用 C 语言实现神经网络需要几步？

一、写在前面的话

本章主要讲讲神经网络的数学基础，并将神经网络中浮夸的概念用合理的顺序整理一下。应该具备的数学基础说多不多：基本上熟悉导数、线代、概率，那么大部分内容就可以看懂了，然而再进行深入学习的话又需要了解一些微分流形的东西，这个东西也是函数导数所衍生的概念，想想也不是很复杂。但好事者给起了很多复杂的名字，比如PCA比如Adam，这无形中也增加了学习成本。以至于很多人学习过程的最大感受就是：我终其一生似乎都在名词的海洋上漂泊。很多年前在学习数据挖掘算法的时候老师就说过：这些算法你们要完全学会得需要几年的时间。几年时间大约也是人生中最有学习兴趣的几年。现在各种渠道充斥着《28天学会XXXX》的就显得有些高调的过分了，28天学习大概只够学习一门简单的编程语言比如Python，想学会算法之上的内容除非你数学很好，是极好的那种，这种人应该是极少的。所以学习神经网络也是一个体力活，需要不断的分析实践。

二、从几何开始

从几何开始大概是学习神经网络最合适的切入点，大部分神经网络的目标就在于构建一个**超曲面**，而几何中的**张量**衍生出了**矩阵**概念，最后作为几何基础的函数和导数是分析神经网络的基础。

首先来看下空间函数：

$$f(x\_1,\cdots,x\_n)$$

GitChat

上面这个函数是定义于N维空间中的**函数**，这个N维空间数学符号为$\mathbb{R}^n$，其中$(x_1,\cdots,x_n)$被称为空间坐标，到这里就是空间几何的入门了，但接下来我们**并不能**分析神经网络。在分析之前还需要一个概念就是**梯度**：

$$\nabla f(x\_1,\cdots,x\_n)=(\frac{\partial f}{\partial x\_1},\cdots\frac{\partial f}{\partial x\_1})$$

这里的梯度就是对每个坐标分量求偏导数，这是最简单的梯度的形式也是欧式空间中的梯度，在其他坐标空间之下如球坐标、柱坐标等，**梯度作为一个向量显然应当保持空间不变性**，这种空间不变性指的是不管在何种坐标系之下梯度方向和大小都不会发生变化。为保持这种不变性梯度分量的数值上就会改变。这种数值上的变化可以用链式求导法则求得：

$$\frac{\partial f}{\partial x\_i}=\sum\_i^N\frac{\partial f}{\partial z\_k}\cdot\frac{\partial z\_k}{\partial x\_i}$$

假设空间坐标变换有函数关系$z_i=z_i(x_1,\cdots,x_n)$

将上面的链式求导写成矩阵形式：

$$\nabla\_x f=\nabla\_z f\cdot A$$

这里为了保持在空间坐标变换下的不变量所乘以的矩阵A就称之为张量，这个张量是有具体表达的：

$$A=\frac{\partial z\_j}{\partial x\_i}$$

所以**张量(Tensor)**在几何中就定义为不随空间变换变化而变化的量，到此为止不再进行更深入的探讨。可以看到Tensor与常说的**矩阵(Matrix)**是有区别的，这种区别使得张量可以说是矩阵的子集。同时也可以理解为什么在一些梯度迭代算法中需要对梯度进行一定的变换形成所谓的**共轭梯度算法**，这是因为按笛卡尔空间梯度计算的话梯度可能空间已经发生了扭曲之后，使得梯度方向并非最优方向。

