目录

课程介绍

本书是一本介绍机器学习与神经网络方面的基础教材。

本书首先介绍了必要的数学基础及其Python代码，以便于对这些数学知识有所遗忘的读者能够快速上手；然后介绍几种典型机器学习算法的Python实现以及scikit-sklearn库对它们的实现，读者既可以在理解算法原理的基础上亲自动手实现以加深熟练程度，也能够直接利用已有的库快速进行模型训练；本书最后介绍了神经网络及其在图像分类上的应用，读者可以了解到在构建多层神经网络的过程中需要注意的问题，以及卷积神经网络在图像分类处理上的原理与实现。

本书大量采用实验案例来配合理论知识的讲解，读者可结合配套的案例源代码来学习。

本书适合作为机器学习与神经网络模型训练方面的入门教材，也可作为高校相关课程的配套课外教材或实验参考书。

第一部分：数学基础

本部分介绍机器学习算法所要求的基本数学知识，主要包括：

- Python(尤其是numpy科学计算包)中对数组(一维、二维、多维)的运算
- 向量、矩阵及它们之间的运算，针对矩阵的典型操作(例如：特征分解等)
- 基本统计量(例如均值、方差等)，基本的概率分布和概率公式(例如贝叶斯公式)
- 导数的概念；导数的分析解法和数值解法；向量和矩阵的求导运算
- 最小二乘法，函数的极值求解，带约束条件的函数极值求解，梯度下降算法等
- 信息熵、向量之间的距离计算等

第一章：Python数组操作

第1节：创建数组

使用单一值创建数组

> 创建具有**10**个元素的全为**0**的数组

- 代码

```
np.zeros(10)
```

- 运行结果

```
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

- 要点

  默认情况下，每个元素是浮点数

> 创建**2x3**的元素值全为**0**的二维数组

- 代码

```
np.zeros((2,3))
```

- 运行结果

```
array([[ 0.,  0.,  0.],[ 0.,  0.,  0.]])
```

- 要点

  (2,3)表示**tupple**类型参数

> 创建**2x3**的元素值全为**0**的二维整数数组

- 代码

```
np.zeros((2,3),dtype=np.int)
```

- 运行结果

```
array([[ 0,  0,  0],[ 0,  0,  0]])
```

- 要点

dtype属性指定元素类型

> 创建**2x3**的元素值全为**1**的二维数组

- 代码

```
np.ones((2,3))
```

- 运行结果

```
array([[ 1., 1., 1.],[ 1., 1., 1.]])
```

- 要点

  使用**tupple**将二维数组长度封装起来

> 创建**2x3**的二维数组，每个元素值都是**5**

- 代码

```
np.full((2,3),5)
```

- 运行结果

```
array([[5, 5, 5],[5, 5, 5]])
```

> 创建**3x3**的二维数组，并且主对角线上元素都是**1**

- 代码

```
np.identity(3)
np.eye(3)
```

- 运行结果

```
array([[ 1., 0., 0.],[ 0., 1., 0.],[ 0., 0., 1.]])
array([[ 1., 0., 0.],[ 0., 1., 0.],[ 0., 0., 1.]])
```

- 要点

  用**identity**方法生成单位矩阵。必须是方阵

> 创建**mxn**的二维数组，并且主对角线上元素都是**1**

- 代码

```
np.eye(3,4)
np.eye(4,3)
```

- 运行结果

```
array([[ 1., 0., 0., 0.],[ 0., 1., 0., 0.],[ 0., 0., 1., 0.]])
array([[ 1., 0., 0.],[ 0., 1., 0.],[ 0., 0., 1.],[ 0., 0., 0.]])
```

- 要点

  用**eye**方法生成对角线元素全为1的矩阵，不必是方阵

> 创建**2x3**的二维数组，不指定初始值

- 代码

```
np.empty((2,3))
np.ndarray((2,3))np.ndarray((2,3))
```

- 运行结果

```
array([[ 1.38808538e-311, 1.38808538e-311, 1.38808538e-311],
       [ 1.38808538e-311, 1.38808542e-311, 1.38808538e-311]])
```

- 要点

  可用上述两种方法之一创建二维数组，但是其中的元素并不会初始化为0

从现有数据初始化数组

下面的例子列出了使用一个数组初始化另一个数组的方式。

> 创建**5**个元素的一维数组，初始化为**1,2,3,4,5**

- 代码

```
np.array([1,2,3,4,5])
```

- 运行结果

```
array([1, 2, 3, 4, 5])
```

- 要点

[ ]操作符实际上生成一个list，将其作为array方法的参数传入后，将据此生成一个array

> 创建**2x3**的二维数组，用指定的元素值初始化

- 代码

```
np.array([[1,2,3],[4,5,6]])
```

- 运行结果

```
array([[1, 2, 3],[4, 5, 6]])
```

- 要点

  注意方括号的使用

> **a**是**(M,N)**数组，根据**a**的维度生成**(M,N)**的全**0**值数组**b**

- 代码

```
a = np.array([[1,2,3],[4,5,6]])
b = np.zeros_like(a)
```

- 运行结果

```
array([[0, 0, 0],[0, 0, 0]])
```

- 要点

  同理有**ones_like**和**full_like**

> 以指定的主对角线元素创建对角矩阵

- 代码

```
np.diag([1,2,3])
```

- 运行结果

```
array([[1, 0, 0],[0, 2, 0],[0, 0, 3]])
```

- 要点

  不必是方阵

切分数组

下面的例子演示了如何将指定数值范围切分成若干份然后形成数组。

> 根据指定的间距，在**[m,n)**区间等距切分成若干个数据点，形成数组

- 代码

```
np.arange(5)
np.arange(1,6)
np.arange(1, 10, 2)
np.arange(0.0, 4.0, 0.5)
```

- 运行结果

```
array([0, 1, 2, 3, 4])
array([1, 2, 3, 4, 5])
array([1, 3, 5, 7, 9])
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5])
```

- 要点
  - 如果只提供一个参数，相当于[0,n)
  - 提供两个参数，相当于指定区间[m, n)
  - 提供三个参数，在区间范围内，还可以指定间隔
  - 支持浮点数间隔

> 根据指定的间距，在**[m,n]**区间等距切分成若干个数据点，形成数组

- 代码

```
np.linspace(0,3,5)
np.linspace(0,3,5,False)
```

- 运行结果

```
array([ 0.  ,  0.75,  1.5 ,  2.25,  3.  ])
array([ 0. ,  0.6,  1.2,  1.8,  2.4])
```

- 要点
  - 不传递第4个参数时，区间实际为：[0,3]，共5个等分点，默认包含终结点
  - 如果第4个参数设为False，则区间实际为：[0,3)，即不包括终结点

> 生成指数间隔**(**而非等距间隔**)**的数组

- 代码

```
np.logspace(1,4,4)
np.logspace(1,4,4,base=2
```

- 运行结果

```
array([ 10., 100., 1000., 10000.])
array([  2.,   4.,   8.,  16.])
```

- 要点
  - 默认情况下按以10为底取指数，取值范围[1,4]，共4个等分点，形成数组{1,2,3,4}；然后按10为底取指数
  - 通过base参数可指定基底为2而不是10

> 生成网格数据点

- 代码

```
nx, ny = (5, 3)
x = np.linspace(0, 1, nx)
y = np.linspace(0, 1, ny) xv,
yv = np.meshgrid(x, y)
```

- 运行结果

```
xv = [[ 0. 0.25 0.5 0.75 1. ] [ 0. 0.25 0.5 0.75 1. ] [ 0. 0.25 0.5 0.75 1. ]]
yv = [[ 0. 0. 0. 0. 0. ] [ 0.5 0.5 0.5 0.5 0.5] [ 1. 1. 1. 1. 1. ]]
```

- 要点
  - 水平方向等分成5个点；竖直方向等分成3个点
  - 通过xv和yv，可形成3(行)x5(列)的网格点；最终的点坐标由xv和yv中的对应索引值组合而成

示例**1**：绘图查看指数间隔的数组间距

- 目标

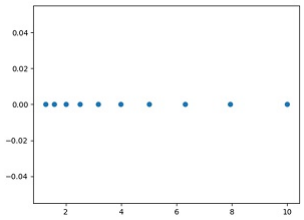
  观察logspace生成的数组中，各个点的间距是变化的

- 代码

```
import numpy as np
import matplotlib.pyplot as plt
N = 10
x = np.logspace(0.1, 1, 10, endpoint=True)
y = np.zeros(N)
plt.plot(x, y, 'o')
```

```
plt.show()
```

- 结果



示例**2**：生成和查看网格数据点

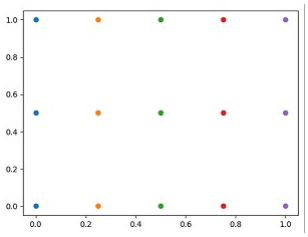
- 目标

  生成[0, 1]之间的二维网格，水平方向5个点，数值方向3个点。作图查看效果

- 代码

```
import numpy as np
import matplotlib.pyplot as plt
nx, ny = (5, 3)
x = np.linspace(0, 1, nx)
y = np.linspace(0, 1, ny)
xv, yv = np.meshgrid(x, y)
plt.plot(xv, yv, 'o')
plt.show()
```

- 结果



数组的引用和拷贝

> 使数组**b**与数组**a**共享同一块数据内存**(**数组**b**引用数组**a)**

- 代码

```
a = np.array([1,2,3,4,5])
b = a
b[0] = 100
a[0]
```

- 运行结果

```
100
```

- 要点

  =运算符在数组操作时是按引用赋值。通过b修改数组的第一个元素，实际上就是修改了a的第一个元素

> 将数组**a**的值做一份拷贝后再赋给**b**，**a**和**b**各自保留自己的数据内存

- 代码

```
a = np.array([1,2,3,4,5])
np.copyto(b, a)
#b = np.copy(a)
#b = np.array(a)
b[0] = 100
a[0]
```

- 运行结果

```
1
```

- 要点

  copyto、copy及array方法都将导致数据复制，因此，通过变量b修改数据将不会影响a中的数据

第2节：修改数组

变更数组维度

下表列出了在数组创建后，调整数组维度的方法

> 查看数组的维度尺寸

- 代码

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
len(a)
a.shape
```

- 运行结果

```
 len(a) = 3
 a.shape=(3,5)
```

- 要点

  len返回第一维的长度：对于一维数组，返回元素个数，二维数组返回行数

> 一维数组变形为**mxn**二维数组

- 代码

```
a = np.array([1,2,3,4,5,6,7,8,9])
b = np.reshape(a, (3,3))
```

- 运行结果

```
b = array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
```

- 要点

  变形前后的元素总个数必须相同；a和b共享同一块数据内存

> 将二维数组调整为一行或一列

- 代码

```
a = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
b = np.reshape(a, (1, -1))
c = np.reshape(a, (-1, 1))
```

- 运行结果

```
b = array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]])
c = array([[1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12]])
```

- 要点

(1, -1)表示：行数为1，列数则根据总元素数量而定
(-1,1)表示：列数位1， 行数则根据总元素数量而定

将行数组转成列数组

- 代码

```
a = np.array([1,2,3,4,5])
b = a[:, np.newaxis]
c = np.reshape(a, (-1, 1))
```

- 运行结果

```
b = array([[1],
       [2],
       [3],
       [4],
       [5]])
```

- 要点

np.newaxis和reshape都可以将行向量转成列向量

二维数组展成连续的一维数组

- 代码

```
c = np.ravel(b)
```

- 运行结果

```
c = array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- 要点

b和c共享同一块数据内存

二维数组展成连续的一维数组(拷贝)

- 代码

```
d = b.flatten()
```

- 运行结果

```
c = array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- 要点

  b和d各自拥有数据内存

> 将原有数组调整为新指定尺寸的数组(拷贝)

- 代码

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
b = np.resize(a, (5,3))
```

- 运行结果

```
b =
[[ 1 2 3]
 [ 4 5 6]
 [ 7 8 9]
 [10 11 12]
 [13 14 15]]
```

- 要点

  resize会生成新的数组，且与原数组分别保留各自内存

> 将原有数组调整为新指定尺寸的数组(拷贝)

- 代码

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
b = np.resize(a, (5,3))
c = np.resize(a, (5,2))
```

- 运行结果

```
b =
[[ 1 2 3]
 [ 4 5 6]
 [ 7 8 9]
 [10 11 12]
 [13 14 15]]
c =
[[ 1 2]
```

```
 [ 3 4]
 [ 5 6]
 [ 7 8]
 [ 9 10]]
```

- 要点

  调整后的元素总数不一定要与原来的元素总数相等。resize仅仅是从第一个元素开始，按照指定的新维度依次排布元素

> 生成转置数组(矩阵)

- 代码

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
b = a.T
#b = np.transpose(a)
```

- 运行结果

```
b = array([
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14],
       [ 5, 10, 15]])
```

- 要点
  - 转置后的数组与原数组共享数据内存
  - 一维数组的转置操作无效

数组的组合、拼接与拆分

> 以竖直方向叠加两个数组

- 代码

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
c = np.concatenate((a, b), axis=0)
d = np.vstack((a, b))运行结果
```

- 运行结果

```
c=
[[1 2]
```

```
 [3 4]
 [5 6]]
d=
[[1 2]
 [3 4]
 [5 6]]
```

- 要点
  - axis=0指定按行(沿行索引方向)拼接；vstack相当于按行(竖直)拼接
  - 二者产生的新数组与原始数组均不共享数据内存

> 以水平方向叠加两个数组

- 代码

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5], [6]])
e = np.concatenate((a, b), axis=1)
f = np.hstack((a, b))
```

- 运行结果

```
e=
[[1 2 5]
 [3 4 6]]
f=
[[1 2 5]
 [3 4 6]]
```

- 要点
  - axis=1指定沿列索引方向拼接。hstack相当于按列(水平)拼接。
  - 水平叠加要求两个数组必须具有相同的行数。所以b必须定义为2行的二维数组

> 竖直方向将二维数组拆分成若干个数组

- 代码

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15],[16,17,18,19,20]])
b = np.split(a, 2)
c = np.vsplit(a, 2)
```

- 运行结果

```
b =
[
  array([
    [ 1, 2, 3, 4, 5],
```

```
    [ 6, 7, 8, 9, 10]]),
  array([
    [11, 12, 13, 14, 15],
    [16, 17, 18, 19, 20]])]
c =
[
  array([
    [ 1, 2, 3, 4, 5],
    [ 6, 7, 8, 9, 10]]),
  array([
    [11, 12, 13, 14, 15],
    [16, 17, 18, 19, 20]])]
```

- 要点
  - split方法默认按行(竖直)拆分；vsplit相当于按行(竖直)拆分。
  - 拆分后共享数据内存。

> 水平方向将二维数组拆分成若干个数组

- 代码

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15],[16,17,18,19,20]])
d = np.split(a, 5, axis=1)
e = np.hsplit(a, 5)
```

- 运行结果

```
d=
[array([[1],[6],[11]]),
 array([[2],[7],[12]]),
 array([[3],[8],[13]]),
 array([[4],[9],[14]]),
 array([[5],[10],[15]])]
e=
[array([[1],[6],[11]]),
 array([[2],[7],[12]]),
 array([[3],[8],[13]]),
 array([[4],[9],[14]]),
 array([[5],[10],[15]])]
```

- 要点

  axis=1指定按列拆分。

访问及修改元素

> 访问二维数组

- 代码

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
b = a[1]
c = a[1,1]
d = a[1][1]
```

- 运行结果

```
b = array([ 6,  7,  8,  9, 10])
c = 7
d = 7
```

- 要点

  通过一个中括号或两个中括号都可以访问二维数组中的元素。

> 访问一维数组的部分元素

- 代码

```
a = np.array([1,2,3,4,5,6,7,8,9,10])
b = a[3:5]
c = a[:5]
d = a[5:]
e = a[-1]
f = a[-2]
g = a[-4:]
h = a[1:6:2]
```

- 运行结果

```
b = array([4, 5])
c = array([1, 2, 3, 4, 5])
d = array([ 6,  7,  8,  9, 10])
e = 10
f = 9
g = array([ 7,  8,  9, 10])
h = array([2, 4, 6])
```

- 要点
  - a[3:5]取元素下标索引在[3,5)区间
  - a[-n:]常用于取数组的后n个元素
  - a[1:6:2]取下标索引在[1:6)区间，间隔为2。

> 访问二维数组的部分元素

- 代码

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
b = a[0:2,2:5]
c = a[:,3]
d = a[:, 2:5]
```

- 运行结果

```
b = array([
  [ 3,  4,  5],
  [ 8,  9, 10]])
c = array([ 4,  9, 14])
d = array([
  [ 3,  4,  5],
  [ 8,  9, 10],
  [13, 14, 15]])
```

- 要点
  - a[0:2,2:5]，取出索引为0,1的行，索引为2,3,4的列
  - a[:, 3]表示取出所有行，取出第3列

删除元素

- 代码

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
b = np.delete(a, 1)
c = np.delete(a, [1,3,5])
```

- 运行结果

```
b = array([ 1,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
c = array([ 1,  3,  5,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

- 要点
  - delete(a, 1)：删除1号索引元素。因为没有指定是删除行还是列(默认是None)，因此将返回一个一维数组
  - 请注意，a本身不会被修改，而是返回修改后的数组
  - delete(a, [1,3,5])：删除索引为1,3,5号的元素

删除行或列

- 代码

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]])
b = np.delete(a,1, axis=0)
c = np.delete(a, [2,3], axis=1)
```

- 运行结果

```
b = array([
       [ 1,  2,  3,  4,  5],
       [11, 12, 13, 14, 15]])
c = array([
       [ 1,  2,  5],
       [ 6,  7, 10],
       [11, 12, 15]])
```

- 要点
  - axis=0指定按行删除
  - axis=1指定按列删除

> 插入元素、行或列

- 代码

```
a = np.array([[1, 1], [2, 2], [3, 3]])
b = np.insert(a, 1, 5)
c = np.insert(a, 1, 5, axis=1)
d = np.insert(a, 1, [0,5,10], axis=1)
e = np.insert(a, len(a),[[4,4]], axis=0)
```

- 运行结果

```
b = array([1, 5, 1, 2, 2, 3, 3])
c = array([
       [1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])
d = array([
       [ 1,  0,  1],
       [ 2,  5,  2],
       [ 3, 10,  3]])
e = array([
       [1, 1],
       [2, 2],
       [3, 3],
       [4, 4]])b = array([1, 5, 1, 2, 2, 3, 3])
c = array([
       [1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])
d = array([
       [ 1,  0,  1],
       [ 2,  5,  2],
       [ 3, 10,  3]])
e = array([
       [1, 1],
```

```
       [2, 2],
       [3, 3],
       [4, 4]])
```

- 要点
  - insert(a, 1, 5)：在1号索引元素后插入值为5的元素；因为没有指定axis，所以将返回一个一维数组
  - insert(a, 1, 5, axis=1)，插入一列，该列元素值全为5

> 追加元素、行或列

- 代码

```
a = np.array([[1, 1], [2, 2], [3, 3]])
b = np.append(a, 4)
e = np.append(a, np.array([[4,4]]), axis=0)
f = np.append(a, np.array([4,4]), axis=0) # 错误
```

- 运行结果

```
b = array([1, 1, 2, 2, 3, 3, 4])
e = array([
       [1, 1],
       [2, 2],
       [3, 3],
       [4, 4]])
```

- 要点
  - 如果指定了axis，那么追加的元素必须与原始数组的维度保持一致。
  - a是3x2数组，所以如果按行追加，那么必须是具有2列的数组。np.array([4,4])只是一个一维数组，不满足要求；而 np.array([[4,4]])是1x2数组，满足要求

> 在一个二维数组后添加一列

- 代码

```
a = np.array([[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15],[16,17,18,19,20]])
b = np.c_[a, np.array([1,1,1])]
```

- 运行结果

```
b = array([
       [ 1,  2,  3,  4,  5,  1],
       [ 6,  7,  8,  9, 10,  1],
       [11, 12, 13, 14, 15,  1]])
```

- 要点

  在机器学习中，c_操作符经常用于在特征矩阵后追加一个Label标签列

第3节：查找、排序和统计

检索符合条件的元素

> 一维数组中，查找不为0的元素

- 代码

```
a = np.array([1,0,0,3,5,0,8])
b = np.nonzero(a)
c = a[b]
```

- 运行结果

```
b = (array([0, 3, 4, 6], dtype=int64),)
c = array([1, 3, 5, 8])
```

- 要点

  b返回非0元素的下标索引，进而可以获取非0元素的值集合

> 二维数组中，查找不为0的元素

- 代码

```
a = np.array([[1,0,0], [0,2,0], [1,1,0]])
b = np.nonzero(a)
c = np.transpose(b)
d = a[b]
```

- 运行结果

```
b = (array([0, 1, 2, 2], dtype=int64), array([0, 1, 0, 1], dtype=int64))
c = array([
       [0, 0],
       [1, 1],
       [2, 0],
       [2, 1]], dtype=int64)
d = array([1, 2, 1, 1])
```

- 要点
  - b是一个tuple，两个array分别存放了非0元素的行下标索引和列下标索引
  - np.transpose方法对于tuple/list等也能求出转置数组；
  - d返回了所有非0元素值，但是是作为一维数组返回的

查找指定条件的元素

- 代码

```
a = np.arange(10)
b = np.where(a>5)
c = a[b]
```

- 运行结果

```
a = array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
b = (array([6, 7, 8, 9], dtype=int64),)
c = array([6, 7, 8, 9])
```

- 要点

  对于二维数组的查找方法类似

返回条件为**True**的元素

- 代码

```
a = np.arange(5)
b = np.array([True, False, False, True, True])
c = a[b]
d = np.arange(10)
e = a > 5
f = a[b]
```

- 运行结果

```
c = array([0, 3, 4])
e = array([False, False, False, False, False, False,  True,  True,  True,  True], dtype=bool)
f = array([6, 7, 8, 9])
```

- 要点
  - 可以通过True/False作为判断条件。条件数组中，True对应的下标索引的元素将被返回。
  - a>5将对a中的每个元素分别判断，并返回判别结果数组

返回指定索引的若干个元素

- 代码

```
a = np.array([4, 3, 5, 7, 6, 8])
indices = [0, 1, 4]
b = np.take(a, indices)
# b = a[indices]
```

- 运行结果

```
b = array([4, 3, 6])
```

- 要点

  [ ]和take方法都可以返回指定下标索引的元素

请注意，上述例子中，所有返回结果与原数组都是相互独立的数据空间

数组排序

> 将数组倒序

- 代码

```
a = np.arange(5)
b = a[::-1]
```

- 运行结果

```
b = array([4, 3, 2, 1, 0])
```

- 要点

  -1指定了步长间隔为-1，意味着从后向前步进

> 一维数组排序

- 代码

```
a = np.array([3, 1, 7, 4, 2, 5, 8])
b = np.sort(a)
```

- 运行结果

```
b = [1 2 3 4 5 7 8]
```

- 要点

  sort并不会直接在原始数组上排序，而是返回一个新的排序后的数组

> 二维数组排序

- 代码

```
a = np.array([[1,4,3],[3,1,8]])
b = np.sort(a, axis=0)
c = np.sort(a, axis=1)
# c = np.sort(a)
```

- 运行结果

```
b = array([
       [1, 1, 3],
       [3, 4, 8]])
c = array([
       [1, 3, 4],
       [1, 3, 8]])
```

- 要点
  - axis=0：每一列数据沿着行索引增长方向升序排列；也就是对每一列排序
  - axis=1：每一行数据沿着列索引增长方向升序排列；也就是对每一行排序

> 以指定索引位置作为分界线，左边元素都小于分界元素，右边元素都大于分解元素

- 代码

```
a = np.array([30, 80, 70, 100, 50, 40, 20, 90])
b = np.partition(a, 0)
c = np.partition(a, 6)
```

- 运行结果

```
b = array([ 20,  80,  70, 100,  50,  40,  30,  90])
c = array([ 20,  30,  70,  80,  50,  40,  90, 100])
```

- 要点
  - b：分界后的数组中索引位置0的元素是分界元素(20)；右边所有元素都大于分界元素
  - c：分界后的数组中索引位置6的元素是分界元素(90)，左边均小于它，右边均大于它
  - 注意：左右两边的元素顺序是不确定的

数组统计

> 查找一维数组中的最大、最小值

- 代码

```
a = np.array([3, 1, 7, 4, 2, 5, 8])
b = np.max(a)
```

```
c = np.min(a)
```

- 运行结果

```
b = 8
c = 1
```

> 查找二维数组总的最大、最小值

- 代码

```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
b = np.max(a)
c = np.max(a, axis=0)
d = np.max(a, axis=1)
```

- 运行结果

```
b = 9
c = array([7, 8, 9])
d = array([3, 6, 9])
```

- 要点
  - axis=0，沿行索引方向查找每一列的极值
  - axis=1，沿列索引方向查找每一行的极值

> 查找极值元素的索引

- 代码

```
a = np.array([3, 1, 7, 4, 2, 5, 8])
b = np.argmax(a)
c = np.array([[1,2,3],[4,5,6],[7,8,9]])
d = np.argmax(c)
e = np.argmax(c, axis=0)
```

- 运行结果

```
b = 6
d = 8
e = array([2, 2, 2], dtype=int64)
```

> 统计数组中非零元素个数

- 代码

```
a = np.array([[1,2,0, 0],[1,0,0,1],[0,0,1,1]])
b = np.count_nonzero(a,axis=0)
c = np.count_nonzero(a,axis=1)
```

- 运行结果

```
a = 6
b = array([2, 1, 1, 2], dtype=int64)
c = array([2, 2, 2], dtype=int64)
```

- 要点
  - axis=0，沿行索引方向统计每一列的非零个数
  - axis=1，沿列索引方向统计每一行的非零个数

> 计算数组算数平均值

- 代码

```
a = np.arange(12).reshape((3,4))
b = np.mean(a)
c = np.mean(a, axis=0)
d = np.mean(a, axis=1)
```

- 运行结果

```
b = 5.5
c = array([ 4.,  5.,  6.,  7.])
d = array([ 1.5,  5.5,  9.5])
```

- 要点
  - axis=0，沿行索引方向计算每一列的平均值
  - axis=1，沿列索引方向计算每一行的平均值

> 计算数组的加权平均值

- 代码

```
a = np.arange(1,10)
b = np.average(a, weights=np.ones(9))
c = np.average(a, weights=np.array([1,0,1,0,1,0,1,0,0]))
```

- 运行结果

```
a = array([1, 2, 3, 4, 5, 6, 7, 8, 9])
b = 5.0
```

```
c = 4.0
```

- 要点
  - 如果没有weights参数，则average与mean的行为一致
  - weights参数指定原数组中的每个元素所占的权重axis=0，沿行索引方向计算每一列的平均值

第二章：线性代数

第1节：向量

向量的写法

在本书中，行向量的数学形式写为：$x=(x_1,x_2,\cdots,x_n)$

列向量的数学形式写为：$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$

或者也写成：$x=(x_1,x_2,\cdots,x_n)^T$，其中，T表示转置

在Python中，使用(M,N)形式表示数组的维度为M行，N列，本书大部分情况下也采用这种形式来描述数组的维度

Python代码 `arr = np.array([1,2,3,4,5])` 定义了一个数组，在大多数情况下可以看成行向量。如果要生成列向量，必须使用$(M,1)$的二维数组

示例**1**：数组**(**行向量**)**创建及基本运算

```
x1 = np.array([1,2,3,4,5])
x2 = np.array([5,4,3,2,1])
r1 = x1 * 5 + 2      # 各元素分别计算
r2 = x1 + x2         # 对应元素分别相加，应保证两个数组元素个数相同
r3 = x1 * x2         # 对应元素分别相乘
r4 = np.multiply(x1, x2)    # 与*操作一致
```

示例**2**：创建列向量

> 直接创建列向量

```
x = np.array([[1],[2],[3],[4],[5]])
```

> 将列向量转成$(1,M)$的行向量

```
x1 = np.array([[1],[2],[3],[4],[5]])
x2 = x1.T
```

注意：转置后，行向量x2不再是一维数组，而是1行M列的二维数组

> 将一个一维数组转成列向量

```
x1 = np.array([1,2,3,4,5])
x2 = x1[:, np.newaxis]
```

```
x3 = x1.T    # 无效
```

注意：转置操作必须针对矩阵或二维数组进行；对一维数组不起作用

> 使用**reshape**方法

```
x1 = np.array([1,2,3,4,5])
x2 = x1.reshape(5,1)
#x2 = np.reshape(x1, (5,1))
```

注意：reshape后，x2是$(M,1)$形式的二维数组

第2节：矩阵

矩阵的写法

(m,n)矩阵一般写成：$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$

矩阵与向量乘积

$Ax = b$

其中A是(m,n)矩阵，x是n维列向量，b是m维列向量。

考虑到向量也可以视为空间坐标，上述式子也可以理解为：某坐标空间中的向量x经过矩阵A的线性组合变换后，变成向量b

矩阵与矩阵的乘积

$A_{(m,n)} \times B_{(n,k)} = C_{(m,k)}$

- 将矩阵B视为由k个列向量$b_1, b_2, \cdots, b_k$ 组成
- 将矩阵A分别与向量b_i计算乘积，其结果依次作为结果矩阵C的第i列
- 这样就相当于将多个向量b_i分别经过矩阵A来做映射

特殊的矩阵

- 零矩阵：所有元素都为0
- 单位矩阵I：左上角到右下角线上所有元素都为1，其余元素都为0
- 对角矩阵$diag(A)$：除了左上角到右下角线上的元素外，其余元素都为0

示例**1**：创建矩阵

> 创建一般矩阵

```
A1 = np.array([[1,2,3],[4,5,6]])
x1 = np.array([1,2,3,4,5])    # 行向量
x2 = np.array([[1],[2],[3],[4],[5]])   # 列向量
```

> 创建特殊矩阵

```
np.zeros((2,3))
np.ones([2,3])
np.identity(3)
np.diag([1,2,3,4])
```

