神经网络表示

只有一个隐藏层的神经网络

神经网络通常有多个隐藏层,但为了更加直观简洁地解释神经网络,所以使用只有一个隐藏层的神经网络来进行讲解.

神经网络的第一层通常是样本的特征参数,我们称之为**输入层**,而中间一层的圆圈我们称之为**神经网络的隐藏层**,而最后一层通常只有一个节点,我们称之为**输出层**

隐藏层的含义就是我们看不到这些中间节点的真正数值
我们可以看到训练集的输入值,也可以看到输出值,但是无法看到中间节点的数值,所以中间的节点层被称为**隐藏层**

神经网络中一些俗成的约定

输入特征的值还有另外一种表示方式,我们表示为

$$a^{[0]} = x$$

a意为**激活**,表示网络中不同层的值,然后会被传递给后面的层,从而产生新的激活值a,所以我们将输入层的值称为$a^{[1]}$

隐藏层也会产生激活值,我们称之为$a^{[1]}$,而隐藏层的第一个节点产生的激活值,我们称之为$a_1^{[1]}$,与此相似的,第二个节点称为$a_2^{[1]}$,以此类推.
$a^{[1]}$在这里是一个n维列向量(n是该层节点的个数)

在输出层会产生$a^{[2]}$,也就是logistic回归中的$\hat{y}$,这又是为什么这个值在之前的计算中又被命名为a

但是之前的计算中我们只揭示了这一层输出,在神经网络中,将会使用 带方括号上标的值来明确表明这个值来自于哪一层

在神经网络中,我们通常将**输入层**称为**第零层**,所以有一个隐藏层的神经网络,我们通常称之为**双层神经网络**,在之前的学习中所接触的logistic回归是一个**单层神经网络**,因为我们通常不把输入层看做一个标准的层

隐藏层和输入层都是有相关的参数的,比如说,在第一个隐藏层中,我们具有参数$w^{[1]}, b^{[1]}$,来表示这是第一层的参数
并且在其中,$w^{[1]}$是一个(4,3)的矩阵,**4表示有四个节点,3表示有3个输入特征**,而$b^{[1]}$则是一个(4,1)的矩阵
类似地,输出层也有相关的参数,并且$w^{[2]}$是一个(1,3)的矩阵,而$b^{[2]}$则是(1,1)

计算神经网络的输出

神经网络的计算过程

在神经网络中,以logistic回归为例,一个节点中通常有两个计算步骤,首先按步骤计算出z,然后第二步计算$sigmoid(z)$激活函数.
所以,神经网络只不过是重复计算这些步骤很多次,得到的a则作为下一层的特征参数

神经网络中计算的向量化

根据以上描述,我们就需要在一层中进行多次这样的计算,使用for循环仍然会十分低效,所以我们仍然需要将这一过程**向量化** 所以我们将行向量$\vec{w}^T$纵向堆叠,得到一个矩阵W^T,将这个矩阵与列向量$\vec{x}$进行矩阵乘法,得到

$$\vec{z} = W^T\vec{x} + \vec{b}$$

接下来,就如同之前讲的,利用python的广播对$\vec{z}$进行运算,得到$\vec{a}$

自己的小总结

实际上,神经网络的正向传播所计算出的激活值,需要保存到反向传播中以计算导数,更新w, b的参数,而再次的正向传播又得出新的激活值,以此不断优化,最终得出最合适的参数和输出

神经网络中多个样本计算的向量化

在多个样本中对于不同层激活值的表示

由于在logistic回归中对于不同样本的标记和**多层神经网络**中对于**不同层**激活值的标记,我们都是用了相同的形式,也就是$x^{[m]}$和$a^{[n]}$
所以在神经网络对于多个样本的处理中,需要再次更新标记的方式

举例
对于第二个样本在第一层中的激活值，我们可以这样表示

$$a^{[1](2)}$$

对于计算的向量化

在这样的神经网络中,如果没有一个对多样本计算的向量化表示,那么就意味着我们再次需要使用显式的for循环来对样本进行遍历,这样的计算无疑仍然是低效的

所以我们需要再次将计算向量化

就如同之前在logistic回归中的向量化计算,我们仍然可以将不同样本的特征向量堆叠在一起形成一个矩阵X,就有了

$$Z^{[i]} = W^{[i]}A^{[i-1]} + b^{[i]}$$

$$A^{[i]} = \sigma(Z^{[i]})$$

这样得到的A实际上是不同样本所生成的激活值列向量横向堆叠形成的矩阵

这样,我们就实现了在神经网络中,对于不同样本计算的向量化

激活函数

tanh函数

为了搭建神经网络,我们可以选择在隐藏层里使用哪一个**激活函数**,而在之前的所有计算中,我们使用的都是$sigmoid$函数,但有时,其他函数的效果要好得多

接下来,我们将看看一些可供选择的函数

在更加一般的情况下,我们可以使用不同的函数,比如$g(z)$
$g(z)$可以是非线性函数,不一定是$sigmoid$函数

$sigmoid$函数的值域介于0到1之间,而有个函数几乎表现比$sigmoid$函数更好,这就是$tanh$函数,或者叫做**双曲正切函数**,其值域介于-1到1之间,表达式为

