[前言 3](#_Toc524971796)

[一. 结构 6](#_Toc524971797)

[1. 神经元 6](#_Toc524971798)

[2. 神经网络 8](#_Toc524971799)

[二. 激活函数 10](#_Toc524971800)

[0. 作用 10](#_Toc524971801)

[1. sigmoid 10](#_Toc524971802)

[2. tanh 12](#_Toc524971803)

[3. ReLU 13](#_Toc524971804)

[4. softmax 15](#_Toc524971805)

[三. 代价函数 17](#_Toc524971806)

[0. 作用 17](#_Toc524971807)

[1. 均方误差 17](#_Toc524971808)

[2. 交叉熵 19](#_Toc524971809)

[3. 对数似然 21](#_Toc524971810)

[四. 初始化 23](#_Toc524971811)

[0. 错误做法：全部初始化为0 23](#_Toc524971812)

[1. 均匀随机分布初始化 24](#_Toc524971813)

[2. 高斯随机分布初始化 25](#_Toc524971814)

[五. 优化 25](#_Toc524971815)

[1. 随机梯度下降 25](#_Toc524971816)

[2. 反向传播算法 28](#_Toc524971817)

[3. 改进的梯度下降 30](#_Toc524971818)

[六. 过拟合 36](#_Toc524971819)

[0. 原因 36](#_Toc524971820)

[1. L2正则化 36](#_Toc524971821)

[2. Dropout 38](#_Toc524971822)

[3. 拓展数据集 39](#_Toc524971823)

[七. 构建流程 40](#_Toc524971824)

[1. 类的定义与初始化 40](#_Toc524971825)

[2. 权重和偏置的随机初始化方法 41](#_Toc524971826)

[3. 代价函数和激活函数的定义 42](#_Toc524971827)

[4. 前向传播 43](#_Toc524971828)

[5. 预测方法 45](#_Toc524971829)

[6. 反向传播 47](#_Toc524971830)

[7. 优化器 48](#_Toc524971831)

[8. 拟合方法 49](#_Toc524971832)

[八. 超参数 50](#_Toc524971833)

[1. 网络形状 50](#_Toc524971834)

[2. 代价函数和激活函数的选择 52](#_Toc524971835)

[3. 学习率 52](#_Toc524971836)

[4. mini-batch大小 53](#_Toc524971837)

[5. 迭代次数上限 54](#_Toc524971838)

[6. 正则化系数 54](#_Toc524971839)

[九. 训练与测试 55](#_Toc524971840)

[1. 数据预处理 55](#_Toc524971841)

[2. 交叉验证 59](#_Toc524971842)

[3. 评估方式 60](#_Toc524971843)

[4. 监控训练过程 62](#_Toc524971844)

[十. 可视化 62](#_Toc524971845)

[1. cost/score变化曲线 62](#_Toc524971846)

[2. 结构图 64](#_Toc524971847)

[3. 拟合结果 70](#_Toc524971848)

[4. 神经元作用 72](#_Toc524971849)

[5. 神经元关注图像 76](#_Toc524971850)

[十一. 保存与加载 79](#_Toc524971851)

[1. 保存 79](#_Toc524971852)

[2. 读取 80](#_Toc524971853)

[十四. 回归 80](#_Toc524971854)

[1. 结构 80](#_Toc524971855)

[2. 激活函数 81](#_Toc524971856)

[3. 代价函数 81](#_Toc524971857)

[4. 数据预处理 82](#_Toc524971858)

# 前言

1. 该笔记目的是记录一些要点，不会详细展开，示例代码只展示核心部分。笔记中的公式是自己码的，补充说明中也包含一些个人想法，如果有错误或缺漏还请告知以期改正。

2. 开始学习神经网络前建议先掌握线性回归和逻辑回归。

推荐Andrew Ng的[Machine Learning](http://study.163.com/course/introduction/1004570029.htm)系列视频课程，对于线性回归和逻辑回归有着较为详细的讲解，但在神经网络上阐释得不够明白，针对神经网络入门，建议先去看[3blue1brown](https://space.bilibili.com/88461692/#/)的深度学习系列视频，简短但很清晰，之后拜读Michael Nielsen的[Neural Networks and Deep Learning](http://neuralnetworksanddeeplearning.com/index.html)（也有中文翻译版，自己找吧）。

3. 所需数学知识：微积分，线性代数，概率论(此处偏少)。

微积分和线性代数的入门学习同样推荐[3blue1brown](https://space.bilibili.com/88461692/#/)的系列视频。

4. 示例代码为python，建议直接使用[Anaconda](https://www.anaconda.com/download/)，省去安装大量科学计算包的时间。

介绍下可能会使用到的包：

[numpy](http://www.numpy.org/)（import numpy as np）

使用Python进行科学计算的基础包，包括：

* 一个强大的N维数组对象Array；
* 比较成熟的（广播）函数库；
* 用于整合C/C++和Fortran代码的工具包；
* 实用的线性代数、傅里叶变换和随机数生成函数。

[pandas](https://pandas.pydata.org/)（import pandas as pd）

提供高性能，易于使用的数据结构和数据分析工具。

[scipy](https://www.scipy.org/scipylib/index.html)（import scipy as sp）

高级的科学计算库，提供稀疏矩阵、插值、积分、优化、傅里叶变换等例程。

[matlibplot](https://matplotlib.org/)（import matplotlib.pyplot as plt）

Python 的 2D绘图库，它以各种硬拷贝格式和跨平台的交互式环境生成出版质量级别的图形。

[pillow](https://pillow.readthedocs.io/en/latest/)（import PIL）

PIL(Python Imaging Library)已经是Python平台事实上的图像处理标准库了。PIL功能

非常强大，但API却非常简单易用。由于PIL仅支持到Python 2.7，加上年久失修，于

是一群志愿者在PIL的基础上创建了兼容的版本，名字叫Pillow，支持最新Python

3.x，又加入了许多新特性。

[numba](http://numba.pydata.org/)(import numba as nb)

Numba使用直接用Python编写的高性能函数为您提供加速应用程序的能力。通过一些注释，面向数组和数学的Python代码可以及时编译为本机机器指令，性能类似于C，C ++和Fortran，无需切换语言或Python解释器。

Numba通过在导入时，运行时或静态（使用包含的pycc工具）使用LLVM编译器基础结构生成优化的机器代码来工作。Numba支持在CPU或GPU硬件上运行Python的编译，并且旨在与Python科学软件堆栈集成

5. 整理者相关：

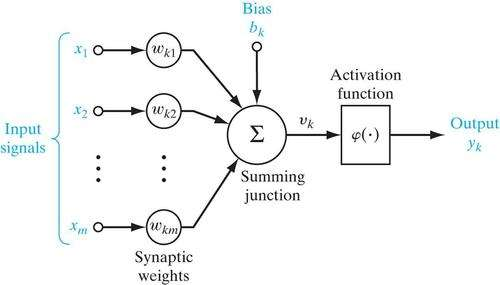
本人[github](https://github.com/jiedawang)上有自己实现的机器学习算法，欢迎围观指正。如果希望私下交流的话，

可以加qq群：609549026，这是一个新建的小圈子，目前还没人，期待热衷于自己实现算法的人加入。

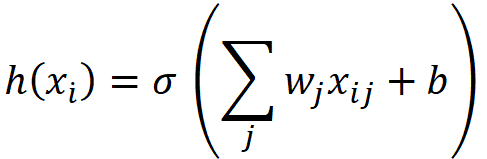
# 一. 结构

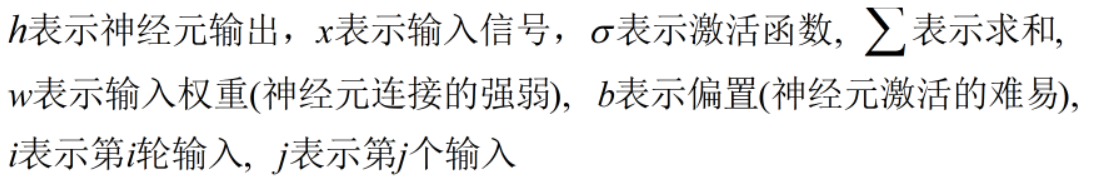
## 1. 神经元

### 图示：



### 公式：





### 公式（向量/矩阵）：

clipboard.png

clipboard.png

**python代码示例：**

h=activation(np.dot(X,w)+b)

### 说明：

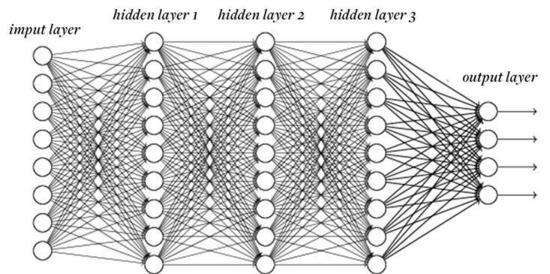
激活函数有多种选择，详细可参考第二章的激活函数部分，入门一般先使用sigmoid，使用sigmoid的单个神经元可以完成线性可分的二分类任务，与二分类逻辑回归非常相似。

python代码示例中向量/矩阵的数据结构和相关运算由numpy库提供，dot表示向量点积或矩阵乘法，示例代码中X的行对应数据记录，列对应属性或者说特征，这种组织方式更好理解，但由此得到的矩阵运算式也会有些区别。

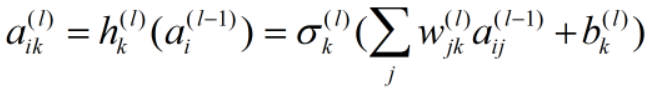
图示网上找的，其中一些表示符号可能与公式中的不一样，但表达的意思是一样的。

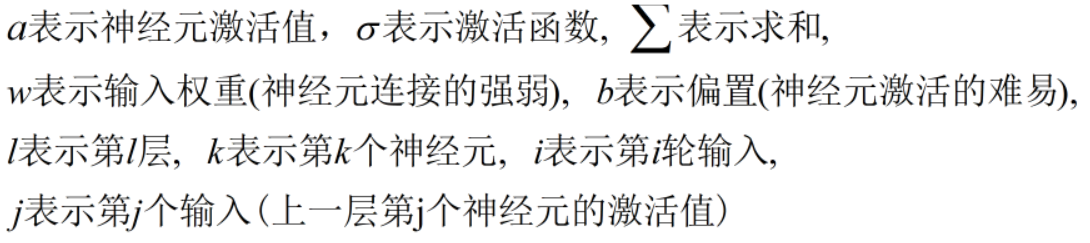
## 2. 神经网络

### 图示：

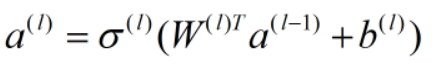


### 公式：





**公式（向量/矩阵）：**



### python代码示例：

for w,b in weights,biases:

a=activation(np.dot(a,w)+b)

### 说明：

公式与单个神经元差不多，增加 l 和 k 表示神经网络的层编号和当前层的神经元编号，第1层为输入层(该层只有激活值，不进行运算)，第L层为输出层(该层运算结果就是整个网络的输出)。

在矩阵形式中，当前层所有神经元的权重向量w合并为当前层的权重矩阵W。

在具体实现中，一般从输入层开始逐层计算每一层激活值直至输出层，这个过程被称为前向传播，神经元激活值的计算逻辑都是一样的，采用矩阵运算会带来更好的性能，同时也有两种做法，一种是一次只处理一个特征向量(单记录)，另一种是直接处理特征矩阵(多记录)，后者一般更高效。