示例**2**：关于*操作符

> 数组与标量进行计算：数组中的每个元素与标量分别计算

```
x1 = np.array([1,2,3,4,5])
x2 = np.array([[1,2,3],[4,5,6]])
x3 = np.mat([[1,2,3],[4,5,6]])
r1 = x1 * 2
r2 = x2 * 2
r3 = x3 * 2
```

> 数组形式矩阵/向量之间的计算：不会当成矩阵乘法来处理，而是按照对应元素相乘

```
A1 = np.array([[1,2,3],[4,5,6]])
A2 = np.array([[1,2],[3,4],[5,6]])
x1 = np.array([[1],[2],[3]])
x2 = np.array([1,2,3])
r1 = A1 * x2      # 正确。r1:2x3。A1每行中的元素，分别乘以x2中的对应元素
r2 = x2 * A1      # 正确。r2:2x3。x2中的每个元素，分别乘以A1中每行的各个元素
r3 = A1 * A2      # 错误。A1和A2的列数不同，因此，A1每行中的元素无法乘以A2中每行的对应元素
r4 = A1 * x1      # 错误。A1只有2行，x1有3行，将导致x1中第3行找不到匹配项
r5 = x1 * A1      # 错误。x1有3行，而A1只有2行，将导致x1中的第3行无法找到匹配
r6 = x1 * x2      # 正确。x1中第一行的每个元素(只有1个)，分别与x2中第一行的每个元素相乘；
                  # x1中第二、三行类推。最终形成3x3矩阵
r7 = x2 * x1      # 正确，x2中第一列的每个元素(只有1个)，分别与x1中第一列的每个元素相乘；
                  # x2中第二、三列类推，最终形成3x3矩阵
```

示例**3**：关于**dot**操作符

数组形式矩阵/向量之间的计算，对于 `a.dot(b)` ：

- 若a, b都是一维数组，执行内积和操作(即对应元素乘积再求和)，同时要求a和b的维数相同
- 若a, b都是二维数组，执行矩阵乘法操作。如果a、b在执行矩阵乘法时行列数不匹配，则产生错误
- 若a为(m,n)数组，b是一维数组，则b必须是n个元素。a的每行和b分别进行内积和
- 若a为具有m个元素的一维数组，b是二维数组，则b必须具有m行。a和b的每列分别进行内积和。
- `np.dot(a, b)` 和 `a.dot(b)` 具有相同的效果

```
A1 = np.array([[1,2,3],[4,5,6]])
A2 = np.array([[1,2],[3,4],[5,6]])
x1 = np.array([[1],[2],[3]])
```

```
x2 = np.array([1,2,3])
x3 = np.array([1,2])
r1 = A1.dot(A2)     # 正确，两个操作数都是矩阵，按照矩阵乘法
r2 = A1.dot(x1)     # 正确，两个操作数都是矩阵，按照矩阵乘法
r3 = A1.dot(x2)     # 正确。A1是2x3矩阵，则x2必须3个元素。
                    # A1中每一行与x2进行内积和运算，结果放到一个数组中，得到array([14, 32])
r4 = A1.dot(x3)     # 错误。A1是2x3矩阵，它的每一行(3个元素)无法与x3(2个元素)进行操作
r5 = x2.dot(A1)     # 错误。x2有3个元素，要求A1必须是3xn矩阵。A1中的行数不匹配
r6 = x3.dot(A1)     # 正确。x3有2个元素，要求A1必须是2xn矩阵。
                    # x3与A1进行矩阵运算，结果放到一个数组中，得到array([9,12,15])
```

数组形式的矩阵和向量运算，应尽可能采用dot来进行

第3节：矩阵性质

矩阵和向量的混合运算性质

对于标量c和c', 向量x 以及矩阵A, B, C：

- ABx可以理解为：将向量x先通过B进行映射(注意B与x先结合)，再将结果通过A进行映射
- $(cAx) = c(Ax) = A(cx)$
- $(cA)B = c(AB) = A(cB)$
- $(c + c')A = cA + c'A$
- $(cc')A = c(c'A)$
- $(A + B)x = Ax + Bx$
- $A + B = B + A$
- $(A + B) + C = A + (B + C)$
- $A(B + C) = AB + AC$
- $(A + B)C = AC + BC$
- $(A + B)^2 = A^2 + AB + BA + B^2$
- $(A + B)(A - B) = A^2 - AB + BA - B^2$
- $(AB)^2 = ABAB$
- 矩阵乘法结合律(假定**AB**、**BC**都合法)：$ABC = (AB)C = A(BC)$

逆矩阵

对于方阵A，它的逆映射对应的矩阵称为A的逆矩阵，记作：A^{-1} 。向量x经过A变换后，再经过A^{-1} 变换就会回到起点。

不是所有的方阵都有逆。从坐标系的角度来理解，如果向量x经过A变换后被降维(扁平化)了，则必然无法再映射回去。这类矩阵称为奇异矩阵。

逆矩阵性质

- $(AA)^{-1} = I$
- $(A^{-1})^{-1} = A$
- $(AB)^{-1} = B^{-1}A^{-1}$

可以这样理解：如果向量x是先通过B再通过A进行的映射，那么要映射回去，就得先通过A^{-1}再通过B^{-1} 。进而有：

- $(ABCD)^{-1} = D^{-1}C^{-1}B^{-1}A^{-1}$
- $(A^k)^{-1} = (A^{-1})^k$

转置矩阵

转置矩阵相对原矩阵，是将元素的行列号互换：$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} => \begin{pmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{pmatrix}$

转置矩阵的性质：

- $(A^T)^T = A$
- $(AB)^T = B^T A^T$
- $(ABCD)^T = D^T C^T B^T A^T$
- $(A^{-1})^T = (A^T)^{-1}$

特征分解

如果存在非零向量ν和常数λ，使得方阵A具有下列性质：$A\nu = \lambda\nu$，则ν称为特征向量，λ被称为这个特征向量对应的特征值。这就是说，原矩阵与特征向量的乘积，相当于将特征向量放大特征值倍数

特征值的性质：

- 特征值衡量了矩阵中某个特征的重要程度；而特征向量则代表了该特征的空间方向
- 如果A是对称矩阵，则各个特征向量是互相正交的，也就是说，它们的对应元素乘积之和为0
- N阶方阵如果存在特征分解，则将会有N个特征向量，并分别对应着一个特征值
- N个特征向量合起来组成特征矩阵V，N个特征值合起来形成特征值向量Λ，上式进一步转成：$A = V diag(\Lambda) V^{-1}$。式中，$diag(\Lambda)$表示由特征值组成的对角矩阵
- 奇异矩阵如果进行特征分解，则必有一个特征值为0

奇异值分解**(SVD)**

将矩阵分解为奇异向量和奇异值：$A = UDV^T$

其中：

- A是一个(m,n)矩阵，U是一个(m,m)方阵，D是一个(m,n)矩阵，V是一个(n,n)矩阵
- U和V都是方阵，D是对角矩阵（不必是方阵）
- 对角矩阵D对角线上的元素被称为矩阵A的奇异值
- 矩阵U的列向量被称为左奇异向量，矩阵V的列向量被称右奇异向量

奇异值分解的性质：

- 每个实数矩阵都有一个奇异值分解，但不一定都有特征分解。例如，非方阵的矩阵没有特征分解，这时我们只能使用奇异值分解
- 奇异值分解使得对于非方阵也能求解出其特征向量和特征值
- 特征向量在U中以列的形式存在，而特征值就是D中的元素

矩阵的秩

将矩阵进行充分的初等变换（也就是利用将一行/列，与一个常数进行四则运算后，叠加到另一个行/列上）后，不全为0的行(或列)的数目，就是该矩阵的秩

秩的特点：

- (m,n)矩阵的秩最大为m和n中的较小者，表示为$min(m,n)$。有尽可能大的秩的矩阵被称为有满秩
- 零矩阵的秩为0
- 可逆矩阵称为满秩矩阵，不可逆矩阵称为降秩矩阵

范数

p范数被定义为：$||x||_p = (\sum_i |x_i|^p)^{\frac{1}{p}}$

向量常见范数：

- 1范数：向量各元素绝对值之和 $||x|| = \sum_i |x_i|$
- 2范数：向量各元素平方和再开方 $||x||_2 = \sqrt{(\sum_i x_i^2)}$
- 正无穷范数：向量各元素绝对值最大值
- 负无穷范数：向量各元素绝对值最小值

矩阵常见范数： 对于(m,n)矩阵A：

- 1范数：列和范数，即对每列向量求和(元素绝对值求和)，返回和的最大值

$||A||_1 = max_j \sum_{i=1}^{m} |a_{i,j}|$

- 正无穷范数：行和范数，即对每行求和(元素绝对值求和)，返回和的最大值

  $||A||_\infty = max_i \sum_{j=1}^{n} |a_{i,j}|$

- 2范数：谱范数，即A^TA矩阵的最大特征值的开平方

  $||A||_2 = \sqrt{\lambda_1}$

  其中，λ_1是A^TA的最大特征值

- F范数：矩阵元素值的平方和再开方

  $||A||_F = \sqrt{(\sum_{i=1}^{m} \sum_{j=1}^{n} a_{i,j}^2)}$

示例**1**：矩阵的逆

```
A1 = np.array([[2, 2, 3],[1, -1, 0],[-1, 2, 1]])
invA1 = np.linalg.inv(A1)
I = A1.dot(invA1)            # 非常接近单位阵
A2 = np.array([[1,2,3],[5, 3, 7], [2, 4, 6]])     # 错误，奇异矩阵
invA2 = np.linalg.inv(A2)
```

示例**2**：转置矩阵

```
A1 = np.array([[2, 2, 3],[1, -1, 0],[-1, 2, 1]])
invA1 = np.linalg.inv(A1)
tA1 = A1.T
tinvA1 = invA1.T
I = tA1.dot(tinvA1)            # 非常接近单位阵
```

示例**3**：特征分解

```
A = np.array([[2, 2, 3],[1, -1, 0],[-1, 2, 1]])
lambdas,V = np.linalg.eig(A)        # lambda为特征值构成的数组；V为所有特征向量按列构成的矩阵
r1 = A.dot(V[:,0])       # 原矩阵与第一个特征向量(特征矩阵中的第一列)相乘
r2 = lambdas[0]*V[:,0]   # 第一个特征值与第一个特征向量(特征矩阵中的第一列)相乘
r3 = A - V.dot(np.diag(lambdas)).dot(np.linalg.inv(V))    # 结果非常接近0
A2 = np.array([[1,2,3],[2,4,6],[-1, 2, 1]])  # 奇异矩阵，第2个特征值接近0
lambdas, V = np.linalg.eig(A2)
```

示例**4**：**SVD**分解

```
A1 = np.array([[2, 2, 3],[1, -1, 0],[-1, 2, 1]])
U,D,VT=np.linalg.svd(A1)        # U为3x3，D包含3个元素，VT为3x3。注意计算出的VT实际上已经是V的转置
r1 = U.dot(np.diag(D)).dot(VT)  # r1与A1非常接近
A2 = np.array([[2,2,3],[4,4,6],[-1, 2, 1]])
U,D,VT=np.linalg.svd(A2)        # 虽然A2是奇异矩阵，但是仍然可以进行奇异分解
```

```
r2 = U.dot(np.diag(D)).dot(VT)  # r2与A1非常接近
A3 = np.array([[2,2,3],[1, -1, 0],[-1, 2, 1], [-3,1,2]])
U,D,VT=np.linalg.svd(A3)        # A3是4x3矩阵，因此U为4x4，D为3个元素，VT为3x3
#r3 = U.dot(np.diag(D)).dot(VT)            # diag(D)为3x3，无法与U进行乘积运算，需要扩充一个行(全是0)
extD = np.vstack((np.diag(D), np.array([0,0,0])))       # 扩展3x3对角矩阵为4x3
r4 = U.dot(extD).dot(VT)        # r4与A3非常接近
```

示例**5**：矩阵的秩

```
A1 = np.array([[2, 1, 3],[6, 6, 10],[2, 7, 6]])
A2 = np.array([[2, 1, 3],[4, 2, 6],[2, 7, 6]])
A3 = np.array([[2, 1, 3],[6, 6, 10],[2, 7, 6],[1,3,5]])
print(np.linalg.matrix_rank(A1))        # 非奇异矩阵，满秩矩阵
print(np.linalg.matrix_rank(A2))        # 奇异矩阵，无法达到满秩
print(np.linalg.matrix_rank(A3))        # 4x3非奇异矩阵，秩为3
```

示例**6**：求范数

```
A1 = np.array([[2, 1, 3],[6, 6, 10],[2, 7, 6]])
print(np.linalg.norm(A1, ord=1)) # 1范数
print(np.linalg.norm(A1, ord=2)) # 2范数
print(np.linalg.norm(A1, ord=np.inf)) # 正无穷范数
print(np.linalg.norm(A1, ord='fro')) # F范数
```

第三章：统计与概率

第1节：基本统计量

均分量

常用的均分量有：

- 均值(mean)：各元素和除以元素个数
- 中值(median)：先将数组升序排列；如果数组有奇数个元素，则直接取回中间元素；如果是偶数个，则取中间两个元素的平均值（因此，该中值可能并不是数组中的元素）
- 百分位数(quantile)：指定一个百分比$x\%$，将数组升序排列后，跳过数组中前$x\%$个元素，返回随后的那个元素。返回的元素值将大于数组中指定百分比的元素。median可以视为quantile(array, 0.5)，也就是取50%百分位数

示例**1**：均分量计算

```
import numpy as np
import collections

arr = np.arange(1, 11, 1)      # [1,2,3,4,5,6,7,8,9,10]
m1 = np.mean(arr)
m2 = np.median(arr)            # (5+6)/2 = 5.5

def quantile(array, percentage):
    """百分位数"""
    index = (int)(percentage * len(array))
    return sorted(array)[index]

m3 = quantile(arr, 0.3)        # 4

def most_common_elements(array):
    """出现频率最高的元素(有可能有多个)"""
    counts = collections.Counter(array)
    max_count = max(counts.values())
    return [el for el, count in counts.items() if count == max_count]

arr2 = np.array([1,2,2,2,3,3,4,4,4,5,6,7,7])
results = most_common_elements(arr2)        # [2,4]
```

偏差相关

常用的偏差定义和计算公式有：

- 偏差(deviation)：数组中每个元素与数组平均值的差，返回一个数组

  $dev(x) = x_i - \bar{x}, i = 1, 2, \cdots, n$

- 方差(variance)：每个元素的偏差平方和，再除以有效元素总数

  $$var(x) = \frac{\sum_{i=1}^{n}(x^{(i)} - \bar{x})^2}{n}$$

  - 无偏估计方差(unbiased estimator)：有效元素总数为$n-1$
  - 有偏估计方差(biased estimator)：有效元素总数为n
  - 方差指示了一个数组中各个元素的离散程度
- 标准差(standard variance)：又叫均方差，方差的平方根。也分为有偏估计和无偏估计标准差

  $$std(x) = \sqrt{var(x)}$$
- 协方差(covariance)：对于长度相同的两个数组，先求出各自的偏差，然后求出两组偏差的对应元素乘积之和，最后除以有效元素总数

  $$cov(x,y) = \frac{\sum_{i}^{n}(\bar{x} - x^{(i)})(\bar{y} - y^{(i)})}{n}$$

  其中：

  - $\bar{y}$：数组y的平均值
  - $\bar{x}$：数组x的平均值
  - $x^{(i)}$：数组x中第i个元素
  - $y^{(i)}$：数组y中第i个元素

  使用向量V_x代表向量x中每个元素与其平均值相减后的结果，使用向量V_y代表向量y中每个元素与其平均值相减后的结果。如果V_x和V_y的协方差为0，则说明这两个向量是正交的；向量V_x中的元素值变化，丝毫不影响向量V_y中的元素
- 相关性(correlation)：对于两个数组(元素个数相同)，计算出协方差以及两个标准差，然后用协方差除以标准差乘积

  $$cor(x,y) = \frac{cov(x,y)}{std(x) * std(y)}$$

  相关性用于衡量两个数组的关联度。1表示最强的正关联；-1表示最强的负关联；0表示完全没有关联

示例**2**：偏差相关计算

```
arr = np.arange(1, 6, 1)      # [1,2,3,4,5]
def deviation(array):
    """偏差"""
    avg = np.mean(array)
    return [el - avg for el in array]

print(deviation(arr))               # 偏差：[-2,-1,0,1,2]
```

```
def unbiased_variance(array):
    """无偏估计方差"""
    deviations = deviation(array)
    return sum([el**2 for el in deviations]) / (len(array) - 1)

def biased_variance(array):
    """有偏估计方差"""
    deviations = deviation(array)
    return sum([el**2 for el in deviations]) /  len(array)

print(unbiased_variance(arr))   # 2.5
print(np.var(arr, ddof=1))      # ddof=1表示计算无偏方差
print(biased_variance(arr))     # 2.0
print(np.var(arr, ddof=0))      # ddof=0表示计算有偏方差

print(np.std(arr, ddof=1))     # 无偏标准差: 1.581
print(np.std(arr, ddof=0))     # 有偏标准差: 1.414

def unbiased_covariance(arrayX, arrayY):
    """无偏协方差"""
    deX = deviation(arrayX)
    deY = deviation(arrayY)
    return sum([el1 * el2 for el1, el2 in zip(deX, deY)]) / (len(arrayX) - 1)

def unbiased_correlation(arrayX, arrayY):
    """无偏相关性"""
    deX = np.std(arrayX, ddof=1)
    deY = np.std(arrayY, ddof=1)
    if(deX > 0 and deY > 0):
        return unbiased_covariance(arrayX, arrayY) / (deX * deY)
    else:
        return 0

arr2 = [2,4,6,8,10]
arr3 = [-2,-4,-6,-8,-10]
arr4 = [6, -100, 3, 20, -90]
print(unbiased_covariance(arr, arr2))       # 5.0
print(unbiased_covariance(arr, arr))        # 2.5
print(unbiased_covariance(arr2, arr2))      # 10.0
print(np.cov(arr, arr2, ddof=1))            # 无偏协方差，返回一个矩阵，矩阵中第[i,j]个元素代表第i个数组和第j个数组之间的协方差

print(unbiased_correlation(arr, arr2))      # 1.0，表示最强正相关
print(unbiased_correlation(arr, arr3))      # -1.0，表示最强负相关
print(unbiased_correlation(arr, arr4))      # -0.19，表示相关性很小
A = np.array([arr, arr2, arr3, arr4])      # 构造4行元素的矩阵
print(np.corrcoef(A, ddof=1))               # 无偏相关行，返回一个矩阵，矩阵中第[i,j]个元素代表第i行和第j行之间的相关性

# 直接使用矩阵形式来计算协方差
DA = A - np.mean(A, axis=1, keepdims=True)
COV = DA.dot(DA.T)/A.shape[1]
print(COV)
print(np.cov(A, ddof=0))
```

第2节：概率与概率分布

概率和概率密度

某个随机事件A发生的概率记为：$P(A)$。它表示在某些给定条件下，事件A发生的可能性。随机事件可分为连续型或离散型。

描述概率值分布状况的函数：

- 概率密度函数(Probability Density Function, PDF)
  - 描述连续型随机变量在某个确定的取值点附近的可能性的函数。
  - PDF本身并不是概率值，而是需要在一定区间积分后才是概率值。
  - 连续型随机变量在某个点的概率为0(虽然其PDF值并不为0)
- 概率质量函数(Probability Mass Function, PMF)
  - 离散型随机变量在各特定取值上的概率。
  - PMF本身就是概率值。
- 累积分布函数(Cumulative Distribution Function, CDF)
  - 反映了随机变量到当前指定值为止的所有概率之和。其值域为[0,1]
  - 对于连续型随机变量：$CDF(x) = \sum_{t=-\infty}^{x} PDF(t)dt$
  - 对于离散型随机变量：$CDF(x) = \sum_{k=1}^{x} PMF(k)$

数学期望

数学期望是试验中每次可能结果的概率乘以其结果的总和，它反映了随机变量的平均值大小

假设X是一个离散型随机变量，其可能取值有： $x_1, x_2, \cdots, x_n$；假设每个取值对应的概率为：

$P(x_k), k = 1, 2, \cdots, n$， 则数学期望定义为：$E(x) = \sum_{k=1}^{n} x_k \cdot P(x_k)$

假设X是一个连续性随机变量，其概率密度函数为$P(x)$，则数学期望定义为：

$E(x) = \int_{x=-\infty}^{+\infty} xP(x)dx$

离散型随机变量的概率分布

离散随机变量的均值与方差计算公式：

均值：$\mu = \sum_{k=1}^{n} x_k \cdot P(x_k) = E(x)$

方差：$\sigma^2 = \sum_{k=1}^{n}(x_k - \mu)^2 \cdot P(x_k) = E[(x-\mu)^2]$

常见的离散分布：

- 0-1分布
  - 只有一个事件，两种结果。该事件发生的概率为$p(0 < p < 1)$，不发生的概率为$q = 1 - p$。注意，p是定值，每次发生该事件的概率都是p
  - 伯努利实验：在同样的条件下，重复、独立的进行的随机实验。其实验只有两种结果：不发生x_0或发生x_1。0-1分布就是对伯努利实验的描述。
  - 概率质量函数：$P(x_k) = p^k(1-p)^{(1-k)}$，其中，$k$的取值只有两个：0或1；相应的，$x_0 = 0, x_1 = 1; P(x_0) = (1-p), P(x_1) = p$
  - 均值：

    $\mu = \sum_k^n x_k \cdot P(x_k) = x_0 \cdot p_0 + x_1 \cdot p_1 = 0 \cdot (1-p) + 1 \cdot p = p$
  - 方差：

    $\sigma^2 = \sum_k^n (x_k - \mu)^2 \cdot P(x_k)$

    $= (x_0 - \mu)^2 \cdot p(x_0) + (x_1 - \mu)^2 \cdot p(x_1)$

    $= (0-p)^2 \cdot (1-p) + (1-p)^2 \cdot p$

    $= p(1-p)$
- 几何分布
  - 在n次伯努利实验中，实验k次才得到第一次成功的机率，即：前$k-1$次都失败，第k次成功的概率
  - 概率质量函数：$P(x_k) = (1-p)^{(k-1)} \cdot p$，其中，$p$是单次实验中，发生该事件的概率
- 二项分布：
  - 重复n次伯努利实验，在n次实验中，发生k次事件的概率
  - 概率质量函数：$P(x_k) = C_n^k p^k (1-p)^{(k-1)} \cdot p$，其中，$p$是单次实验中，发生该事件的概率。$C_n^k = \frac{(P_n^k)}{k!} = \frac{n!}{k!(n-k)!}$
  - 均值：$\mu = np$， 方差：$\sigma^2 = np(1-p)$
- 泊松(Poisson)分布 日常生活中，大量事件是有固定频率的，比如：
  - 某医院平均每小时出生3个婴儿
  - 某网站平均每分钟有2次访问
  - 某超市平均每小时销售4包奶粉等

它们的特点是，我们可以预估这些事件的总数，但是没法知道具体的发生时间。例如：平均每小时出生3个婴儿，但无法准确得知下一个小时会出生几个。泊松分布就是描述某段时间内，事件具体的发生概率

- 概率质量函数：$P(N(t)=n)=\frac{(\lambda t)^n e^{-\lambda t}}{n!}$
  - N表示某种函数关系，t表示时间(为单位时间的倍数)，λ表示事件的平均频率(已知)，n表示待计算的事件次数。例如，如果平均每小时出生3个婴儿，则$\lambda=3$，以1小时为单位时间。
  - 计算：1小时出生3个婴儿的概率。此时，
    $n=3,t=1,P(N(1)=3)=\frac{(3\cdot 1)^3 e^{-3\cdot 1}}{3!}\approx 0.224$
  - 计算：接下来2个小时，一个婴儿都不出生的概率。此时，
    $n=0,t=2,P(N(2)=0)=\frac{(3\cdot 2)^0 e^{(-3\cdot 2)}}{0!}\approx 0.0025$
  - 计算：接下来1个小时，至少出生2个婴儿的概率。
    $P(N(1)\geq 2)=1-P(N(1)=0)-P(N(1)=1)=0.8$
- 绝大部分情况下，泊松分布将被规范化为单位时间内的概率，此时$t=1$，λ表示单位时间内发生时间的次数。其概率质量函数为：$P(x)=\frac{\lambda^x e^{-\lambda}}{x!}$ 其中，x相当于之前的n，其取值为0,1,2,3....
- 均值：$\mu=\lambda$， 方差：$\sigma^2=\lambda$

连续型随机变量的额概率分布

- 均匀分布(Uniform Probability Distribution)
  - 该随机变量所有可能值的出现概率相同。
  - 概率密度函数： $pdf(x)=\frac{1}{b-a},(a\leq x\leq b)$，假设该随机变量的取值范围在[a, b]区间
  - 累积分布函数：

$$CDF(x)=\begin{cases}\frac{x-a}{b-a}, & (a\leq x\leq b)\\ 0, & (x<a)\\ 1, & (x>b)\end{cases}$$

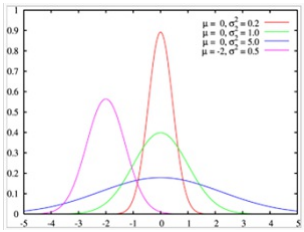
  - 均值：$\mu=\frac{a+b}{2}$， 方差：$\sigma^2=\frac{(b-a)^2}{12}$
- 指数分布
  - 泊松分布是单位时间内独立事件发生次数的概率分布，指数分布是独立事件的时间间隔的概率分布，或者说某个事件在时间(x)内发生的概率
    - 概率密度函数： $PDF(x)=\lambda e^{-\lambda x},(x>0)$。其中，$\lambda$称为率参数(rate parameter)，即单位时间内发生事件的次数（频率）

- 另一种写法：$PDF(x)=\frac{1}{\theta}e^{\frac{-x}{\theta}},(x>0)$。其中，$\theta=\frac{1}{\lambda}$
- 累积分布函数：$CDF(x)=1-e^{-\lambda x}$。注意，该函数是概率密度函数的积分
- 均值：$\mu=\theta$, 方差：$\sigma^2=\theta^2$

如之前泊松分布中的例子，假设平均每小时出生3个婴儿($\lambda=3$)，并且刚才过去的那个小时正好已经生了3个婴儿，那么接下来的15分钟，有婴儿出生的概率是多少？

以小时为单位，15分钟为1/4小时，使用累积分布函数来计算0~1/4小时区间内事件出现的概率：$CDF(1/4)=1-e^{-\lambda x}=1-e^{-3\cdot 1/4}\approx 0.53$

- 正态分布
  - 正态分布又叫高斯分布。若随机变量X服从一个数学期望μ，方差为σ^2的正态分布，则记为：$N(\mu,\sigma^2)$。其中，μ决定了正态分布中心线的位置，σ决定了它的幅度。
  - 概率密度函数：$PDF(x)=\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$。在不同的$\mu$和$\sigma$组合下，其图像如下所示：

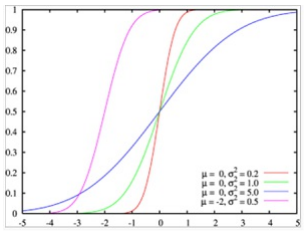


密度函数关于平均值对称

$\mu=0,\sigma^2=1$时，被称为标准正态分布。此时：

- 函数曲线下68.268949%的面积在平均值左右的一个标准差范围内
- 95.449974%的面积在平均值左右两个标准差2σ的范围内
- 99.730020%的面积在平均值左右三个标准差3σ的范围内
- 99.993666%的面积在平均值左右四个标准差4σ的范围内

  - 累积分布函数：$CDF(x)=\frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{x}e^{-\frac{(x-\mu)^2}{2\sigma^2}}dx$。图像如下：



第3节：随机数

关于随机变量

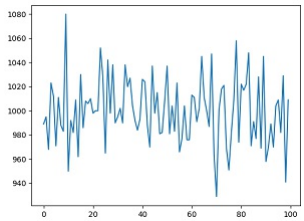
随机变量可以随机地取不同值的变量。一个随机变量只是对可能的状态的描述；它必须伴随着一个概率分布来指定每个状态的可能性

示例**1**：均匀分布产生随机数：**rand, random, randint**

- 代码

```
import numpy as np
import collections as col
import matplotlib.pyplot as plt

total_count = 100000
section_count = 100
min_value = 0
max_value = 100

data = np.random.randint(min_value, max_value, total_count)    # 0~100之间的整数，随机生成100000个
#data = (np.random.rand(total_count)*100).astype(int)          # 随机生成0~1之间随机数100000个，放大100倍，再转成整数
#data = (np.random.random(total_count)*100).astype(int)        # 随机生成0~1之间随机数1000个，放大100倍，再转成整数
counts = col.Counter(data)

x = np.arange(section_count)
y = np.zeros(section_count)
for i in x:
    y[i] = counts[i]

plt.plot(x, y)
plt.show()
```

- 分布结果



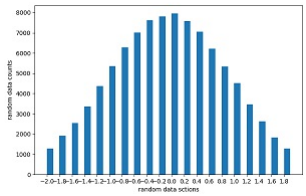
示例**2**：标准正态分布随机数：**randn**

- 代码

```
import numpy as np
```

```
import collections as col
import matplotlib.pyplot as plt
total_count = 100000
section_count = 20
section_max_value = 2.0 #仅统计-2.0~2.0之间的样本数
section_min_value = -2.0
section_interval = (section_max_value - section_min_value) / section_count
data = np.random.randn(total_count)
counts = np.array(section_count)
x = np.arange(section_min_value, section_max_value, section_interval)
y = np.zeros(len(x))
# 按照-2.0~-1.9, -1.9~-1.8..... -0.1~0.0, 0.0~0.1, 0.1~0.2....1.9~2.0的区间计算data在各区间的数量
for value in data:
if value > section_max_value or value < section_min_value:
continue
index = int((value - section_min_value) / section_interval) # 计算区间编号。只取计算结果的整数部分作为编号
y[index] += 1
plt.xticks(np.arange(section_min_value, section_max_value, section_interval))
bar_width = section_interval/2
plt.bar(x+bar_width/2, y, bar_width)
plt.xlabel("random data sctions")
plt.ylabel("random data counts")
plt.show()
```