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

实际上这个函数就是将$sigmoid$函数平移了下来,并使其穿过原点

如果我们使$g(z) = tanh(z)$,这样做的效果几乎比$sigmoid$函数更好,因为这样做使得激活函数的输出更加接近于0

有时我们需要训练自己的学习算法时,可能需要平移所有数据,让数据平均值为0,这时使用$tanh$函数就有类似于数据中心化的效果,使得数据的平均值接近0而不是0.5,这实际上使得下一层的学习变得更加方便

不同层对于激活函数的使用

所以在之后的学习中，我们不再使用$sigmoid$函数,因为$tanh$函数在几乎所有场合都更加优越,**但存在一个例外**,那就是**输出层**,因为我们希望输出的结果介于0到1之间,而不是-1到1,所以，使用$sigmoid$唯一的一个例外就是在进行二元分类输出层的时候

我们在神经网络中,不同层的激活函数可以不一样,有时候为了表示不同层的激活函数,我们仍然可能使用方括号上标来表示不同层的激活函数$g(z)$可能不同

ReLU函数

无论是$sigmoid$函数还是$tanh$函数,如果z非常大或者非常小,那么导数的梯度可能就很小,接近0,就会拖累**梯度下降算法**,所以我们在**机器学习**中有一个很受欢迎的函数叫做$ReLU$函数,表达式为

$$a = max(0, z)$$

所以只要z为正,导数就是1,当z为负时,斜率就为0

如果z刚好为0,那么导数是没有定义的,**但是在编程实现中**,你得到Z刚好为**0矩阵**的概率很低,所以在实践中并不需要担心这一点,而且也可以通过对z赋值,比如,当$z = 0$时,将其导数赋值为0或1,通过这样的操作也是可行的

选择激活函数时的经验法则

根据不同激活函数的特性,我们可以做出不同的激活函数选择
所以当我们在进行二元分类时,$sigmoid$函数就非常适合作为输出层的激活函数,然后其他所有隐藏层都可以使用$ReLU$函数,实际上这也是很多人的选择,虽然有时也有人选择$tanh$函数

而$ReLU$函数存在一个缺点,就是当z为负时,导数等于0,虽然这在实践中并没有什么问题,但是$ReLU$函数实际上存在另一个版本,叫做**带泄露的$ReLU$函数**,当z为负时,函数不再

为0,而是有一个很平缓的斜率,比如0.01,这个函数通常来说比$ReLU$函数更好,但实际上使用的频率并没有这么高

对于$ReLU$和**带泄露的**$ReLU$函数,有一个好处在于,对于很多z空间,激活函数的导数与0相差很远
所以在实践中使用$ReLU$函数,神经网络的学习速度通常会快很多,主要原因在于$ReLU$函数没有这种在梯度趋近于0时减缓学习速度的效应
当z为负时,$ReLU$函数的斜率为0,但在实践中,有足够多的隐藏单元令z大于0,所以对于大多数训练样本来说还是很快的

选择参数

在建立神经网络时通常有很多不同的选择,**比如隐藏单元数,激活函数,还有初始化权重**,在很多时候,很难去定下一个准则来确定什么参数最适合你的问题

为什么需要非线性激活函数

简单的理由

事实证明,如果想要让神经网络能够计算出有趣的函数,就必须使用非线性的激活函数

我们以正向传播的过程为例

假如我们直接令$g(z) = z$,这就是一个线性的激活函数,在学术上称为恒等激活函数,因为我们直接输出了输入值,那么这个模型的输出仍然只是我们输入的特征x的线性组合,我们的多层神经网络也不过是对线性组合的再输出

在**深度神经网络**,也就是含有许多隐藏层的神经网络中,如果我们使用线性激活函数,或者干脆没有激活函数,那么不论我们的神经网络有多少层,我们一直都只是在计算线性激活函数,所以不如直接去掉全部隐藏层

所以,如果**全部使用非线性激活函数**,那么这个模型的复杂度实际上与我们之间所做的只有一层的**标准logistic**回归是一样的

所以我们必须要引入**非线性激活函数**

一些例外

但我们仍然有地方可以使用线性激活函数,那就是如果我们需要学习的是**回归问题**,比如预测房价,那么输出是一个实数,而不是0和1,此时使用线性激活方程也许就是可行的,**但是在除了输出层以外的其他隐藏层**,我们仍然需要使用非线性的激活函数