在分类问题中，输出层每个神经元对应一个分类，整个输出层可以表达属于每个分类的概率，可以取最大概率的一个分类作为模型的输出。

笔记中的大部分内容是基于分类问题的，关于在回归问题上的应用请看第十四章节。

# 二. 激活函数

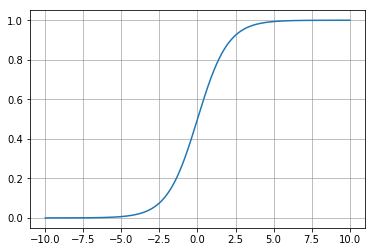
## 0. 作用

激活函数（Activation functions）负责将神经元的输入映射到输出端。激活函数将非线性特性引入到网络中，对于模型去学习、理解非常复杂和非线性的关系来说具有十分重要的作用。没有激活函数的神经网络每层只对输入进行线性组合，就算叠加了若干层，依旧是线性的。

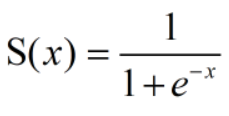
大部分激活函数在定义上是输入一个值输出一个值，属于一元函数。

## 1. sigmoid

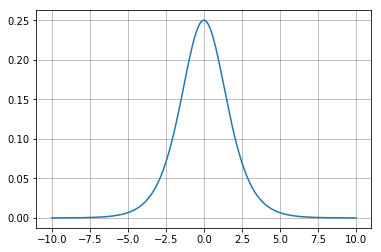
### 函数图像：



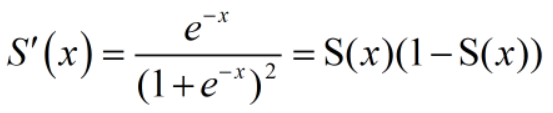
### 函数：



### 导函数图像：



### 导函数：



### python代码示例：

s=1./(1.+np.exp(-x))

ds=s\*(1-s)

### 说明：

sigmoid又称逻辑函数，值域为(0,1)，从图像可以看出，函数增长中间快两边慢，即神经元的激活值越小或是越大，变化都会变得越困难，神经元容易饱和，这往往会导致学习缓慢。

计算自然指数时使用np.exp(-x)比np.e\*\*(-x)更快。

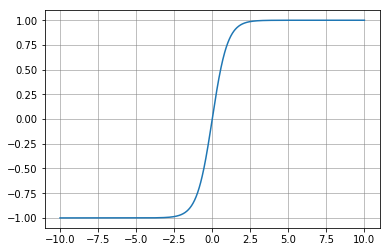
由于函数值域大于0，该激活函数不利于描述负相关。

sigmoid可以用于分类网络的输出层，用于表示属于该分类的概率。

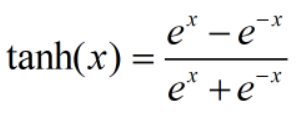
当输出层使用sigmoid时，代价函数使用交叉熵最合适。

## 2. tanh

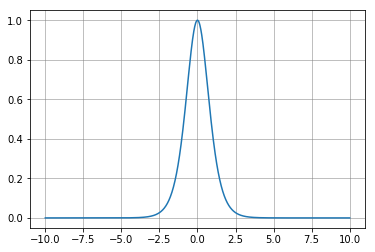
### 函数图像：



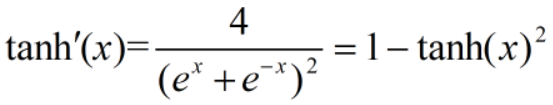
### 函数：



### 导函数图像：￼



### 导函数：



### python代码示例：

e=np.exp(x)

t=(e-1./e)/(e+1./e)

dt=1-t\*t

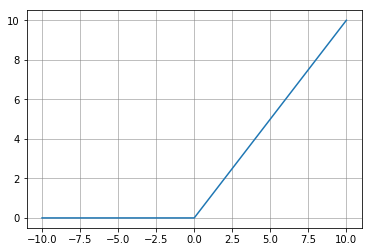
### 说明：

tanh又称双曲正切函数，值域为(-1,1)，函数图像和sigmoid很像，相当于sigmoid的一个改进版本，能够描述负相关，往往效果优于sigmoid。

tanh一般只用于隐含层，不用于输出层。

## 3. ReLU

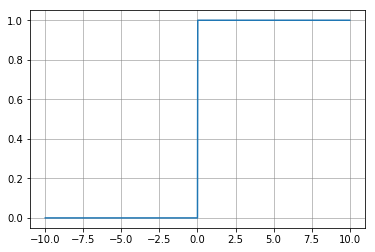
### 函数图像：



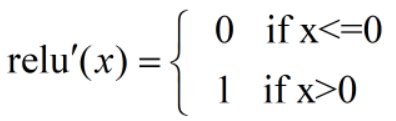
### 函数：

clipboard.png

### 导函数图像：￼



### 导函数：



### python代码示例：

r=x.copy()

r[r<0.]=0.

dr=np.zeros\_like(r)

dr[r>0.]=1.

### 说明：

ReLU又称线性整流函数，值域为[0,正无穷)。ReLU在输入为正数时不存在饱和问题，且计算上更快速，但也有缺点，输入是负数时，ReLU是完全不激活的，这就表明一旦输入负数，反向传播时梯度一直为0，该神经元就会死掉，一些ReLU的变种没有该问题；

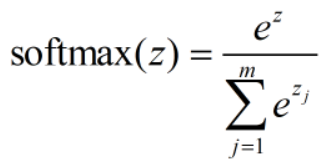
该激活函数能够避免深层网络中的梯度消失，梯度消失的原因可以结合反向传播的计算理解，反向传播过程中梯度的计算会不断乘上激活函数的导数，而总是小于1且多数情况下比1小得多的导数会让梯度值越来越小；

ReLU有很多改进的变种：比如负数输入也有输出的Leaky ReLU，考虑高斯噪声的Noisy ReLU等，可以根据实际需要去选择。

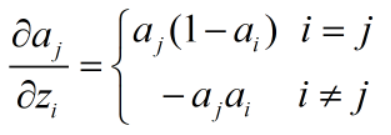
ReLU一般只用于隐含层，不用于输出层。

## 4. softmax

### 函数：



### 偏导函数：



### python代码示例：

e=np.exp(x)

a=(e.T/e.sum(axis=1)).T

a\_i=a.reshape((a.shape[0],a.shape[1],1))

a\_j=a.reshape((a.shape[0],1,a.shape[1]))

da=-a\_j\*a\_i

for i in range(a.shape[1]):

da[:,i,i]=a[:,i]\*(1-a[:,i])

### 说明：

softmax和sigmoid很相似，sigmoid可以认为是softmax的一种特例。softmax能将网络输出转换为总和为1的联合概率分布，但如果分类并不互斥则没有必要使用该激活函数。

softmax函数的输入输出都是向量，属于多元函数，和其他激活函数不同。

softmax单独计算梯度是不合适的，因为输入输出都是向量，所以偏导矩阵的计算和存储会占用很多资源，一般会将softmax和对数似然代价函数放在一起，使用化简后的公式计算输出层的梯度。

softmax只用于分类网络的输出层。

# 三. 代价函数

## 0. 作用

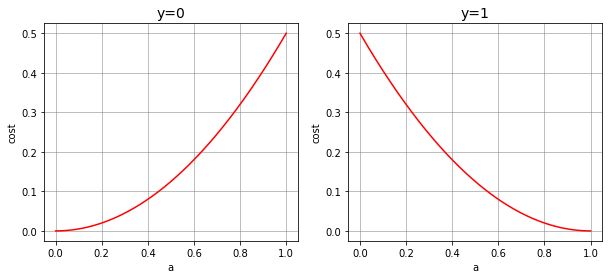
代价函数（cost function）用于衡量模型的预测值与真实值之间的误差大小，优化模型参数使得代价函数最小化是我们最基本的目标。 一个完整的优化目标函数由代价函数和正则化项组成（正则化项用于防止过拟合，这个之后会提到），整个目标函数用于衡量经验风险+结构风险，中间有一个参数用于平衡两者，定义好目标函数后就可以使用可行的优化算法对其优化以提高模型的表现。一开始时可以先不管正则化，直接将代价函数最小化作为优化目标。

另外还有一个名词叫损失函数（loss function），不同人对此的定义可能不一致，有的人是把代价函数和损失函数当作同一个东西，有的人则是认为损失函数定义在单个样本上，而代价函数是一个数据集上损失函数的平均，希望严谨一些可以采用后一种认识。

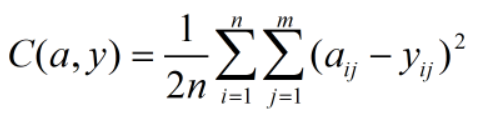
代价函数在定义上是输入两个向量输出一个值，属于多元函数。

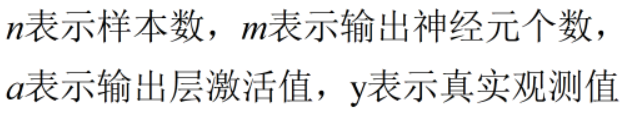
## 1. 均方误差

### 函数图像（单个样本单个特征）：

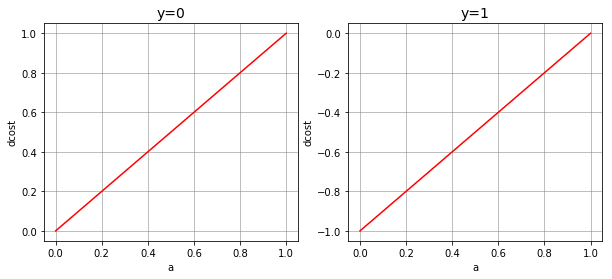


### 函数：

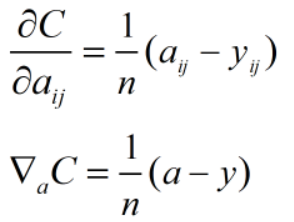




### 偏导函数图像（单个样本单个特征）：



**偏导数/梯度：**



### python代码示例：

re=a-y

c=0.5\*(re\*re).sum()/a.shape[0]

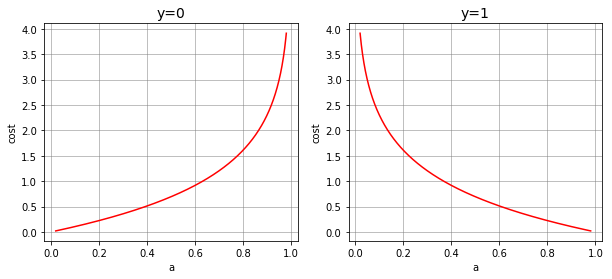
dc=re/a.shape[0]

### 说明：

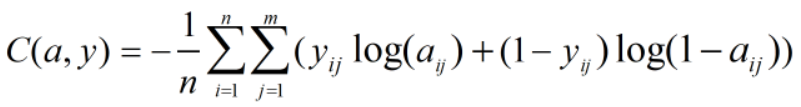
均方误差简称MSE，该代价函数在入门时会首先使用，计算每个样本在所有输出上误差的平方和，再对所有样本取平均，这种衡量误差的方式易于理解，平方和换成绝对值和也是一种可行的衡量方式，但绝对值不便于优化，一般还是使用平方和。