- 分布结果



第4节：贝叶斯公式

条件概率

- 某个随机事件A发生的概率，记为：$P(A)$
- 两个随机事件A和B都发生(同时发生)的概率记为：$P(A,B)$或$P(B,A)$
- 条件概率：在随机事件A发生的前提下，再发生随机事件B的概率，记为：$P(B|A)$
- 如果A、B是各自独立的事件，则：$P(A,B)=P(A)\cdot P(B)$
- 如果A、B不独立，且$P(A)$不为0，则：$P(B|A)=\dfrac{P(B,A)}{P(A)}$
- 全概率公式：$P(B)=P(B|A)\cdot P(A)+P(B|\overline{A})\cdot P(\overline{A})$ 其中，$P(B|\overline{A})$ 是指当A不发生时B发生的概率
- 如果A是由$A_1,A_2,\cdots,A_k$ 多种分类组成（而不仅是非此即彼的两种类别），则全概率公式变为：$P(B)=P(B|A_1)\cdot P(A_1)+P(B|A_2)\cdot P(A_2)+\cdots+P(B|A_k)\cdot P(A_k)$ 其中，

  $P(A_1)+P(A_2)+\cdots+P(A_k)=1$

贝叶斯公式

通用公式：$P(B|A)=\dfrac{P(A|B)\cdot P(B)}{P(A)}$

示例1：患病概率计算

- 假设某种病在人群中的发病率是0.001，即1000人中大概会有1个人得病，则有：

  $P(\text{患病})=0.1\%$ 即：在没有做检验之前，我们预计的患病率为$P(\text{患病})=0.1\%$，这个就叫作先验概率

- 假设现在有一种该病的检测方法，其检测的准确率为95%；即：如果真的得了这种病，该检测法有95%的概率会检测出阳性，但也有5%的概率检测出阴性；或者反过来说，如果没有得病，采用该方法有95%的概率检测出阴性，但也有5%的概率检测为阳性。用条件概率表示即为：

  $P(\text{显示阳性}|\text{患病})=95\%$

- 现在我们想知道的是：在做完检测显示为阳性后，某人的患病率$P(\text{患病}|\text{显示阳性})$，这个其实就称为后验概率。
- 根据贝叶斯公式，可以得知：

$$
\begin{aligned}
P(\text{患病}|\text{显示阳性}) =& \frac{P(\text{显示阳性}|\text{患病}) \cdot P(\text{患病})}{P(\text{显示阳性})} \\
=& \frac{P(\text{显示阳性}|\text{患病}) \cdot P(\text{患病})}{P(\text{显示阳性}|\text{患病}) \cdot P(\text{患病}) + P(\text{显示阳性}|\text{无病}) \cdot P(\text{无病})} \\
=& \frac{95\% \cdot 0.1\%}{95\% \cdot 0.1\% + 5\% \cdot 99.9\%} \\
=& 1.86\%
\end{aligned}
$$

示例**2**：新生儿性别概率计算

假设：一个家庭中，生男孩或生女孩的概率相同，且第二个小孩的性别与第一个小孩的性别无关(彼此独立)。

- 列出下列事件：
  - 样本空间：{ (男，女), (男，男), (女，女), (女，男) }
  - 设A=第一个小孩是女孩。设B=第二个小孩是女孩
  - 根据样本空间的分析，可知：$P(A) = 1/2, P(B) = 1/2,$
  - 因为A和B是独立事件，所以：$P(A, B) = P(A) * P(B) = 1/4$。这个结果从样本空间中也能看出来
- 计算1：如果某家庭生了两个小孩，其中第一个小孩是女孩，那么在这种情况下，第二个小孩是女孩的概率是多少？
  - $P(\text{第二个小孩是女孩}|\text{第一个小孩是女孩}) = P(B|A)$，因为$A$和$B$是独立事件，因此$P(B|A) = P(A, B)/P(A) = (1/4)/(1/2) = 1/2$
  - 从另一个角度来看，因为每个小孩是男或女的概率相同，因此，第二个小孩是女孩的概率当然也是1/2
- 计算2：如果某家庭生了两个小孩，并且已知两个小孩中至少有一个是女孩，那么两个小孩都是女孩的概率是多少？
  - 设C=两个小孩都是女孩，D=两个小孩中至少有一个是女孩。C和D并不是彼此独立的
  - 根据样本空间，$P(C) = 1/4, P(D) = 3/4$
  - $P(C|D) = P(D|C) \cdot P(C)/P(D) = 1 \cdot (1/4)/(3/4) = 1/3$

第四章：导数与微分

第1节：基本概念

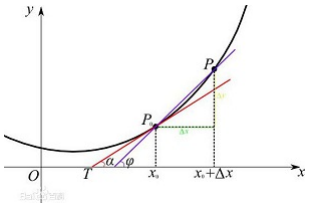
导数定义

设函数$y=f(x)$在点x_0的某个邻域内有定义，当自变量x在x_0处有增量Δx，且$(x_0+\Delta x)$也在该邻域内时，相应的函数取得增量：$\Delta y=f(x_0+\Delta x)-f(x_0)$。

如果当$\Delta x\to 0$时，$\frac{\Delta y}{\Delta x}$之比的极限存在，则称函数$y=f(x)$在点$x_0$处可导，并且该极限值称为函数$y=f(x)$在点$x_0$处的导数，可记作：$f'(x_0)$，或者：$y'|x=x_0$，或者：$\frac{dy}{dx}|x=x_0$

导数的数值计算方法：$f'(x_0)=\lim_{\Delta x\to 0}\frac{\Delta y}{\Delta x}=\lim_{\Delta x\to 0}\frac{f(x_0+\Delta x)-f(x_0)}{\Delta x}$

函数$y=f(x)$在点x_0的导数，就是函数$f(x)$在点$(x_0,f(x_0))$处的切线的斜率：



在很多情况下，用数值方法计算导数会导致较大的计算量。因此，也常用分析法来计算导数。在分析法中，首先需要获得函数$f(x)$的导函数：

- 如果函数$y=f(x)$在开区间内每一点都可导，就称函数在$f(x)$在该区间内可导。此时$f(x)$对于在区间内的每一个确定的x值，都对应着一个确定的导数值，从而构成一个新函数，即原来函数$y=f(x)$的导函数，记作：$y',f'(x),\frac{dy}{dx}$ 或者$\frac{df(x)}{dx}$。导函数也简称导数

基本函数的导函数

原函数	导函数
$y=C$ (C为常数)	$y'=0$
$y=x^n$ (n为常数)	$y'=n\cdot x^{n-1}$
$y=a^x$ (a为常数)	$y'=a^x\cdot\ln a$
$y=\log_a^x$ (a为常数)	$y'=\frac{1}{x\ln a}$
$y=sinx$	$y'=cosx$

$y=cosx$	$y'=-sinx$
$y=tanx$	$y'=sec^2x$
$y=cotx$	$y'=-csc^2x$

复杂函数的导函数

$(Cu)'=Cu'$ (C为常数)

$(u\pm v)'=u'\pm v'$

$(uv)'=u'v+uv'$

$(u/v)'=\dfrac{u'v-uv'}{v^2}$

复合函数的链式求导

链式求导法则：复合函数$F(x)$对自变量x的导数，等于该函数对中间变量t的导数$F'(t)$，乘以中间变量对自变量的导数$T'(x)$

示例**1**：计算函数$F(x)=(3x^2-2x+1)^2$的导数

设：$t=3x^2-2x+1$ 则：$F(t)=t^2,t(x)=3x^2-2x+1$ 因此：

$F'(x)=F'(t)\cdot t'(x)=(2\cdot t)\cdot(3\cdot 2\cdot x-2+0)=2\cdot(3x^2-2x+1)\cdot(6x-2)$

示例**2**：计算函数$F(x)=\frac{1}{1+e^{-x}}$的导数

设：$t=1+e^{-x}$ 则：$F(t)=t^{-1},t(x)=1+e^{-x}$ 因此：

$F'(x)=F'(t)\cdot t'(x)=-t^{-2}\cdot(e^{-x}\cdot(-1))$

$$=\frac{1}{1+e^{-x}}\cdot\frac{e^{-x}}{1+e^{-x}}=\frac{1}{1+e^{-x}}\cdot(1-\frac{1}{1+e^{-x}})$$

$$=F(x)\cdot(1-F(x))$$

偏导数

针对一个变量求偏导时，可以把其它变量视为常数

示例**3**：求函数$f(x,y)=x^2+2xy+y^2$ 的偏导数

需要针对自变量x和y分别来求。 也可以看成在xoy平面内，分别求沿x方向的导数和y方向的导数。

$$\frac{\partial f}{\partial x}=\frac{\partial(x^2+2xy+y^2)}{\partial x}=\frac{\partial x^2+\partial(2xy)+\partial y^2}{\partial x}=2x+2y+0=2x+2y$$

$$\frac{\partial f}{\partial y}=\frac{\partial(x^2+2xy+y^2)}{\partial y}=\frac{\partial x^2+\partial(2xy)+\partial y^2}{\partial y}=0+2x+2y=2x+2y$$

第2节：求和公式及向量的导数计算

求和公式的导数计算

示例1：

设集合X有n个元素$x_1, x_2, \cdots, x_n$，集合Y有n个元素$y_1, y_2, \cdots, y_n$，现有变量a和b及函数：

$f(a,b) = (ax_1 + b - y_1) + (ax_2 + b - y_2) + \cdots + (ax_n + b - y_n) = \sum_{i=1}^{n}(ax_i + b - y_i)$

计算$\frac{\partial f}{\partial a}$ 和 $\frac{\partial f}{\partial b}$

解答： 在计算时，请注意：

- x_i和y_i 都应视为常量。并不是要针对它们来求导
- 把a和b视为求导自变量
- 遵守复合函数的导数计算法则

$\frac{\partial f}{\partial a} = \sum_{i=1}^{n} x_i$ $\frac{\partial f}{\partial b} = \sum_{i=1}^{n} 1 = n$

示例2：

已知：$f(a,b) = \frac{1}{2n}\sum_{i=1}^{n}(b + ax_i - y_i)^2$ 计算：$\frac{\partial f}{\partial a}$ 和 $\frac{\partial f}{\partial b}$

解答：

$\frac{\partial f}{\partial a} = \frac{1}{2n}\sum_{i=1}^{n}[2 \cdot (b + ax_i - y_i) \cdot \frac{\partial (b + ax_i - y_i)}{\partial a}] = \frac{1}{n}\sum_{i=1}^{n}[(b + ax_i - y_i) \cdot x_i]$

$\frac{\partial f}{\partial b} = \frac{1}{2n}\sum_{i=1}^{n}[2 \cdot (b + ax_i - y_i) \cdot \frac{\partial (b + ax_i - y_i)}{\partial b}] = \frac{1}{n}\sum_{i=1}^{n}(b + ax_i - y_i)$

示例3：

已知m维行向量w和m维列向量x，定义函数：$f(w,x) = wx$ 计算：$\frac{\partial f}{\partial w}$和$\frac{\partial f}{\partial x}$

解答：

w是向量，因此$\frac{\partial f}{\partial w}$可以分解成$f$对$w$中的每一个分量分别求偏导，然后再组合成向量：

$\frac{\partial f}{\partial w} = (\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \cdots, \frac{\partial f}{\partial w_m})$

$= (\frac{\partial (w_1x_1 + w_2x_2 + \cdots + w_mx_m)}{\partial w_1}, \frac{\partial (w_1x_1 + w_2x_2 + \cdots + w_mx_m)}{\partial w_2}, \cdots, \frac{\partial (w_1x_1 + w_2x_2 + \cdots + w_mx_m)}{\partial w_m})$

在针对w的每个分量求导数时，其它分量均视为常数，因此：

$\frac{\partial f}{\partial w} = (x_1, x_2, \cdots, x_m) = x^T$

类似的，可得：

$$\frac{\partial f}{\partial x} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{pmatrix} = w^T$$

示例**4**：

设w为m维行向量，包含标量元素$w_1, w_2, \cdots, w_m$，函数f定义如下：$f(w) = ww^T$ 计算：$\frac{\partial f}{\partial w}$

解答：

$$\begin{aligned}
\frac{\partial f}{\partial w} &= (\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \cdots, \frac{\partial f}{\partial w_m}) \\
&= (\frac{\partial (w_1^2 + w_2^2 + \cdots + w_m^2)}{\partial w_1}, \frac{\partial (w_1^2 + w_2^2 + \cdots + w_m^2)}{\partial w_2}, \cdots, \frac{\partial (w_1^2 + w_2^2 + \cdots + w_m^2)}{\partial w_m}) \\
&= (\frac{\partial w_1^2}{\partial w_1}, \frac{\partial w_2^2}{\partial w_2}, \cdots, \frac{\partial w_m^2}{\partial w_m}) \\
&= \frac{\partial w w^T}{\partial w} \\
&= 2w
\end{aligned}$$

示例**5**：

已知(m, n)矩阵W和n维列向量x，定义函数：$f(W, x) = Wx$ 计算：$\frac{\partial f}{\partial W}$

解答：

- 考虑$\frac{\partial f}{\partial W}$

  f是(m个元素的)列向量，因此$\frac{\partial f}{\partial W}$ 可以先分解成f中的每一个元素对W求偏导：

  $$\frac{\partial f}{\partial W} = \begin{pmatrix} \frac{\partial f_1}{\partial W} \\ \frac{\partial f_2}{\partial W} \\ \vdots \\ \frac{\partial f_m}{\partial W} \end{pmatrix}$$

- 考虑$\frac{\partial f_i}{\partial W}$

  W是(m, n)矩阵，f_i是标量，因此，$\frac{\partial f_i}{\partial W}$ 可以分解成f_i对W中的每一个元素求偏导：

$$\frac{\partial f_i}{\partial W} = \begin{pmatrix} \frac{\partial f_i}{\partial W_{11}} & \frac{\partial f_i}{\partial W_{12}} & \cdots & \frac{\partial f_i}{\partial W_{1n}} \\ \frac{\partial f_i}{\partial W_{21}} & \frac{\partial f_i}{\partial W_{22}} & \cdots & \frac{\partial f_i}{\partial W_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_i}{\partial W_{m1}} & \frac{\partial f_i}{\partial W_{m2}} & \cdots & \frac{\partial f_i}{\partial W_{mn}} \\ \end{pmatrix}$$

在计算特定元素$\frac{\partial f_i}{\partial W_{kj}}$ 时，只需要考虑$\boldsymbol{f_i}$中来源于W_{kj}元素的那些项(其余项针对W_{kj}的导数都为0)。而根据$\boldsymbol{f = Wx}$，$\boldsymbol{f_i}$的值仅仅与W中的第i行有关，W_{kj}也仅仅与x_j进行乘积计算，因此:

- 当$i \neq k$时，$\frac{\partial f_i}{\partial W_{kj}} = 0$
- 当$i = k$时，$\frac{\partial f_i}{\partial W_{kj}} = x_j$

  注意$\frac{\partial f_i}{\partial W_{kj}}$和$x_j$都是标量，进而可得$W$中的第$i$行元素的导数为：

- 当$i \neq k$时，$\frac{\partial f_i}{\partial W_{k}} = 0$
- 当$i = k$时，$\frac{\partial f_i}{\partial W_{k}} = x$

综合起来如下(仅第i行不为0)：

$$\frac{\partial f_i}{\partial W} = \begin{pmatrix} 0 & 0 & ... & 0 \\ 0 & 0 & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ x_1 & x_2 & \cdots & x_n \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & 0 \\ \end{pmatrix}$$

- $\frac{\partial f}{\partial W}$ 尺寸大小

通过上面的推算过程可知，$\frac{\partial f}{\partial W}$实际上是个$m$维向量，向量中每个元素都是一个$(m, n)$矩阵。

因此$\frac{\partial f}{\partial W}$共$m \times m \times n$个元素

第3节：矩阵的链式求导

矩阵链式求导的复杂性

在机器学习算法中，经常需要针对矩阵进行求导，例如：

已知矩阵$X:(N,D)$，矩阵$W:(D,M)$，以及矩阵$Y=XW:(N,M)$

假设标量 $L=\sum_{i=1}^{N}\sum_{j=1}^{M}f(y_{ij})$，其中，$y_{ij}$表示矩阵$Y$中的第$(i,j)$个元素，$f(y_{ij})$是基于$y_{ij}$的某种函数定义。也就是说，$L$是有许多个$f(y_{ij})$函数值求和而得。

计算：$\frac{\partial L}{\partial X}$和$\frac{\partial L}{\partial W}$

注意：

- 因为X维度为(N,D)，而L是标量，因此$\frac{\partial L}{\partial X}$实际上可看成$L$针对$X$中的每一个元素分别求导，最终形成$(N,D)$矩阵
- 因为W维度为(D,M)，而L是标量，因此$\frac{\partial L}{\partial W}$形成$(D,M)$矩阵
- Y维度为(N,M)，X维度为(N,D)，$\frac{\partial Y}{\partial X}$ 计算方法是：Y矩阵中的每个元素都需要分别针对X矩阵中的每个元素求导，最终将会有$N\times M\times N\times D$个导数值，这些导数值将构成**Jacobian**矩阵。
- 在机器学习中，N、D、M的值可能会比较大，例如：$N=64,D=4096,M=4096$，则$N\times M\times N\times D=64\times 1024\times 1024\times 1024$个元素，如果每个元素用32位浮点数存储(4字节)，则共消耗$4\times 64\times 1024\times 1024\times 1024=256GB$内存，这是不现实的
- 因此，必须要找出一种办法，在不计算Jacobian矩阵的情况下，求解偏导数。而因为Y是X和W经过线性变换而得的，故存在着化简的可能性

矩阵链式求导公式

- 已知矩阵$X:(N,D)$，矩阵$W:(D,M)$，以及矩阵$Y=XW:(N,M)$，若L是关于Y的函数，则：

  $\frac{\partial L}{\partial X}=\frac{\partial L}{\partial Y}W^T$

  $\frac{\partial L}{\partial W}=X^T\frac{\partial L}{\partial Y}$

- 针对上一结论的变形：已知矩阵$W:(N,D)$，矩阵$X:(D,M)$，以及矩阵$Y=WX:(N,M)$，若L是关于Y的函数，则：

$\frac{\partial L}{\partial W}=\frac{\partial L}{\partial Y}X^T$

$\frac{\partial L}{\partial X}=W^T\frac{\partial L}{\partial Y}$

推导上述矩阵链式求导公式

考查$N=2,D=4,M=3$的特定情况： $X=\begin{pmatrix}x_{11}&x_{12}&x_{13}&x_{14}\\x_{21}&x_{22}&x_{23}&x_{24}\end{pmatrix}$

$W=\begin{pmatrix}w_{11}&w_{12}&w_{13}\\w_{21}&w_{22}&w_{23}\\w_{31}&w_{32}&w_{33}\\w_{41}&w_{42}&w_{43}\end{pmatrix}$

$Y=XW$

$=\begin{pmatrix}x_{11}w_{11}+x_{12}w_{21}+x_{13}w_{31}+x_{14}w_{41} & x_{11}w_{12}+x_{12}w_{22}+x_{13}w_{32}+x_{14}w_{42}\rightarrow\\ \rightarrow x_{11}w_{13}+x_{12}w_{23}+x_{13}w_{33}+x_{14}w_{43} & \\ x_{21}w_{11}+x_{22}w_{21}+x_{23}w_{31}+x_{24}w_{41} & x_{21}w_{12}+x_{22}w_{22}+x_{23}w_{32}+x_{24}w_{42}\rightarrow\\ \rightarrow x_{21}w_{13}+x_{22}w_{23}+x_{23}w_{33}+x_{24}w_{43} & \end{pmatrix}$

$=\begin{pmatrix}y_{11}&y_{12}&y_{13}\\y_{21}&y_{22}&y_{23}\end{pmatrix}$

$L=f(y_{11})+f(y_{12})+f(y_{13})+f(y_{21})+f(y_{22})+f(y_{23})=\sum_{i,j}f(y_{ij})$

上式中，$f(y_{ij})$是关于y_{ij}的函数，这些函数值累加起来形成目标函数L

假设已经求得$\frac{\partial L}{\partial Y}$，请计算$\frac{\partial L}{\partial W}$和$\frac{\partial L}{\partial X}$

- 观察已知的$\frac{\partial L}{\partial Y}$

  考虑到：$L=f(y_{11})+f(y_{12})+f(y_{13})+f(y_{21})+f(y_{22})+f(y_{23})$ 可知：$\frac{\partial L}{\partial f(y_{ij})}=1$ L是标量，Y是$(2,3)$矩阵，导数也应是$(2,3)$矩阵，表示为：

  $\frac{\partial L}{\partial Y}=\begin{pmatrix}\frac{\partial L}{\partial y_{11}}&\frac{\partial L}{\partial y_{12}}&\frac{\partial L}{\partial y_{13}}\\\frac{\partial L}{\partial y_{21}}&\frac{\partial L}{\partial y_{22}}&\frac{\partial L}{\partial y_{23}}\end{pmatrix}$

  $=\begin{pmatrix}\frac{\partial L}{\partial f(y_{11})}\cdot\frac{\partial f(y_{11})}{\partial y_{11}}&\frac{\partial L}{\partial f(y_{12})}\cdot\frac{\partial f(y_{12})}{\partial y_{12}}&\frac{\partial L}{\partial f(y_{13})}\cdot\frac{\partial f(y_{13})}{\partial y_{13}}\\\frac{\partial L}{\partial f(y_{21})}\cdot\frac{\partial f(y_{21})}{\partial y_{21}}&\frac{\partial L}{\partial f(y_{22})}\cdot\frac{\partial f(y_{22})}{\partial y_{22}}&\frac{\partial L}{\partial f(y_{23})}\cdot\frac{\partial f(y_{23})}{\partial y_{23}}\end{pmatrix}$

  $=\begin{pmatrix}\frac{\partial f(y_{11})}{\partial y_{11}}&\frac{\partial f(y_{12})}{\partial y_{12}}&\frac{\partial f(y_{13})}{\partial y_{13}}\\\frac{\partial f(y_{21})}{\partial y_{21}}&\frac{\partial f(y_{22})}{\partial y_{22}}&\frac{\partial f(y_{23})}{\partial y_{23}}\end{pmatrix}$

- 考查$\frac{\partial L}{\partial x_{11}}$

$$
\begin{aligned}
L =& f(y_{11}) + f(y_{12}) + f(y_{13}) + f(y_{21}) + f(y_{22}) + f(y_{23}) \\
=& f(x_{11}w_{11} + x_{12}w_{21} + x_{13}w_{31} + x_{14}w_{41}) + f(x_{11}w_{12} + x_{12}w_{22} + x_{13}w_{32} + x_{14}w_{42}) + \\
& f(x_{11}w_{13} + x_{12}w_{23} + x_{13}w_{33} + x_{14}w_{43}) + f(x_{21}w_{11} + x_{22}w_{21} + x_{23}w_{31} + x_{24}w_{41}) + \\
& f(x_{21}w_{12} + x_{22}w_{22} + x_{23}w_{32} + x_{24}w_{42}) + f(x_{21}w_{13} + x_{22}w_{23} + x_{23}w_{33} + x_{24}w_{43})
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L}{\partial x_{11}} =& \frac{\partial (f(y_{11}) + f(y_{12}) + f(y_{13}) + f(y_{21}) + f(y_{22}) + f(y_{23}))}{\partial x_{11}} \\
=& \frac{\partial f(y_{11})}{\partial x_{11}} + \frac{\partial f(y_{12})}{\partial x_{11}} + \frac{\partial f(y_{13})}{\partial x_{11}} + \frac{\partial f(y_{21})}{\partial x_{11}} + \frac{\partial f(y_{22})}{\partial x_{11}} + \frac{\partial f(y_{23})}{\partial x_{11}}
\end{aligned}
$$

而根据链式法则(注意，此时y_{11} 和x_{11} 都已是一元变量，因此可以使用通常意义上的链式法则)：

$$
\frac{\partial f(y_{11})}{\partial x_{11}} = \frac{\partial f(y_{11})}{\partial y_{11}} \cdot \frac{\partial y_{11}}{\partial x_{11}} = \frac{\partial L}{\partial y_{11}} \cdot \frac{\partial y_{11}}{\partial x_{11}}
$$

因此：

$$
\begin{aligned}
\frac{\partial L}{\partial x_{11}} =& \frac{\partial L}{\partial y_{11}} \cdot \frac{\partial y_{11}}{\partial x_{11}} + \frac{\partial L}{\partial y_{12}} \cdot \frac{\partial y_{12}}{\partial x_{11}} + \frac{\partial L}{\partial y_{13}} \cdot \frac{\partial y_{13}}{\partial x_{11}} + \frac{\partial L}{\partial y_{21}} \cdot \frac{\partial y_{21}}{\partial x_{11}} + \frac{\partial L}{\partial y_{22}} \cdot \frac{\partial y_{22}}{\partial x_{11}} + \frac{\partial L}{\partial y_{23}} \cdot \frac{\partial y_{23}}{\partial x_{11}} \\
=& \frac{\partial L}{\partial y_{11}} \cdot w_{11} + \frac{\partial L}{\partial y_{12}} \cdot w_{12} + \frac{\partial L}{\partial y_{13}} \cdot w_{13}
\end{aligned}
$$

类似的：

$$
\begin{aligned}
\frac{\partial L}{\partial x_{12}} =& \frac{\partial L}{\partial y_{11}} \cdot \frac{\partial y_{11}}{\partial x_{12}} + \frac{\partial L}{\partial y_{12}} \cdot \frac{\partial y_{12}}{\partial x_{12}} + \frac{\partial L}{\partial y_{13}} \cdot \frac{\partial y_{13}}{\partial x_{12}} + \frac{\partial L}{\partial y_{21}} \cdot \frac{\partial y_{21}}{\partial x_{12}} + \frac{\partial L}{\partial y_{22}} \cdot \frac{\partial y_{22}}{\partial x_{12}} + \frac{\partial L}{\partial y_{23}} \cdot \frac{\partial y_{23}}{\partial x_{12}} \\
=& \frac{\partial L}{\partial y_{11}} * w_{21} + \frac{\partial L}{\partial y_{12}} \cdot w_{22} + \frac{\partial L}{\partial y_{13}} \cdot w_{23}
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L}{\partial x_{21}} =& \frac{\partial L}{\partial y_{11}} \cdot \frac{\partial y_{11}}{\partial x_{21}} + \frac{\partial L}{\partial y_{12}} \cdot \frac{\partial y_{12}}{\partial x_{21}} + \frac{\partial L}{\partial y_{13}} \cdot \frac{\partial y_{13}}{\partial x_{21}} + \frac{\partial L}{\partial y_{21}} \cdot \frac{\partial y_{21}}{\partial x_{21}} + \frac{\partial L}{\partial y_{22}} \cdot \frac{\partial y_{22}}{\partial x_{21}} + \frac{\partial L}{\partial y_{23}} \cdot \frac{\partial y_{23}}{\partial x_{21}} \\
=& \frac{\partial L}{\partial y_{21}} \cdot w_{11} + \frac{\partial L}{\partial y_{22}} \cdot w_{12} + \frac{\partial L}{\partial y_{23}} \cdot w_{13}
\end{aligned}
$$

等等

- 表达成矩阵形式

$$
\begin{aligned}
& \frac{\partial L}{\partial X} \\
=& \begin{pmatrix} \frac{\partial L}{\partial x_{11}} & \frac{\partial L}{\partial x_{12}} & \frac{\partial L}{\partial x_{13}} & \frac{\partial L}{\partial x_{14}} \\ \frac{\partial L}{\partial x_{21}} & \frac{\partial L}{\partial x_{22}} & \frac{\partial L}{\partial x_{23}} & \frac{\partial L}{\partial x_{24}} \end{pmatrix} \\
=& \begin{pmatrix} \frac{\partial L}{\partial y_{11}} \cdot w_{11} + \frac{\partial L}{\partial y_{12}} \cdot w_{12} + \frac{\partial L}{\partial y_{13}} \cdot w_{13} & \frac{\partial L}{\partial y_{11}} \cdot w_{21} + \frac{\partial L}{\partial y_{12}} \cdot w_{22} + \frac{\partial L}{\partial y_{13}} \cdot w_{23} \rightarrow \\ \rightarrow \frac{\partial L}{\partial y_{11}} \cdot w_{31} + \frac{\partial L}{\partial y_{12}} \cdot w_{32} + \frac{\partial L}{\partial y_{13}} \cdot w_{33} & \frac{\partial L}{\partial y_{11}} \cdot w_{41} + \frac{\partial L}{\partial y_{12}} \cdot w_{42} + \frac{\partial L}{\partial y_{13}} \cdot w_{43} \\ \frac{\partial L}{\partial y_{21}} \cdot w_{11} + \frac{\partial L}{\partial y_{22}} \cdot w_{12} + \frac{\partial L}{\partial y_{23}} \cdot w_{13} & \frac{\partial L}{\partial y_{21}} \cdot w_{21} + \frac{\partial L}{\partial y_{22}} \cdot w_{22} + \frac{\partial L}{\partial y_{23}} \cdot w_{23} \rightarrow \\ \rightarrow \frac{\partial L}{\partial y_{21}} \cdot w_{31} + \frac{\partial L}{\partial y_{22}} \cdot w_{32} + \frac{\partial L}{\partial y_{23}} \cdot w_{33} & \frac{\partial L}{\partial y_{21}} \cdot w_{41} + \frac{\partial L}{\partial y_{22}} \cdot w_{42} + \frac{\partial L}{\partial y_{23}} \cdot w_{43} \end{pmatrix} \\
=& \begin{pmatrix} \frac{\partial L}{\partial y_{11}} & \frac{\partial L}{\partial y_{12}} & \frac{\partial L}{\partial y_{13}} \\ \frac{\partial L}{\partial y_{21}} & \frac{\partial L}{\partial y_{22}} & \frac{\partial L}{\partial y_{23}} \end{pmatrix} \cdot \begin{pmatrix} w_{11} & w_{21} & w_{31} & w_{41} \\ w_{12} & w_{22} & w_{32} & w_{42} \\ w_{13} & w_{23} & w_{33} & w_{43} \end{pmatrix} \\
=& \frac{\partial L}{\partial Y} W^T
\end{aligned}
$$

