AI 线性分类问题(1)

Cangye@geophyx.com

1 向量张量与矩阵

矩阵是由空间张量所衍生出的概念，所以一些名词诸如特征向量之类的带有明显的坐标的含义。这些概念的理解与分析对于之后的神经网络的数学模型的分析有着相当重要的意义。但概念之间又有些许不同。比如在空间度量下的向量长度的概念：

$$|\xi|^2 = g_{ij}\xi^i\xi^j \tag{1}$$

在直角坐标系下表示为：

$$|\xi|^2 = \xi^i\xi^i \tag{2}$$

这是二范数的形式：$||\xi|| = \xi^i\xi^i$而二范数是与空间坐标变换无关的。在机器学习的相关概念中，向量长度的概念都是坐标无关的，这是与数学中定义不同的部分。但是在迭代算法的设计与理解过程中,类似坐标变换$x^i = x^i(z_1, z_2, \cdots, z_n)$确实又扮演者非常重要的作用，十分明显的例子诸如梯度法和共轭梯度法。 SVM是在空间坐标平面的基础上去理解的，本质上也是一个坐标变换。数学语言太干瘪，用例子来说：

2 单层感知器

单层感知器的数学模型：

$$y = f(\vec{x}\vec{W} + \vec{b}) \tag{3}$$

图形化表示如下： @— @—— @—- @—— 在实现函数逼近的过程中，通常的做法是选取一个代价函数。比如$\vec{x}, d$代表一个输入输出对。实现过程中需要做的就是使得预期输出与实际输出的差距尽可能的小：

$$\varepsilon = (d - y(\vec{x}))^2 \tag{4}$$

这里选择的代价函数使类似于向量的内积的形式的，但是并没有乘以坐标变换张量g_{ij}，选择平方作为代价函数并不是因为需要用到其长度的概念，而是使得我们的优化过程变成明确的凸优化过程。可见这个代价函数的选取并不是确定的。看下图优化过程： @https://zhuanlan.zhihu.com/p/26260132/edit

$$y = f(\vec{x}\vec{W} + \vec{b}) \tag{5}$$

通过问题的转换使得我们要的点在二维曲面的最低点。这是凸优化的转换过程，在很多高端的数值模拟方法中都有用到，这也是为什么'ansys'(数值模拟软件)的求解器可以用于金融数据分析。之后的过程很简单，我们只需要不断的进行梯度的求解：

$$grad = E(\nabla\mathcal{E}) = E\begin{bmatrix} \nabla_w\mathcal{E}^T \\ \nabla_b\mathcal{E}^T \end{bmatrix} \tag{6}$$

这是我们求解迭代的过程，用图形表示为： @https://zhuanlan.zhihu.com/p/26260132/edit

3 单层感知器实现

用Python实现上述过程：

```
class LMS():
    def Sigmoid(self,x):
        return 1/(1+np.exp(-x))
    def DSigmiod(self,x):
        return np.exp(-x)/(1+np.exp(-x))**2
    def __init__(self,shape=[2,1]):
        self.shape=shape
        self.W=np.random.random(shape)
        self.b=np.random.random(shape[1])
    def train(self,data,vali,eta):
        nu=np.dot(data,self.W)+np.tile(self.b,np.shape(vali))
        para=(vali-self.Sigmoid(nu))*self.DSigmiod(nu)
        para=np.reshape(para,[-1])
        x=np.transpose(data)
        grad_t=np.multiply(x,para)
        grad=np.transpose(grad_t)
        grad_ave=np.average(grad,axis=0)
        grad_ave=np.reshape(grad_ave,self.shape)
        self.W=np.add(self.W,eta*grad_ave)
        self.b=np.add(self.b,eta*np.average(para))
    def valid(self,data):
        return self.Sigmoid(np.dot(data,self.W))
```

或者调用TensorFlow

4 多层感知器

$$\frac{\mathcal{E}}{w^{[N-1]}} = f'(y^{[N-2]} \cdot w^{[N-1]}) * w^{[N]} \cdot [f'(y^{[N-1]} \cdot w^{[N]}) * (d - y^{[N]})^T] \cdot y^{[N-1]} = \mathcal{M}^{[N-1]} y^{[N-1]} \tag{7}$$

5 多层感知器实现

Python实现

```
    def forward(self,data):
        self.y[0][:]=data
        temp_y=data
        for itrn in range(self.layer-1):
            temp_v=np.dot(temp_y,self.W[itrn])
            temp_vb=np.add(temp_v,self.b[itrn])
            temp_y=self.sigmoid(temp_vb)
            self.y[itrn+1][:]=temp_y
            self.d_sigmoid_v[itrn+1][:]=self.d_sigmiod(temp_vb)
        return self.y[-1]
    def back_forward(self,dest):
        self.e[self.layer-1]=dest-self.y[self.layer-1]
        temp_delta=self.e[self.layer-1]*self.d_sigmoid_v[self.layer-1]
```

```
14            temp_delta=np.reshape(temp_delta,[-1,1])
15            self.dW[self.layer-2][:]=np.dot(np.reshape(self.y[self.layer-2],[-1,1]),np.transpose(
                  temp_delta))
16            self.db[self.layer-2][:]=np.transpose(temp_delta)
17            #print(self.dW[self.layer-2])
18            for itrn in range(self.layer-2,0,-1):
19                sigma_temp_delta=np.dot(self.W[itrn],temp_delta)
20                temp_delta=sigma_temp_delta*np.reshape(self.d_sigmoid_v[itrn],[-1,1])
21                self.dW[itrn-1][:]=np.dot(np.reshape(self.y[itrn-1],[-1,1]),np.transpose(temp_delta))
22                self.db[itrn-1][:]=np.transpose(temp_delta)
```