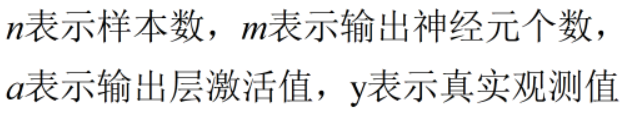
## 2. 交叉熵

### 函数图像（单个样本单个特征）：

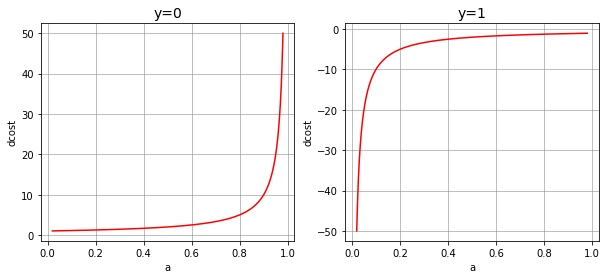


**函数：**

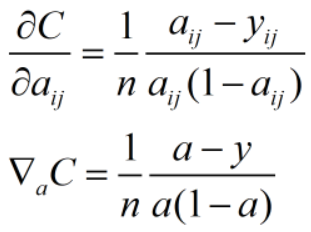




### 偏导函数图像（单个样本单个特征）：



### 偏导数/梯度：



### python代码示例：

c=-(y\*np.log(a)+(1-y)\*np.log(1-a)).sum()/a.shape[0]

dc=(a-y)/a/(1-a)/a.shape[0]

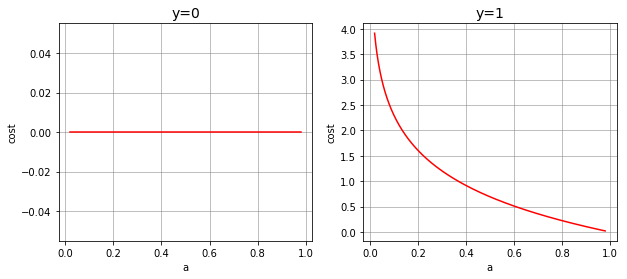
### 说明：

交叉熵代价满足预测偏差越大，代价增长越迅速，前期能够带来更快的学习速度；

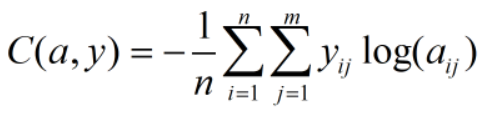
在输出层激活函数使用sigmoid时，该代价函数能够完美抵消输出层激活函数导数a(1-a)造成的 梯度值减小，不过，仅限输出层，所以隐含层使用relu，输出层使用sigmoid，代价函数使用交叉熵，是常见的一种组合。

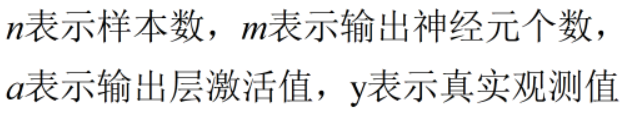
## 3. 对数似然

### 函数图像（单个样本单个特征）：

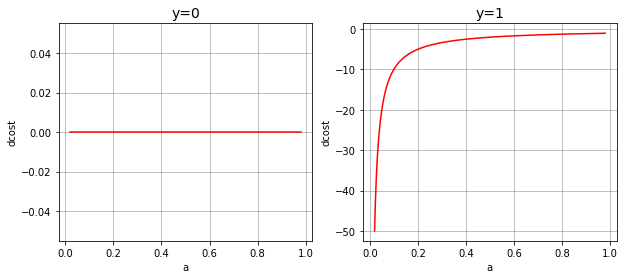


**函数：**

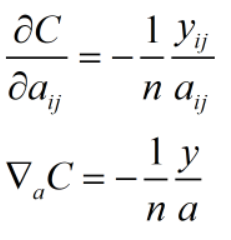




### 偏导函数图像（单个样本单个特征）：



### 偏导数/梯度：



### python代码示例：

c=-(y\*np.log(a)).sum()/a.shape[0]

dc=-y/a/a.shape[0]

### 说明：

对数似然函数和交叉熵函数类似，但只考了正确类损失，不考虑错误类损失。

对数似然代价一般搭配softmax输出层使用，当正确类的输出值a接近于1时，y=1，C接近于0，当输出值a距离a越大时，C值越大。

# 四. 初始化

## 0. 错误做法：全部初始化为0

### 说明：

权重和偏置全部初始化为0时，会导致输入无法对网络的输出产生影响，且所有神经元激活值都

是一样的，反向传播时计算得到的梯度每一层也会完全一样，每一层的所有神经元保持一样的更

新，网络失去非对称性，结果等同于每层只有一个神经元。

只要权重和偏置其中一个进行随机初始化就能避免这种情况，但通常都是优先将权重随机初始化，因为效果更好。只将偏置随机初始化，虽然能够使神经元激活值不一样，从而能正常地反向传播，但由于一开始输入无法对输出产生影响，导致没有充分利用输入信息，影响了初期的学习速度。

只要权重的初始化合适，偏置的初始化可以选择简单地置0。

## 1. 均匀随机分布初始化

### python代码示例：

#方法一

w=np.random.random(w\_shape)\*0.001

#方法二

w=np.random.random(w\_shape)-0.5

### 说明：

方法一生成的权重都是正数，输入值的加权和会是一个比较大的正数，如果使用sigmoid激活函数，激活值很容易聚集在1附近，这样会影响学习速度，所以通常会乘上一个很小的数，降低加权和的值，使激活值靠近中间导数较大的部分；

方法二将权重初始化为(-0.5,0.5)范围内的随机数，输入值的加权和大多会接近0，也能达到不 错的效果。

## 2. 高斯随机分布初始化

### python代码示例：

w=np.random.randn(w\_shape)/np.sqrt(w\_shape[0])

### 说明：

该方法将w初始化为均值为 0，标准差为 1/sqr(input\_n) 的高斯随机分布，能够加快收敛，有时也能带来准确率地提升，效果通常比均匀随机分布的初始化效果要好。

# 五. 优化

## 1. 随机梯度下降

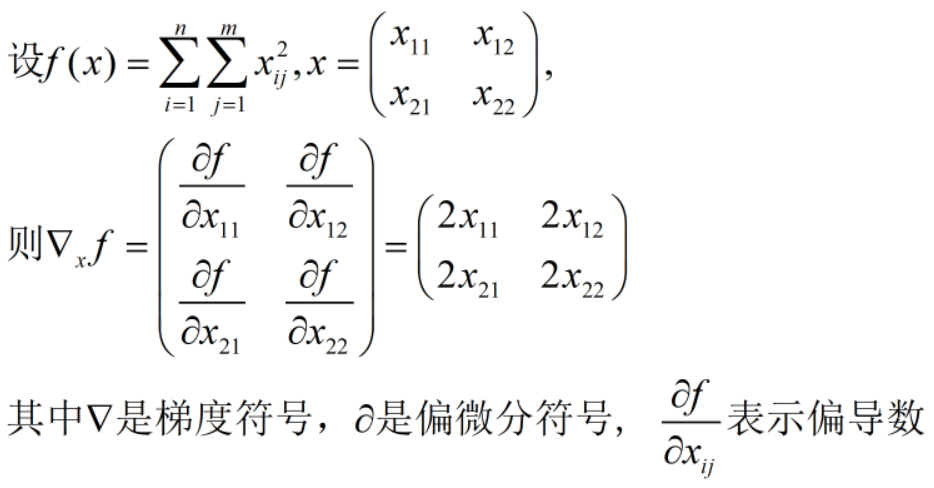
### 梯度下降：

梯度下降法是依赖梯度信息对参数进行迭代优化的方法，旨在逐步逼近局部最小值。

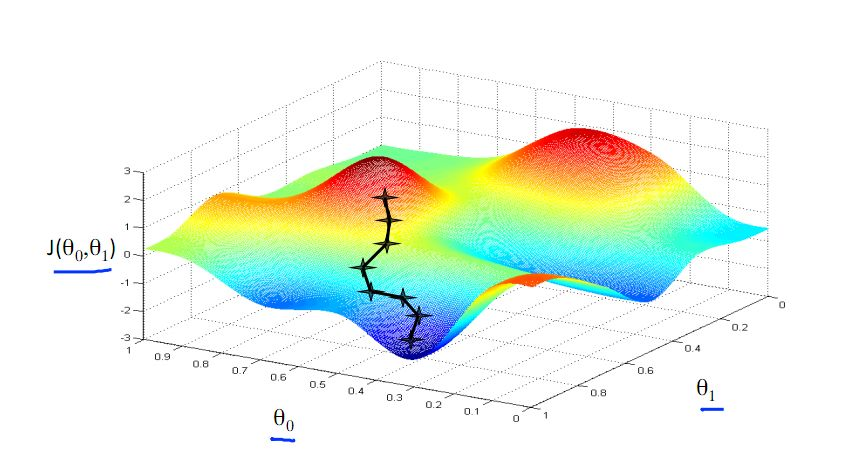
梯度是多元函数中自变量偏导数的集合，一般写作向量/矩阵形式，梯度指出一个方向，函数在该点处的方向导数沿着该方向取得最大值，即函数在该点处沿着该方向变化最快，如果函数图像是个二维坡面，该方向就是该点处坡面最陡峭的方向，沿该方向的负方向下坡速度最快。

偏导数，简单来说是对于一个多元函数，选定一个自变量并让其他自变量保持不变（即看作常量），只考察因变量与选定自变量的变化关系。

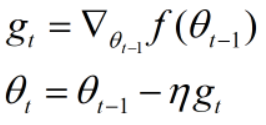
导数，简单来说就是"瞬时变化量"，这种说法并不严谨，但可以先这么理解，实际上导数是在考察自变量的微小变化对因变量的影响。比如汽车行驶距离关于时间的函数，其中某个时间点的导数是速度，二阶导数是加速度等等，而自变量与导数的关系也可以看作一个函数，称为导函数。



### 图示：



### 公式：



### 小批量：

实际应用中，由于数据量可能很大，每次下降都使用全部训练样本去计算梯度会很慢，所以一种通常的做法是将训练集分成一个个小的子集，逐个使用每个子集去计算梯度并优化。这样虽然会使梯度下降的方向有些偏差，但大致方向还是正确的，再加上很快的迭代速度，往往能更快收敛。

每次优化使用全部样本的称为 批量梯度下降(BGD)，使用单个样本的称为 随机梯度下降(SGD)，使用一个子集(mini-batch)的称为 小批量梯度下降(MBGD)，但现在一般都将小批量梯度下降称为随机梯度下降了。

### python代码示例：

for i in range(iter\_max):

#随机排序数据集

random\_idx=np.random.permutation(samples\_n)

X,Y=X[random\_idx],Y[random\_idx]

#使用每个小批量子集进行更新

for j in range(batches\_n):

X\_=X[j\*batch\_size:(j+1)\*batch\_size,:]

Y\_=Y[j\*batch\_size:(j+1)\*batch\_size,:]

#梯度计算

delta\_w,delta\_b=back\_prop\_(X\_,Y\_,weights,biases)

#更新权重和偏置

for j in range(len(delta\_b)):

weights[j]-=learning\_rate\*delta\_w[j]

biases[j]-=learning\_rate\*delta\_b[j]