- 类似的可得：$\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Y}$

- 有了上述推论，在不计算Jacobian矩阵的情况下，也能比较方便的求得偏导数。上述结论虽然是在$N = 2, D = 4, M = 3$情况下得出的，但一般也具有普遍性

第4节：反向传播图解法求导

图解法

在链式求导过程中，如果“链条”较长，那么很容易出现错漏。图解法提供了正向计算和反向求导计算的流程图，有助于理解求导过程

示例**1**：

已知$f(x,y,z)=(x+y)\cdot z$，请计算$x=-2,y=5,z=-4$时的导数$\frac{\partial f}{\partial x}$，$\frac{\partial f}{\partial y}$和$\frac{\partial f}{\partial z}$

- 直接计算

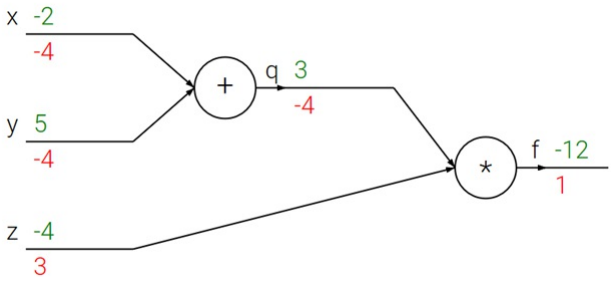
  令$q=(x+y),f=qz$，很显然有：

  $\frac{\partial f}{\partial x}=\frac{\partial f}{\partial q}\frac{\partial q}{\partial x}=z\cdot 1=z=-4$

  $\frac{\partial f}{\partial y}=\frac{\partial f}{\partial q}\frac{\partial q}{\partial y}=z\cdot 1=z=-4$

  $\frac{\partial f}{\partial z}=q=x+y=3$

- 画出导数计算流图如下：



- 解释
  - 上图中，横线上方的绿色数值代表该变量的值，横线下方的红色数值代表该标量相对最终计算结果(f)的导数值
  - 最右边节点相对自己的导数恒为1
  - 从右向左计算各个节点的导数
  - 两个变量相乘，其针对最终计算结果(f)的导数，等于下游(右侧)节点导数值乘以另一个变量。（有置换的效果）
  - 两个变量相加，其针对最终计算结果(f)的导数，等于下游节点导数值。相当于下游节点的导数完整的流回给各个变量

示例**2**：

已知$f(w,x)=\frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$，使用导数计算流计算当

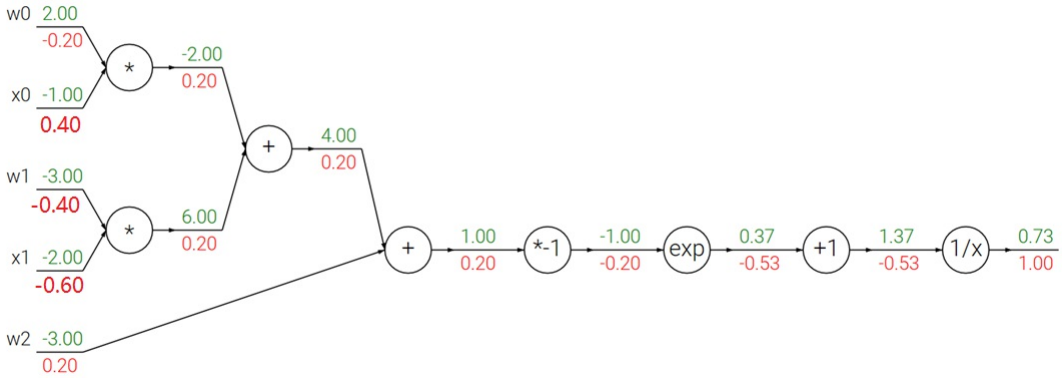
$w_0=2.0,x_0=-1.0,w_1=-3.0,x_1=-2.0,w_2=-3.0$时的各个导数值

- 回顾下列常用导函数

  $f(x)=\frac{1}{x}=>f'(x)=-\frac{1}{x^2}$

  $f(x)=e^x=>f'(x)=e^x$

- 做出导数流图



示例**3**：

已知W为(N,M)矩阵，x为M维列向量，$f(W,x)=||Wx||^2=\sum_{i=1}^{N}(Wx)_i{}^2$，计算当

$W=\begin{pmatrix}0.1&0.5\\-0.3&0.8\end{pmatrix},x=\begin{pmatrix}0.2\\0.4\end{pmatrix}$ 时的$\frac{\partial f}{\partial W}$和$\frac{\partial f}{\partial x}$

- 推导计算

  令：$y=Wx=\begin{pmatrix}w_{11}x_1+w_{12}x_2+\cdots+w_{1m}x_m\\w_{21}x_1+w_{22}x_2+\cdots+w_{2m}x_m\\\vdots\\w_{n1}x_1+w_{n2}x_2+\cdots+w_{nm}x_m\end{pmatrix}=\begin{pmatrix}y_1\\y_2\\\vdots\\y_n\end{pmatrix}$ 显然，y是N维列向量

  $$\begin{aligned}f=&(w_{11}x_1+w_{12}x_2+\cdots+w_{1m}x_m)^2+(w_{21}x_1+w_{22}x_2+\cdots+w_{2m}x_m)^2+\cdots+\\&(w_{n1}x_1+w_{n2}x_2+\cdots+w_{nm}x_m)^2\\=&\sum_{k=1}^{n}y_k{}^2\end{aligned}$$

  从上式可知，当$k\neq i$时，$\frac{\partial y_k}{\partial w_{ij}}=0$

  而当$k=i$时，$\frac{\partial y_k}{\partial w_{ij}}=\frac{\partial(w_{i1}x_1+w_{i2}x_2+\cdots+w_{im}x_m)}{\partial w_{ij}}=x_j$

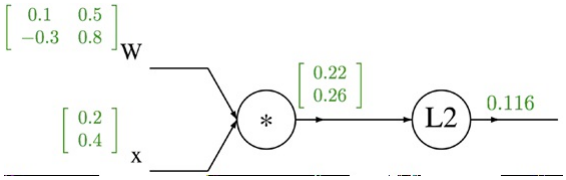
  可得：$\frac{\partial f}{\partial w_{ij}}=\sum_{k=1}^{n}\frac{\partial f}{\partial y_k}\cdot\frac{\partial y_k}{\partial w_{ij}}=\frac{\partial f}{\partial y_i}\cdot\frac{\partial y_i}{\partial w_{ij}}=2y_i\cdot x_j$

  从而，针对第i行w_i：$\frac{\partial f}{\partial w_i}=2y_i\cdot x^T$，其结果是一个$M$维行向量(与$x^T$一致)

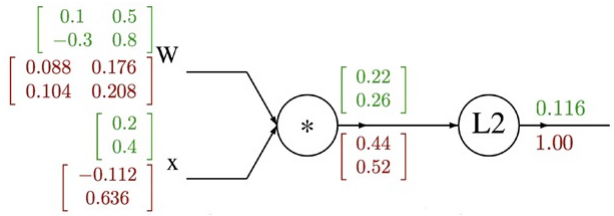
最终，针对整个矩阵W：$\frac{\partial f}{\partial W} = 2yx^T$ 其中，y是N维列向量，x^T是M维行向量，yx^T产生(N, M)矩阵

同理可得：$\frac{\partial f}{\partial x} = 2W^Ty$ 其中，W^T是(M, N)矩阵，y是N维列向量，W^Ty产生M维列向量 上述结论也可以通过上一节的矩阵链式求导公式得出。

- 使用导数计算流 正向计算：



反向求导： 计算$\frac{\partial f}{\partial W}$及$\frac{\partial f}{\partial x}$ 时，将上面推算出的$\frac{\partial f}{\partial W} = 2yx^T$和$\frac{\partial f}{\partial x} = 2W^Ty$代入求解：



第5节：求函数极限值

利用导函数求极值点

求函数$f(x)$的极值的步骤

1. 计算其导函数$f'(x)$，然后令$f'(x)=0$，得出此时的解x_0
2. 计算导函数$f'(x)$在$x=x_0$处的导数(也就是$f(x)$的二阶导数）$f''(x_0)$
    - 如果$f''(x_0)>0$，则x_0为极小值点
    - 如果$f''(x_0)<0$，则x_0为极大值点
    - 如果$f''(x_0)=0$，则x_0为非极值点，而是拐点

示例**1**：利用导数计算$f(x)=x^2$的极值

$f'(x)=2x$，令$f'(x)=0$，得：$x_0=0$

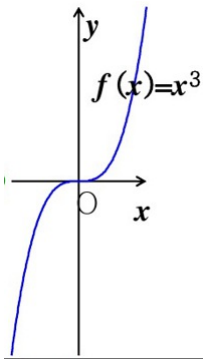
$f''(x_0)=2$，因此$x_0=0$是极小值点

极小值为$f(x_0)=f(0)=0$

示例**2**：利用导数计算$f(x)=x^3$的极值

$f'(x)=3x^2$，令$f'(x)=0$，得：$x_0=0$

$f''(x_0)=6x_0=0$，因此这个点不是极值点而是拐点
可通过其图像看到这一点：

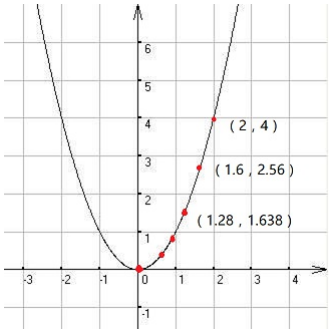


采用数值计算求函数极值

选择一个初始点，然后计算该点的导数，再通过导数和步长推进到下一个点，直到两个点之间的差值很小为止

示例**3**：以数据计算方法计算$f(x)=x^2$的极值

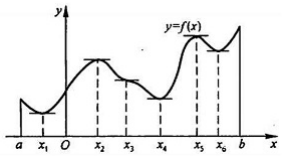
- 首先做出函数图像如下：



- 按照下列步骤执行：

1. 选定一个初始值，例如$x_0=2$
2. 计算在该点的导数，$f'(x_0)=4$
3. 按照下列公式调整x的新值：$x_1=x_0-\alpha f'(x_0)$
    - α称为步进系数(学习速率)，用来控制步长大小。例如设置为0.1, 0.001等
    - $f'(x_0)$主要用来控制步长的方向。在本例中是通过正负号来控制
    - 注意，要与导数的方向相反（所以用减号），否则将会离极值点越来越远
4. 计算$f(x_1)$，并且与$f(x_0)$对比，根据某个规则来判断是否已经收敛，例如：二者的差值小于某个临界误差，例如0.000001。如果尚未收敛，则继续进行上述循环
5. 如果在指定循环次数（例如1000次）之后仍然没有收敛，则可以认为该函数没有极值

- 说明
    - 上图中，从点(2,4)出发，将逐步逼近(0,0)。
    - 如果初始值取为负数，则导数也为负，根据$x_1=x_0-\alpha f'(x_0)$，x_1将大于x_0，也就是向原点方向移动，这也是正确的。
    - 因此，通过$x_1=x_0-\alpha f'(x_0)$来不断逼近，是一种合适的方法
- 注意：如果函数本身有多个极值点，那么逼近法找到的是其中的一个，未必是最低的极值点：



示例**1**：编写一个程序，以数值方法计算函数的极小值

- 尝试调整alpha, difference, max_iter等参数的值，观察其收敛性及收敛速度
- 代码参考：【04/01minimize_calculation.py】

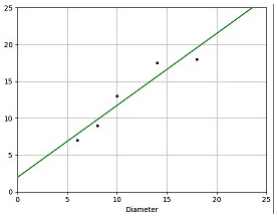
-

第五章：数值优化

第1节：最小二乘法曲线拟合

单变量线性拟合

- 假设平面上有若干个数据点(x_i, y_i)，其中$i = 1, 2, \cdots, n$。x作为唯一的自变量，y作为因变量。作一条曲线，使得“各数据点到该曲线的距离最近”：



- 如何定义“各数据点到该曲线的距离最近”
  - 首先定义残差概念如下：$r_i = h(x_i) - y_i$，其中$h(x_i)$代表拟合后的曲线函数在第i个数据点x_i处的计算结果
  - 根据范数的定义，可知：
    - 正无穷范数(残差绝对值的最大值)：$F(+\infty) = max(|r_i|)$
    - 1范数(残差绝对值之和)：$F(1) = \sum_i^n |r_i|$
    - 2范数(残差平方和)：$F(2) = \sum_i^n r_i^2$
  - 一般使用2范数的大小来判断“各数据点到该曲线的距离最近”，这种方法就称为最小二乘法(Least Square Method)
- 拟合曲线的类型
  - 可以使用直线、二次曲线、高阶曲线等进行拟合
  - 无论哪种曲线，都可以表示成：$h(x) = w_0 + w_1 x + w_2 x^2 + \cdots + w_m x^m$，其中，$m$指定了拟合曲线的最大阶数
  - 最小二乘法的目标，就是要找到向量$w = (w_0, w_1, \cdots, w_m)$，使得针对所有的数据点，残差的2范数最小
  - 当$m = 1$时，拟合结果就是一条直线

最小二乘法直线拟合公式的推导

- 目标函数：$F(w_0, w_1) = \sum_i^n (w_0 + w_1 x_i - y_i)^2$，现在要求使$F$最小的$w_0$和$w_1$
- 要计算函数的极值，可以采用求一阶偏导，并使之为0

$$\begin{cases} \frac{\partial F}{\partial w_0} = 2\sum_i^n (w_0 + w_1 x_i - y_i) = 0 \\ \frac{\partial F}{\partial w_1} = 2\sum_i^n (w_0 + w_1 x_i - y_i) x_i = 0 \end{cases}$$

- 公式推导
  - 根据$\frac{\partial F}{\partial w_0} = 0$，可得：$w_0 = \bar{y} - w_1 \bar{x}$，其中：
    - $\bar{y}$：训练数据中y的平均值
    - $\bar{x}$：训练数据中x的平均值
  - 将$w_0 = \bar{y} - w_1 \bar{x}$代入到$\frac{\partial F}{\partial w_1} = 0$，可得：

    $\frac{\partial F}{\partial w_1} = \sum_{i=1}^n (\bar{y} - w_1 \bar{x} + w_1 x_i - y_i) x_i$

    $= \bar{y} \sum x_i - \sum x_i y_i - w_1 (\bar{x} \sum x_i - \sum x_i^2)$

    $= 0$

    进而可得：

    $$w_1 = \frac{\bar{y} \sum x_i - \sum x_i y_i}{\bar{x} \sum x_i - \sum x_i^2} = \frac{n\bar{y}\bar{x} - \sum x_i y_i}{n\bar{x}^2 - \sum x_i^2} = \frac{\sum x_i y_i - n\bar{x}\bar{y}}{\sum x_i^2 - n\bar{x}^2}$$

    注意，上式中用到了下列事实：

    $$\sum x_i = n\bar{x}$$

  - 引入两个重要的推论式：

    $$\sum_{i=1}^n (x_i - \bar{x})^2 = \sum x_i^2 - n\bar{x}^2$$

    $$\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y}) = \sum x_i y_i - n\bar{x}\bar{y}$$

    证明如下：

    $\sum_{i=1}^n (x_i - \bar{x})^2$

    $= (x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \cdots + (x_n - \bar{x})^2$

    $= (x_1^2 - 2x_1\bar{x} + \bar{x}^2) + (x_2^2 - 2x_2\bar{x} + \bar{x}^2) + \cdots + (x_n^2 - 2x_n\bar{x} + \bar{x}^2)$

    $= (x_1^2 + x_2^2 + \cdots + x_n^2) + n\bar{x}^2 - 2\bar{x}(x_1 + x_2 + \cdots + x_n)$

    $= \sum x_i^2 + n\bar{x}^2 - 2\bar{x} * n * \bar{x}$

    $= \sum x_i^2 - n\bar{x}^2$

$$
\begin{aligned}
&\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})\\
=&(x_1-\bar{x})(y_1-\bar{y})+(x_2-\bar{x})(y_2-\bar{y})+\cdots+(x_n-\bar{x})(y_n-\bar{y})\\
=&(x_1y_1+\bar{x}\bar{y}-x_1\bar{y}-y_1\bar{x})+(x_2y_2+\bar{x}\bar{y}-x_2\bar{y}-y_2\bar{x})+\cdots+(x_ny_n+\bar{x}\bar{y}-x_n\bar{y}-y_n\bar{x})\\
=&(x_1y_1+x_2y_2+\cdots+x_ny_n)+n\bar{x}\bar{y}-\bar{y}(x_1+x_2+\cdots+x_n)-\bar{x}(y_1+y_2+\cdots+y_n)\\
=&\sum x_iy_i+n\bar{x}\bar{y}-\bar{y}\cdot n\cdot\bar{x}-\bar{x}\cdot n\cdot\bar{y}\\
=&\sum x_iy_i-n\bar{x}\bar{y}
\end{aligned}
$$

- 结合上述w_0和w_1计算结果及推论式，进一步得到：

$$
\begin{cases}
w_0=\bar{y}-w_1\bar{x}\\
\\
w_1=\dfrac{x_iy_i-n\bar{x}\bar{y}}{\sum x_i^2-n\bar{x}^2}=\dfrac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=1}^{n}(x_i-\bar{x})^2}
\end{cases}
$$

回顾协方差和方差的概念，上述结果也可以写成：

$$
\begin{cases}
w_0=\bar{y}-w_1\bar{x}\\
\\
w_1=\frac{cov(x,y)}{var(x)}
\end{cases}
$$

其中，$cov(x,y)$是数组x,y的协方差，$var(x)$是数组x的方差

示例**1**：使用协方差-方差公式拟合数据

- 代码参考：【05/01least_square_fit_single_variant.py】

线性拟合的一般解法

假设有m个样本，每个样本包括一个n维度向量(多变量)和一个对应的结果值(y)，且$m\geq n+1$

$$
S=\begin{pmatrix}
x_11 & x_12 & \cdots & x_1n\\
x_21 & x_22 & \cdots & x_2n\\
\vdots & \vdots & \ddots & \vdots\\
x_m1 & x_m2 & \cdots & x_mn
\end{pmatrix}
$$

$y=(y_1,y_2,\cdots,y_m)^T$

需要拟合成下列线性形式：$F(x)=Sa+b$，其中，a是n维向量，b是标量

- 样本数据变形

  令：$X=\begin{pmatrix}
  1 & x_11 & x_12 & \cdots & x_1n\\
  1 & x_21 & x_22 & \cdots & x_2n\\
  \vdots & \vdots & \vdots & \ddots & \vdots\\
  1 & x_m1 & x_m2 & \cdots & x_mn
  \end{pmatrix}$

也就是说，在S第1列插入一列全为1的列

$w = (w_0, w_1, w_2, \cdots, w_{n-1}, w_n)^T$

则拟合后的函数$f = Sa + b$转变成：$f(X, w) = Xw$

全为1的列在X中的位置是可以调整的，只要保证与w中的w_0位置对应就行

- 残差函数

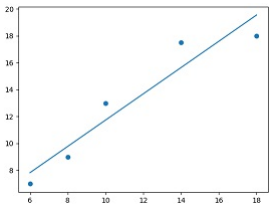
  $Err(w, X, y) = Xw - y$

- 最小二乘解法

  - 矩阵解法为(本课程不作推导证明)：$w = (X^T X)^{-1} X^T y$
  - 通过数值迭代算法逐渐逼近求解，例如梯度下降法、高斯牛顿法等
  - 使用scipy.optimize.leastsq直接求解

示例**2**：使用矩阵解法求解直线拟合公式

- 矩阵解法需要先在原来的坐标矩阵S中加入一个全为1的列，形成扩展后的矩阵X
- 拟合成直线，则w：$[w_1, w_0]$，注意，w中元素的顺序应该与X中各列的顺序相对应
- 与示例1的结果进行对比，二者非常相近。对于直线拟合，既可以采用例题1中的协方差-方差求解法，也可以采用矩阵求解法
- 拟合结果如下图所示：



- 代码参考：【05/02least_square_fit_matrix_single_variant.py】

示例**3**：使用**scipy**提供的**leastsq**方法求解直线拟合公式

- 要提供残差函数，也就是：$Err(w, x) = f(w, x) - y$
- 而函数$f(w, x)$应根据拟合的曲线而定：
- 提供w的初始估计值
- 代码参考：【05/03scipy_leastsq_fit_single_variant.py】

示例**4**：使用矩阵解法求解高阶曲线拟合公式

- 高阶曲线看起来不是线性的，但是如果对每阶数据进行变换，则仍然可以按线性来处理
- 应先将X坐标数据分别生成2阶、3阶...n阶相应的坐标数据，即：

$$X = \begin{pmatrix} x_1^n & x_1^{n-1} & \cdots & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_m^n & x_m^{n-1} & \cdots & x_m & 1 \end{pmatrix}$$

$w : [w_n, \cdots, w_1, w_0]$

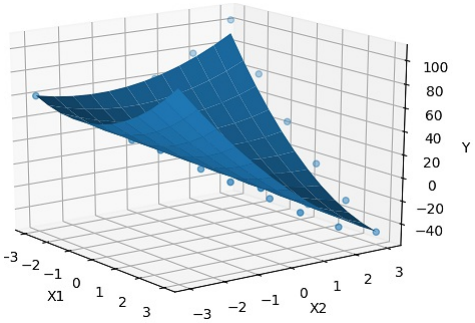
- 可按照Xw线性方法求解
- 代码参考：【05/04least_square_fit_matrix_high_order.py】

示例**5**：使用**scipy**提供的**leastsq**方法求解高阶曲线拟合公式

- 代码参考：【05/05scipy_leastsq_fit_high_order.py】

示例**6**：使用**scipy**提供的**leastsq**方法求解多维、高阶拟合公式

- 对于多维、高阶数据，通过扩展X矩阵，仍然可以视为线性拟合，因此也仍然可以使用矩阵方法来求解。本例直接采用leastsq方法求解。
- 原始样本数据来源于$y = 3x_1^2 - 5x_1x_2 + 3x_2^2 - 8x_1 - 10x_2 + 6$
- 拟合出的二元、二阶多项式一共有6项，每项的系数最终都接近上式中的系数
- 在使用leastsq进行拟合前，需要准备好数据矩阵，多项式中的每一项的数据值需要先计算出来
- 使用了mpl_toolkits.mplot3d.Axes3D来绘制三维散点图和曲面图



- 代码参考：【05/06scipy_leastsq_fit_multi_variants.py】

不适合矩阵方法求解的情形

- 矩阵方法需要计算逆矩阵，这个操作开销很大，而且有些矩阵无法求逆。因此当样本数量较大，维度较多时，不宜采用矩阵方法
- 如果拟合的公式不是线性的，则不能使用矩阵方法
- 因此，一般来说，采用数值迭代算法更为普遍

第2节：梯度下降算法

提出问题

假设下列函数定义：

$F(w)=\frac{1}{2m}\sum_{i=1}^{m}[f_w(x^{(i)}-y^{(i)})]^2=\frac{1}{2m}\sum_{i=1}^{m}(w_0+w_1x^{(i)}-y^{(i)})^2$

其中：

- m：样本数据点数量
- f_w：以w为系数的拟合函数
- $x^{(i)}$：第i个数据的横坐标值(数据中的自变量)
- $y^{(i)}$：第i个数据的纵坐标值(数据中的因变量)
- $f_w(x^{(i)})$：将$x^{(i)}$ 代入到拟合函数计算的结果）
- $F(w)$：也称为目标函数

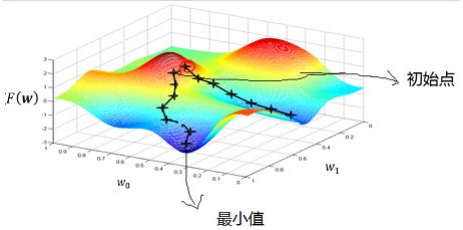
根据最小二乘法可知，针对函数$F(w)$，要设法求出最优的w，从而使得$F(w)$的值最小。求解过程可以使用梯度下降法来实现。

请务必注意：

- 要求出$F(w)$的极值，就是要求出最优的w
- $x^{(i)}$和$y^{(i)}$ 都来自与已知的样本数据，因此对于函数$F(w)$来说，它们并不是变量；w才是变量

工作原理

下图展示了梯度下降算法的工作原理：



- 选择一个初始点(即给定初始的w)，作为当前工作点
- 计算该点出的梯度值(导数$\frac{\partial F}{\partial w}$)，沿着梯度最大的方向，移动当前工作点到新的位置
- 经过若干次移动后，如果新的工作点位置对应的$F(w)$值与上一个位置几乎没有差异，则可以认为找到了极值点

- 请注意，一般来说，采用梯度下降法求取极小值（而不是极大值）

计算步骤

1. 给定目标函数$F(w)$和学习速率(learning rate)：α
2. 随机初始化w_0和w_1，计算出$F(w_0, w_1)$
3. 计算出能够使$F(w)$下降最快的w偏移量/梯度：∇w，叠加到初始w_0和w_1上
    i. 针对w_0和w_1分别计算梯度∇w_0和∇w_1
    ii. 按下列公式调整w_0和w_1：
        - $w_0 = w_0 - \alpha \nabla w_0$
        - $w_1 = w_1 - \alpha \nabla w_1$
        - 请注意，w_0和w_1的变化方向要与导数的方向相反
    iii. 利用调整后的w_0和w_1，再次计算$F(w_0, w_1)$
4. 循环第3步过程，直到最近两次计算出来的目标函数的差值小于某个误差限
5. 此时的w_0和w_1就是目标函数的最优解

注意：

- 可能需要尝试及合理选择α。α太大，有可能越过了最优点；α太小，收敛速度会非常慢
- 当前主流的梯度算法，能够随着梯度下降循环的进行，动态调整α值，从而能够较快的收敛，又能避免跨越最优点的情形
- 找到的最优点，也有可能是局部最优点而非全局最优点

批量梯度下降法

- 梯度下降法中最重要的一步是计算某个工作点的梯度∇w。有多种用于计算∇w的方法，其计算结果也会有所不同
- 批量梯度下降法是指：在计算∇w时，要把所有样本点数据的梯度都计算进去，也就是直接针对$F(w)$来计算偏导(注意$F(w)$中包含了所有样本点的数据值)，其计算公式如下：

$$
\begin{aligned}
\nabla w_0 &= \frac{\partial F(w)}{\partial w_0} \\
&= \frac{\partial \frac{1}{2m}\sum_{i=1}^{n}(w_0 + w_1 x^{(i)} - y^{(i)})^2}{\partial w_0} \\
&= \frac{\frac{1}{2m}\cdot 2 \cdot \sum_{i=1}^{m}[(w_0 + w_1 x^{(i)} - y^{(i)}) \cdot \partial(w_0 + w_1 x^{(i)} - y^{(i)})]}{\partial w_0} \\
&= \frac{1}{m}\sum_{i=1}^{m}[(w_0 + w_1 x^{(i)} - y^{(i)}) \cdot (1 + 0 + 0)] \\
&= \frac{1}{m}\sum_{i=1}^{m}(w_0 + w_1 x^{(i)} - y^{(i)}) \\
\nabla w_1 &= \frac{\partial F(w)}{\partial w_1} \\
&= \frac{\partial \frac{1}{2m}\sum_{i=1}^{n}(w_0 + w_1 x^{(i)} - y^{(i)})^2}{\partial w_1} \\
&= \frac{\frac{1}{2m}\cdot 2 \cdot \sum_{i=1}^{m}[(w_0 + w_1 x^{(i)} - y^{(i)}) \cdot \partial(w_0 + w_1 x^{(i)} - y^{(i)})]}{\partial w_1} \\
&= \frac{1}{m}\sum_{i=1}^{m}[(w_0 + w_1 x^{(i)} - y^{(i)}) \cdot (0 + x^{(i)} + 0)] \\
&= \frac{1}{m}\sum_{i=1}^{m}[(w_0 + w_1 x^{(i)} - y^{(i)}) \cdot x^{(i)}]
\end{aligned}
$$

- 在梯度下降算法的每一步，计算出∇w后，代入下式中进行下一步计算：

$$w_0 = w_0 - \alpha \nabla w_0$$

$$w_1 = w_1 - \alpha \nabla w_1$$

- 特点
  - 算法实现比较简单
  - 对于相同的样本数据点，每次优化会得到相同的优化结果
  - 梯度计算量比较大

示例**1**：使用批量梯度下降法拟合直线

- 代码参考：【05/07batch_gradient_descent_single_variant.py】

示例**2**：使用批量梯度下降法拟合多维数据

- 将批量梯度下降算法拆分到一个新的代码文件中以方便调用
- 为输入的数据矩阵X扩展一个全为1的列，这样方便与向量w进行乘积运算
- 在计算∇w时，可以一次性计算所有w分量的梯度：$\nabla w = X^T \cdot (Xw - y)$
- 尝试调整tolerance和learning_rate的值，查看循环所需的次数。一般应有下列情况：
  - 给定learning_rate=0.01，调整tolerance(1e-5, 1e-6, 1e-7,1e-8等)，可看到，拟合精度越高，需要的循环次数越多

    - 给定tolerance=1e-7，调整learning_rate(0.1, 0.01, 0.005, 0.001等)，可看到，如果rate太大，有可能直接溢出造成不收敛；rate太小，就需要更多的循环次数，甚至在规定的循环次数内无法收敛
- 代码参考：【05/08batch_gradient_descent_multi_variants.py】和【05/bgd_resolver.py】