所以,通常来说,**只有输出层**可以使用线性激活函数,还有可能就是一些与压缩相关的问题,除此之外,使用线性激活函数的情况非常少见

激活函数的导数

针对反向传播中的情况选择激活函数

首先我们可以来看$sigmoid$函数,对于任意给定的z,都会有相应的导数对应于这个值,前面已经计算过,对于任意的z的导数都有

$$g^{'}(z) = \sigma(z)(1 - \sigma(z))$$

如果z很大,那么$\sigma(z)$就会无限接近1,导数也就会无限接近0
同样的,当z非常小时,导数也会无限接近于0

我们来看$tanh$函数,我们可以有导数计算公式

$$g^{'}(z) = 1 - (tanh(z))^2$$

与$sigmoid$函数类似,当z很大或者很小时,导数都无限接近于0

最后来看$ReLU$函数以及带泄露的$ReLU$函数,对于$ReLU$函数,有导数

$$g^{'}(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geqslant 0 \end{cases}$$

正如之前所说，函数的导数在0上时实际上没有定义，但是在**编程实现时**实际上是没有问题的
因为z全部为0的可能性非常小,实际上将z设置为多少实际上无关紧要

同样的,对于带泄露的$ReLU$函数,就有

$$g^{'}(z)=\{\begin{matrix}0.01, & z<0\\ 1, & z\geqslant 0\end{matrix}$$

神经网络的梯度下降法

单隐层神经网络中的梯度下降法

我们的单隐层神经网络会有$W^{[1]},b^{[1]},W^{[2]},b^{[2]}$这些参数,一个神经网络的成本函数

首先假设我们是在进行二元分类,那么,成本函数的定义就会是

$$J(W^{[1]},b^{[1]},W^{[2]},b^{[2]})=\frac{1}{m}\sum_{i=1}^{n}L(A^{[2]},Y)$$

实际上这个函数和之前的logistic回归几乎完全一样

在训练神经网络时，随机初始化参数很重要，而不是全部初始化为0，当我们把参数初始化为某些值后，每个梯度下降的循环都会计算预测值，然后计算导数，实际上这一过程十分类似于之前的logistic回归

如何计算这些偏导数

我们知道了如何计算预测值，也知道如果使用向量化计算实现，所以现在我们需要计算这些导数
首先,我们给出单隐层神经网络正向传播的过程

$$Z^{[1]}=W^{[1]}A^{[0]}+b^{[1]}$$

$$A^{[1]}=g(Z^{[1]})$$

$$Z^{[2]}=W^{[2]}A^{[1]}+b^{[2]}$$

$$A^{[2]}=\sigma(Z^{[2]})$$

根据这一过程,我们给出反向传播的过程

$$dZ^{[2]}=\frac{1}{m}(A^{[2]}-Y)$$

$$dW^{[2]} = \frac{1}{m}A^{[1]T}(A^{[2]} - Y)$$

由于对于db的计算很难以公式的方式表达，所以这里使用编程时的代码实现表示

```
db = np.sum(dZ2,axis=1,keepdim=True)/m
```

使用的`keepdim=1`是为了保证输出的是一个矩阵,而不是数组,在这行代码中,我们输出的实际上是一个(1,1)的矩阵,但以后会考虑到一些多维的情况\

接下来我们需要计算第一层的导数
但是我们可以看到,对于$dZ^{[1]}$的计算公式是这样的

$$dZ^{[1]} = W^{[2]}dZ^{[2]} * g^{[1]'}(Z^{[1]})$$

公式中的"*"表示逐元素乘积 所以最后有

$$dW^{[1]} = \frac{1}{m}dZ^{[1]}X^T$$

```
db1 = np.sum(dZ1,axis = 1,keepdim = True)
```

随机初始化

为什么不能全部初始化为0

对于logistic回归,我们可以将参数初始化为0,但是如果将神经网络各参数数组全部初始化为0,那么梯度下降算法将会完全无效

我们先来看看如果将参数初始化为0会怎么样

如果我们将参数初始化为0,那么无论样本如何,我们得到的$A1$都会是**完全相同**每行,并且在计算反向传播时,同样的道理,$dZ1$的每一行也是完全相同的

将参数全部设置为0,这意味着**同一层的不同节点实际上是在做完全相同的计算**,效果等同于只有一个节点

因为每一次迭代后,我们会发现,更新后的数据,每一行的数据永远是相同的,这样我们会发现,实际上**无论有多少个隐藏单元,这些单元都在重复相同的计算,那么多个隐藏单元的设**

置实际上没有任何意义

随机初始化参数

所以,我们可以使用随机数来初始化参数

```
W = np.random.randn((n_h,n_x))*0.01
```

而对于b来讲并不存在这样的问题，所以我们可以将b初始化为0

```
b = np.zeros((n_h,1))
```

通常来说，权重矩阵应当设置为很小的随机值,因为如果我们使用的是$sigmoid$或者$tanh$等激活函数,W太大会使得激活值太大,从而导致梯度很小,算法的迭代速度严重降低

实际上,如果我们的激活单元中如果没有使用这两个函数,不用特地去将这些值设置为很小的值,但是**如果我们是在进行二分分类,那么输出单元的激活函数一定是$sigmoid$函数**,所以使用0.01作为系数是很合理的

其实我们还有比0.01更好用的常数,但我们训练的是单隐层的神经网络,是一个浅层的神经网络,所以0.01是可以使用的,但**如果我们训练的是一个非常深的神经网络**,那么就可能需要尝试一些其他的常数