上面说到了矩阵，其可以看成一系列数值、函数的有序集合，用矩阵表示公式是方便的，比如我们的线性方程组：

$$A \cdot x = b$$

坐标的**仿射变换**也可以用矩阵的形式表示：

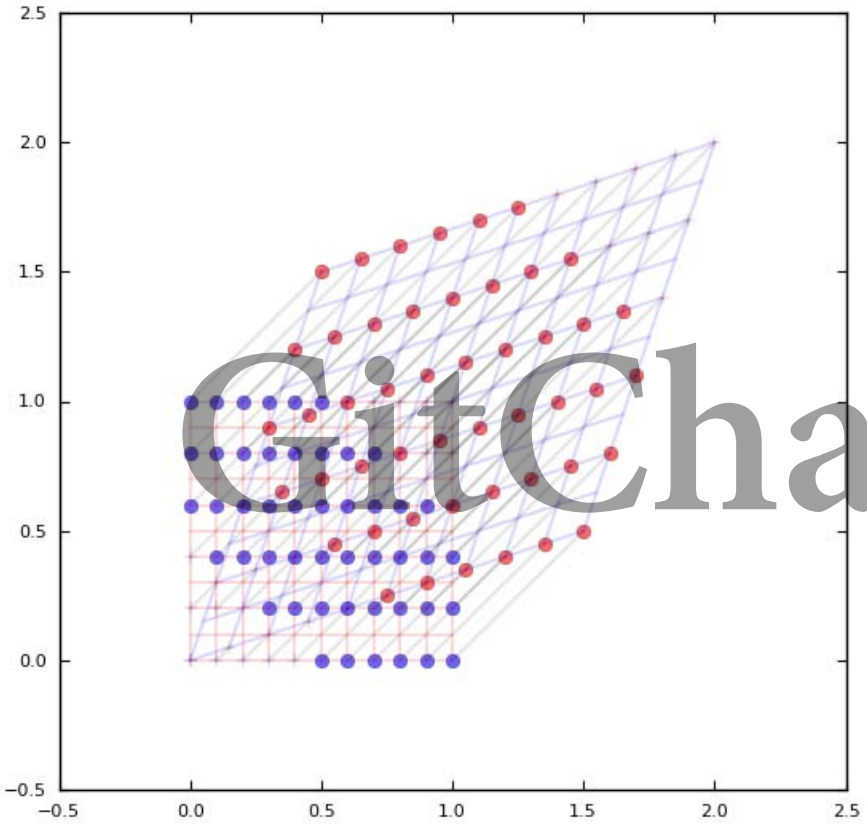
$$z = A \cdot x$$

张量中点乘是一种**缩并**运算，这种缩并代表着求和运算，但是总写求和符号是非常麻烦的，很多数学书籍中将其写成爱因斯坦**约定求和**的方式：

$$\sum_i \frac{\partial f}{\partial z\_k} \cdot \frac{\partial z\_k}{\partial x\_i} = \frac{\partial f}{\partial z\_k} \cdot \frac{\partial z\_k}{\partial x\_i}$$

在乘法计算中相同指标代表求和可以省略求和符号，这在数学分析中是非常有力的工具。

再回到上面的仿射变换，仿射变换代表了空间的旋转拉伸变换：



GitChat

再补充一些群的概念

深入学习机器学习不可避免的会涉及到流形和群，所以这里再补充一些概念。

上面说到，梯度是向量，这种向量的变换方式是与张量A做运算，还有另一种向量其变换方式与梯度向量不同比如速度：

$$v = [\dot{x}\_1, \cdots, \dot{x}\_n]$$

这里用到了另一种简写：

$$\dot{x}\_1 = \frac{dx}{dt}$$

这样在坐标变换x=>z的情况，依然用链式求导法则：

$$\dot{x}_i = \frac{dx_i}{dt} = \frac{\partial x_i}{\partial z_j}\frac{dz_j}{dt} = \frac{\partial x_i}{\partial z_j}\dot{z}$$

上面就用到了约定求和，可以将其写成矩阵形式：

$$v_x = B \cdot v_z$$

空间中向量长度是一个很重要的概念：

$$|v|^2 = v_i \cdot v_i$$

这种定义方式很熟悉，在计算机科学中常将其称为**二范数**。

上面讲过这只是笛卡尔空间中的长度，真实空间长度还需要乘以一个二阶张量：

$$|v|^2 = g_{ij}v_iv_j$$

具体$G = [g_{ij}]$形式的推导有兴趣的可以自己推导一下，利用链式求导法则就可以。

这里要补充的是，如果变换$x = x(z_1, \cdots, z_n)$保持G的形式不变，则将变换称之为**度量**$G = [g_{ij}]$之下的一个**运动**。