### 说明：

神经网络中的梯度计算使用反向传播算法，详见第二节；

learning\_rate是学习率，用于控制梯度下降的步长，batch\_size是小批量子集大小，都属于超

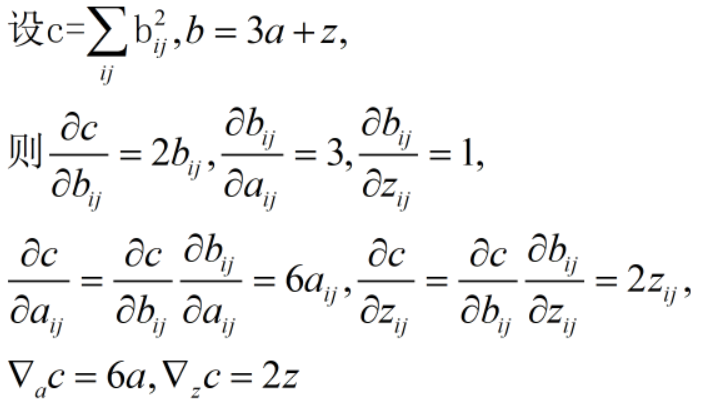
参数，需要调试，详见后面的超参数章节；

梯度下降逼近的是局部最小值，并不能保证达到全局最优，也有查找全局最优的算法（比如模拟退火），但通常计算开销大，而且不一定能应用在神经网络上。很多时候，局部极小值也能带来不错的结果了。

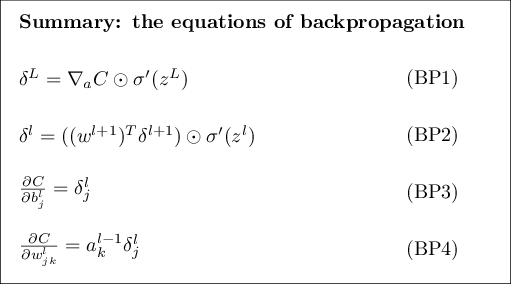
## 2. 反向传播算法

### 链式法则：

神经网络的作用相当于一连串嵌套的函数，每一层经过一个线性函数和一个激活函数，最后输出层再加一个代价函数，通过求导的链式法则可以从最后一层向前计算代价函数关于每一层权重和偏置的梯度，这种方法称为反向传播算法。



### 公式：



### python代码示例：

delta\_w,delta\_b=[],[]

cost,cost\_grad=cost\_(a[-1],Y,grad=True)

for i in range(len(a)-1,0,-1):

if i==len(a)-1:

delta\_z=cost\_grad\*a\_grad[i-1]

#输出层激活函数和代价函数是如下组合时，

#梯度计算可以合并化简为a-y:

#softmax+对数似然；sigmoid+交叉熵；无激活+均方误差

#delta\_z=a[-1]-Y

else:

delta\_z=np.dot(delta\_z,self.weights[i].T)\*a\_grad[i-1]

delta\_w.append(np.dot(a[i-1].T,delta\_z)/delta\_z.shape[0])

delta\_b.append(delta\_z.mean(axis=0))

delta\_w.reverse()

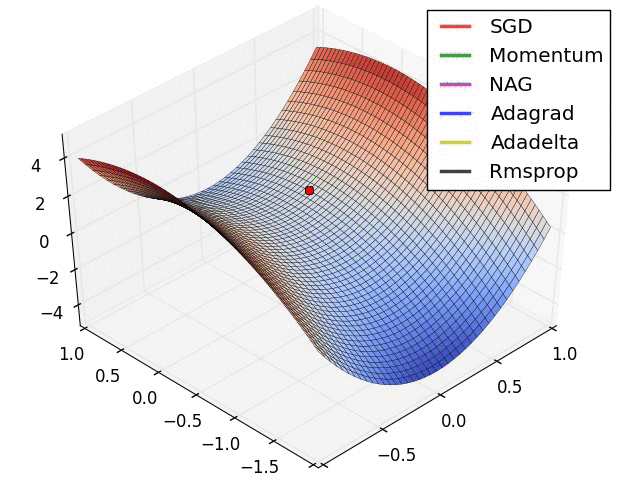
delta\_b.reverse()

### 说明：

此处的python实现是对mini\_batch的所有样本一起进行计算，最后再做平均，所以计算代价关于输出值的梯度时不需要除以样本数。一开始可以先尝试一次算一个样本。

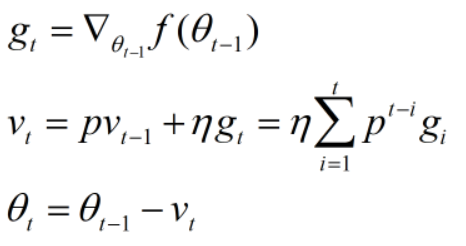
## 3. 改进的梯度下降

### 图示：

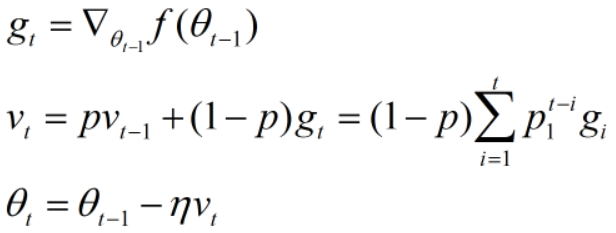


### (1) 动量梯度下降(Momentum)：

### 公式：



### 公式的一种变形：



### python代码示例：

#梯度计算

delta\_w,delta\_b=back\_prop\_(X\_,Y\_,weights,biases)

#更新权重和偏置

for j in range(len(delta\_b)):

v\_w[j]=p\*v\_w[j]+learning\_rate\*delta\_w[j]

v\_b[j]=p\*v\_b[j]+learning\_rate\*delta\_b[j]

weights[j]-=v\_w[j]

biases[j]-=v\_b[j]

### 说明：

该方法和sgd的区别就是利用了历史梯度信息，此处的更新量v就可以看作速度，当前速度与之前速度会在相同的方向上累积，不同的方向上抵消，结果就是速度的指向会被往极值点的方向牵引。

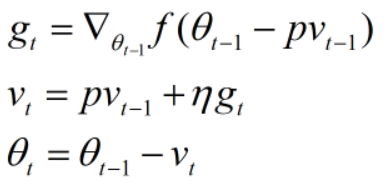
动量因子p控制对历史梯度的利用率，等于0时与常规sgd无异，等于1时会无限制的累积历史速度，作用有点类似于摩擦力带来的速度衰减。

该算法比sgd更像是小球滚下山坡过程的近似，但不严格，因为速度的变化不连续。

注意，v使用前需要初始化为0向量。

### (2) Nesterov加速梯度下降(NAGD):

### 公式：



### python代码示例：

#梯度计算

for j in range(len(delta\_b)):

weights[j]=weights[j]-p\*v\_w[j]

biases[j]=biases[j]-p\*v\_b[j]

delta\_w,delta\_b=back\_prop\_(X\_,Y\_,weights,biases)

#更新权重和偏置

for j in range(len(delta\_b)):

v\_w[j]=p\*v\_w[j]+learning\_rate\*delta\_w[j]

v\_b[j]=p\*v\_b[j]+learning\_rate\*delta\_b[j]

weights[j]-=learning\_rate\*delta\_w[j]

biases[j]-=learning\_rate\*delta\_b[j]

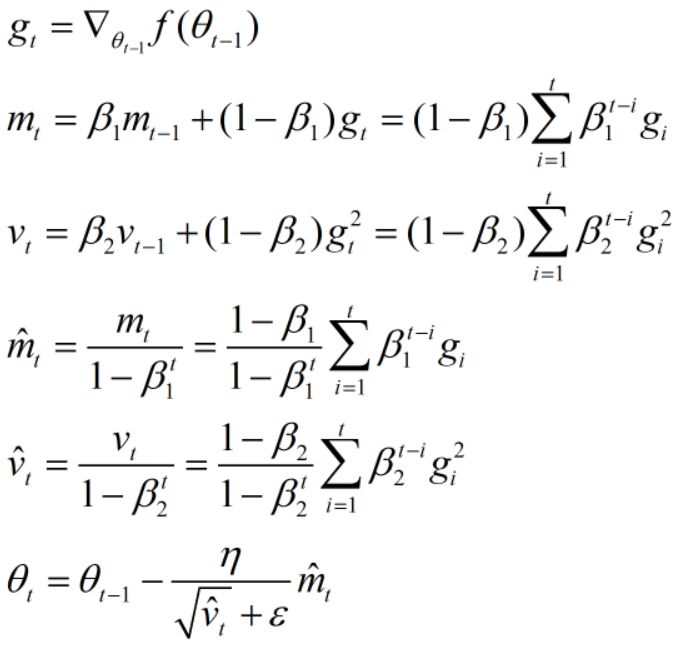
### 说明：

该方法是对动量法的改进，根据动量项修正当前梯度的计算，相当于预测了历史累积速度将把自己带到什么位置，再计算该位置的梯度，实现上就是先根据动量项进行一次参数更新，然后计算更新后位置的梯度，再更新剩余部分。

注意，v使用前需要初始化为0向量。

### (3)自适应矩估计(Adam):

### 公式：



### python代码示例：

#梯度计算

delta\_w,delta\_b=back\_prop\_(X\_,Y\_,weights,biases)

#更新权重和偏置

for j in range(len(delta\_b)):

m\_w[j]=beta1\*m\_w[j]+(1.-beta1)\*delta\_w[j]

m\_b[j]=beta1\*m\_b[j]+(1.-beta1)\*delta\_b[j]

v\_w[j]=beta2\*v\_w[j]+(1.-beta2)\*(delta\_w[j]\*\*2)

v\_b[j]=beta2\*v\_b[j]+(1.-beta2)\*(delta\_b[j]\*\*2)

#偏差校正

m\_w\_=m\_w[j]/(1-beta1\*\*t)

m\_b\_=m\_b[j]/(1-beta1\*\*t)

v\_w\_=v\_w[j]/(1-beta2\*\*t)

v\_b\_=v\_b[j]/(1-beta2\*\*t)

weights[j]-=learning\_rate/(np.sqrt(v\_w\_)+eps)\*m\_w\_

biases[j]-=learning\_rate/(np.sqrt(v\_b\_)+eps)\*m\_b\_

t+=1

### 说明：

一种自适应学习率算法，它利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率。

Adam的优点主要在于经过偏置校正后，每一次迭代学习率都有个确定范围，使得参数比较平稳。Adam多数情况下能表现得比较好，但不绝对，在没有时间精细调参时可以优先使用。

其实对照一下其他算法更好理解该算法，AdaGrad对不同参数应用不同的学习率，更新量的分母是所有历史梯度平方的和，即越频繁更新的参数分母会越大，学习率会变小，防止更新过头，更新不频繁的参数的学习率则会变大，加速更新，但有个问题是无限制的累积会导致分母整体都越来越大，学习率最后变得过小导致更新困难。RMSprop对此进行了改进，历史梯度平方的记忆有一个衰减率，这样就不会无限制地累积了。Adam在RMSprop的基础上又引入了动量项，即更新量的分子和momentum是差不多的，除此之外还引入了偏差校正，防止一开始更新量过小，经过偏差校正的m/v实际上是在求一个历史梯度/历史梯度平方的加权平均数，权重随时间衰减。

