型，接下来我们借助torch的nn模块来实现线性回归模型。

在PyTorch中nn模块定义了很多的函数供我们使用，这些函数基本上涵盖了构建神经网络会使用的各种层，当然nn模块中还包含了大部分我们平时使用的损失函数。

我们来看第一个版本的代码：

import torch  
from torch.autograd import Variable  
  
x = Variable(torch.linspace(-1,1,100).type(torch.FloatTensor))  
x = torch.unsqueeze(x,dim=1)  
y = 3.14\*x + 0.8\* torch.rand(x.size())  
  
*# 1.定义超参数*epoch = 10000  
learning\_rate = 0.001  
  
*# 2.定义模型  
# 输入输出特征均为1维，该函数会初始化权重信息*model = torch.nn.Linear(in\_features=1, out\_features=1)  
  
*# 3.定义损失函数  
# 使用均方误差作为损失函数，size\_average = False表示使用总误差*criterion= torch.nn.MSELoss(size\_average= False)  
  
*# 4.定义优化函数  
# model.parameters()会自动提取模型中的参数*optimizer= torch.optim.SGD(model.parameters(),lr=learning\_rate)  
  
*# 5.开始训练*for i in range(epoch):  
 *# 预测* y\_predictions = model(x)  
 *# 损失计算* loss = criterion(y\_predictions, y)  
 if i % 500 ==0:  
 print(loss)  
 *# 梯度归零* optimizer.zero\_grad()  
 *# 反向传播* loss.backward()  
 *# 更新参数* optimizer.step()

*# 打印参数信息*  
for name, param in model.named\_parameters():  
 print(name,param)  
 print('\_\_\_\_\_\_\_\_\_')

输出为：

tensor(325.2811, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

tensor(5.3676, grad\_fn=<MseLossBackward0>)

weight Parameter containing:

tensor([[3.1783]], requires\_grad=True)

\_\_\_\_\_\_\_\_\_

bias Parameter containing:

tensor([0.4176], requires\_grad=True)

\_\_\_\_\_\_\_\_\_

可以看出我们最终学习到的W的值为3.1783，B的值为0.4176。

还有一种方法是参考网上的，基本上都是一样的写法：

import torch  
from torch.autograd import Variable  
  
x = Variable(torch.linspace(-1,1,100).type(torch.FloatTensor))  
x = torch.unsqueeze(x,dim=1)  
y = 3.14\*x + 0.8\* torch.rand(x.size())  
  
*# 1.定义超参数*epoch = 10000  
learning\_rate = 0.001  
  
*# 2.定义模型  
# 输入输出特征均为1维，该函数会初始化权重信息*class LinearModel(torch.nn.Module):  
 def \_\_init\_\_(self):  
 super(LinearModel, self).\_\_init\_\_()  
 self.linear = torch.nn.Linear(1, 1)  
 def forward(self, x):  
 y\_predictions = self.linear(x)  
 return y\_predictions  
model = LinearModel()  
  
*# 3.定义损失函数  
# 使用均方误差作为损失函数，size\_average = False表示使用总误差*criterion= torch.nn.MSELoss(size\_average= False)  
  
*# 4.定义优化函数  
# model.parameters()会自动提取模型中的参数*optimizer= torch.optim.SGD(model.parameters(),lr=learning\_rate)  
  
*# 5.开始训练*for i in range(epoch):  
 *# 预测* y\_predictions = model(x)  
 *# 损失计算* loss = criterion(y\_predictions, y)  
 if i % 500 ==0:  
 print(loss)  
 *# 梯度归零* optimizer.zero\_grad()  
 *# 反向传播* loss.backward()  
 *# 更新参数* optimizer.step()  
  
print('w: ',model.linear.weight.item())  
print('b:', model.linear.bias.item())  
  
*# 预测*x\_test = torch.Tensor([[2.5]])  
y\_test = model(x\_test)  
print("y\_pred = ", y\_test.data)

# 输出

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

tensor(5.8470, grad\_fn=<MseLossBackward0>)

w: 3.186012029647827

b: 0.3983287513256073

y\_pred = tensor([[8.3634]])

1. **总结**

本次文章我们借助PyTorch实现了线性回归模型，整体上来说是比较容易的。在实现这个项目的时候，最好先学习一下线性回归模型的理论知识，这样能更好的理解文章中的要点。

**参考资料**：李沐、Aston Zhang《动手学深度学习》