随机梯度下降法

计算梯度时，从训练数据中随机选择一组数据，仅针对该数据求导。假设选择的数据是

$(x^{(k)}, y^{(k)})$，则：

$\frac{\partial F(w)}{\partial w_0} = 2 \cdot (w_0 + w_1 x^{(k)} - y^{(k)})$

$\frac{\partial F(w)}{\partial w_1} = 2 \cdot (w_0 + w_1 x^{(k)} - y^{(k)}) \cdot x^{(k)}$

特点：

- 计算梯度时无需对所有样本数据计算，因此降低了计算量
- 但是因为是随机选择的数据点梯度，每次优化的结果可能略有区别

示例**3**：使用随机梯度下降法拟合多维数据

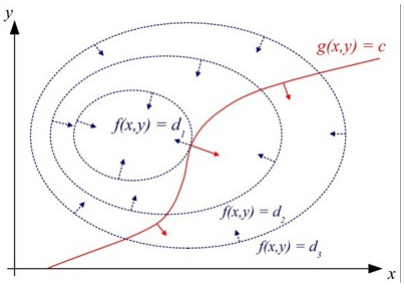
- 要注意，learning_rate不能太大，否则无法收敛
- 代码参考：【05/09stochastic_gradient_descent_multi_variants.py】

第3节：拉格朗日乘子法

原理

拉格朗日乘子法（Lagrange multiplier）用于解决在某种约束条件下对某个函数求极值的问题。

- 假设某目标函数：$f(x,y)$， 在无约束条件下，只要求出该函数导数为0时的(x_0,y_0)值，然后再代入$f(x_0,y_0)$求取极值即可。
- 如果在求极值的同时，还需要满足约束条件$g(x,y)=c$，或$g(x,y)-c=0$；就需要使用拉格朗日乘子法。
- 下图画出了目标函数$f(x,y)$的等值线图(蓝色虚线)，同时画出了约束函数$g(x,y)$的曲线图（红色实线）。$g(x,y)=c$与某条等值线的切点（图中的d_1点），就是$f(x,y)$的极值点。只需要把该极值点的(x,y)代入回$f(x,y)$中，就能求得目标函数$f(x,y)$在约束条件$g(x,y)=c$下的极值。
- 同时请注意，函数$f(x,y)$和$g(x,y)=c$在切点d_1处的法向量共线(但不一定同向)



计算方案

假设目标函数$f(x,y)$以及约束条件：$g(x,y)=c$。 要求目标函数的极值，等价于求下列函数的极值： $F(x,y,\lambda)=f(x,y)+\lambda(g(x,y)-c)$ 其中，λ是一个不为0的标量

- 将F函数分别对x,y,λ求偏导数，并使其偏导数为0

$$\begin{cases}\frac{\partial F}{\partial x}=0\\ \frac{\partial F}{\partial y}=0\\ \frac{\partial F}{\partial \lambda}=0\end{cases}$$

- 多个约束条件
  - 在多个约束条件$g_1(x,y)=c_1$和$g_2(x,y)=c_2$ 情况下，

    $F(x,y,\lambda_1,\lambda_2)=f(x,y)+\lambda_1(g_1(x,y)-c_1)+\lambda_2(g_2(x,y)-c_2)$
  - 再对x,y,λ_1,λ_2 分别求偏导数

示例**1**：已知目标函数：$f(x_1,x_2,x_3)=2x_1^2+3x_2^2+7x_3^2$，约束函数：$g_1=2x_1+x_2-1$，$g_2=2x_2+3x_3-2$，计算其极值

- 应用乘子法之后的函数：

  $$F(x_1,x_2,x_3,\lambda_1,\lambda_2)=2x_1^2+3x_2^2+7x_3^2+\lambda_1(2x_1+x_2-1)+\lambda_2(2x_2+3x_3-2)$$

- 列出偏导方程式：

  $$\begin{cases}\frac{\partial F}{\partial x_1}=4x_1+2\lambda_1=0\\ \frac{\partial F}{\partial x_2}=6x_2+\lambda_1+2\lambda_2=0\\ \frac{\partial F}{\partial x_3}=14x_3+3\lambda_2=0\\ \frac{\partial F}{\partial \lambda_1}=2x_1+x_2-1=0\\ \frac{\partial F}{\partial \lambda_2}=2x_2+3x_3-2=0\end{cases}$$

- 可解得：$\lambda_1=-0.45,\lambda_2=-1.41,x_1=0.227,x_2=0.546,x_3=0.302$，对应的极值为$f(0.227,0.546,0.302)$

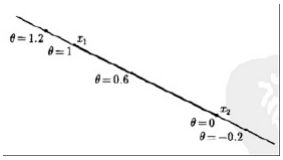
第4节：凸优化

凸集合

- 平面上，如果集合C内任意两点间的线段均在集合C内，则称集合C为凸集。

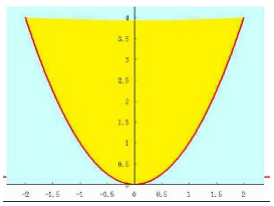
  数学表达为： 对于 $\forall x_1, x_2 \in C$ 及 $\theta \in [0,1]$，C是凸集得充要条件是：$\theta x_1 + (1-\theta)x_2 \in C$

  - 简单的说，对于平面上的集合来说，如果两点x_1和x_2之间连接的直线上所有点都在集合C内，则集合C是凸集
  - 直观查看直线上的点：

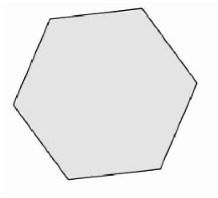
    

    - 当$\theta = 0$时，该点就是x_2
    - 当$\theta = 1$时，该点就是x_1
    - 当$0 < \theta < 1$时，$\theta x_1 + (1-\theta)x_2$可以视为在$x_1$和$x_2$之间比例$\theta$处的点。例如，当$\theta = 0.5$时，正好是$x_1$和$x_2$的中间点
    - 当$\theta > 1$或$\theta < 0$时，点在x_1和x_2外侧
  - 推广到N维情况，对于$\forall x_1, x_2, \cdots, x_k \in C, \theta_i \in [0,1]$；且$\sum_{i=1}^{k} \theta_i = 1$，则$C$是凸集的必要条件是：$\sum_{i=1}^{k} \theta_i x_i \in C$
- 举例

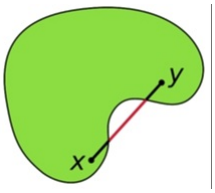
  - 下图$y = x^2$函数曲线包围的黄色区域部分是凸集

  - 下图多边形包含的灰色区域部分是凸集

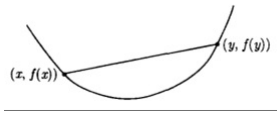


- 下边绿色部分不是凸集



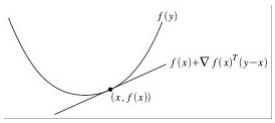
凸函数

- 如果函数f的定义域是凸集，且满足：当$0 \leq \theta \leq 1$时，有
  $f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)$，则称函数$f$为凸函数



  - 从上图(凸函数曲线图)看，在凸函数上任意选取两点连成直线，则直线上的所有点都在凸函数曲线的上方

- 使用一阶导数判别函数是否凸函数：如果函数f一阶可导，且对于其定义域内的x_1, x_2，存在：$f(x_2) \geq f(x_1) + \nabla f(x_1)^T(x_2 - x_1)$，则函数$f$为凸函数



  - 上式中，$\nabla f(x_1)^T$ 表示函数f在x_1处的梯度

- 典型凸函数
  - 仿射函数(线性变换+偏移)：$Ax + b$，A为mxn矩阵，x为n维列向量，b为n维列向量
  - 指数函数：$e^a x$
  - 幂函数：x^a，其中，$x \in R_+, a \geq 1$或$a \leq 0$
  - 负对数函数：$-logx$
  - 负熵函数：$xlogx$
  - 范数函数：$||x||_p$
  - 非负加权求和：$g(x) = w_1 f_1(x) + \cdots + w_n f_n(x)$；其中，$w > 0$，且$f_i(x)$都是是凸函数；则$g(x)$也是凸函数

- 凸函数重要特性：对凸函数求极值，其局部最优解，就是全局最优解

凸函数的最优解

- 凸优化问题：设x为n维向量，目标函数$f(x)$，以及m个不等式约束条件$g(x)$、p个等式约束条件$h(x)$：

$$\begin{cases} target: & min(f(x)) \\ s.t.: & g_i(x) \le 0, i = 1, 2, \cdots, m \\ & h_i(x) = 0, i = 1, 2, \cdots, p \end{cases}$$

  - 要求$f(x), g_i(x)$是凸函数，$h_i(x)$是仿射函数
  - 为方便起见，将不等式全都转换成"小于等于"
  - 在上述约束条件下，求目标函数的极小值的过程，就是凸优化。
- 无约束条件下的最优化
  - 当m和p都为0时，相当于没有约束条件。则原凸优化问题转为求解：$\nabla f(x) = 0$ 求得的x^*和对应的$f(x^*)$就是对应的极值点和极值
- 仅有等式约束条件下的最优化
  - 当$m = 0$而$p \neq 0$时，则原凸优化问题可以使用上一节拉格朗日乘子法来求解，求得的x^*和对应的$f(x^*)$就是对应的极值点和极值
- 包含等式、不等式约束条件下的最优化
  - 如果$m \neq 0$，则需要引入拉格朗日对偶问题来求解

最初问题、原问题与拉格朗日对偶问题

- 最初问题：

$$\begin{cases} target: & min(f(x)) \\ s.t.: & g_i(x) \le 0, i = 1, 2, \cdots, m \\ & h_i(x) = 0, i = 1, 2, \cdots, p \end{cases}$$

- 原问题-重新表述最初问题：
  - 引入拉格朗日乘子法，构造拉格朗日函数

    $F(x, \alpha, \beta) = f(x) + \sum_{i=1}^{m} \alpha_i g_i(x) + \sum_{i=1}^{p} \beta_i h_i(x)$ 其中：$\alpha \ge 0$
  - 先将F看成是α, β的函数，并且求取F的最大值F^*，即：

    $F^* = \max\limits_{\alpha, \beta} F(x, \alpha, \beta)$

设此时α, β的最优解分别为：α^*, β^*。也就是说：

$F^* = \underset{\alpha,\beta}{max} F(x, \alpha, \beta) = F(x, \alpha^*, \beta^*)$

如果x满足$g(x)$和$h(x)$的所有约束条件，即所有的：$g(x) \leq 0$，所有的：$h(x) = 0$，则显然有：$F^* = f(x)$，并且此时应该有：$\alpha^* = 0$

- 与最初问题等价的原问题： 定义：$\theta_p(x) = F^*(x) = \underset{\alpha,\beta}{max} F(x, \alpha, \beta) = f(x)$

  最初问题中要对$f(x)$求极小值，就相当于是对$\theta_p(x)$求极小值

  也就是说，在上述约束条件下要求$min(f(x))$，等价于求:

  $min_x \theta_p(x) = \underset{x}{min}\ \underset{\alpha,\beta}{max} F(x, \alpha, \beta)$

  或者说，首先求使得拉格朗日函数F取最大值的α^*, β^*，然后代入到$F(x, \alpha^*, \beta^*)$，求使得F取最小值p^*时的x^*：

  $p^* = \underset{x}{min} \theta_p(x) = \theta_p(x^*)$

- 对偶问题

  - 先把F看成是x的函数，求使得F取最小值时的x^*，然后再定义下列关于α,β的函数：

    $\theta_d(\alpha, \beta) = \underset{x}{min} F(x, \alpha, \beta) = F(x^*, \alpha, \beta)$

  - 对偶问题表述形式： 计算出最优的α, β，使得θ_d取最大值，也就是：

    $\underset{\alpha,\beta}{max} \theta_d(\alpha, \beta) = \underset{\alpha,\beta}{max}\ \underset{x}{min} F(x, \alpha, \beta)$

    也就是说，对偶问题表述为：首先求使得拉格朗日函数F取最小值的x^*，然后代入到$F(x^*, \alpha, \beta)$，求使得F取最大值d^*时的α^*, β^*：

    $d^* = \underset{\alpha,\beta}{max} \theta_d(\alpha, \beta) = \theta_d(\alpha^*, \beta^*)$

- 原问题与对偶问题的关系

  - 容易观察到，对于任意的x, α, β，存在下列事实：

    $\theta_d(\alpha, \beta) = \underset{x}{min} F(x, \alpha, \beta) \leq F(x, \alpha, \beta) \leq \underset{\alpha,\beta}{max} F(x, \alpha, \beta) = \theta_p(x)$

    因此： $\underset{\alpha,\beta}{max} \theta_d(\alpha, \beta) \leq \underset{x}{min} \theta_p(x)$

即： $d^* = \underset{\alpha,\beta}{max}\ \underset{x}{min}\ F(x,\alpha,\beta) \leq \underset{x}{min}\ \underset{\alpha,\beta}{max}\ F(x,\alpha,\beta) = p^*$

也就是说：原问题的最小值大于或等于对偶问题的最大值。这意味着：如果要求原问题的最小值，可以先求出对偶问题的最大值。 而如果$d^* = p^*$，那么就可以用对偶问题的最大值来作为原问题的最小值。

- 原始问题最优解与对偶问题最优解之间的关系 定义Duality Gap(对偶间隙)如下：

  $\nabla = d^* - p^*$

  如果$\nabla = 0$，称为强对偶。在这种情况下，如果x^*, α^*, β^*是对偶问题的最优解就是原问题的最优解。

- 如何达到强对偶

  - 充要条件(Slater条件)：存在x^*，使得：$g_i(x^*) < 0, i = 1, 2, \cdots, m$，也就是说所有不等式全都取<号，不取=号。另一方面，对于没有不等式约束条件的原问题，其拉格朗日对偶函数直接就具备强对偶特性
  - 必要条件(KKT条件)：存在x^*, α^*, β^*，使得下列条件都满足：

$$
\begin{cases}
g_i(x^*) \leq 0, & i = 1, 2, \cdots, m \\
\alpha_i^* \geq 0, & i = 1, 2, \cdots, m \\
h_i(x^*) = 0, & i = 1, 2, \cdots, p \\
(\frac{\partial F(x, \alpha^*, \beta^*)}{\partial x}|_{x=x^*}) = 0 \\
\sum_{i=1}^{m} \alpha_i^* g_i(x^*) = 0
\end{cases}
$$

Python针对凸优化的求解函数

- scipy.optimize.minimize方法可以针对带等式约束条件和不等式约束条件的目标函数求解最优值。但是它往往要求目标函数中只有一组待优化的变量
- 如果目标函数中有多组待优化的变量，直接求解会比较困难，这时就可以引入拉格朗日对偶问题，并利用KKT条件($\frac{\partial F(x, \alpha^*, \beta^*)}{\partial x}|_{x=x^*}) = 0$化简目标函数，使之能消除掉过多的优化变量组，这样就易于求解了。
- KKT条件中的其它4个等式、不等式约束条件是比较方便在程序中写出来的，可以被minimize方法直接使用
- 在对偶问题中，要求的是最大值（从而作为原问题的最小值），如果使用minimize方法，需要取其相反数，然后求最小值

第5节：scipy.optimize

示例**1**：计算$f(x)=x^2+x$和$f(x)=x^3$的极小值

- minimize_scalar方法用于求单变量函数的极小值

```
import numpy as np
from scipy.optimize import minimize_scalar

def f1(x):
  return x*x+x

def f2(x):
  return x*x*x

res = minimize_scalar(f1)
print("函数【f=x*x+x】极值点：", res.x, "最小值：", res.fun)
res = minimize_scalar(f2)
print("函数【f=x*x*x】极值点：", res.x, "最小值：", res.fun) # 事实上该函数并没有收敛
```

示例**2**：计算函数$f(x)=100-x^2$ 的最大值

- minimize方法只能求极小值，如果要求极大值，可以将函数乘以-1
- minimize方法提供了args参数，可以向目标函数传递除了第一个参数以外的其它参数。本例中将-1传给了sign参数

```
import numpy as np
from scipy.optimize import minimize_scalar

def f1(x, sign):
   return sign*(100 - x*x)

res = minimize_scalar(f1, args=(-1))
print("函数极值点：", res.x, "最大值：", res.fun)
```

示例**3**：计算函数$f(x)=x^3$在区间**[3,5]**的极小值

```
import numpy as np
from scipy.optimize import minimize_scalar

def f1(x):
    return x*x*x

res = minimize_scalar(f1, bounds=(3.0, 5.0), method='bounded')
print("函数极值点：", res.x, "最小值：", res.fun)
```

示例**4**：计算函数$f(x_1, x_2) = 3x_1^2 + 5x_1x_2 + 2x_2^2 - 8x_1 - 10x_2 + 6$在$x_1 \in [0, +\infty)$和$x_2 \in [0, +\infty)$区间上的的极小值

- 如果是多变量函数，则必须使用minize方法
- 在定义目标函数时，传入的第一个参数x是一个向量，分别代表了每个变量
- 必须要给定初始值
- 在指定bounds时，必须以tuple的形式依次描述每个变量的值域。对于$+\infty$和$-\infty$，可以使用None来代替，取决于它与区间另一端的位置关系
- 要设定bounds，求解器必须是'L-BFGS-B'、'TNC'或者'SLSQP'

```
import numpy as np
from scipy.optimize import minimize

# f(x1, x2) = 3*x1*x1 + 5*x1*x2 + 2*x2*x2 - 8*x1 - 10*x2 + 6
def f1(x):
   return 3*x[0]*x[0] + 5*x[0]*x[1] + 2*x[1]*x[1] - 8*x[0] - 10*x[1] + 6

init_x = [1.5, 0.5]
bnds = ((0, None), (0, None))
res = minimize(f1, init_x, bounds=bnds, options={'disp':True})
print("函数极值点：", res.x, "最小值：", res.fun)
```

示例**5**：计算函数$f(x_1, x_2, x_3) = 2x_1^2 + 3x_2^2 + 7x_3^2$在约束条件：$2x_1 + x_2 = 1$和$2x_2 + x_3 = 1$的极值

- constraints参数可以指定约束条件
- 对于"="约束条件，设定type为'eq'，并给出令$h(x) = 0$的约束表达式
- 对于"≤"约束条件，设定type为'ineq'，并给出令$h(x) \leq 0$的约束表达式。不存在">"条件约束，在这种情况下，要设法转为"≤"约束
- 如果使用约束条件，那么求解器必须是'COBYLA'或'SLSQP'

```
import numpy as np
from scipy.optimize import minimize
def target(x):
   return 2*x[0]*x[0] + 3*x[1]*x[1] + 7*x[2]*x[2]

cons = (
   {'type':'eq', 'fun':lambda x: np.array([2*x[0]+x[1]-1])},
   {'type':'eq', 'fun':lambda x: np.array([2*x[1]+3*x[2]-2])})
init_x = [0.1,0.2,0.5]    # 假定的初始最优值
res = minimize(target, init_x, method='SLSQP', constraints=cons)
print(res)
```

示例**6**：使用**minimize**求解拟合公式的最优系数向量

- 本例使用minimize来替代leastsq方法求最优解
- 将系数向量w视为待求解的变量，目标函数应该是关于w而不是x的函数
- 代码参考：【05/10scipy_minimize_fit_multi_variants.py】

第六章：其它

第1节：信息论

信息论主要研究的是对一个信号能够提供信息的多少进行量化。1948年，香农引入信息熵，将其定义为离散随机事件的出现概率。一个系统越是有序，信息熵就越低；反之，一个系统越是混乱，信息熵就越高。所以说，信息熵可以被认为是系统有序化程度的一个度量

信息熵

如果一个随机变量X的可能取值为：$X = x_1, x_2, \cdots x_n$，其概率分布分别为：$P(x_i)$。则随机变量X的熵定义为：

$H(X) = \sum_{i=1}^{n} P(x_i) log(P(x_i))$

- 信息熵的值越大，说明越不确定
- 在信息论中，常常用以2为底的对数来计算，计算出来的单位称为bits；在机器学习中，常常用自然对数来计算，计算出来的单位称为nats。在本课程中，都采用自然对数来进行计算

条件熵

条件熵表示在条件X下Y的信息熵，记作：

$H(Y|X) = \sum_{i=1}^{n} P(x_i) H(Y|x_i) = \sum_{i=1}^{n} \{P(x_i) * [- \sum_{j=1}^{m} P(Y_j|x_i) log(P(Y_j|x_i))]\}$

- x_i：条件X的每种可能取值，假设共有n种取值
- $P(x_i)$：具有x_i取值的样本在总样本种所占的比例
- $H(Y|x_i)$：选取所有包含x_i取值的样本，基于随机变量Y来计算信息熵
- $Y_j|x_i$：所有包含x_i取值的样本中，特征Y的第j种取值(假设共有m种取值)
- $P(Y_j|x_i)$：所有包含x_i取值的样本中，Y_j取值所占的比例

信息增益

- 信息增益描述了一个特征带来的信息量的多少，常用于决策树的构建和特征选择。一个特征往往会使一个随机变量Y的信息量减少，减少的部分就是信息增益
- 信息增益 = 信息熵 - 条件熵
- 信息增益越大，就越说明该特征对确定某个事件的贡献越大(降低了某个事件的不确定性)，或者说，该特征是某个事件的主要特征

示例**1**：假设有下列训练样本：

编号	天气	温度	湿度	风力	是否外出打球
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
3	Overcast	Hot	High	Weak	Yes
4	Rain	Mild	High	Weak	Yes
5	Rain	Cool	Normal	Weak	Yes
6	Rain	Cool	Normal	Strong	No
7	Overcast	Cool	Normal	Strong	Yes
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

设置"是否外出打球"这一随机变量为Y

- 信息熵计算

  先统计 $P(y=no)=5/14;P(y=yes)=9/14;n=2$

  $H(Y)=-5/14*log(5/14)-9/14*log(9/14)=0.6518$

- 条件熵计算

  设湿度为随机变量X，统计：$P(x=high)=7/14=1/2;P(x=normal)=7/14=1/2$

  $$H(Y|X_{湿度})=P(x=high)*H(Y|x=high)+P(x=normal)*H(Y|x=normal)$$

  而：

  $$\begin{aligned}H(Y|x=high)&=-P(Y=yes|x=high)*log(P(Y=yes|x=high)-\\&\quad P(Y=no|x=high*log(P(Y=no|x=high)\\&=-3/7*log(3/7)-4/7*log(4/7)=0.6829\\H(Y|x=normal)&=-6/7*log(6/7)-1/7*log(1/7)=0.4101\end{aligned}$$

  最终：

  $$H(Y|X_{湿度})=1/2*0.6829+1/2*0.4101=0.5465$$

- 信息增益计算
  - 信息增益 = 信息熵 - 条件熵=0.6518-0.5465=0.1053
  - 也就是说，引入了湿度这个变量之后，就使得是否打球这个变量的信息熵就从0.6518减小到了0.5465，变得更加确定了
- 确定最关键的特征变量
  - 信息增益越大，该特征变量就越关键
  - 考虑到系统的总信息熵是固定不变的，因此，某特征变量的条件熵越小，该变量就越关键

第2节：距离计算

向量之间的几种距离计算

对于两个n维向量$A=(x_{11},x_{12},\cdots,x_{1n})$和$B=(x_{21},x_{22},\cdots,x_{2n})$，它们之间的举例有多种定义方式：

- 曼哈顿距离(manhattan)

  $d=\sum_{k=1}^{n}|x_{1k}-x_{2k}|$

- 欧式距离(euclidean)，也是L2范数

  $d=\sqrt{\sum_{k=1}^{n}(x_{1k}-x_{2k})^2}$

- 闵可夫斯基距离(minkowski)，它其实包含了上述两种距离的定义

  $d=[\sum_{k=1}^{n}|x_{1k}-x_{2k}|^p]^{\frac{1}{p}}$

- 车比雪夫距离(chebyshev)，也就是无穷范数

  $d=max(|x_{1k}-x_{2k}|)$

- 夹角余弦，用于衡量两个向量方向上到差异。夹角余弦越大，表明向量夹角越小。当两个向量方向一致时，夹角余弦取值为1；完全相反时，取值为-1

  $cos\theta=\frac{A*B}{|A|*|B|}=\frac{\sum_{k=1}^{n}x_{1k}*x_{2k}}{(\sum_{k=1}^{n}x_{1k}^2)^{\frac{1}{2}}*(\sum_{k=1}^{n}x_{2k}^2)^{\frac{1}{2}}}$

- 汉明距离(hamming)，比较两个向量中对应位置值不相同的个数

  例如，[5, 7, 9, 0]与[5,10,8,0]之间的汉明距离为2

空间的距离

直线的方向向量和法向量：

- 与该条直线平行的非零向量称为方向向量。记为v
- 与该条直线垂直的非零向量称为法向量。记为n
- v与n的内积为0
- 直线$Ax+By+C=0$的法向量为：(A,B)，方向向量为：$(-B,A)$

直线及平面间的距离：

- 二维平面上，点$P(x_0, y_0)$到直线$Ax + By + C = 0$之间的距离为：

  $$d = \frac{|Ax_0 + By_0 + C|}{(A^2 + B^2)^{\frac{1}{2}}}$$

- 二维平面上，两条平行线$Ax + By + C_1$=0和$Ax + By + C_2 = 0$之间的距离为：

  $$d = \frac{|C_1 - C_2|}{(A^2 + B^2)^{\frac{1}{2}}}$$

- 扩展到N维空间，两个平行超平面$W^T X + C_1 = 0$和$W^T X + C_2 = 0$的距离为：

  $$d = \frac{|C_1 - C_2|}{(W^T W)^{\frac{1}{2}}}$$

  其中，W和X为具有N个元素的列向量 使用符号：$||W|| = \sqrt{W^T W}$ 则上式也可表示成：

  $$d = \frac{|C_1 - C_2|}{||W||}$$

第二部分：机器学习算法与实现

本部分介绍机器学习算法的原理、Python代码实现以及如何从sklearn包中直接调用各类算法，主要包括：

- 一元和多元线性回归
- 逻辑回归
- 支持向量机SVM
- KNN分类及K-Means聚类
- 朴素贝叶斯分类
- 决策树和随机森林
- 浅层神经网络

第七章：线性回归

第1节：单变量线性回归

提出问题

假设某披萨店的披萨价格和披萨直径之间有下列数据关系：

训练样本	直径(英寸)	价格(美元)
1	6	7
2	8	9
3	10	13
4	14	17.5
5	18	18

根据上面的训练数据，我们能否推断(预测)出某个直径的披萨可能的售价呢？例如，12英寸的披萨可能售卖多少钱？

分析问题

把直径看成自变量x(以后也称特征值)，价格看成因变量y，可以先通过作图看出二者的关系：

可以看到：

- 价格y随着直径x的变化，大致呈现线性变化；
- 如果根据现有的训练数据能够拟合出一条直线，使之与这些训练数据的各点都比较接近，那么根据该直线，就可以计算出在任意直径披萨的价格
- 本节代码参考【07/01plot_pizza_data.py】

解决方法

采用Python scikit-learn库中提供的sklearn.linear_model.LinearRegression对象来进行线性拟合

- 思路
  - 拟合出来的直线可以表示为：$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 = \theta_0 + \theta_1 x_1$
    - x_0 表示Inercept Term，一般设置为1即可
    - x_1 表示影响计算结果的的第一个因素(或称特征，在本例中就是直径)。在单变量线性

回归中，只有x_1

- θ_0表示截距，θ_1表示斜率。这两个参数都是需要通过拟合求出来的
- $h_\theta(x)$称为判别函数(Hypothesis Function)或推理式，也就是线性拟合的模型结果函数

- 步骤
  - 准备训练数据

```
xTrain = np.array([6,8,10,14,18])[:, np.newaxis]
yTrain = np.array([7,9,13,17.5,18])
```

LinearRegression支持单变量和多变量回归。对于多变量回归，xTrain显然是矩阵形式。因此，即使只有一个变量，LinearRegression也要求输入的特征值以矩阵形式(列向量)存在。

  - 创建模型对象

```
model = LinearRegression()
```

  - 执行拟合

```
hypothesis = model.fit(xTrain, yTrain)
```

判别函数(hypothesis)对象中包含了大量的属性和方法，可用于针对该模型的后续操作

  - 获取判别函数的参数（截距和斜率）

```
print("theta0=", hypothesis.intercept_)
print("theta1=", hypothesis.coef_)
```

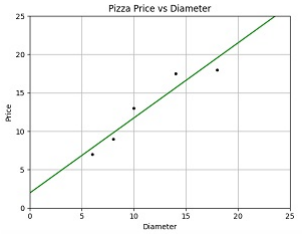
  - 预测新的数据

```
model.predict([[12]])
model.predict([[0],[10],[14],[25]])
```

    - 将待预测的数据放置在一个矩阵(或列向量)中
    - 可以批量预测多个数据

- 结果

根据判别函数，绘制拟合直线，并同时现实训练数据点：



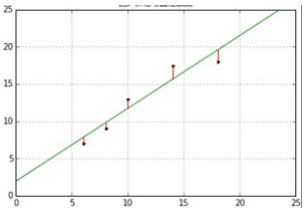
- 拟合的直线较好的穿过训练数据
- 根据新拟合的直线，可以方便的求出各个直径下对应的价格(预测结果)
- 本节代码参考【07/02sklearn_linear_fit_pizza_single_variant.py】

模型评价

拟合出来的判别函数效果如何：对训练数据的贴合度如何？对新数据的预测准确度如何？

先给出下列定义：

- 残差(residuals)：判别函数计算结果与实际结果之间的差异，如下图中的红色线段部分。一般是计算残差平方和



- R方(r-squared)：又称确定系数(coefficient of determination)。在通过训练数据得出了判别函数后，对于新的数据，如何评估该假设函数的表现呢？可以使用与训练数据不同的另一组数据（称为检验/测试数据）来进行评估。R方就是用来进行评估的一种计算方法。在Pyhton的scikit-learn中，是这样定义R方的（针对给定的测试数据）：