Adam的更新量受限于学习率，归一化的数据一般设置学习率0.001~0.01就行了，未归一化的数据可能需要更大的学习率(不推荐用未归一化的数据)。

该算法的作者建议beta1取默认值为0.9，beta2为0.999，eps为10−8。

注意，m和v使用前需要初始化为0向量，t初始化为1。

### (4) 其他：

Adagrad,RMSProp,Adadelta,

Adamax,Nadam

# 六. 过拟合

## 0. 原因

过拟合是因为模型太过复杂，导致足以强行记住所有训练用的个例，而没有去学习其中的普遍特征，导致泛化能力不强，在非训练数据上表现得不好，这不是我们想要的，所以需要去避免。

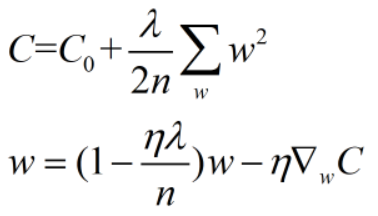
## 1. L2正则化

### 作用：

L2正则化是对优化的目标函数做了些改变，在代价函数的基础上引入了衡量模型复杂度的惩罚项：权重的平方和，并通过一个系数控制误差和复杂度之间的平衡。该项会导致梯度下降时权重不断衰减，这样模型会倾向于学习出更小的权重值，这意味着样本个例上的噪声对结果的扰动会更小，从而学习到更普遍的特征。

另一个使用权重的绝对值和的称作L1正则化，这种正则化会增加最重要的权重的值，其余的权重值衰减至0，可以用于特征选择。

### 公式：



### python代码示例：

#目标函数

cost=cost0+0.5\*l2\_alpha\*(w\*w).sum()/sample\_n

#梯度下降时增加衰减项

for j in range(len(delta\_b)):

delta\_w[j]+=l2\_alpha\*weights[j]/sample\_n

weights[j]-=learning\_rate\*delta\_w[j]

biases[j]-=learning\_rate\*delta\_b[j]

### 说明：

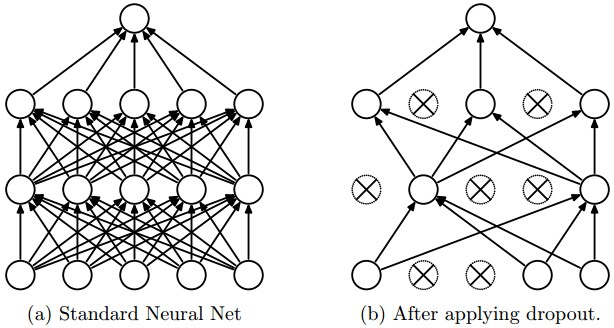
正则项在原版公式中需要除以样本数，因此带来的结果就是单纯地拓充训练数据集也要修改正则化系数，其实大可以将 正则化系数/样本数 合并为一个系数，这样直接设置一个较小的系数就行了。

## 2. Dropout

### 作用：

弃权(Dropout)，这是一种相对激进的方案，每次学习时随机屏蔽一半的神经元（输入输出层不需要屏蔽），只优化另一半的神经元，有点随机森林的思想在里面，这种做法会削弱神经元之间的依赖性，迫使模型去学习更普遍的特征，精确到个例的学习往往在这种机制下难以留存。

### 图示：



### python代码示例：

#前向传播时增加

mark=np.random.rand(a.shape[1])<dropout\_p

a\*=(mark/dropout\_p)

a\_grad\*=(mark/dropout\_p)

### 说明：

对每层的激活值应用dropout进行随机屏蔽（另一种做法是针对权重，但操作激活值更简便些），

dropout\_p是每个神经元被屏蔽的概率，为每个神经元生成一个0~1的随机数，小于dropout\_p的进行屏蔽，只作用于隐含层，输入输出层不需要屏蔽。对于未被屏蔽的神经元在前向传播时需要进行激活值的缩放，乘上dropout\_p即可，因为前向传播时是使用全部神经元进行计算的（计算完成后再进行屏蔽），而反向传播只更新未屏蔽的神经元，所以要保持数学期望上的一致，或者换种做法，在反向传播时未屏蔽的神经元的激活值和梯度除以dropout\_p，效果一样。

本人在实际尝试使用dropout时，并未得到预期的效果，反而收敛变慢了，最终得到的准确率也更低。除开代码可能有错误外，本人尚未尝试过复杂的网络，这或许是dropout不奏效的原因。

## 3. 拓展数据集

### 作用：

通过更多不一样的训练样本来避免模型去记忆训练样本个例也是一种有效的做法，但问题在于数据不是那么好获取的，进一步拓充数据集往往很困难。一种可行的做法是对原训练数据做一些处理，自己构造新的数据，这在图像数据集上还是比较好操作的，比如旋转，变色，加噪点等，有的数据则难以确定怎么处理合适，所以根据具体情况来选择是否使用该方案。

暂未尝试过该类操作，没有代码示例。

# 七. 构建流程

## 1. 类的定义与初始化

### python代码示例：

class MultilayerPerceptron:

def \_\_init\_\_(self,layers=(784,100,10),learning\_rate=0.01,iter\_max=100)

#此处可以增加参数校验

#其他初始化内容也可以放在此处

self.layers=layers

self.learning\_rate=learning\_rate

self.iter\_max=iter\_max

### 说明：

定义类并重写初始化方法，将配置信息保存在对象属性中以方便后面调用。参数校验根据实际需要来增加，只是展示核心逻辑供别人学习研究可以不加，如果要实际使用最好还是加一下，不然参数类型出错了会摸不着头脑。

多层感知器(MultilayerPerceptron)是全连接前馈神经网络的一种常见别名。

## 2. 权重和偏置的随机初始化方法

### python代码示例：

def random\_init\_(self):

w,b=[],[]

for i in range(len(self.layers)-1):

w\_shape=(self.layers[i],self.layers[i+1])

b\_shape=self.layers[i+1]

w\_=np.random.randn(w\_shape[0],w\_shape[1])/np.sqrt(w\_shape[0])

b\_=np.zeros(b\_shape)

w.append(w\_)

b.append(b\_)

self.weights,self.biases=w,b

### 说明：

权重和偏置的初始化可以放在\_\_init\_\_中，需要完整的定义了输入/隐含/输出层的参数（可以是一个参数，比如此处的layers，或是多个参数共同定义），也可以放在训练方法中，根据训练数据自动生成输入输出层形状。

## 3. 代价函数和激活函数的定义

### python代码示例：

#激活函数：sigmoid

#通过设置grad=True可以返回梯度信息

def sigmoid\_(self,z,grad=False):

a=1./(1.+np.exp(-1.\*z))

if grad==False:

return a

else:

return a,a\*(1-a)

#代价函数：均方误差

def mean\_sqr\_err\_(self,a,Y,grad=False):

re=a-Y

c=0.5\*(re\*re).sum()/Y.shape[0]

if grad==False:

return c

else:

return c,re

### 说明：

此处实现只展示了sigmoid激活函数和mse代价函数，一开始可以先固定为这两个，如希望自由设置除实现相应函数外还需定义相关参数，可在初始化时进行函数的绑定，或是在训练时根据设置选择相应的函数运行，建议前一种方式。

梯度值的计算可以和函数值放在一个方法中，也可以分开，分开可能会产生一些重复计算。

建议输出层激活函数和代价函数使用一些固定的搭配，比如 sigmoid+交叉熵，softmax+对数似然，无激活+均方误差 等。这样输出层在反向传播时的梯度计算可以化简为a-y，尤其是对于softmax这种激活函数，单独计算梯度是很浪费资源的，一个特征数n的向量计算得到的偏导矩阵大小为n\*n。

## 4. 前向传播

### python代码示例：

#前向传播

#a为输入神经元激活值，shape=(sample\_n,input\_size)

#return\_al=True时，返回所有层的激活值和梯度信息

def forward\_prop\_(self,a,return\_al=False):

layers\_n=len(self.layers)

if return\_al==False:

for i,w\_,b\_ in zip(range(1,layers\_n),self.weights,self.biases):

if i<layers\_n-1:

a=self.activation\_(np.dot(a,w\_)+b\_)

else:

a=self.output\_activation\_(np.dot(a,w\_)+b\_)

return a

else:

al,al\_grad=[a],[]

for i,w\_,b\_ in zip(range(1,layers\_n),self.weights,self.biases):

if i<layers\_n-1:

a,a\_grad=self.activation\_(np.dot(a,w\_)+b\_,grad=True)

else:

#输出层不单独计算梯度信息

a=self.output\_activation\_(np.dot(a,w\_)+b\_)

a\_grad=None

al.append(a)

al\_grad.append(a\_grad)

return al,al\_grad

### 说明：

此处实现中的activation\_和output\_activation\_就是激活函数，一开始可以直接替换为sigmoid函数，在可配置的情况下需要将配置中的指定的函数绑定到该对象属性上。

预测和训练时的前向传播不太一样，预测时隐含层的激活值和每层的梯度信息不是必须的，省略可以减少计算量和空间占用，而训练时是需要的，所以可以分成两部分。

输出层如前一节所说，激活函数和代价函数单独计算梯度不划算，但如果不使用softmax，也是可以单独计算并在反向传播时统一整合的。

## 5. 预测方法

### python代码示例：

#预测

def predict(self,X,return\_a=False):

#此处可以增加参数校验和数据预处理

#前向传播

a=self.forward\_prop\_(X)

#整合结果

if return\_a==False:

output=self.prob\_to\_label\_(a,self.classes)

else:

output=a.reshape((-1,)+self.output\_shape)

#只有一条记录时去除多余的维度

if output.shape[0]==1:

output=output[0]

return output

#将分类概率转换为分类标签

def prob\_to\_label\_(self,prob,classes):

prob\_max=np.argmax(prob.T,axis=0)

if len(classes)>0:

#向量返回向量，标量返回标量

if type(prob\_max)==type(np.array(1)):

classes=np.array(classes)

output=np.full(len(prob\_max),'').astype(classes.dtype)

for i in range(len(classes)):

output[prob\_max==i]=classes[i]

else:

output=classes[prob\_max]

#没有提供分类标签时返回分类索引

else:

output=prob\_max

return output

### 说明：

prob\_to\_label\_用于将分类概率矩阵转换为单列的分类标签向量，预测方法可以选择返回分类概率或分类标签，classes是所有分类标签，需要与输出层神经元一一对应，可以在训练时从训练数据中提取，也可以在预测时作为参数传入。

预测方法完成后整个网络已经可以实现预测了，不过由于还没优化结果会很糟糕，可以测试一下已有部分能否正常运行。

## 6. 反向传播

### python代码示例：

#反向传播

#反向逐层计算w和b的梯度

def back\_prop\_(self,al,al\_grad,Y):

delta\_w,delta\_b=[],[]

for i in range(len(al)-1,0,-1):

if i==len(al)-1:

#输出层的计算不一样

#输出层将代价函数和激活函数合并在一起计算梯度

delta\_z=al[-1]-Y

#单独计算后再合并是如下

#delta\_z=cost\_grad\*al\_grad[i-1]

else:

delta\_z=np.dot(delta\_z,self.weights[i].T)\*al\_grad[i-1]

delta\_w.append(np.dot(al[i-1].T,delta\_z)/delta\_z.shape[0])

delta\_b.append(delta\_z.mean(axis=0))

delta\_w.reverse()

delta\_b.reverse()

return delta\_w,delta\_b

### 说明：

反向传播根据链式求导法则计算输入样本在代价函数上关于每一层权重和偏置的平均梯度。基于矩阵的运算式容易让初学者晕头转向，建议配合矩阵的形状来确定计算的正确性，必要时写两个小矩阵验证一下。