我们常用的一个相似度标准有一个**闵可夫斯基距离**：

$$d = \sqrt[p]{|v_1 - v_2|^p}$$

涉及到了闵可夫斯基空间的内容，数学中其特点就是向量长度定义为：

$$|v|^2 = v_1v_1 - v_2v_2 - v_3v_3 - v_4v_4$$

这与计算机科学中的**闵可夫斯基距离**是有区别的，闵可夫斯基空间中$v_1 = ct$是狭义相对论空间。

GitChat

三、张量和矩阵

前面已经多次用到了张量和矩阵的概念了，所以也能理解为什矩阵中会有一些特征向量之类的名词，这是因为矩阵和张量空间的联系所致。空间中的向量可以用几个向量叠加的形式比如：

$$v = v_i\vec{e_i}$$

这里的$\vec{e_i}$是坐标基向量，坐标基可以看成是长度为1的有序实数对，v_i是相应分量上的长度。坐标变换也就是可以看成是对坐标基的变换。对于矩阵可以看成是几个向量的集合：

$$A = [v_1, \cdots, v_m]$$

这些向量v可以用k个向量表示其中$m >= k$，此时就可以对向量数量进行了压缩，这也是**数据压缩中**所诉求的：用更少的数据存储更多内容。

下面讲讲如何进行这种数据的压缩：

首先的就是一个**矩阵的特征值分解**问题，我们可以将一个方阵变为三个矩阵相乘：

$$A_{n\times n} = E\Lambda E^{-1}$$

其中Λ代表一个对角矩阵。对于$rank(A) < n$的情况，这里rank就是**矩阵的秩**，也就是对角矩阵中有多少个非0值，可以用少于N个向量来表示矩阵A中所包含的所有向量，所以这些向量就称为**特征向量**。

但是对于非方阵$B_{m\times n}$情况下，显然无法进行特征值分解，这就需要对矩阵进行部分变换：

$$B^T \cdot B = A\_n \times n$$

对A进行常规的特征值分解：

$$A\_n \times n = E\Lambda E^{-1}$$

由此找到了n个**正交基**$[v\_1, \cdots, v\_n] = E$

若将正交基用B进行变换，

$$m\_i = B \cdot v\_i$$

将向量进行相乘：

$$\vec{m\_i} \cdot \vec{m\_j} = \vec{v\_i}^T \cdot B^T \cdot B \cdot \vec{v\_j}$$
$$= \vec{v\_i}^T \cdot A \cdot \vec{v\_j} = \vec{v\_i}^T \cdot \lambda\_j \cdot \vec{v\_j} = \lambda\_j \delta\_{ij}$$

化简过程用到了特征值的性质：

$$A \cdot v = \lambda \cdot v$$

可以看到，经过线性变换B之后，向量仍然是正交的。m_i向量的个数与矩阵A的秩相同。将m_i进行标准化：

$$\mu\_i = \frac{B \cdot v\_i}{|B \cdot v\_i|} = \frac{B \cdot v\_i}{\sqrt{\lambda\_i}}$$

将上面式子写成：

$$B \cdot v\_i = \sqrt{\lambda\_i}\mu\_i$$

再写成矩阵的形式：

$$BV = U\Lambda$$

所以

$$B = U\Lambda V^T (V\text{中向量都是单位正交的})$$

因此任意矩阵都可以写成三个矩阵相乘的形式。这就是**矩阵的奇异值分解(SVD)**，其中特征值的大小就代表不同特征向量的权重，对于小的权重可省略。所以SVD可以方便的进行数据压缩，但是从另一方面来说，所谓数据压缩就是去除线性相关性比较高的向量，那么如何度量这种相关性呢，一种方法就是将相关矩阵对角化：