  $SS_{tot} = \sum_{i=1}^{m}(y^{(i)} - \overline{y})^2$

  $SS_{res} = \sum_{i=1}^{m}[y^{(i)} - h_\theta(x^{(i)})]^2$

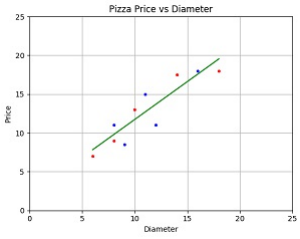
  $R^2 = \frac{1-SS_{res}}{SS_{tot}}$

  - m：测试数据集中的数据组数
  - $y^{(i)}$：测试数据集中第i组数据的y值（实际价格）
  - $\overline{y}$：测试数据集中y的平均值
  - $h_\theta(x^{(i)})$：将$x^{(i)}$代入到判别函数计算的结果，也就是根据模型算出的y值（计算价格）
  - SS_{tot}：针对测试数据计算出来偏差平方和
  - SS_{res}：针对测试数据计算出来的残差平方和

    - 一般来说，R方越大(不会超过1)，说明模型效果越好。如果R方较小或为负，说明效果很差
- 在Python中如何对单变量线性回归模型的效果进行评估
    - 手动计算

      假设hpyTrain代表针对训练数据的预测y值，hpyTest代表针对测试数据的预测y值

        - 训练数据残差平方和：`ssResTrain = sum((hpyTrain - yTrain) ** 2)`
        - 测试数据残差平方和：`ssResTest = sum((hpyTest - yTest) ** 2)`
        - 测试数据偏差平方和：`ssTotTest = sum((yTest - np.mean(yTest)) ** 2)`
        - R方：`Rsquare = 1 - ssResTest / ssTotTest`
    - LinearRegression对象提供的方法
        - 训练数据残差平方和：`model._residues`
        - R方：`model.score(xTest, yTest)`
    - 查看拟合效果

        - 红色为训练数据点，蓝色为测试数据点，绿色为判别函数(拟合直线)
        - 计算出的R方为0.662，效果一般
        - 计算出训练数据的相关性为0.954，测试数据的相关性为0.816。可以发现，根据数据集的不同，直径与价格之间的相关性波动较大。这也能解释为何针对测试数据的R方事实上不够理想
    - 本节代码参考【07/03sklearn_linear_fit_pizza_single_variant_eval.py】

第2节：线性回归手工计算

成本函数

- 在使用训练数据来训练模型时，用于定义判别函数与实际值的误差。成本函数计算结果越小，说明该模型与训练数据的匹配程度越高
- 设定了某个模型后，只要给定了成本函数，就可以使用数值方法求出成本函数的最优解（极小值），从而确定判别函数模型中各个系数
- 一元线性回归(直线拟合)的成本函数：

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2$$

  - m：训练数据集中的数据组数。
  - h_θ：训练出来的判别函数
  - $x^{(i)}$：训练数据集中的第i组数据的x值（直径）
  - $y^{(i)}$：训练数据集中第i组数据的y值（实际价格）
  - $h_\theta(x^{(i)})$：将$x^{(i)}$代入到判别函数计算的结果，也就是根据判别函数算出的y值（预测价格）

示例**1**：使用协方差和方差公式生成一元线性回归模型

- 系数计算公式：

$$\theta_1 = \frac{\sum_i^m(\bar{x} - x^{(i)})(\bar{y} - y^{(i)})}{\sum_i^m(\bar{x} - x^{(i)})^2} = \frac{cov(x, y)}{var(x)}$$

$$\theta_0 = \bar{y} - \theta_1\bar{x}$$

  - $\bar{y}$：训练数据中y的平均值
  - $\bar{x}$：训练数据中x的平均值
  - $cov(x, y)$：向量x和y的协方差
  - $var(x)$：向量x的方差
- 使用该方法计算出来的判别函数参数，与LinearRegression对象的计算结果一致。
- 本节代码参考【07/04cov_var_fit_pizza_single_variant.py】

示例**2**：使用批量梯度下降法生成一元线性回归模型

- 梯度计算：

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{\partial[\frac{1}{2m}\sum_{i=1}^{m}(\theta_0+\theta_1 x^{(i)}-y^{(i)})^2]}{\partial \theta_0}$$

$$= \frac{\frac{1}{2m} * 2 * \sum_{i=1}^{m}[(\theta_0+\theta_1 x^{(i)}-y^{(i)}) * \partial(\theta_0+\theta_1 x^{(i)}-y^{(i)})]}{\partial \theta_0}$$

$$= \frac{1}{m}\sum_{i=1}^{m}[(\theta_0+\theta_1 x^{(i)}-y^{(i)}) * (1+0+0)]$$

$$= \frac{1}{m}\sum_{i=1}^{m}(\theta_0+\theta_1 x^{(i)}-y^{(i)})$$

$$\frac{\partial J(\theta)}{\partial \theta_1} = \frac{\partial[\frac{1}{2m}\sum_{i=1}^{m}(\theta_0+\theta_1 x^{(i)}-y^{(i)})^2]}{\partial \theta_1}$$

$$= \frac{\frac{1}{2m} * 2 * \sum_{i=1}^{m}[(\theta_0+\theta_1 x^{(i)}-y^{(i)}) * \partial(\theta_0+\theta_1 x^{(i)}-y^{(i)})]}{\partial \theta_1}$$

$$= \frac{1}{m}\sum_{i=1}^{m}[(\theta_0+\theta_1 x^{(i)}-y^{(i)}) * (0+x^{(i)}+0)]$$

$$= \frac{1}{m}\sum_{i=1}^{m}[(\theta_0+\theta_1 x^{(i)}-y^{(i)}) * x^{(i)}]$$

- 将上述导数带入到$\theta_0' = \theta_0 - \alpha\frac{\partial J(\theta)}{\partial \theta_0}$； $\theta_1' = \theta_1 - \alpha\frac{\partial J(\theta)}{\partial \theta_1}$ 公式中，按照梯度下降法的一般规则调整
- 本节代码参考【07/05bgd_fit_pizza_single_variant.py】

示例**3**：使用随机梯度下降法生成一元线性回归模型

- sklearn.linear_model.SGDRegressor对象提供了使用随机梯度下降算法进行线性回归的实现
- 本节代码参考【07/06sklearn_sdg_regressor_pizza_single_variant.py】

第3节：多变量线性回归

提出问题

在上一章中，披萨价格仅与直径有关，按照这一假设，其预测的结果并不令人满意(R方=0.662)。本章再引入一个新的影响因素：披萨辅料级别(此处已经把辅料级别调整成数值，以便能够进行数值计算)。训练数据如下：

训练样本	直径(英寸)	辅料级别	价格(美元)
1	6	2	7
2	8	1	9
3	10	0	13
4	14	2	17.5
5	18	0	18

另外提供测试数据如下：

测试样本	直径(英寸)	辅料级别	价格(美元)
1	8	2	11
2	9	0	8.5
3	11	2	15
4	16	2	18
5	12	0	11

如何使用线性回归训练数据，并且判断是否有助于提升预测效果呢？

分析问题

对于一个自变量x_1的情形，y与x的关系用一条直线就可以拟合(假设有一定线性相关性)。对于有两个自变量x_1, x_2的情形，y与x的关系就需要用一个平面来拟合。如果有更多的自变量，虽然无法在三维空间中展现，但仍然可以用数学的方式来描述它们之间的关系。

- 判别函数：$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$
    - x_0称为Inercept Term，一般设置为1即可
    - $x_1, x_2, \cdots, x_n$表示影响y的各个因素。假设共有n个影响因素
- 判别函数的矩阵表述形式：

  考虑到：

  $h_\theta(x^{(1)}) = \theta_0 x_0^{(1)} + \theta_1 x_1^{(1)} + \theta_2 x_2^{(1)} + \cdots + \theta_n x_n^{(1)}$

  $h_\theta(x^{(2)}) = \theta_0 x_0^{(2)} + \theta_1 x_1^{(2)} + \theta_2 x_2^{(2)} + \cdots + \theta_n x_n^{(2)}$

  ……

$$h_\theta(x^{(m)}) = \theta_0 x_0^{(m)} + \theta_1 x_1^{(m)} + \theta_2 x_2^{(m)} + \cdots + \theta_n x_n^{(m)}$$

其中：$x_j^{(i)}$ 表示第i组数据的第j个自变量。注意，各组数据的第0个自变量均为Intercept Term，直接设置为1

以矩阵运算的方式表示上面的各组公式，可得：

$$h_\theta = X * \theta$$

其中：$X = \begin{pmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{pmatrix}$，$\theta = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{pmatrix}$

- 成本函数的矩阵运算形式：

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m}(X * \theta - y)^T(X * \theta - y)$$

- 梯度变化（偏导数）：

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{\partial[\frac{1}{2m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)})^2]}{\partial \theta_0}$$

$$= \frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)}) * x_0^{(i)}$$

$$\frac{\partial J(\theta)}{\partial \theta_1} = \frac{\partial[\frac{1}{2m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)})^2]}{\partial \theta_1}$$

$$= \frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)}) * x_1^{(i)}$$

$\cdots$

$$\frac{\partial J(\theta)}{\partial \theta_n} = \frac{\partial[\frac{1}{2m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)})^2]}{\partial \theta_n}$$

$$= \frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x_1^{(i)} + \cdots + \theta_n x_n^{(i)}) * x_n^{(i)}$$

上式中，$x_0^{(i)} = 1$

将上述各组式子用矩阵形式表达如下：

$$\frac{\partial J(\theta)}{\partial \theta} = X^T * (X * \theta - y)$$

解决方法

示例**1**：采用**Python scikit-learn**库中提供的**sklearn.linear_model.LinearRegression**对象来进行多变量线性回归

- 与单变量线性回归类似，但要注意训练数据此时是$m \times n$(m是训练数据条数，n是自变量个数)，在本例中，是5x2的矩阵： `xTrain = np.array([[6,2],[8,1],[10,0],[14,2],[18,0]])`
- 针对测试数据的预测结果，其R方约为0.77，已经强于单变量线性回归的预测结果
- 本节代码参考【07/07sklearn_linear_fit_pizza_multi_variants.py】

示例**2**：采用批量梯度下降法

本节代码参考【07/08bgd_fit_pizza_multi_variants.py】和【07/bgd_resolver.py】

第4节：模型训练要点

Feature Scaling问题

在多变量情况下，各个变量的值域可能有很大区别。例如，$x1$的值域可能在(1,10)之间，而$x2$的值域可能(1,10000)之间。值域差异过大，很容易造成在计算过程中溢出或无法收敛。而通过对数据进行归一化(Normalization)处理，可以较好的解决这个问题

- 查看数据【07/house_price.csv】
  - Area：房屋面积；Rooms：房间数；Price：房屋出售价格。其中Area和Rooms是自变量；Price是因变量
  - Area的值一般在"数千"这个级别上，而Rooms则是个位数，值域差异明显
- 使用sklearn.linear_model.LinearRegression处理：
  - 无需对自变量进行归一化处理，也能得到良好的结果。针对训练数据的R方约为0.73
- 使用自定义的批量梯度下降法来处理：
  - 在未对自变量归一化处理的情况下，运算出现异常，无法收敛
  - 归一化处理后，能够得到与LinearRegression类似的结果
- 归一化方法：
  - 有多种种归一化方法，一般可采用下列办法： $x = \frac{x-\bar{x}}{std(x)}$
  - 其中，$\bar{x}$该列数据的平均值，$std(x)$是该列数据的标准差
- 本节代码参考【07/09house_price_normalization.py】

学习速率(learning rate)对计算的影响

- 一般情况下，学习速率越小，其收敛的可能性越大（不容易因为步长过大，而“错过”极值点），但是需要更多次循环才能到达极值点。
- 学习速率越大，能够朝着极值点更快前进，但有可能因错过极值点而造成无法收敛
- 在实际的梯度下降算法中，会先选择一个较大的learning rate，随着不断逼近极值点，逐渐减小learning rate。

高阶拟合

在拟合数据点时，一般来说，对于一个自变量的，拟合出来是一条直线；对于两个自变量的，拟合出来时一个直平面。这种拟合结果是严格意义上的“线性”回归。但是有时候，采用“曲线”或“曲面”的方式来拟合，能够对训练数据产生更逼近的效果。这就是“高阶拟合”。

例如，对于一个自变量的情形，可以采用3阶拟合：$h_\theta(x) = \theta_0 x^0 + \theta_1 x^1 + \theta_2 x^2 + \theta_3 x^3$

对于两个自变量的情形，可以采用更复杂的高阶拟合：

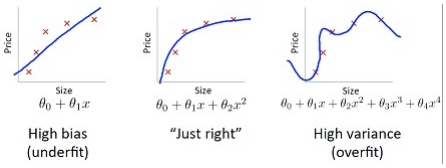
$h_\theta(x) = \theta_0 x_1^0 x_2^0 + \theta_1 x_1^1 + \theta_2 x_2^1 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1^1 x_2^1$

- python提供的高阶拟合方法
  - numpy.polifit方法
  - LinearRegression + PolynomialFeatures联合使用
- 如何以"线性回归"的方式来拟合高阶曲线
  - 以单变量高阶拟合为例。训练数据中，仅有一列自变量数据x_1
  - 通过Intercept Term，扩展一列全是1的自变量数据x_0
  - 将x_1的每个元素分别平方，得到第三列自变量数据x_2
  - 将x_1的每个元素分别立方，得到第四列自变量数据x_3
  - 将(x_0, x_1, x_2, x_3)看成具有四个自变量的多线性回归情形，通过LinearRegression对象进行多变量线性回归计算
  - 生成x_2, x_3等高阶自变量数据，可借助PolynomialFeatures对象的fit_transform方法来实现。
  - 高阶曲线对于训练数据的拟合程度较好，但对于测试数据，却不一定有较好的R方
- 本节代码参考【07/10polynomial_features_linear_fit.py】

---

bias和variance

观察下列拟合（单变量）：



- 左边图采用直线拟合，很显然有许多训练数据点不能很好的被拟合，将导致训练数据的残差很大。这种情况称为underfit（欠拟合），或者：High bias
- 右边图采用4次曲线拟合，完美的通过了训练数据的每个点。但是，如果针对另一批测试数据做预测，这条高阶曲线将产生非常大的误差。这种情况称为overfit（过拟合）,或者：High variance
- 中间图采用2次曲线拟合，虽然不能完美拟合每个数据点，但是对测试数据也能产生较好的预测效果。这时比较合适的情形
- 因此，在计算模型参数时，既要防止High bias，也要防止High variance

第八章：逻辑回归和分类

第1节：双类别逻辑回归

提出问题

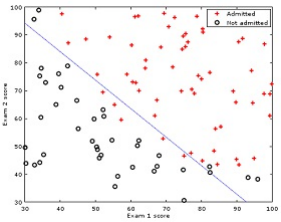
假设某班学生的两门考试成绩(exam1 score, exam2 score)与最终评价是否合格(passed)的数据如下(部分数据)：

训练样本	Exam 1 Score	Exam 2 Score	Passed
1	34.6	78.0	0
2	30.3	43.9	0
3	35.8	72.9	0
4	60.2	86.3	1
5	79.0	75.3	1

根据上面的训练数据，如果再提供一组新的分数(例如：65, 58)，则该学生是否通过呢？

分析问题

查看数据图像：



- 图中，水平方向为Exam 1 Score，垂直方向为Exam 2 Score。红色+点表示Passed=1，黑色圆圈点表示Passed=0。可以观察到，所有数据点较为明显的分成两个类别(通过或不通过)
- 前两章讲的线性回归，主要都是针对训练数据和计算结果均为数值的情形。但在很多情况下，结果不是数值，而是某种分类。例如，考试成绩通过或不通过；或者识别数字图片中的数字为0、1、2、3、4...8、9等。在这种情况下，可以将每个类别作为一个数值结果，然后通过模型计算自变量与该分类数值结果之间的关系。
- 由于分类结果较少，因此不太适合采用线性回归的拟合方法。而逻辑回归则提供了解决办法。
- 本节代码参考【08/01plot_exam_score_data.py】

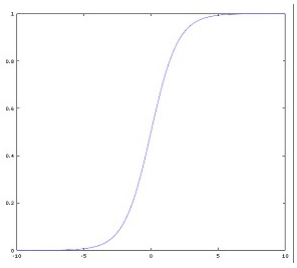
算法介绍

- Hypothesis Function

$$h_\theta(x) = g(x_0\theta_0 + x_1\theta_1 + \cdots + x_n\theta_n) = g(x * \theta)$$

  - x代表一组(行)数据，该组数据共有n个变量

- x_0为Intercept Item，一般设置为1
- g称为sigmoid函数，其定义为：$g(z)=\frac{1}{1+e^{-z}}$，该函数图像如下：



可以看到，当$z=0$时，$g(z)$的值为0.5。低于0.5的$g(z)$可以认为预测为false，高于0.5的预测为true。

- Cost Function:

$J(\theta)=\frac{1}{m}\sum_{i=1}^{m}[-y^{(i)}log(h_\theta(x^{(i)})-(1-y^{(i)})log(1-h_\theta(x^{(i)}))]$

使用矩阵运算表达如下：

$J(\theta)=\frac{-y^T*log(g(X*\theta))-(1-y^T)*log(1-g(X*\theta))}{m}$

- Gradient ：

$\theta_j=\theta_j-\alpha(\frac{\partial J(\theta)}{\partial\theta_j})$

$\frac{\partial J(\theta)}{\theta_j}=\frac{1}{m}\sum_{i=1}^{m}[(h_\theta(x^{(i)})-y^{(i)})x_j^{(i)}]=\frac{1}{m}(X^T*(g(X*\theta)-y))$

- 计算决策边界线(Decision Boundary)：

决策边界线上所有的点，其预测出来的y值（$h_\theta(x)$）正好为0.5，即：

$\frac{1}{1+e^{-z}}=0.5=>z=x*\theta=0$

当$n=2$时有：$\theta_0+\theta_1x^{(1)}+\theta_2x^{(2)}=0$，该边界线是一条直线

解决方法

示例**1**：使用**sklearn.linear_model.LogisticRegression**

- 设置逻辑回归算法的某些属性

```
model = LogisticRegression(solver='lbfgs')
```

使用lbfgs算法来执行回归计算。默认使用liblinear。注意，这两种算法的结果并不相同

- 执行计算

  ```
  model.fit(X, y)
  ```

- 执行预测

  ```
  model.predict(newX)
  ```

  返回值是newX矩阵中每行数据所对应的结果。如果是1，则表示passed；如果是0，则表示unpassed

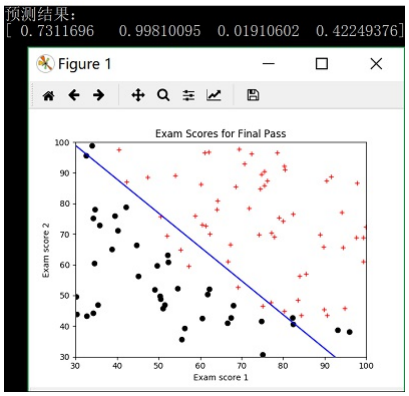
- 获得模型参数值

  ```
  theta0 = model.intercept_[0]
  theta1 = model.coef_[0,0]
  theta2 = model.coef_[0,1]
  ```

- 本节代码参考【08/02sklearn_logistic_exam_scores.py】

示例**2**：梯度下降法手动实现

- 仔细阅读Cost Function和Gradient Function的矩阵运算实现
- 先对数据进行归一化处理，有助于计算收敛
- 训练数据和测试数据需要手动添加Intercept Item列。先进行归一化，再手动添加Intercept Item
- 测试数据也需要进行相应的归一化处理，才能预测。
- 绘制边界线时，边界线上的点横坐标数据也需要先归一处理，然后求出归一化的纵坐标，最后再回算出纵坐标的正常值
- 运行结果：



预测结果在0.5以上，可认为passed=1；否则passed=0

- 本节代码参考【08/03bgd_logistic_exam_scores.py】和【02/bgd_resolver.py】

示例**3**：使用**scipy.optimize**优化运算库

- 调用minimize方法时，需要提供jac参数，并将其设置为梯度计算函数
- scipy.optimize库中提供的算法会比我们自己实现的算法更高效、灵活、全面

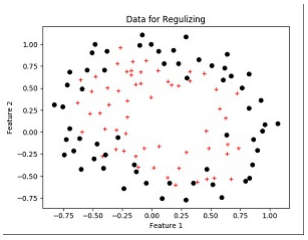
- 本例中没有对数据进行归一处理，因此导致minimize方法执行过程中溢出(尽管可能也能收敛)。请自行添加归一化处理功能
- 本节代码参考【08/04scipy_minimize_logistic_house_price.py】

第2节：Regulization

Overfit问题

如果Feature数量太多(n很大)，使得θ向量中的元素数量相应也很大，那么可能造成Overfitting，即：Hypothesis Function完美的贴合每一个Training Set，但是对于新的参数，却很难给出较好的Prediction。

- 数据文件【08/data_for_regulizing.csv】中包含两个Feature自变量和一个因变量。其中，因变量的值为1或0。因变量为1的点用红色+号表示，为0的点用黑色圆点表示。其数据点图像为下图所示：

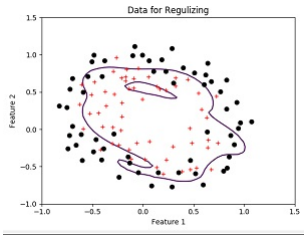


- 使用逻辑回归对上述数据进行分类。很显然，这不是简单的线性关系（或者说是线性不可分的）。因此采用$h_(\theta) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$这样简单的假设函数是无法满足要求的。考虑采用高阶函数，例如6阶函数(一共28项)：

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_1 x_2 + \theta_5 x_2^2 + \theta_6 x_1^3 + \theta_7 x_1^2 x_2 + \theta_8 x_1 x_2^2 + \theta_9 x_2^3 + \cdots$$
$$+\theta_{21} x_1^6 + \theta_{22} x_1^5 x_2 + \theta_{23} x_1^4 x_2^2 + \theta_{24} x_1^3 x_2^3 + \theta_{25} x_1^2 x_2^4 + \theta_{26} x_1 x_2^5 + \theta_{27} x_2^6$$

  将x_1和x_2两个自变量的值，按照上述多项式的各个项进行乘方、乘积运算，可以得到28组不同的结果，因而就形成了28个Feature

- 使用上一节介绍的scipy.optimize库执行逻辑回归计算，仍按照之前的Cost Function和Gradient代入到minimize函数进行计算，只是Feature的个数从2个增加到28个。可以看到下图所示的边界曲线



  这条边界曲线非常曲折，试图将每个点都区分开来，从而使训练数据达到最好的匹配。但是也看到，这个边界将造成overfit，导致预测新数据时准确性大幅下降。

- 本节代码参考【08/05sklearn_logistic_overfit.py】

规范化逻辑回归（**Regulized Logistic Regression)**

为了解决上面出现的overfit问题，我们希望限制θ中每个参数的大小。

- 修改Cost Function如下:

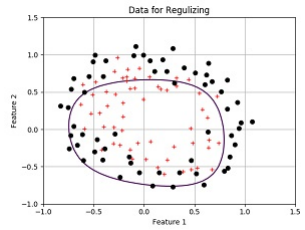
$$J(\theta)=\frac{1}{m}\sum_{i=1}^{m}[-y^{(i)}log(h_\theta(x^{(i)}))-(1-y^{(i)})log(1-h_\theta(x^{(i)}))]+\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

  - Cost Function函数中加入了θ^2项(这一项也可称为penalty项)，这就使得每一个θ不能太大，从而在一定程度上防止了Overfitting。
  - λ用于调整θ的权重，如果λ很大，则θ被迫非常小，从而防止Overfitting，但有可能造成Underfitting；如果λ太小。则Overfitting的可能性会大大增加；如果$\lambda=0$，则与之前未采用Regulization方法时完全相同
  - 请注意：θ_0(Intercept Item)不参与penalty项的计算
- 修改Batch Gradient Descent如下

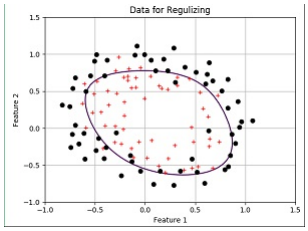
$$\theta_j=\theta_j-\alpha(\frac{\partial J(\theta)}{\partial\theta_j})$$

$$\frac{\partial J(\theta)}{\theta_j}=\frac{1}{m}\sum_{i=1}^{m}[(h_\theta(x^{(i)})-y^{(i)})x_j^{(i)}]+\frac{\lambda}{m}\theta_j$$

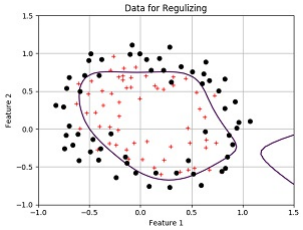
  - 上面的式子适用于θ_1及以上的参数迭代计算。
  - 对于θ_0，仍与未采用Regulization方法时的梯度计算公式相同
- 引入了Regulization后，我们可以观察到下列的拟合结果：
  - $\lambda=10$



  - $\lambda=0.1$



  - $\lambda=0.001$



    - 可见，λ越大，overfit可能性越小，但underfit的可能性增加；λ越小，overfit可能性增加，但underfit可能性减小

- 本节代码参考【08/06scipy_minimize_logistic_regulation.py】

---

sklearn.linear_model.LogisticRegression的Regulazation处理：

- LogisticRegression本身已经考虑了Regulazation问题，因此在调用本对象进行拟合时，其overfit和underfit均控制得较好
- 在构造LogisticRegression对象时，可以传入penalty参数，即指定Regulazation方式。默认值L2就是采用权重平方和。而参数C就是$\frac{1}{\lambda}$
- 本节代码参考【08/07sklearn_logistic_regulation.py】

第3节：多类别逻辑回归

提出问题

上一节介绍的逻辑回归能够较好的判断结果是1或0的情形(两种类别)。但是现实中，结果往往有多种类别。
例如，判断某个手写出来的阿拉伯数字是0~9中的哪一个，这就有10个类别。
问题：给定一张手写阿拉伯数字的图片，判断出该图片是0~9中的哪一个数字

分析问题

使用机器学习对图片进行分类预测，主要是要通过某种机器学习算法，对大量已知图片数据进行训练，得出相应的假设公式。具体来说，可分为下列步骤：

1. 准备训练数据
   为了使机器学习具有一定的准确性，需要提供足量的训练数据。本例中，我们准备提供0~9这10个数字的手写图片总共5000张，并且：
   - 每张图片都已经标记好其对应的数字值（称为“分类标签”或Label）
   - 为了便于计算机统一处理，每张图片都是28x28像素
   - 每张图片都是灰度图（即：每个像素的取值从0~255，0为白色，255为黑色），这样就省去了处理RGB彩色的负担
2. 准备好训练数据的特征值矩阵
   程序必须把训练图片读取到某种数据结构中才能处理。
   - 对于一张图片，我们准备一个一维数组，将该图片对应的数字(Label)放在数组的第一个元素；然后将图片中每个像素点的值，按行依次连续存放在数组后面的元素中。最终该数组的共有1+28x28=785个元素
   - 将这5000张图片的数据放置在一个二维数组中，容量是：5000x785。这样就构成了一个训练矩阵。
   - 因为像素值在0~255之间，跨度较大，因此必须对训练数据进行归一化(Normalization)
   - 在本例中，我们直接从灰度图像素中提取特征值，这是最简单的提取方法。未来还会有更为复杂的提取方法
3. 使用逻辑回归进行分类
   - 有了特征值和Lable矩阵，就可以使用某种机器学习算法进行训练了。
   - 逻辑回归能够较好的进行分类。但是上一章讲的逻辑回归只能分成两个类别。因此需要考虑如何讲2-Classes的分类扩展到N-Classes的分类
4. 使用得到的假设公式进行预测
   - 训练完成后，就得到了假设公式
   - 将待预测的图片（也必须是28x28的灰度图），读取到一个一维数组中(784个元素，没有

Label）。然后就可以代入到假设公式中进行预测。预测的结果应该能告知是0~9中的哪个数

5. 使用测试数据进行验证
    - 为了判断该假设公式的有效性，需要另外找一批图片（本例中500张）进行验证。
    - 测试数据也需要先进行归一化处理
    - 本例仅统计出预测正确的图片数量占总图片数量的比重（正确率）

上述分析中，最困难的是第3步。因为目前尚不知晓如何进行N-Classes的分类

解决方法**1：1 vs All** 分类算法

该算法实际上行要对0~9是个数字分别进行逻辑回归。假设Feature矩阵为X(5000行x768列)，Label矩阵为y(5000行x1列)。

- 计算方法
    - 在对数字0进行计算时，Feature矩阵中所有不是数字0的行，在y中对应的值全设为0；否则设为1。针对Feature矩阵和Label，采用上一章讲的minimize方法求假设函数h_0，该假设函数共有769(768+1)个θ参数
    - 对数字1进行计算时，Feature矩阵中所有不是数字1的行，在y中对应的值全设为0；否则设为1。再次使用minimize方法求假设函数h_1
    - 依次类推，对于每个数字，都需要生成一个假设函数(因而共有10组θ参数)。最后产生θ的矩阵（10行x769列)，每行代表某个数字的假设函数参数
    - 预测新图片时，分别使用每个假设函数依次对图片进行预测，得到10个可能性值(范围在0~1之间)。这10个值正好就是0~9这10种数字的可能性。可能性最高的那个值，就是其对应的数字
- 关于Cost Function的计算
    - 逻辑回归计算中，很容易造成Cost Function计算溢出(NaN)。注意Cost Function中有这样一项(X是Feature矩阵)：$\frac{-y^T * log(g(X*\theta)) - (1-y^T) * log(1-g(X*\theta))}{m}$
        - 如果 $g(X*\theta)=0$，那么$log(g(X*\theta))$将会是NaN；如果$g(X*\theta)=1$，那么$log(1-g(X*\theta))$将会是NaN
        - 根据Sigmoid函数的定义和图像，可以大致估算出，如果$X*\theta$的值大于10或小于-10，则$g(X*\theta)$将会趋近于1或0
        - 因此，为保证没有溢出，需要尽可能限制$X*\theta$的大小范围在[-10,10]之间
- 归一化处理
    - 如果Feature矩阵X中各元素的值较大，那么$X*\theta$的结果也会比较大，从而使Cost Function溢出。因此，有必要对X进行归一化处理，减小元素值域范围
    - 在线性回归一章中提到可以使用： $x_i = \frac{x_i - \bar{x}_i}{std(x_i)}$ 方法来进行归一化，其中x_i是第i个Feature