## 7. 优化器

### python代码示例：

#优化方案：随机梯度下降

def sgd\_(self,X\_,Y\_):

#前向传播计算全部神经元的激活值和激活函数梯度信息

al,al\_grad=self.forward\_prop\_(X\_,return\_al=True)

#反向传播计算全部权重和偏置的更新量

delta\_w,delta\_b=self.back\_prop\_(al,al\_grad,Y\_)

#考虑学习率和L2正则化更新权重和偏置

for j in range(len(delta\_b)):

delta\_w[j]+=self.l2\_alpha\*self.weights[j]

self.weights[j]-=self.learning\_rate\*delta\_w[j]

self.biases[j]-=self.learning\_rate\*delta\_b[j]

### 说明：

此处优化器实现的是带L2正则化的随机梯度下降，仅根据反向传播的结果进行相应规则的更新，mini-batch的选取并没有放在此处。正则化项原定义中需要除以样本数，这里略去了，只通过一个系数控制。

## 8. 拟合方法

### python代码示例：

#拟合

def fit(self,X,y,test\_X=None,test\_y=None,

monitor\_cost=False,monitor\_score=False):

#此处可以增加参数校验和数据预处理

#迭代优化

for i in range(self.iter\_max):

#随机排序数据集

random\_idx=np.random.permutation(self.samples\_n)

X,Y,y=X[random\_idx],Y[random\_idx],y[random\_idx]

#使用每个小批量子集进行更新

for j in range(batches\_n):

#按batch\_size抽取小批量子集

X\_,Y\_=X[j\*batch\_size:(j+1)\*batch\_size,:],

Y[j\*batch\_size:(j+1)\*batch\_size,:]

#应用优化方案

self.optimizer\_(X\_,Y\_)

#此处可添加代价和评分变化的监控部分

#提前结束迭代和学习率衰减等机制也在此处

### 说明：

optimizer\_是优化器，一开始可以固定为sgd。batch\_size是mini-batch的大小，即每次优化使用的训练数据数量，为防止每个样本使用的频次差异过大，将训练集按batch\_size分割为一个个子集按个使用，使用完全部样本算作一轮迭代。数据预处理根据需要来设置，归一化/one-hot编码/数据集变形等都可以添加在此处。进度监控，提前结束迭代和学习率衰减在实现上没什么难度，具体机制自己斟酌，此处略。

Y是分类模式下y经过one-hot编码得到的，Y用于训练时的计算，而y用于进度监控中计算评分。拟合方法完成后就可以尝试训练了。

# 八. 超参数

## 1. 网络形状

### 作用：

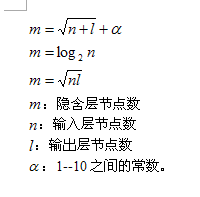
该参数用于控制网络中神经元的数量和分布，即定义输入层有几个神经元，隐含层有几层，每层有几个神经元，输出层有几个神经元。

### 建议：

可以简单地通过一个元组来定义，例如(784,16,16,10)，然后根据定义好地形状去初始化权重和偏置矩阵。如果希望更灵活些，也可以将输入和输出层的形状单独设置，例如根据RGB彩色图片数据的格式设置输入层形状为(height,width,3)，然后可以根据该设置对输入数据进行变形，转换为标准的(samples\_n,features\_n)形式，就可以同样地处理了。在此基础上，还可以不预先定义输入输出层形状，而是在训练时根据数据集形状自动生成。输入层的神经元个数根据输入数据的特征数量决定。输出层的神经元个数根据处理问题的类型决定，回归为一个，分类则与类别数量相同。

隐含层形状是真正需要去调试的变量，时间允许的话可以采用逐个增加并观察优化曲线的方式，同时一开始先缩减训练集大小以加快调试速度。通过一些经验公式来确定起点效率更高，详见下方的图。或许也可以从隐含层神经元较多的网络开始，然后应用一些防止过拟合的技巧进行训练，之后可以根据权重大小，筛选一些相对被孤立的神经元丢弃，这只是个人的一些猜想，尚未验证是否可行。

有一种有趣的算法叫做增强拓扑进化神经网络(NEAT)，是通过遗传算法去自动构建神经网络的结构，类似自然选择，但计算量大，不适合构建复杂的网络。



## 2. 代价函数和激活函数的选择

### 建议：

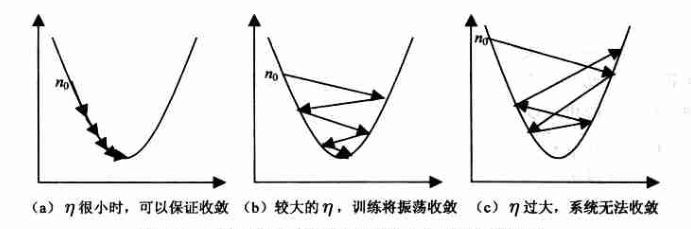
入门时一般激活函数选择sigmoid，代价函数选择均方误差，之后尝试别的函数以获得更好的效果。sigmoid和tanh都会导致梯度减小，所以在深层网络下效果不好，这时要考虑替换为relu或其变种。交叉熵代价通常会有比均方误差更好的表现，但计算上有溢出的可能，需要添加容错机制。

## 3. 学习率

### 作用：

用于控制迭代优化过程的步长，此处就是梯度下降的步长。步子大，学得快，但过大无法正常收敛，每一次更新都会跑到坡面的另一侧的更高处，越跑越远，偏离目标；步子小能够保证其稳定收敛，但需要的迭代次数也会变多，时间开销增加。所以需要选择一个适中的学习率。

### 图示：



### 建议：

除手动调整外，学习率衰减是个可行的方案，具体怎么设置需要斟酌，简单的做法是持续的衰减，只要迭代就会衰减，另一个可行的做法是监控cost变化，一开始先默认较高的学习率，这样可以在初期以较快的速度下降，当出现cost回升意味着靠近局部最低点了，这时按预设比率减小学习率，再上升再减小，这样就能相对稳定地靠近极值。至于默认学习率，衰减条件，衰减比率等怎么设置就要通过实际调试去确定了，算是更高层的超参数，但比起频繁地手动调整学习率还是省事多了。有时学习率过大，在某些数据集上迭代几次后cost计算就溢出了，这种异常需要捕获并处理，如不想直接结束而继续训练，除同样衰减学习率外最好权重偏置也重置一下，毕竟使cost计算溢出的权重和偏置偏离正确值太远了。

还有一种做法，使用自适应学习率的优化算法，如adadelta，该算法不需要设置全局学习率。

## 4. mini-batch大小

### 作用：

用于随机梯度下降的小批量子集大小，最小值为1，一次使用一个样本，最大值为训练集大小，一次使用全部训练样本，通常会去选择一个介于两者之间的合适的值。

### 建议：

如果训练集本来就小，那也不需要选择了，直接用全部样本就行。mini-batch指出的是梯度下降的大致方向，太小会使得梯度下降过程变得更不稳定，而且每次批量处理的样本数越少，训练速度也会越慢，极端情况下就是一个一个地去处理样本，效率可想而知，所以一般大小都设置在100以上。mini-batch太大同样影响效率，因为单次下降所需时间太长，虽然每步下降得更精确，但这个提升没法弥补迭代太慢带来的低效率。sklearn中选择的就是一个相对固定的值200，样本数小于这个值直接使用全量样本，不然就是200。

## 5. 迭代次数上限

### 作用：

控制迭代优化次数的上限，因为不能让模型无止境地训练下去，如果一定迭代次数后仍不能达到预期的效果就没必要继续执行了，此时应该调整参数重新训练。

### 建议：

通过一些机制使不满足要求的模型提前结束迭代是一个常用的方案，例如cost下降幅度没有达到设定的阈值，连续n次迭代cost没有进一步降低，cost连续升高，测试集表现一直没有提升等等，这样默认迭代次数可以设一个较大的值，且不需要频繁更改。

## 6. 正则化系数

### 作用：

控制目标优化函数中误差代价和模型复杂度代价之间的平衡，值越大，学习到的模型越简单，过大容易欠拟合，值越小，对训练集误差的容忍度越低，过小容易过拟合。

### 建议：

按照原本的定义，正则化系数乘上学习率是一轮迭代结束后权重衰减的比率，而这个值除以总样本数就是平均到每个样本的衰减比率，该系数需要根据训练样本集大小而不断调整，可以考虑合并 正则化系数/样本数 这一项，简化设置。

# 九. 训练与测试

## 1. 数据预处理

### (1) 特征缩放(归一化)：

将所有特征值缩放到一个较小的取值范围内但保留大小关系，目的是消除量纲，不同特征之间取值范围差异过大，会导致计算出来的梯度取值范围差异也过大，导致在不同方向上的更新速度不平衡，很难去选择一个合适的学习率。

### python代码示例：

#获取缩放的参照标准，一般为输入训练集X

def scaler\_reference(X):

ref=pd.DataFrame()

ref['min']=X.min()

ref['max']=X.max()

ref['mean']=X.mean()

ref['std']=X.std()

return ref

#两种不同的缩放方式

def minmax\_scaler(X,ref):

return (X-ref['min'])/(ref['max']-ref['min'])

def standard\_scaler(X,ref):

return (X-ref['mean'])/ref['std']

### (2) 特征映射：

通过特征之间的组合计算构造新的特征，比如原本有x1,x2两个特征，可以再构造x1\*x2,sin(x1)\*cos(x2)之类的特征。神经网络本身就具备处理非线性问题的能力，所以对特征映射的需求相对较少。

### (3) 缺失值处理：

有很多可选的方法，最简单粗暴的做法就是直接剔除带空值的记录，但如果数据量本来就少，再去剔除部分数据会造成负面影响。另一种简单的做法就是均值填充，即连续数值取均值，离散数值取出现频次最高的，在一定程度上使缺失位置的值具有普遍性，在训练中产生的干扰也会比较小。

### (4) one-hot：

又称独热编码、一位有效编码，是将有k种取值的离散变量编码为k列取值状态的技巧，编码后每列对应一个取值，用0或1表示该行数据是否是该取值，每行只有一个1，起到离散变量连续化的作用。一般用于处理离散特征或目标类别。

### python代码示例：

values=np.unique(array)

result=np.zeros((len(array),len(values))).astype('int')

for i in range(len(values)):

result[array==values[i],i]=1

### (5) 图片数据操作：

主要使用PIL库，涉及图片的创建/读取/缩放/保存/显示，灰度图和彩色图的像素点绘制，与numpy array的转换，灰度图和彩色图之间的转换,更复杂的尚未尝试。

### python代码示例：

from PIL import Image

#图片形状

shape=(height,width)

#创建灰度/彩色图片

image1=Image.new('L',shape)

image2=Image.new('RGB',shape)

#缩放

image=image.resize((100,100))

#读取和保存

image=Image.open(file\_path)

image.save(file\_path,'png')

#显示

#(在ipython直接输入图片对象名也是可以显示图片的)

plt.imshow(image)

#添加灰度/彩色像素点

#第一个参数为位置，第二个参数为灰度值或RGB值

#单个像素点的操作效率不是很高，转换为numpy数组进行批量操作更高效

image1.putpixel((x,y),l)

image2.putpixel((x,y),(r,g,b))