比如数据矩阵X，假设均值为0，那么相关矩阵就可以写成：

$$S\_x = \frac{1}{n-1} X \cdot X^T = U\Lambda V^T V \Lambda U^T$$

由此，可以对X乘以U^T即可使得矩阵对角化，这是**PCA方法**。

四、函数分析

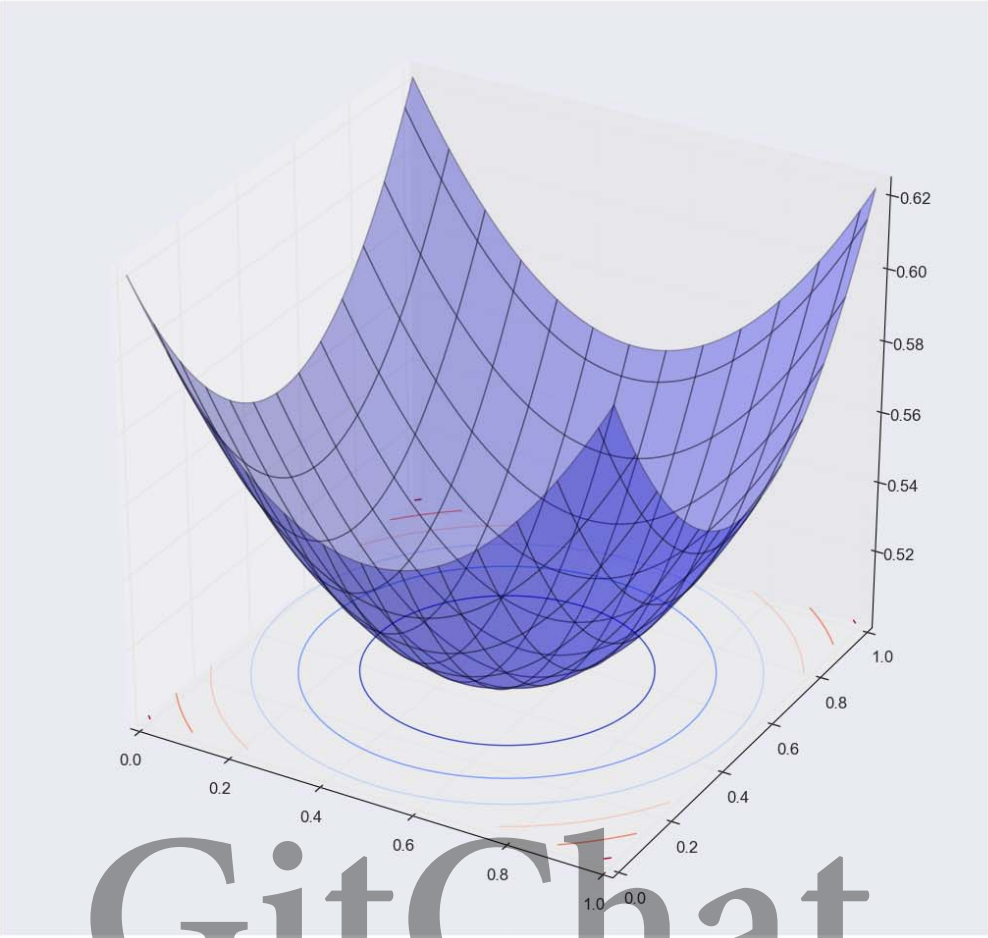
很多机器学习的训练过程中都需要建立一个靠谱的函数，称之为**代价函数**，这个代价函数也可以称之为**泛函**，泛函存在的目的就在于将复杂的数学问题简化为求解函数最小值的问题，举一个**最优化问题**的例子，假设存在一个解线性方程的问题：