列向量，$barx_i$是第i个Feature的平均值，$std(x_i)$是第i个Feature的标准差。但是，在本例中，这将有可能导致x_i为负，从而没有意义

    - 本例中，我们最终采用：$X = \frac{X}{max(X)}$ ，每个元素值域都被限定在[-1,1]之间
- 运行结果

针对测试数据的预测准确性在90%，还是具有一定的效果

- 本节代码参考【08/08scipy_minimize_logistic_digits_multi_classes.py】

解决方法2：使用sklearn.linear_model.LogisticRegression进行多分类预测

非常方便的是，LogisticRegression对象直接支持多分类模型，并且无需手工进行归一化处理，也不用给定θ的初值。但要注意：

- 创建LogisticRegression对象时，请尽量指定multi_class='multinomial' 属性
- 也可以手工先对数据进行归一化处理，能够在一定程度上影响计算结果 运行结果：
    - 无归一化处理：

    - 加入归一化

- 本节代码参考【08/09sklearn_logistic_digits_multi_classes.py】

第4节：模型性能分析

提出问题：

当我们使用一批训练数据，训练出一个假设函数(模型)后，一般会有几个疑问：

- 这个模型的效果(性能)如何呢，或者借助该模型预测出来的结果，可信度有多高呢？
- 如果预测结果不理想，那么可以从哪些方面去改进呢？

用于模型验证和评价的数据

假设手头有1000条原始数据，可按照6:2:2的比例进行分割。

- 60%的Training Examples：用于训练Hypothesis函数。要通过调整λ, learning rate等，分别计算$J_{train}(\theta)$，从而得到若干个候选的Hypothesis函数
- 20%的Cross Validation Examples：对前面的每个候选Hypothesis函数，分别计算$J_{cv}(\theta)$，取最小的一个作为最终Hypothesis
- 20%的Test Examples：用于计算前述选出的Hypothesis的$J_{test}(\theta)$及预测正确率

$J_{cv}(\theta)$，$J_{test}(\theta)$的计算公式与$J_{val}(\theta)$相同，只是将数据集换成对应的验证数据集或测试数据集

模型性能尺度

假设某Logistic Regression模型$h_\theta(x)$，如果y=1，代表发现了cancer，y=0代表没有cancer。对于给定的test data，我们预测错误率只有1%（99%的正确率），那么可以认为这个模型很好吗？

假设在真实情况下，只有0.5%的几率有cancer；那么我们完全可以设计一个极简单的算法（永远预测y=0），这样的话，也只有0.5%的错误率（比我们之前的$h_\theta(x)$还要好得多）。但是，这个极简单的算法就会更好吗？

因此，除了使用我们通常认为的错误率/正确率来衡量一个Hypothesis外，还应有其它更为重要的验算指标。

- 预测和实际结果统计表(假设仅考虑2-Classes分类的情形)

		Actual Class	
		1	0
Predicted Class	1	True Positive	False Positive
	0	False Negative	True Negative

- Actual Class表示数据集中实际的结果；Predicted Class则表示通过Hypothesis预测的结果
- True：表示实际结果与预测结果一致(预测正确)；False：表示实际结果和预测结果不一致(预测错误)
- Positive：表示预测结果为1；Negative表示预测结果为0
- 上述各中情况将分别简写为：TP, FP, TN, FN。建议可以这样理解记忆：针对一条测试数据，
  - TP：模型预测结果为P(Positive, 1)，而且预测正确（该测试数据实际结果也为Positive/1)
  - FP：模型预测结果为P(Positive, 1)，但是预测错误（该测试数据实际结果应为Negtive/0)
  - TN：模型预测结果为N(Negtive, 0)，而且预测正确（该测试数据实际结果也为Negtive/0)
  - FN：模型预测结果为N(Negtive, 0)，但是预测错误（该测试数据实际结果应为Postive/1)

- Metric计算
  - 正确率：

    $Accuracy = \frac{\#TP+\#TN}{\#TP+\#FP+\#TN+\#FN}$

    体现了：预测正确的样本数占总样本数的比重

  - 精度：$Precision = \frac{\#TP}{\#TP+\#FP}$

    体现了：对于所有预测为1的样本中，实际真正为1的样本所占的比例。它反映“误报”程度：精度越高，误报越小

  - 查全率：$Recall = \frac{\#TP}{\#TP+\#FN}$

    体现了：对于所有实际为1的样本中，预测也为1的样本所占的比例。它反映“漏报”程度：查全率越高，漏报越少

  - $F1Score = \frac{2PR}{P+R}$

    其中，P为Precision, R为Recall

在针对Cross Validation Data进行对比时，应选择F1 Score最大的那个Hypothesis来作为最优解

对于前面的例子，如果某个算法简单预测所有$y = 0$，那么其查全率为0，F1 Score=0，因此模型性能最差

- 在Precision和Recall中取平衡

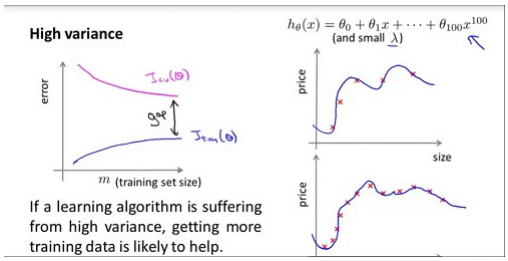
  假设某Logistic Regression算法，其Hypothesis输出：$0 \leq h_\theta(x) \leq 1$。设计某个阈值$K$，使得当$h_\theta(x) \geq K$时，预测$y=1$；当$h_\theta(x) < K$时，预测$y=0$。

  - 通常，我们设计K为0.5。也就是说，当可能性超过0.5时就做出Positive的预测结果
  - 如果设置K=0.7，意味着预测更加保守，只有高可信度才会被预测为1。 因此误报可能性降低，Precision提升；漏报可能性增加，Recall降低。
  - 如果设置K=0.3，意味着预测趋于大胆，低可信度也会被预测为1。因此误报可能性增加，Precision降低；漏报可能性减少，Recall上升。
  - 大致可以得出结论：Precisioin和Recall是有矛盾的，它们很难同时都达到最高。因此需要取折中(F1 Score)

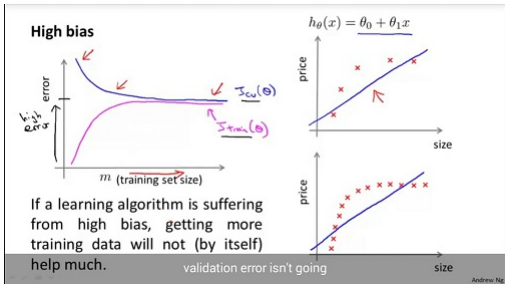
训练样本数量对性能的影响

有时候模型的预测性能很差，有可能是模型算法的问题，也有可能是样本的问题。其中，样本数量不足可能是影响模型性能的一个重要因素。本段介绍如何判断是否样本数量太少。
以训练样本数量m(training example)为横坐标，以模型最终计算出来的$J(\theta)$为纵坐标作图，该图称为学习曲线(Learning Curves)

- 对于High Variane（过拟合），其$J_{train}(\theta)$一般很小，且随着m的增加缓慢的增加。



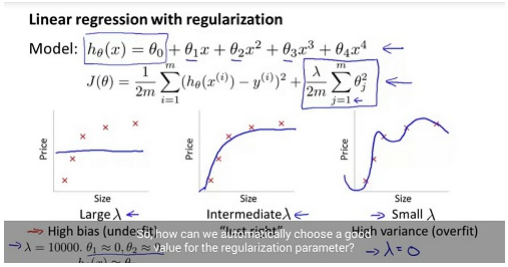
- 对于High Bias（欠拟合），其$J_{train}(\theta)$刚开始很小，但是随着m的增加，会迅速增加；$J_{cv}(\theta)$通常很大，而且随着m的增加缓慢的减少，最终二者比较接近且值都较大。在这种情况下，增加更多的训练数据没有任何帮助，而是要考虑改进算法或选取更多的Feature。



- 当$J_{train}(\theta)$趋于稳定时的m数，可作为比较合适的训练样本数量。

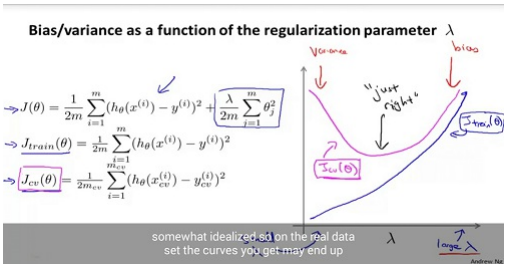
Regularization参数λ对模型性能的影响

线性回归和逻辑回归都可以在Cost Function后面增加一个penalty项(下图以线性回归为例):



- 如果λ很大，则为了使$J(\theta)$较小，所有的θ不得不趋近于0（除了θ_0，它不参与计算Regularization）。因此将会造成严重的High Bias(Underfit)， $J_{train}(\theta)$和$J_{cv}(\theta)$都会很大
- 如果λ很小，则对于θ几乎没有什么约束。这将会造成High Variane(Overfit)。$J_{train}(\theta)$非常小，但$J_{cv}(\theta)$会很大

如果以λ为横坐标，$J(\theta)$为纵坐标，可以画出类似于下图的关系。$J_{cv}(\theta)$最小值所对应的λ值最适合作为选择项。

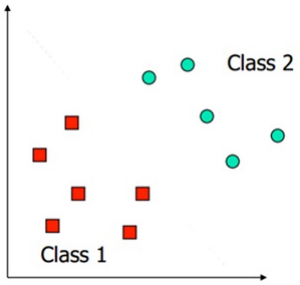


第九章：SVM

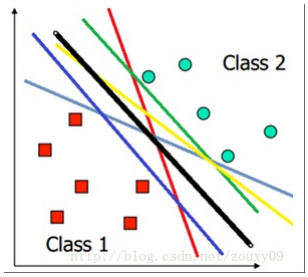
第1节：工作原理

提出问题

观察下列训练数据的二分类图：



很明显，可以用一条直线分割Class1和Class2，并且利用该分界线，来判断新的数据点到底属于哪个类别。但是这样的分割线可以画出很多条：

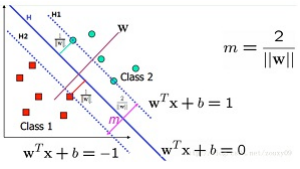


那么，那一条分界线是最合理的呢？

分析问题

支持向量机(Support Vector Machine, SVM)通过寻找一个超平面来对样本进行分割；且分割线相对样本中的正例和反例均保持最大的间隔(margin)

以2个Feature的训练数据(也就是平面上的二维坐标点)为例：



- 超平面
  - 假设中间的蓝色实线H作为分割超平面(二维情况下其实是一条线)，两边的虚线H1超平面和H2超平面与H平行，并且经过离H最近的训练数据点
  - 位于H1和H2上的点，被称为支持向量(可能有多个)
- 超平面的数学表述

- 对于二维数据点，可以使用公式$a_1x_1 + a_2x_2 + b = 0$ 来描述直线H。其中x_1是横坐标，x_2是纵坐标(为了表示Feature，这里特意不使用y作为纵坐标标记)
- 设x是包含(x_1, x_2)的列向量，w是包含(a_1, a_2)的列向量，则H直线$a_1x_1 + a_2x_2 + b = 0$ 又可以表示为：$w^T * x + b = 0$
- 在多维(超过二维)的情况下，$w^T * x + b = 0$ 仍然是成立的。w向量的长度是维度数(Feature数)
- 超平面H1所在直线：$w^T * x + b = 1$，超平面H2所在直线：$w^T * x + b = -1$
- 对于任意正样本$(y = +1)$，都应出现在H1的右边，也就是：$w^T * x + b \geq 1$；而对于任意负样本$(y = -1)$，都应出现在H2的左边，也就是：$w^T * x + b \leq -1$

- SVM的优化目标
  - 超平面H1和H2之间的距离为：$m = \frac{2}{(w^Tw)^{\frac{1}{2}}} = \frac{2}{||w||}$ ，其中， $||w|| = (w^Tw)^{\frac{1}{2}}$
  - 现在，要使得这两个超平面之间的距离最远，实际上就是要使得$||W||$最小。为了方便计算，一般设为使得 $\frac{||w||^2}{2}$ 最小
  - 另一方面，所有训练数据点，要么在H1的右边，要么在H2的左边(H1和H2之间不能有数据点)，即：$y * (w^T * x + b) \geq 1$其中，y表示样本的分类结果(+1或-1)
  - 最终得到SVM的优化目标及限制条件如下：

$$\begin{cases} target: min(\frac{||w||^2}{2}) \\ s.t.: 1 - y^{(i)} * (w^T * x^{(i)} + b) \leq 0 \end{cases}$$

    - m是样本的个数
    - $x^{(i)}$ 是第i个样本的Feature向量
    - $y^{(i)}$ 是第i个样本的分类结果(+1或-1)
  - 上述可以使用凸优化来进行求解，以得出最优的w和b，从而得到判别式$h(x) = w^T * x + b$
- 使用SVM进行预测
  - 给定待预测的Feature向量x_t，采用下列方法：$h(x_t) = sign(w^T * x_t + b)$。其中$sign$表示取符号。
    - 如果$h(x_t) > 0$，则返回+1
    - 如果$h(x_t) < 0$，则返回-1。

解决方法

示例**1**：使用**sklearn.svm**对象来求解**SVM**问题的例子

- 本节代码参考【09/01sklearn_svm_classify】

SVM与凸优化求解

优化目标

从上节分析可知，SVM优化的原问题如下：

$$\begin{cases} target: min(\frac{||w||^2}{2}) \\ \quad s.t.: 1 - y^{(i)} * (w^T * x^{(i)} + b) \leq 0, (i = 0, 1, \cdots, m) \end{cases}$$

这是典型的凸优化问题

- 为什么不直接使用scipy.optimize.minimize直接对原问题求最优解
  - 原问题其实也可以使用scipy.optimize.minimize来求最优解
  - 但是，转换为对偶问题后，将更容易引入核函数（后面介绍）

凸优化求解步骤

- 构造拉格朗日函数

  $F(w, b, \alpha) = \frac{||w||^2}{2} + \sum_{i=1}^{m} \alpha_i (1 - y^{(i)} * (w^T * x^{(i)} + b))$

  - 参数说明
    - m是样本的个数
    - $x^{(i)}$ 是第i个样本的Feature向量，假设有n维
    - $y^{(i)}$ 是第i个样本的分类结果(+1或-1)
    - 上式子中，x和y都是训练数据(已知量)，不要把它们看成自变量；而w(n维向量)，α(由$\alpha_1, \alpha_2, \cdots, \alpha_m$ 组成的向量)和b(实数)才是需要优化的变量
  - 原问题优化目标是：先针对α优化，使F能取最大值；然后针对w和b优化，使F取到最小值： $p^* = \underset{w,b}{min} \underset{\alpha}{max} F(w, b, \alpha)$ 请注意，此处将w和b放在一起，因为偏置项b本身就是与权重w同时出现的。

- 对偶问题 对偶问题优化目标是：先针对w和b优化，使F取到最小值；然后针对α优化，使F能取最大值

  $d^* = \underset{\alpha}{max} \underset{w,b}{min} F(w, b, \alpha)$

  要使原问题与对偶问题具备强对偶性，必须附加KKT条件

- 满足KKT条件中的$\frac{\partial F(x, \alpha^*, \beta^*)}{\partial x}|_{x=x^*} = 0$

- 注意，上式条件中的x对应着w和b 由：$\frac{\partial F(w,b,\alpha)}{\partial w}=0$

  $\frac{\partial F(w,b,\alpha)}{\partial w}=\frac{\frac{1}{2}w^Tw+\sum_{i=1}^m\alpha_i(1-y^{(i)}*(w^T*x^{(i)}+b))}{\partial w}==w-\sum_{i=1}^m\alpha_iy^{(i)}x^{(i)}=0$

  得到条件式**(1)**：$w=\sum_{i=1}^m\alpha_iy^{(i)}x^{(i)}$

  由：$\frac{\partial F(w,b,\alpha)}{\partial b}=0$

  $\frac{\partial F(w,b,\alpha)}{\partial b}=\sum_{i=1}^m\alpha_iy^{(i)}=0$

  得到条件式**(2)**：$\sum_{i=1}^m\alpha_iy^{(i)}=0$

- 将条件式(1)和(2)代入到拉格朗日函数

  $F(w,b,\alpha)=\frac{1}{2}w^Tw+\sum_{i=1}^m\alpha_i(1-y^{(i)}*(w^T*x^{(i)}+b))$

  $=\frac{1}{2}(\sum_{i=1}^m\alpha_iy^{(i)}x^{(i)})^T*(\sum_{i=1}^m\alpha_iy^{(i)}x^{(i)})+$

  $\sum_{i=1}^m\alpha_i-\sum_{i=1}^m[(\alpha_iy^{(i)}(\sum_{i=1}^m\alpha_iy^{(i)}x^{(i)})^Tx^{(i)})]-b\sum_{i=1}^m\alpha_iy^{(i)}$

  $=\frac{1}{2}\sum_{i=1}^m(\alpha_iy^{(i)})*\sum_{j=1}^m[\alpha_jy^{(j)}(x^{(i)})^Tx^{(j)}]-$

  $\sum_{i=1}^m\{(\alpha_iy^{(i)})*\sum_{j=1}^m[\alpha_jy^{(j)}(x^{(i)})^Tx^{(j)}]\}+\sum_{i=1}^m\alpha_i-b\sum_{i=1}^m\alpha_iy^{(i)}$

  $=\sum_{i=1}^m\alpha_i-\frac{1}{2}\sum_{i=1}^m(\alpha_iy^{(i)})*\sum_{j=1}^m[\alpha_jy^{(j)}(x^{(i)})^Tx^{(j)}]$

  此时的F函数已经不再包含变量w和b。从而，对偶问题中首先要求的：$\min\limits_{w}F(w,b,\alpha)$，直接就是上式，而与$w$和$b$无关！

- 从而对偶问题就变成了：

  $$\begin{cases}target:\max\limits_{\alpha,b}\{\sum_{i=1}^m\alpha_i-\frac{1}{2}\sum_{i=1}^m(\alpha_iy^{(i)})*\sum_{j=1}^m[\alpha_jy^{(j)}(x^{(i)})^Tx^{(j)}]\}\\s.t.:\begin{cases}\alpha_i\geq0\\\sum_{i=0}^m\alpha_iy^{(i)}=0\end{cases}\end{cases}$$

  该目标函数仅仅包含α，是比较容易通过数值方法计算极值而求解的。 请注意，我们费了很大力气将原问题转成对偶问题，使用KKT条件，最后得到只有一组变量α的优化目标函数。对于大多数优化算法来说，都是针对一组变量来进行优化！

- 求解上述对偶问题(其实是典型的Quadratic Programming，或者QP问题)，得到α的最优解。
  - scipy.optimize.minimize方法提供了针对上述问题的求解器
  - 因为对偶问题的目标函数是要求最大值，因此需要取负号，以便minimize求取最小值
- 根据条件式(1)：$w=\sum_{i=1}^m\alpha_iy^{(i)}x^{(i)}$ 得到w的最优解
- 求解最优的b值

- 最优的b值，就是分割超平面所在的b值。从SVM的基本工作原理图中可以看出，它位于$y=-1$和$y=1$两个超平面的正中间。
- 设$w^T*x_p+b=1$代表$y=1$的超平面，$w^T*x_n+b=-1$代表$y=-1$的超平面；很显然，x_p 和x_n 正好都是支持向量
- 根据SVM超平面分界图，可知：所有$y>1$的样本点x，都满足：$w^T*x>w^T*x_p$；所有$y<-1$的样本点x，都满足：$w^T*x<w^T*x_p$
- 因此，x_p就是所有$y=1$的样本中，w^T*x最小的样本点；而x_n就是所有$y=-1$的样本中，w^T*x最大的样本点。可以通过分别计算$y=1$和$y=-1$的所有样本而找到x_p和x_n(这些点就是支持向量)
- 最后计算最优的b值： $b=\frac{-(w^T*x_p+w^T*x_n)}{2}$

- 根据强对偶的特点，对偶问题的最优解w,α,b，就是原问题的最优解 最终得到原问题的判别函数：$h(x)=sign(w^T*x+b)$

解决方法

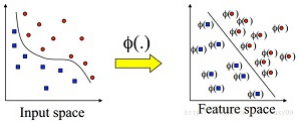
示例**1**：按照拉格朗日对偶问题使用**scipy.optimize.minimize**求**SVM**最优解的例子

- 本节代码参考【09/02convex_optimize_classify.py】

第3节：核函数

线性可分与线性不可分

- 如果无法使用一条直线(或者一阶变量)来作为分割超平面，那么称为该样本数据线性不可分
- 线性不可分的数据集无法使用前述SVM直接计算，此时需要将原有数据的Feature(X)通过某个函数进行转换，使转换后的样本能够线性可分

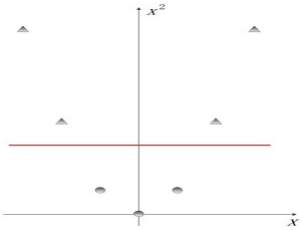


Cover定理

将复杂的模式分类问题非线性地投射到高阶空间将比投射到低阶空间更可能是线性可分

- 下图中的一阶数据，-1,0,1三个点属于一个分类，其余数据点属于另一个分类。显然，这是线性不可分的

- 对各点投射到二阶空间(取平方)，可以看到，此时的数据已经是线性可分了



- 因此，使用一个非线性映射$\phi(x)$将全部原始数据x变换到另一个特征空间，在这个空间中，样本就很可能变得线性可分了。当然，引入高阶问题会使计算变复杂

核函数

映射函数$\phi(x)$是很难直接定义的，所以一般不会直接指定该函数 回顾拉格朗日对偶问题：

$$\begin{cases} target: \underset{\alpha,b}{max} \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}(\alpha_i y^{(i)}) * \sum_{j=1}^{m}[\alpha_j y^{(j)}(x^{(i)})^T x^{(j)}] \\ s.t.: \begin{cases} \alpha_i \geq 0 \\ \sum_{i=0}^{m}\alpha_i y^{(i)} = 0 \end{cases} \end{cases}$$

- 可以看到，只有一项计算与样本的Feature相关，就是任意两个样本的内积：$(x^{(i)})^T x^{(j)}$。因

此，映射后，这个内积就变成：$\phi(x^{(i)})^T\phi(x^{(j)})$

- SVM绕过了$\phi(x)$函数本身的定义，而是直接使用核函数取代映射后的内积计算，即：

  $K(x_i,x_j)=\phi(x_i)^T\phi(x_j)$

- 常用的核函数叫径向基函数(RBF)，也叫高斯核函数：$K(x_1,x_2)=e^{-\frac{||x_1-x_2||^2}{2\sigma^2}}$
- 在给定了核函数之后，无需再计算出$\phi(x)$本身值，就能计算出其映射后的内积，从而能够使SVM的计算过程顺利进行，最终的对偶问题为：

$$\begin{cases} target: \underset{\alpha,b}{max} \sum_{i=1}^{m}\alpha_i-\frac{1}{2}\sum_{i=1}^{m}(\alpha_i y^{(i)})*\sum_{j=1}^{m}[\alpha_j y^{(j)}K(x^{(i)},x^{(j)})] \\ s.t.: \begin{cases} \alpha_i \geq 0 \\ \sum_{i=0}^{m}\alpha_i y^{(i)}=0 \end{cases} \end{cases}$$

- 最终的判别函数为：

  $h(x)=sign(w^T*x+b)=sign(\sum_{i=1}^{m}\alpha_i y^{(i)}K(x^{(i)},x)+b)$

解决方法

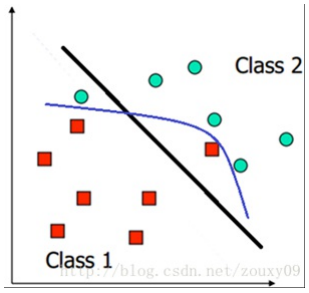
示例**1**：观察不同的核函数对分类边界的映像

- 本节代码参考【09/03sklearn_svm_kernel_compare.py】

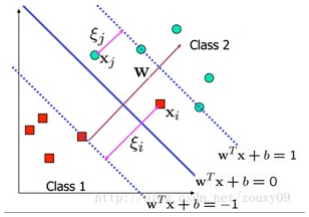
第4节：软间隔与过拟合

过拟合问题

- 对于线性不可分的样本，虽然可以通过核函数引入高阶变量使之变成线性可分的，但是也可能会造成过拟合：



- 因此，有时候宁愿牺牲少数几个样本数据，尽量确保总体趋势上的正确性。这就需要修改前面提到的限制条件。如下图所示，允许少量点x_i, x_j 等位于H1和H2超平面之间。这可以通过设定一个不小于0的松弛变量(slack variable) ξ来达到:



此时的优化目标变成：$min(\frac{||w||^2}{2} + C\sum_{i=1}^{m}\xi_i)$，对应的限制条件变成：

$y * (w^T * x + b) + \xi - 1 \geq 0$

带松弛变量的SVM解法

- 原问题

$$\begin{cases} target: min(\frac{||w||^2}{2} + C\sum_{i=1}^{m}\xi_i) \\ s.t.: 1 - \xi_i - y^{(i)} * (w^T * x^{(i)} + b) \leq 0, (i = 0, 1, \cdots, m) \\ -\xi_i \leq 0 \end{cases}$$

C称为“离群点权重”或者惩罚因子，它的值越大，表明离群点的后果越严重，而计算出来的结果，将尽可能消除离群点(H1和H2间隔更小，偏向过拟合)

- 拉格朗日对偶函数

$$\begin{cases} F(w,\alpha,\beta,\xi,b)=\frac{||w||^2}{2}+C\sum_{i=1}^{m}\xi_i+\sum_{i=1}^{m}\alpha_i(1-\xi_i-y^{(i)}*(w^T*x^{(i)}+b))-\sum_{i=1}^{m}\beta_i\xi_i \\ \alpha_i>0 \\ \beta_i>0 \\ \xi>0 \\ \frac{\partial F(w,\alpha,\beta,\xi,b)}{\partial w}=0 \\ \frac{\partial F(w,\alpha,\beta,\xi,b)}{\partial b}=0 \\ \frac{\partial F(w,\alpha,\beta,\xi,b)}{\partial \xi}=0 \end{cases}$$

- 通过式中等于0的三个偏导方程消去w,ξ,b，得到已于求解的对偶问题：

$$\begin{cases} target: \underset{\alpha,b}{max}\{\sum_{i=1}^{m}\alpha_i-\frac{1}{2}\sum_{i=1}^{m}(\alpha_i y^{(i)})*\sum_{j=1}^{m}[\alpha_j y^{(j)}(x^{(i)})^T x^{(j)}]\} \\ \quad s.t.: \begin{cases} \alpha_i\geq 0 \\ \alpha_i\leq C \\ \sum_{i=0}^{m}\alpha_i y^{(i)}=0 \end{cases} \end{cases}$$

  - 可以看到，最优的α其实与ξ无关
  - SMO算法更适合求解含有$C\geq\alpha_i\geq 0$约束条件的问题
  - 仍然可以使用核函数来代替 $(x^{(i)})^T x^{(j)}$

第**5**节：处理多分类问题

1 vs All方法

- 针对所有样本数据，先分成1类和其它类，使用SVM计算边界超平面L1
- 再分成2类和其它类，再次计算超平面L_2。依次类推直到L_k (k为分类数)
- 预测新数据时，分别跟$L_1, L_2, \cdots, L_k$进行判断，如果有一个超平面直接判别属于某个类别，则完成预测
- 如果有多个超平面判别属于各自不同类别，则一般只能任选一个
- 如果没有任何超平面判别属于某个类别，则一般只能视为无法分类数据

---

1 vs 1方法

- 先只对1类和2类的数据进行计算，得到1类和2类的分割超平面
- 然后对2类和3类计算，之后1类和3类...等等。直到两两类别分别完成计算
- 预测新数据时，对两两类别之间的分割超平面分别进行匹配。统计有多少次判别将其划归为1类，多少次判为2类....。判定所属类别次数最多的，就是预测的结果类别

示例**1**：使用多分类**SVM**来计算数字图片分类

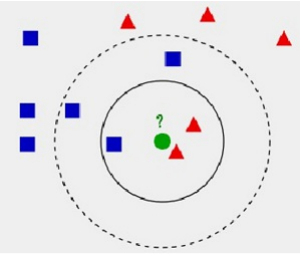
- 本节代码参考【09/04sklearn_svm_digits_multi_classes.py】

第十章：K分类核聚类

第1节：K近邻分类

提出问题

已知N维空间中若干个点的坐标，以及这些点所属的类别(子空间)。给定新的点坐标，如何判断该点应被划入哪个类别(子空间)?