#与numpy的转换

#单张灰度图片的数据格式:(height,weight)

#单张彩色图片的数据格式:(height,weight,3)

array=np.asarray(image)

image=Image.fromarray(np.uint8(array))

#灰度图与彩色图的转换

#彩色图转换为灰度图后色彩信息就丢失了，再转换回来还是灰色调

image1=image1.convert('RGB')

image2=image2.convert('L')

#也可以尝试通过numpy计算进行转换

image1\_=np.asarray(image1).astype('float64')

image1\_=image1\_.repeat(3).reshape(image1\_.shape+(3,))

image=Image.fromarray(np.uint8(image1\_))

image2\_=np.asarray(image2).astype('float64')

image2\_=(11\*image2\_[:,:,0]+16\*image2\_[:,:,1]+5\*image2\_[:,:,2])/32

image2=Image.fromarray(np.uint8(image2\_))

## 2. 交叉验证

### 作用：

为了能够评估模型的泛化能力，会将数据集拆分为训练集和测试集(比例一般是4:1)，使用训练集训练模型，然后用测试集去评估模型，过拟合现象需要通过该种方式才能得到监控。

除将数据拆分成两份的做法外，还有更严格的k折交叉验证，将数据集均分成k份，挨个取一份作测试集，其余的作训练集，重复k次验证，这种做法比较耗时，时间充足且追求高质量时可以使用。

### python代码示例：

#numpy

random\_idx=np.random.permutation(data.shape[0])

split\_idx=math.ceil(0.8\*len(data))

train\_X,train\_y=data[random\_idx[:split\_idx],:-1],data[random\_idx[:split\_idx],-1]

test\_X,test\_y=data[random\_idx[split\_idx:],:-1],data[random\_idx[split\_idx:],-1]

#pandas

train=data.sample(frac=0.8,random\_state=None)

test=data[~data.index.isin(train.index)]

train\_X,train\_y=train.iloc[:,:-1],train.iloc[:,-1]

test\_X,test\_y=test.iloc[:,:-1],test.iloc[:,-1]

## 3. 评估方式

### (1) 准确率：

正确分类样本数/全部样本数，最基本的评估方式。

### python代码示例：

score=y[y==pred\_y].shape[0]/y.shape[0]

### (2) 混淆矩阵：

也称误差矩阵，是表示精度评价的一种标准格式，用n行n列的矩阵形式来表示。

混淆矩阵的列和行分别对应预测类别和真实类别，每一个值对应将某个类别预测为某个类别的数量，通过该矩阵可以得知一个类别容易被错分为什么类别以作针对性的调整。

### python代码示例：

pred\_dist=np.zeros((len(classes),len(classes)))

for i in range(len(classes)):

for j in range(len(classes)):

bool\_index=(y==values[j])&(pred\_y==values[i])

pred\_dist[i][j]=len(y[bool\_index])\*1.0/len(y)

pred\_dist=pd.DataFrame(pred\_dist,

columns='y\_'+pd.Series(values).astype('str'),

index='p\_'+pd.Series(values).astype('str'))

## 4. 监控训练过程

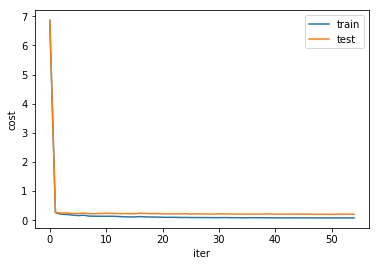
监控训练过程中的cost和score的变化，在此之上可以应用诸如学习率衰减，提前结束迭代等自动化方案。监控一般基于训练集或测试集的全部数据，而非一个mini-batch，所以会对训练速度有些影响，但可靠的监控会减少调试的次数，整体上还是值得的。

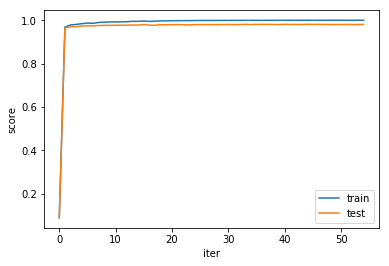
# 十. 可视化

（第一个是必须的，其余可以跳过）

## 1. cost/score变化曲线

### 图示：





### python代码示例：

#在优化过程中记录cost和score

for i in range(iter\_max):

...(此处进行优化)

cost1=compute\_cost\_(X,Y)

cost2=compute\_cost\_(test\_X,test\_Y)

cost\_h.append([cost1,cost2])

score1=compute\_score\_(X,y)

score2=compute\_score\_(test\_X,test\_y)

score\_h.append([score1,score2])

cost\_h=pd.DataFrame(cost\_h,columns=['train','test'])

score\_h=pd.DataFrame(score\_h,columns=['train','test'])

#cost优化曲线

cost\_h.iloc[:,:].plot()

plt.xlabel('iter')

plt.ylabel('cost')

plt.show()

#score优化曲线

score\_h.iloc[:,:].plot()

plt.xlabel('iter')

plt.ylabel('score')

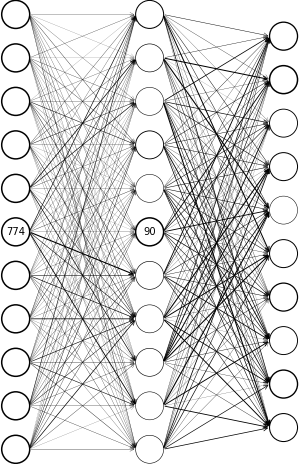
plt.show()

### 说明：

此种可视化最简单但适用范围也是最广的，可以用于评估超参数是否合适，多数情况下都会使用。训练过程中将每次迭代的cost/score计算出来后存在dataframe里，然后直接调用.plot()即可绘制。dataframe是pandas的一种数据结构，是对二维的numpy array的做了进一步的封装。

## 2. 结构图

### 图示：



### python代码示例：

#绘制一层神经元

def plot\_layer(ax,weights,biases,layer,height,first=False):

#当前层神经元数量/前一层神经元数量

nn,pnn=weights.shape[1],weights.shape[0]

#屏蔽大部分神经元的显示

unshow,unshow\_p=0,0

show\_max=10

if nn>show\_max:

unshow=nn-show\_max

nn=show\_max+1

if pnn>show\_max:

unshow\_p=pnn-show\_max

pnn=show\_max+1

if height>show\_max:

height=show\_max+1

#计算偏移(x是横向轴，y是纵向轴，左下角是原点)

x\_off=layer\*0.4

y\_off=(height-nn)/2

y\_off\_p=(height-pnn)/2

#绘制第一层隐含层前先绘制输入层

if first==True:

for i in range(pnn):

plot\_neuron(ax,None,1,y\_off\_p,(x\_off-0.4,0.2\*(y\_off\_p+pnn-i-1)))

if (i==show\_max/2)&(pnn==show\_max+1):

ax.text(x\_off-0.4,0.2\*(y\_off\_p+pnn-i- 1),str(unshow\_p),va="center",ha="center")

#遍历该层神经元

if pnn==show\_max+1:

weights=np.r\_[weights[:int(show\_max/2+1)],weights[-int(show\_max/2):]]

if nn==show\_max+1:

weights=np.c\_[weights[:,:int(show\_max/2+1)],weights[:,-int(show\_max/2):]]

biases=np.r\_[biases[:int(show\_max/2+1)],biases[-int(show\_max/2):]]

biases[int(show\_max/2)]=1

weights=np.abs(weights)

weights=weights/weights.max()

biases=np.abs(biases)

if biases.max()==0:

biases[:]=1

else:

biases=biases/biases.max()

for j in range(nn):

plot\_neuron(ax,weights[:,j],biases[j],y\_off\_p,(x\_off,0.2\*(y\_off+nn-j-1)))

if (j==show\_max/2)&(nn==show\_max+1):

ax.text(x\_off,0.2\*(y\_off+nn-j-1),str(unshow),va="center",ha="center")

#绘制一个神经元

def plot\_neuron(ax,w,b,y\_off\_p,xy,text=' '):

#绘制该神经元=

style\_neuron = dict(boxstyle="circle", color='white', ec='black',lw=0.5+b)

ax.annotate(text,xy=(0,0),xycoords='axes fraction',

xytext=(xy[0],xy[1]),

textcoords='axes fraction',va="center",ha="center",

bbox=style\_neuron,fontsize=15)

#绘制该神经元的所有输入连接

if type(w)!=type(None):

pnn=w.shape[0]

for i in range(pnn):

style\_connect = dict(arrowstyle="<-", color='black',lw=0.1+w[i])

ax.annotate('',xy=(xy[0]-0.4+0.04,0.2\*(y\_off\_p+pnn-i-1)),

xycoords='axes fraction',

xytext=(xy[0]-0.04,xy[1]),

textcoords='axes fraction',

va="center",ha="center",arrowprops=style\_connect)

#绘制神经网络

def plot\_network(model):

layers=model.layers

height=max(layers)

plt.figure(1,facecolor='white')

axprops=dict(xticks=[], yticks=[])

ax=plt.subplot(111,frameon=False,\*\*axprops)

for i in range(len(model.weights)):

if i==0:

plot\_layer(ax,model.weights[i],model.biases[i],i,height,first=True)

else:

plot\_layer(ax,model.weights[i],model.biases[i],i,height,first=False)

plt.show()

#绘制结构图

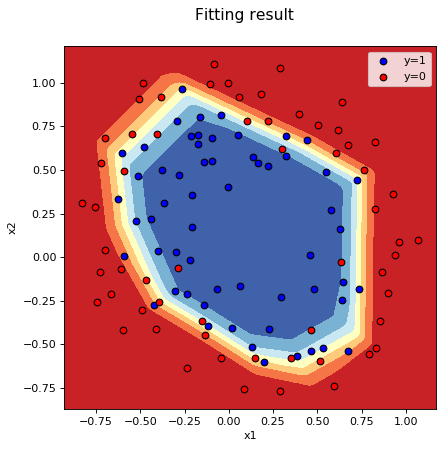
plot\_network(mlp0)

### 说明：

该可视化方法用于理解和展示神经网络的构造。在神经元数量很多时，全量绘制会很耗时而且观感很差，所以超出指定数量的神经元建议屏蔽显示。如果做成桌面小程序，可以用于动态展示训练时神经网络的变化，但只是好看而已，对调试模型没太大帮助。可以根据权重的正负将连线绘制成不同的颜色。

## 3. 拟合结果

### 图示：



### python代码示例：

#X值域

x1\_min,x1\_max=X[:,0].min(),X[:,0].max()

x2\_min,x2\_max=X[:,1].min(),X[:,1].max()

#生成网格数据，用于绘制等高线图或曲面图

X0=np.zeros((101\*101,2))

X0\_1,X0\_2=np.mgrid[

x1\_min-0.1:x1\_max+0.1:101j,

x2\_min-0.1:x2\_max+0.1:101j

]

X0[:,0]=X0\_1.reshape(101\*101)

X0[:,1]=X0\_2.reshape(101\*101)

#设置图像大小

plt.figure(figsize=(6,6),dpi=80)

#模型预测等高线图

y0=mlp0.predict(X0,return\_a=True)

y0=y0.reshape(101,101,2)

plt.contourf(X0\_1,X0\_2,y0[:,:,1],cmap=plt.cm.RdYlBu)