$$A \cdot x = b$$

上面方程是难于直接求解的(这个问题还是有其他解法的)，因此将其转化为x的函数：

$$f(x) = x^T Ax - 2bx$$

GitChat



GitChat

对于二维向量x其泛函是一个空间二次曲面，而泛函极小值，也就是梯度为0的时候：

$$\nabla f(x) = Ax - b = 0$$

就对应于方程的解，当然神经网络中泛函常用空间距离或者熵就可以了，这里并没有具体涉及到泛函建立的过程。泛函分析还可以用于证明求解**高斯分布是最大熵分布**。

对于一般多维函数，可以对其进行Taylor展开：

$$f(x) = f(x\_0) + a\_1\nabla(f(x\_0))dx + a\_2dx^T Hdx + o(dx^3)$$

上面是多维函数的展开的数学形式，其中H就是Hessian矩阵。为了寻找函数的最小值，只要对其不断的在负梯度方向上寻找就可以了：

$$x\_t + 1 = x\_t - \lambda\nabla f(x)$$

这就是常说的**最速下降法**，在最小值附近还有一个特征是函数改变量$\Delta f = f(x) - f(x_0)$最小，这对Tayler展开两边对dx求偏导数就可以得到：

$$dx = x\_t + 1 - x\_t = -H^T \cdot \nabla f$$

所以这又是一种梯度迭代方法，称之为**牛顿法**，还有一种，可以预估函数的线性梯度，并在计算中将其减去：

$$g = f + \frac{\partial f\_i}{\partial x\_j}(x - x\_t)\_i = f + J(x - x\_t)$$

此时就可以将函数g的目标定位增长为0，此时对其求x的偏导：

$$dx = x\_t + 1 - x\_t = (J^T \cdot J)^T J^T f(x)$$

这就成为**高斯牛顿法**

最后来看看多层神经网络的情况，用下降法做基本思想，这里将代价函数定义为空间距离：

$$\varepsilon(w) = |d - y| = \sqrt{\sum\_i (d\_i - y\_i)^2}$$

目标就是求解该函数的梯度：

$$\nabla_w \varepsilon = \frac{\varepsilon}{w^{[1]}} + \cdots + \frac{\varepsilon}{w^{[N]}}$$

由于多层神经网络是分层的所以求解偏导数时将其用上标表示不同层的权值，来观察任意相邻两层之间的关系：

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

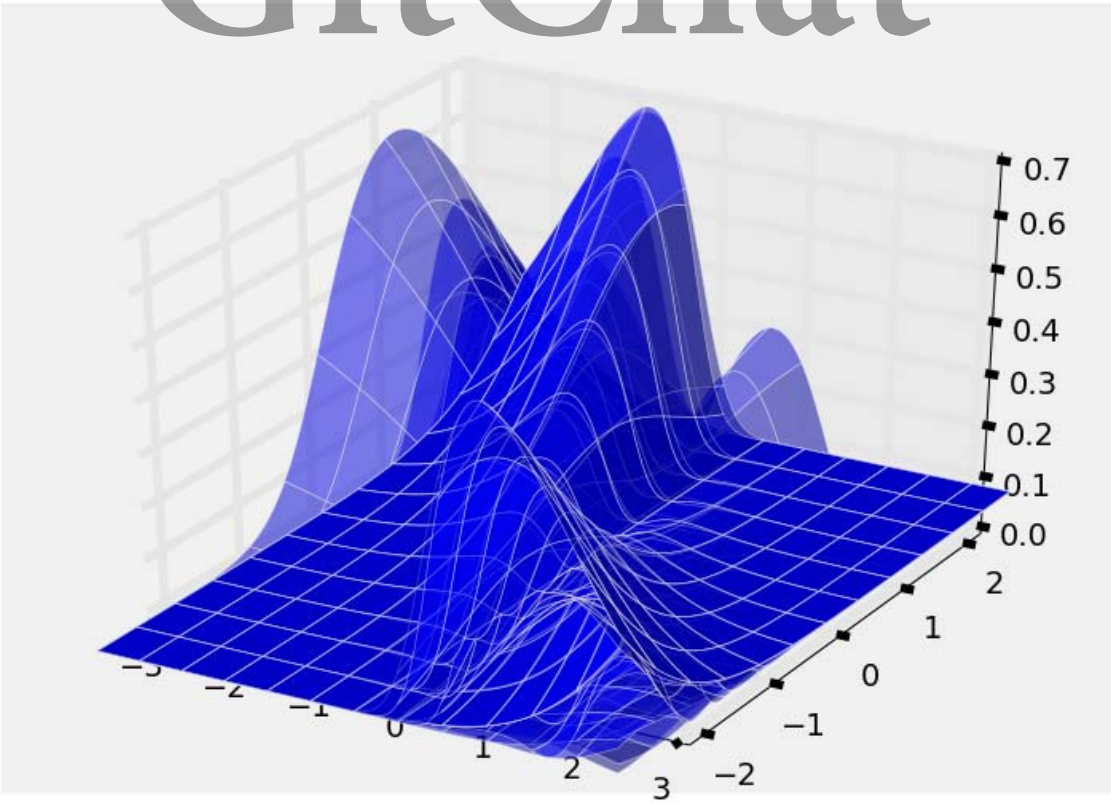
$$\frac{\varepsilon}{w^{[N-1]}} = f'(y^{[N-2]} \cdot w^{[N-1]}) * w^{[N]} \cdot [f'(y^{[N-1]} \cdot w^{[N]}) * (d - y^{[N]})^T] \cdot y^{[N-1]} = \mathcal{M}^{[N-1]} y^{[N-1]}$$

二者矩阵之间的关系如下：

$$\mathcal{M}^{[N-1]} = f'(x) * [w^{[N]} \cdot \mathcal{M}^{[N-1]}]$$

上面的导数是从最后一层的ε向第一层递推传递的，因此这里称为反向传播算法。
这里理论部分讲的差不多了，还需要有个问题，就是在计算的时候实际上所形成的最优化曲面是一个随着样本不断变化的曲面：

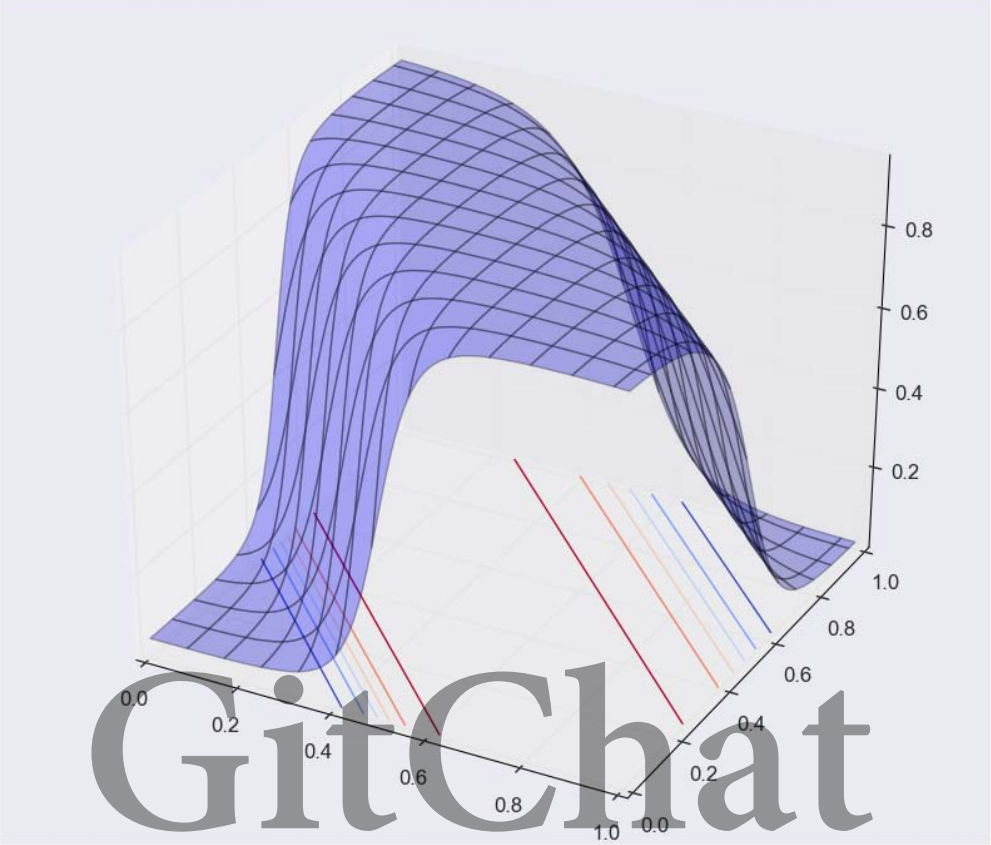
GitChat



这也就是说实际上梯度是带有一定的随机性，为了减少这种随机性，更好的预测梯度的方向，在训练过程中可以一次加入多个样本，这个样本的大小称为为**BATCHSIZE**，这种学习方式称为**批学习**，批学习是容易进行并行化的，而一个个去输入样本的话称之为**在线学习**，这两种学习的学习曲线有所不同。

五、实现

其实到上面应该就可以结束了，但是提到了BP算法实现，这里为了方便用了Python，先来看看其结果：



显然多层网络可以解决抑或问题。

```
"""
@author: Cangye@hotmail.com
"""

import numpy as np

class BPAlg():
    def sigmoid(self,x):
        """
        Define active function sigomid
        """
        return 1/(1+np.exp(-x))
    def d_sigmiod(self,x):
        """
        Define df/dx
        """
        return np.exp(-x)/(1+np.exp(-x))**2
    def __init__(self,shape):
        """
        Initialize weights
        """
        self.shape=shape
```

```
        self.layer=len(shape)
        self.W = []
        self.b = []
        self.e = []
        self.y = []
        self.dW = []
        self.v = []
        self.db = []
        self.d_sigmoid_v = []
        for itrn in range(self.layer-1):
            self.W.append(np.random.random([shape[itrn], shape[itrn+1]]))
            self.dW.append(np.random.random([shape[itrn], shape[itrn+1]]))
            self.b.append(np.random.random([shape[itrn+1]]))
            self.db.append(np.random.random([shape[itrn+1]]))
        for itr in shape:
            self.e.append(np.random.random([itr]))
            self.y.append(np.random.random([itr]))
            self.v.append(np.random.random([itr]))
            self.d_sigmoid_v.append(np.ones([itr]))
    def forward(self, data):
        """
        forward propagation
        """
        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):
        """
        back propagation
        """
        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]
        temp_delta = np.reshape(temp_delta,[-1,1])
        self.dW[self.layer-2][:] = np.dot(np.reshape(self.y[self.layer-2],
[-1,1]),np.transpose(temp_delta))
        self.db[self.layer-2][:] = np.transpose(temp_delta)
        for itrn in range(self.layer-2, 0, -1):
            sigma_temp_delta = np.dot(self.W[itrn],temp_delta)
            temp_delta = sigma_temp_delta*np.reshape(self.d_sigmoid_v[itrn],[-1,1])
            self.dW[itrn-1][:] = np.dot(np.reshape(self.y[itrn-1], [-1,1]),
np.transpose(temp_delta))
            self.db[itrn-1][:] = np.transpose(temp_delta)
    def data_feed(self, data, dest, eta):
        NDT = len(data)
        for itrn in range(NDT):
            self.forward(data[itrn])
            self.back_forward(dest[itrn])
            for itrn in range(self.layer-1):
                self.W[itrn][:] = self.W[itrn] + eta*self.dW[itrn]
                self.b[itrn][:] = self.b[itrn] + eta*self.db[itrn]
```

GitChat