#数据集散点图

plt.scatter(X[:,0][y==1],X[:,1][y==1],c='b',edgecolors='black',label='y=1')

plt.scatter(X[:,0][y==0],X[:,1][y==0],c='r',edgecolors='black',label='y=0')

plt.legend()

plt.xlabel('x1')

plt.ylabel('x2')

plt.suptitle('Fitting result',fontsize=14,y=0.96)

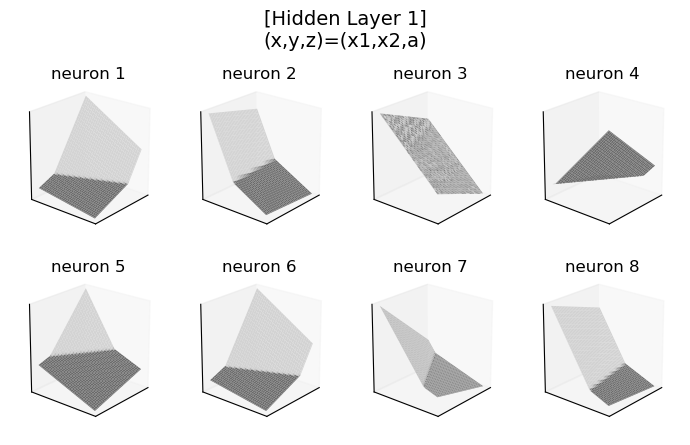
plt.show()

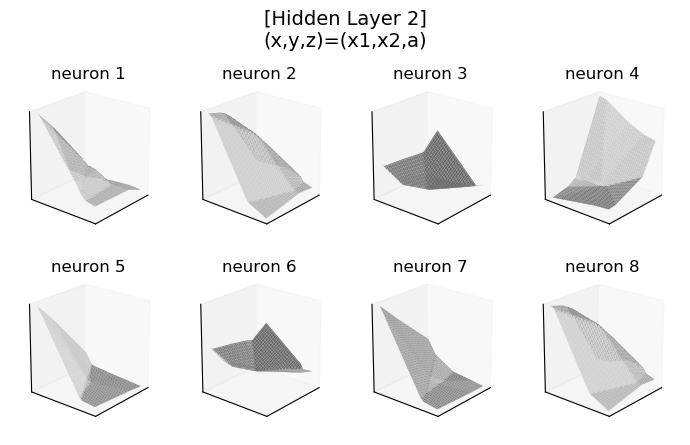
### 说明：

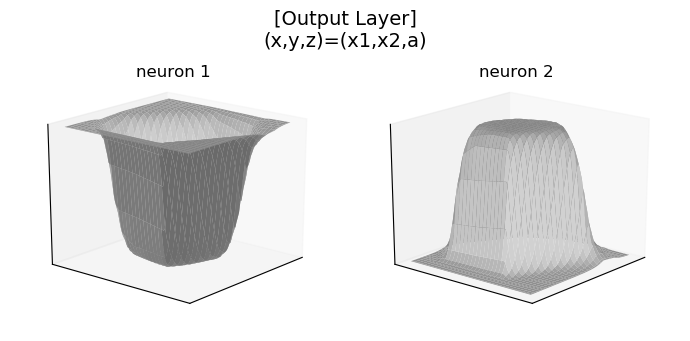
该可视化方法用于观察拟合的效果。通过绘制数据集的散点图和模型的决策平面，可以直观的展示两者的契合程度。限制较大，仅限有两个特征的数据集。比较适合学习研究算法性质以及向别人教授或展示时使用。如果做成桌面小程序，可以动态展示拟合状态的变化。

## 4. 神经元作用

### 图示：







### python代码示例：

#神经元作用的曲面图

def plot\_surface(fig,X,Y,Z,nrows=1,ncols=1,index=1,title='',elev=None,azim=None):

axes1=fig.add\_subplot(nrows,ncols,index,projection='3d')

axes1.set\_title(title,fontsize=12)

#拟合曲面图

axes1.plot\_surface(X,Y,Z,color='lightgrey',alpha=1.0)

axes1.set\_xticks([])

axes1.set\_yticks([])

axes1.set\_zticks([])

axes1.view\_init(elev=elev, azim=azim)

#绘制所有神经元的作用曲面图

def plot\_neurons\_action(model,X):

x1,x2=X[:,0].reshape(101,101),X[:,1].reshape(101,101)

al,al\_grad=model.forward\_prop\_(X,return\_al=True)

for k in range(1,len(al)):

size=al[k].shape[1]

row=math.ceil(size/4)

subplot\_idx=[]

for i in range(1,row+1):

for j in range(1,5):

idx=4\*(i-1)+j

if idx>size:

break

if row==1:

subplot\_idx.append((1,size,idx))

else:

subplot\_idx.append((row,4,idx))

fig=plt.figure(figsize=(7,2.0+1.0\*row),dpi=100)

a=al[k].reshape((101,101,size))

for rol,col,idx in subplot\_idx:

plot\_surface(fig,x1,x2,a[:,:,idx-1],rol,col,idx,

'neuron %d'%idx,elev=20, azim=40)

if k==len(al)-1:

plt.suptitle('[Output Layer]\n(x,y,z)=(x1,x2,a)',fontsize=14,y=1.12)

else:

plt.suptitle('[Hidden Layer %d]\n(x,y,z)=(x1,x2,a)'%k,fontsize=14,y=1.10)

plt.tight\_layout()

plt.show()

#X0的生成同上一小节

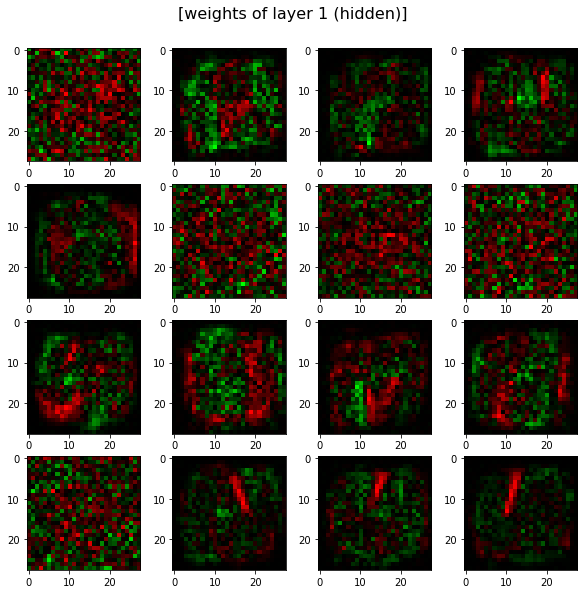
plot\_neurons\_action(mlp0,X0)

### 说明：

该可视化方法用于理解神经元处理非线性问题的能力。从图示中可以看出，第一个隐含层基本是没有处理非线性问题的能力的，激活值与输入特征的关系曲面就是一个弯折的平面(ReLU激活函数)，但经过每一层对上一层结果的组合，可以逐步逼近复杂的关系曲面。限制较大，仅限有两个特征的数据集。比较适合学习研究算法性质以及向别人教授或展示时使用。如果做成桌面小程序，可以动态展示神经元作用的变化。

## 5. 神经元关注图像

### 图示：



### python代码示例：

#神经元关注图像（绿色正相关，红色负相关）

def w\_plot(model,layer,w\_p):

if layer==0:

print('input layer can not be visualized')

return None

elif layer==len(model.biases):

layer\_type='output'

else:

layer\_type='hidden'

row=math.ceil(model.weights[layer-1].shape[1]/4)

plt.figure(figsize=(10,2.4\*row))

plt.suptitle('[weights of layer %d (%s)]'%(layer,layer\_type),

fontsize=16,y=0.95-0.0025\*row)

if layer==1:

w\_p=model.weights[layer-1]-model.biases[layer-1]/(28\*28)

else:

w\_p=np.dot(w\_p,model.weights[layer-1])-model.biases[layer-1]/(28\*28)

for i in range(model.weights[layer-1].shape[1]):

w\_=w\_p[:,i].reshape(28,28)

scalar=256/np.abs(w\_).max()

image=Image.new('RGB',(28,28))

for y in range(28):

for x in range(28):

if w\_[x,y]>0:

r,g=0,int(w\_[x,y]\*scalar)

else:

r,g=int(-w\_[x,y]\*scalar),0

image.putpixel((x,y),(r,g,0))

#image.resize((28,28))

plt.subplot(row,4,i+1)

plt.imshow(image)

plt.show()

return w\_p

w\_p=None

for i in range(len(mlp0.biases)):

w\_p=w\_plot(mlp0,i+1,w\_p)

### 说明：

该可视化方法用于探究神经网络在图像识别上的运作原理。然而，在全连接网络中，神经元关注的图像对人来说多数情况下是难以理解的。第一层隐含层可以直接将根据权重生成图像，但后面的层该如何处理还尚不明确，上面的代码是对权重进行了累积，但由于没有考虑偏置和激活函数的作用，可能生成的图像并不正确，所以目前只推荐使用该方法生成第一层隐含层图像。

# 十一. 保存与加载

## 1. 保存

### python代码示例：

import json

data = {"param1 name": param1,

"param2 name": [array.tolist() for array in param2],

"param3 name": param3}

f = open(file\_path, "w")

json.dump(data, f)

f.close()

### 说明：

此处代码实现将模型对象的参数和属性存入dict，再通过json保存至文件。使用pickle也可以将对象保存至文件，但更改类可能导致依赖旧定义的文件载入失败，所以改用json，额外再增加一些处理逻辑就可以健全地处理旧版本的模型文件。

一些参数和属性可能使用numpy的数组结构存储，需要转换为list结构，不然json不能识别。

## 2. 读取

### python代码示例：

f = open(file\_path, "r")

data = json.load(f)

f.close()

model=MultilayerPerceptron(param1=data['param1 name'])

model.param2=[np.array(item) for item in data['param2 name']]

model.param3=tuple(data['param3 name'])

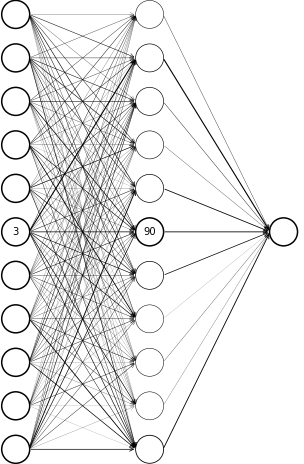
### 说明：

此处代码实现通过json读取文件并根据文件中的数据创建一个新模型。有的数据类型经过json保存后会发生变化，比如tuple类型，需要转换回来。

# 十四. 回归

## 1. 结构

### 图示：



### 说明：

回归神经网络在结构上与分类神经网络的区别就在于输出层被固定为1个神经元。

## 2. 激活函数

回归神经网络的隐含层激活函数没有额外限制，但输出层不能用激活函数（或是用恒等式，梯度为1向量）。这是因为回归输出不应当有值域限制，使用常规的激活函数会导致输出被限制到某个范围内，这不符合我们的需要。

## 3. 代价函数

回归神经网络的代价函数一般使用均方误差(MSE)，详见前面第三章节第一小节。在回归问题上，神经网络外大多数其他模型都是使用了均方误差，所以此处也不需要另辟蹊径。

## 4. 数据预处理

归一化（即特征缩放）是必须的，不然很难优化，详见第九章节第一小节。无论是分类问题还是回归问题，归一化都是一个必要的处理过程。另外回归问题下的目标向量是连续值，不需要使用one-hot编码处理。