分析问题

- KNN算法基本思想： 已知一批数据集及其对应的分类标签，输入测试数据，将测试数据的特征与训练集中对应的特征进行相互比较，找到训练集中与之最为相似的前K个数据，则该测试数据对应的类别就是K个数据中出现次数最多的那个分类。具体步骤：

  (1) 计算测试数据与各个训练数据之间的距离； (2) 按照距离的递增关系进行排序； (3) 选取距离最小的K个点； (4) 确定前K个点所在类别的出现频率； (5) 返回前K个点中出现频率最高的类别作为测试数据的预测分类。

  与之前线性回归、逻辑回归和朴素贝叶斯模型不同，KNN算法不会形成假设函数。每次预测时，必须即时跟所有训练数据进行计算，因此工作量很大。

- K值的选取：
  - K可以视为一个hyper-parameter(超参数)，一般需要通过交叉验证的方法来选取最优值
  - 如果K值太小就意味着整体模型变得复杂，容易发生过拟合(High Variance)，即如果邻近的实例点恰巧是噪声，预测就会出错，极端的情况是K=1，称为最近邻算法，对于待预测点x，与x最近的点决定了x的类别
  - K值的增大意味着整体的模型变得简单，极端的情况是K=N，那么无论输入实例是什么，都简单地预测它属于训练集中最多的类。这样的模型过于简单，容易发生欠拟合(High Bias)
- K-D树方法：
  - K近邻法的最简单实现是线性扫描（又称暴力法），这时要计算输入实例与每一个训练实例的距离，当训练集很大时，计算非常耗时，这种方法是不可行的
  - 为了提高K近邻搜索的效率，可以考虑使用特殊的结构存储训练数据，以减少计算距离的次数。K-D树提供了一种最基本的方法

    - 可将K-D树看成类似于二叉查找树的数据结构，每个训练数据存储在这个数据结构中。在查找数据时，采用类似于中序遍历的算法。（但实际过程要复杂许多）。
    - K-D树搜索的平均复杂度为$O(logM)$。其中,M为样本数量。但是当Feature数量N很大时，搜索性能将急剧下降。一般K-D树适于M>>N的情形
- KNN的优缺点：
    - 精度较高，对异常值不太敏感
    - 特别适合多分类的情况
    - 简单易实现
    - 很多情况下比朴素贝叶斯的效果更好
    - 计算复杂度高，空间复杂度高

解决方法

示例**1**：实现简单的**KNN**算法

- 给定若干组数据及其对应分类，再给定一组测试数据，使用KNN计算出该测试数据最接近的分类
- 本节代码参考【10/01simple_knn_classify.py】

示例**2**：使用**sklearn.neighbors.NearestNeighbors**

- NearestNeightbors能够直接获得最接近的K个分类(以该分类对应的下标索引形式返回)，以及对应的距离
- 选择KNN的实现算法(指定algorithm参数)
    - 'brute'：暴力法
    - 'kd-tree'
    - 'ball-tree'
- 设置权重模式(指定weights参数)
    - 'uniform'：所有点的权重相同
    - 'distance'：越接近的点，权重越大
    - [callable]：允许通过一个自定义的函数来确定各个点的权重
- 设置距离计算模式(指定metric参数)
    - 默认为'minkowski'。其它可选如：euclidean,manhattan,chebyshev等
- 本节代码参考【10/02sklearn_nearest_neighbors_classify.py】

示例**3**：使用**sklearn.neighbors.NearestNeighbors**识别手写数字图片

- 本节代码参考【10/03knn_digits_classify.py】和【10/digits_training.csv】、【10/digits_testing.csv】

第2节：K-Means聚类算法

提出问题

对于给定了Label的训练数据，可以使用KNN来分类，但是对于没有给定Label的训练数据，如何根据其中的Feature，对数据进行分类，使分类下的样本数据看上去比较接近呢？

分析问题

- 监督学习和无监督学习
  - 以往的回归、朴素贝叶斯、SVM等都是有类别标签y的，也就是说样例中已经给出了样例的分类。这类机器学习称为监督学习
  - 而聚类的样本中却没有给定y，只有特征x；其目的是找到每个样本x潜在的类别y，并将同类别y的样本x放在一起
- K-Means算法
  - 计算过程
    1. 估计样本中的总分类个数K
    2. 随机选取K个聚类质心点$\mu_1, \mu_2, \cdots, \mu_k$
    3. 对于每一个样本，分别计算到每个质心点的距离$d_1, d_2, \cdots, d_k$，取最近的一个距离，作为该样本暂时所属的分类k。可以选择euclidean距离作为参考依据。至此，每个样本都归入某个类别下
    4. 对于每个类别，计算其所辖的每个样本到其质心的距离之和，作为累积距离偏差W_k
    5. 针对每个类别，重新计算质心：$\mu_k = \frac{\sum_{i=1}^{N_k} x_i}{N_k}$，其中，$N_k$表示当前该类别下的样本数量
    6. 利用新的质心，再次计算累积距离偏差W_k'
    7. 对于两次计算出的W_k，如果其差值小于指定的误差限(例如0.00001)，则可认为已经收敛；否则再次进行步骤5)和6)
  - 算法的优缺点
    - 算法简单，易于实现
    - 初始质心选择很重要，有可能会导致找不到全局收敛解(只找到局部收敛解)；但是又没有什么好的办法来优化质心的选择
    - K值的选取也很重要。K值越大，对训练数据的分类将很好，但有可能造成overfit。也并没有太好的办法确定K值

解决问题

示例**1**：自行实现**K-Means**算法

- 本节代码参考【10/04kmeans_points_cluster.py】

示例**2**：使用**sklearn.cluster.KMeans**

- 本节代码参考【10/05sklearn_kmeans_points_cluster.py】

第十一章：朴素贝叶斯分类

第1节：离散型

提出问题

如何从一条短信内容判断它是垃圾短信(Spam)的可能性？ 假设已经有大量短信文本，并且已知每条短信是否垃圾短信。那么再给出一条新短信文本，如何根据已有短信信息对这条新短信做出判断？

分析问题

- 朴素贝叶斯方法

  对于一个分类问题，给定样本特征x，样本属于类别y的概率，根据贝叶斯理论计算如下：

  $P(y|x) = \frac{P(x|y)P(y)}{P(x)}$ 假设x维度为N(即：有N个特征)，c代表y的可能分类，其维度为K(即：y有K个分类);若提供一个x向量，则计算出该x向量所对应的y是类别k的可能性为：

  $$P(y=c_k|x) = \frac{P(x|y=c_k)*P(y=c_k)}{P(x)} = \frac{P(x_1|y=c_k,x_2|y=c_k,\cdots,x_n|y=c_k)*P(y=c_k)}{P(x_1,x_2,\cdots,x_n)}$$

  "朴素"是指，这n个特征在概率上彼此独立，即：

  $$P(x_1|y=c_k,x_2|y=c_k,\cdots,x_n|y=c_k) = P(x_1|y=c_k)*P(x_2|y=c_k)*\cdots*P(x_n|y=c_k)$$

  根据全概率公式：

  $P(x_1,x_2,\cdots,x_n)$

  $= P(x_1,x_2,\cdots,x_n|y=c_1)*P(y=c_1) + P(x_1,x_2,\cdots,x_n|y=c_2)*P(y=c_2)+$
  $\cdots + P(x_1,x_2,\cdots,x_n|y=c_k)*P(y=c_k)$

  $= \sum_{k=1}^{K}[P(x_1,x_2,\cdots,x_n|y=c_k)*P(y=c_k)]$

  $= \sum_{k=1}^{K}[P(x_1|y=c_k)*P(x_2|y=c_k)*\cdots*P(x_n|y=c_k)*P(y=c_k)]$

  $= \sum_{k=1}^{K}[P(y=c_k)*\prod_{i=1}^{n}P(x_i|y=c_k)]$

  最终可得：

  $P(y=c_k|x)$

  $= \frac{P(x_1|y=c_k)*P(x_2|y=c_k)*\cdots*P(x_n|y=c_k)*P(y=c_k)}{P(x_1)*P(x_2)*\cdots*P(x_n)}$

  $= \frac{[\prod_{i=1}^{N}P(x_i|y=c_k)]*P(y=c_k)}{\sum_{k=1}^{K}[P(y=c_k)*\prod_{i=1}^{N}P(x_i|y=c_k)]}$

- 朴素贝叶斯如何用于文本信息分类： 对于m条文本信息(假设全是英文)，如果已经知道它们一共分为K类(例如，分为垃圾信息和非垃圾信息两类)，一般的做法是：

- 建立一个词汇表：$Dict[N]$ 1) 将每一条文本信息拆分成若干个单词 2) 将每个单词加入到词汇表中，如果词汇表中已经存在该单词，则只保留一个 3) 遍历所有文本，建立起完整的词汇表，假设其词汇总数为N，这就是Feature的数量
- 计算先验概率 1) 遍历所有文本信息的分类，计算从1~K，每个类别所占的比重(概率)：$Probability_{c=k}$ 。例如，1号类别占总数的20%，2号占15%，等等 2) 按照信息分类，在每一个分类下的所有文本信息中，先统计该类下的单词总数($N_{c=k}$)，然后针对词汇表中的每个单词，依次计算其占$N_{c=k}$ 的比重(概率)。从而得到：$Probability\_Dict[N]_{c=k}$ 3) 计算出所有类别下，每个单词的出现概率：$Probability\_Dict[K,N]$
- 预测新信息的类别 1) 将新信息拆分成若干个单词 2) 建立一个数组：$Test\_Feature[N]$，对照单词表，如果某个单词在单词表位于第$index$个元素，则$Test_Feature[index]=1$；如果不存在则设为0 3) 根据贝叶斯公式：$P(y=c_k|x)=\frac{\sum[\prod_{i=1}^{N}P(x_i|y=c_k)]*P(y=c_k)}{\sum_{k=1}^{K}[P(y=c_k)*\prod_{i=1}^{N}P(x_i|y=c_k)]}$ ，对于每个类别，其分母部分都是相同的，因此只需要比较分子部分，最大的那个即是其所属类别。 4) 将$Test_Feature$与$Probability_Dict[i,N]$求内积，然后乘以$Probability_{c=i}$，得到上式中分子部分(类别为i时)。 5) 将i从0 K分别计算，结果最大的那个就是所属类别

注意事项

上述过程并不复杂，但是在实际计算时会产生两个问题：

- 平滑处理
  - 如果词汇表中，某个单词(Feature)在某些分类下占总单词数的比重为0，例如：对于分类2，词汇表中的第8个单词"hello"从来没有出现过，那么$P(x_8|y=c_2)=0$，进而贝叶斯公式中的整个分子部分都为0。这就意味着，无论该文本中是否存在其它单词，都会使其属于2号分类的概率为0。这是不合理的
  - 因此，需要确保每个Feature在每个分类中出现的次数"大于0"。一般的做法是，采用下列公式计算该单词在该类别中的先验概率：

$$P(x_i|y=c_k)=\frac{T_{x_i|c_k}+\alpha}{T_{c_k}+\lambda*\alpha}$$

    - $T_{x_i|c_k}$ 表示第i个特征在第k个分类中的出现次数；T_{c_k}表示第k个分类中的单词总数；
    - λ为x的特征数量。对于文本分类来说，λ可取成单词表的容量。
    - α称为平滑参数，若取值为1，则称为拉普拉斯平滑(Laplace smoothing)；若取值在0~1之间，则称为Lidstone smoothing
  - 采用平滑处理后，不会再出现概率为0的情形，并且仍然可以保证所有特征的概率之和仍为1

    - 例如，假设某个分类下的单词总数为$T_{c_k}=2000$，而词汇表共有500个单词，则可设置$\lambda=500,\alpha=1$
    - 如果有1个单词出现了2000次(意味着只有这一个单词出现了，其它单词都没有出现过)，那么这该单词出现的概率为 (2000+1)/(2000+500x1)=2001/2500
    - 词汇表中剩余的499个单词，每个出现的概率为：1/2500，共计499/2500。二者之和正好为1
- 对数概率运算
    - 贝叶斯公式的分子部分： $\prod_{i=1}^{N}P(x_i|y=c_k)$ 存在概率连续乘积的情况。如果N比较大(Feature较多)，则连乘将导致结果越来越小，最后超出了浮点数运算范围而无法继续计算
    - 观察下列事实： $P_1*P_2*P_3*\cdots*P_n=e^{log(P_1)+log(P_2)+log(P_3)+\cdots+log(P_n)}$ 该式子将小数的连乘转换成自然对数结果的和，这样，即使每项都不大，但却避免了乘积更小的问题。
    - 借助这一点，在计算概率连乘时，应采用对数概率形式：

      $log[\prod_{i=1}^{N}P(x_i|y=c_k)]$

      $=log(P(x_1|y=c_k))+log(P(x_2|y=c_k))+\cdots+log(P(x_N|y=c_k))$

      $=\sum_{i=1}^{N}log(P(x_i|y=c_k))$

      $P(y=c_k|x)$

      $=\frac{[\prod_{i=1}^{N}P(x_i|y=c_k)]*P(y=c_k)}{\sum_{k=1}^{K}[P(y=c_k)*\prod_{i=1}^{N}P(x_i|y=c_k)]}$

      $=\frac{P(y=c_k)*e^{\sum_{i=1}^{N}log(P(x_i|y=c_k))}}{\sum_{k=1}^{K}[P(y=c_k)*e^{\sum_{i=1}^{N}log(P(x_i|y=c_k))}]}$

      这将大大改善浮点运算的效能

解决方法

示例**1**：简单的文本感情色彩分析

- 假设有很多条文本，并且已知每条文本的是否是带侮辱性的文本。根据已有的数据，识别出一条新文本是否带侮辱性的文本。
- 在预测新文本时，仅对按分类计算出的概率的分子部分进行比较，省去了计算分母的麻烦。
- 本节代码参考【11/01naive_bayes_classify_text.py】

示例**2**：使用**sklearn.naive_bayes.MultinomialNB**的文本感情色彩分析

- 本节代码参考【11/02sklearn_naive_bayes_multinomialnb_classify_text.py】

示例**3**：垃圾信息分类

- 在message.csv文本中，包含有大量的短信。每行数据包括两个字段：短信内容，以及该短信是否垃圾信息的标志(1或0)。
- 现在希望将这些数据拆分成训练数据集和测试数据集，根据训练数据集做出一个预测模型；然后在测试数据集上来验证其效果。
- 本例可采用典型的二分类朴素贝叶斯算法来实现。为此，定义了BinaryNaiveBayesClassifier类。该类根据贝叶斯公式直接计算出每个分类下的概率，超过0.5的，就认为归属于该分类。
- 本例中，设置了$\alpha = 1$
- 本节代码参考【11/03naive_bayes_classifier_spam.py】、【11/naive_bayes_classifier.py】，以及测试数据【11/messages.csv】

示例**4**：使用**sklearn.naive_bayes.MultinomialNB**的垃圾信息分类

- MultinomialNB对象的α属性，可以用于设置或获取相应的α值
- 将本例的结果与案例3进行对比，可以看到，在$\alpha = 1$的情况下，两个计算结果非常接近
- 本节代码参考【11/05sklearn_naive_bayes_multinomialnb_spam.py】以及测试数据【11/messages.csv】

第2节：连续型

提出问题

通过一些测量的特征，包括身高、体重、脚的尺寸，判定一个人是男性还是女性。
已有训练数据如下表：

性别 y	身高(英尺) x1	体重(磅) x2	脚的尺寸(英尺) x3
男	6	180	12
男	5.92	190	11
男	5.58	170	12
男	5.92	165	10
女	5	100	6
女	5.5	150	8
女	5.42	130	7
女	5.75	150	9

与之前例子中的单词个数统计不同，身高、体重、脚的尺寸等数据是连续分布的，无法用计数来统计。例如，分别统计身高为6、6.01、6.02、6.03....的人所占的比重，是没有意义的。因此，对于连续型随机变量，无法像离散型变量一样使用$\frac{T_{x_i|c_k}}{T_{c_k}}$来计算每个Feature在某个分类中的概率。

分析问题

- 连续型随机变量对应有概率密度函数，可以通过概率密度函数来计算在某一点的概率密度（注意，连续型随机变量的PDF计算出来的某一点的值，并不是直接的概率）。我们将采用概率密度函数值代替离散情形下的概率值。此时的朴素贝叶斯公式变为：

  $$P(y=c_k|x)=\frac{pdf(x_1|y=c_k)*pdf(x_2|y=c_k)*\cdots*pdf(x_n|y=c_k)*P(y=c_k)}{evidence}$$

  - 分母evidence一般不必计算，只需比较不同分类中分子的大小，就能判断出所属的类别
  - 即使是连续型随机变量，每种分类所占的比重$P(y=c_k)$仍然是可以计算的。例如，我们可以从训练样本数据种计算出本例中两个类别(男、女)的概率为

    $P(y=c_1)=P(y=c_2)=0.5$
  - 只要能够确定概率密度函数pdf，就能够计算出后验概率
- 在大部分情况下，可以假定随机变量的分布遵循正态分布。因此我们可以直接采用正态分布的概率密度函数。正态分布的pdf需要指定均值μ和方差σ^2，这两个参数可以根据训练数据计算出来，例如：

  $$\mu(x_1|y=1)=\frac{\sum_{i=1}^{m_{(y=1)}}x_1^{(i)}}{m_{(y=1)}}=(6+5.92+5.58+5.92)/4=5.855$$

$$\begin{aligned}\sigma^2_{(x_1|y=1)} =& \frac{\sum_{i=1}^{m_{(y=1)}}[x_1^{(i)} - \mu(x_1|y=1)]^2}{m_{(y=1)} - 1} \\ =& ((6-5.855)^2 + (5.92-5.855)^2 + (5.58-5.855)^2 + (5.92-5.855)^2)/3 \\ =& 0.03503\end{aligned}$$

  - $y=1$代表分类为"男"的情形，$y=0$代表分类为"女"的情形
  - $m_{(y=1)}$ 表示$y=1$情形下的数据条数

  按照这种方法，可以依次计算出每个Feature在每种分类下的正态分布的μ和σ^2：

性别 y	均值(身高) x1	方差(身高) x1	均值(体重) x2	方差(体重) x2	均值(脚的尺寸) x3	方差(脚的尺寸) x3
男性	5.855	3.5033e-02	176.25	1.2292e+02	11.25	9.1667e-01
女性	5.4175	9.7225e-02	132.5	5.5833e+02	7.5	1.6667e+00

- 使用正态分布的概率密度函数计算后验概率。例如，要求身高、体重、脚尺寸分别为6、130、8的测试样本所述的分类，可以这么做：

  $$pdf(x_1|y=1) = \frac{1}{\sigma_{x_1|y=1} * \sqrt{2\pi}} * e^{-\frac{(x-\mu(x_1|y=1))^2}{2\sigma_{x_1|y=1}{}^2}} = \frac{1}{\sqrt{0.035033} * \sqrt{2\pi}} * e^{-\frac{(6-5.855)^2}{2*0.035033}} = 1.5789$$

  依次类推，计算出$pdf(x_2|y=1)$和$pdf(x_3|y=1)$，再结合$P(y=1)=0.5$，可计算出：

  $P(y=1|x) = \frac{6.1984^{-9}}{evidence}$

  $P(y=2|x) = \frac{5.3778^{-4}}{evidence}$

  可以看出，该测试样本是女性的概率要远远高于男性

解决方法

示例**1**：使用**sklearn.naive_bayes.GaussianNB**对连续性随机变量进行朴素贝叶斯分类

- 本节代码参考【11/05sklearn_naive_bayes_gaussiannb_classify_gender.py】

第十二章：决策树和随机森林

第1节：决策树

提出问题

给定一批样本，每个样本列出了在若干种特定因素影响下，样本最终的分类结果。每种因素的取值是有限的几个值。
对于新给定的影响因素，如何判断其所属的类别？

分析问题

- 决策树的基本概念
  - 决策树是通过一系列规则对数据进行分类的过程。它提供一种在什么条件下会得到什么值的类似规则的方法。决策树分为分类树和回归树两种，分类树对离散变量做决策树，回归树对连续变量做决策树
  - 样本所有特征中有一些特征在分类时起到决定性作用，决策树的构造过程就是找到这些具有决定性作用的特征，根据其决定性程度来构造一个树--决定性作用最大的那个特征作为根节点，然后递归找到各分支下子数据集中次大的决定性特征，直至子数据集中所有数据都属于同一类
  - 构造决策树的过程本质上就是根据数据特征将数据集分类的递归过程，我们需要解决的第一个问题就是，当前数据集上哪个特征在划分数据分类时起决定性作用
  - 要获取对分类结果起决定性作用的特征，可以借助信息论中的条件熵或信息增益来进行判断。对每个特征分别计算信息增益，增益最大的那个就可以视为起最决定作用的特征
- ID3：基于信息论的决策树算法
  - 算法步骤：

    1. 以所有样本为工作数据集
    2. 分别计算每个特征的条件熵，选取条件熵最小的那个特征作为根节点，并假设该特征有$n1$种取值$x_1, x_2, \cdots, x_{n1}$
    3. 在根节点下设置$n1$个分支$D(D_1, D_2, \cdots, D_{n1})$，分支$D_i$的工作数据集设置为所有包含特征值$x_i$的样本；
    4. 针对每个分支D_i

       1) 检查该分支下的所有工作数据集，如果每条数据的判别结果都相同，则直接以该判别结果作为叶子节点，结束该分支的构建；如果判别结果有多个，则继续执行2)
       2) 在该分支的工作数据集中，计算剩余特征(去掉了父节点特征后的)的条件熵，选取条件熵最小的那个特征作二级节点
       3) 如果被选中的二级节点的条件熵为0，则说明该特征已经可以完全判别样本，以该

二级节点特征的取值作为分支，其取值对应的判别结果作为叶子节点，完成该分支树的构建

4) 如果二级节点条件熵大于0，则根据二级节点特征的取值，重新设置样本中包含该特征值工作数据集，开始新一轮的计算

5. 重复执行4，直到所有分支都确定了叶子节点，从而完成决策树的构建

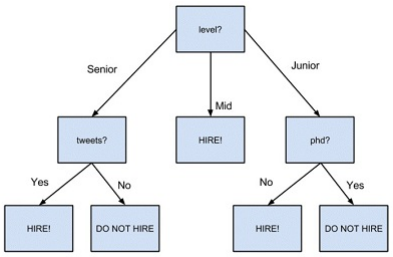
- 一旦建立了决策树，对于新的数据，可以从根节点开始逐个检索得出最终的判别结果
- 缺点：偏向于具有大量值的属性--就是说在训练集中，某个属性所取的不同值的个数越多，那么越有可能拿它来作为分类属性，而这样做有时候是不合理的的。此外，它不能处理特征值是连续变量的情形

- ID3的改进算法
  - C4.5是ID3的一个改进算法，用信息增益率来选择属性。C4.5支持对连续特征变量的离散化处理。它的缺点是效率低，只适合于在内存中计算
  - CART算法的全称是Classification And Regression Tree，采用的是Gini指数（选Gini指数最小的特征）作为分类标准。本课程不作进一步解释

解决方法

示例**1**：基于**ID3**算法的决策树构建过程演示

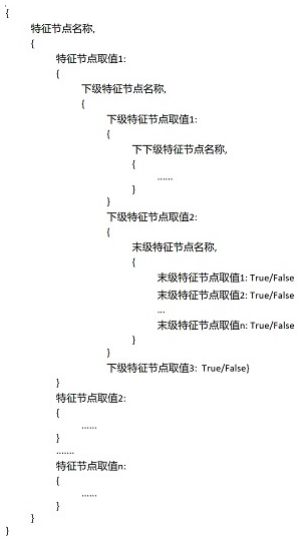
- 本例演示如何计算条件熵，选择根节点、二级节点，以及如何判断分支构建结束
- 最终构成的树结构如下图所示：



- 本节代码参考【12/01hr_decision_id3_step_by_step_demo.py】

示例**2**：**ID3**算法的自定义决策树分类器

- 实现了一个最简单的，支持True/False两中分类结果判别的决策树
- build_tree_id3方法：用于构造决策树。决策树的结构形如：



每个节点下都增加一个名为None的分支，该分支下只包含叶子节点，叶子节点直接返回其父节点下的所有样本的主要判别结果

- classify方法：用于根据已经构造好的决策树对新数据分类
  - 待预测的数据中，很可能有样本数据中不存在的特征名称或特征值，因此，决策树分类器必须要能够应对这种情况
  - 本例子中为简单起见，待预测数据中无论是出现了意外的特征名/特征值，还是缺失某些特征名/特征值，都将视为特征值为None，从而将选择决策树当前检索节点的None分支继续搜索
- 本节代码参考【12/02hr_decision_id3_tree_test_demo.py】及【12/id3_tree.py】

示例**3**：自定义决策树并计算预测准确率

- 本节代码参考【12/03car_decision_id3_tree_test_demo.py】,【12/id3_tree.py】及【12/car.csv】数据文件

示例**4**：使用**sklearn.tree.DecisionTreeClassifier**对前述数据进行分类

- 在进行训练之前，要把字符串形式的Feature值转换成数值形式；一个特征下的不同特征取值，分别用不同的数值代表
- 字符串形式的Label也要转换成数值形式的分类
- 本例提供了14条训练数据和2条验证数据。请注意，验证数据也需要进行数值化转换。因此，本例中先将所有数据都进行数值化转换；但最后执行训练时，仅取前14条
- 本例的数值化转换代码暂未考虑验证数据中特征值缺失的问题！
- 本节代码参考【12/04hr_decision_tree_classifier_numerized_demo.py】

示例**5**：使用**sklearn.feature_extraction.DictVectorizer**辅助进行数值化转化

- 该对象能够将字符串形式的特征值提取出来，展开成数值向量形式
- 请注意，对于缺失特征值的情形，可能给出不同于以上案例的判别结果
- 本节代码参考【12/05hr_decision_tree_classifier_dictvectorizer_demo.py】

示例**6**：从**csv**文件中读取数据建立决策树并预测

- 本节代码参考【12/06car_tree_decision_demo.py】及【12/car.csv】数据文件

第2节：随机森林

基本原理

- 单一决策树很容易产生过拟合；如果采用多棵决策树，共同投票来做决定，往往会比采用单一决策树具有更好的效果。
- Bagging策略：从样本集（假设样本集N个数据点）中重采样选出n个样本（有放回的采样，样本数据点个数仍然不变为N），对这n个样本建立分类器（ID3\C4.5\CART等方法），重复以上两步m次，获得m个分类器，最后根据这m个分类器的投票结果，决定数据属于哪一类。
- 使用步骤：
  - 样本的随机：从样本集中用Bagging策略随机选取n个样本
  - 特征的随机：从所有属性中随机选取K个属性，选择最佳分割属性作为节点建立CART决策树（也可以是其他类型的分类器，比如SVM、Logistics）
  - 重复以上两步m次，即建立了m 棵CART决策树
  - 这m个CART形成随机森林，通过投票表决结果，决定数据属于哪一类（投票机制有一票否决制、少数服从多数、加权多数）

解决问题

示例**1**：使用自定义随机森林

- 生成多棵决策树
  - 每棵决策树使用不完全相同的样本数据。每次生成决策树之前，先将原始样本数据的顺序打散，然后取前若干条数据作为样本
  - 因为特征数量不多，因此本例没有对特征进行随机选取，而是直接使用了所有特征
- 投票
  - random_forest.forest_classify方法用于在多个决策树之间做出判别并根据投票结果给出最终判别
- 本节代码参考【12/01car_id3_random_forest_test_demo.py】、【12/random_forest.py】以及【12/car.csv】数据文件

示例**2**：使用**sklearn.ensemble.RandomForestClassifier**

- 本节代码参考【12/02car_radom_forest_demo.py】以及【12/car.csv】数据文件

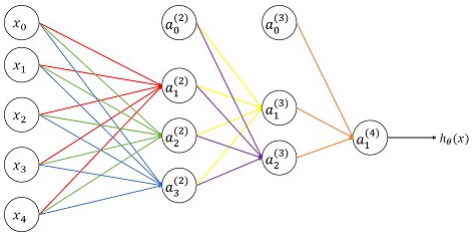
第十三章：神经网络

第1节：神经网络基本结构

前馈神经网络

前馈神经网络是一种最简单的神经网络，各神经元分层排列。每个神经元只与前一层的神经元相连。接收前一层的输出，并输出给下一层．各层间没有反馈。
适于处理复杂的非线性分类情况。相比线性回归、logistic回归，提高灵活性的同时，又不太会有过拟合。

对于一组训练数据(x, y)，其中，x包含4个Feature，分别记为x_1, x_2, x_3, x_4
如下图所示的神经网络一共分为4层：



- 各层解析
  - x也被称为神经网络的Layer 1(第一层)，又称：Input Layer。该层也可以记为$a^{(1)}$ 。其中x_1, x_2, x_3, x_4 分别也可记为$a_1^{(1)}, a_2^{(1)}, a_3^{(1)}, a_4^{(1)}$
  - $a^{(2)}$ 是网络中的Layer 2, $a^{(3)}$ 是网络中的Layer 3。这两层又称：Hidden Layer
  - $a^{(4)}$是网络中的Layer 4，也就是最后一层。该层又称：Output Layer，它就是Hypothesis函数
  - 本例中Output Layer最终只有1个元素。此时该元素代表该组x所对应的y值是1的几率。在Multi Classfication中，Output Layer可能有多个元素，那么第k个元素就代表该组x所对应的y值是k的几率。取几率最大的那个k值作为预测结果。
  - a_0或x_0称为Bias Unit，一般赋值为1。（类似于Linear Regression和Logistic Regression中的Intercept Item）
- 计算任务
  - 从$a^{(1)}$到$a^{(2)}$，需要乘以第一层模型参数$\theta^{(1)}$；从$a^{(2)}$到$a^{(3)}$，需要乘以第二层模型参数$\theta^{(2)}$；从$a^{(3)}$ 到$a^{(4)}$，需要乘以第二层模型参数$\theta^{(3)}$
  - 请注意，后一层节点中的每个元素，都跟前一层节点的所有元素有关。每一层节点均可视为具有N个元素的列向量
  - 每层的$\theta^{(i)}$都是一个$N^{(i+1)} \times N^{(i)}$ 矩阵，其中$N^{(i)}$ 表示前一层的节点数量(含Interception Item)；$N^{(i+1)}$ 表示后一层的节点数量(不包含Interception Item)。以下将该矩阵写成符

号： Θ

- 完整计算过程
  - 先给定Θ的初始值，然后从输入层开始，执行正向计算(Foword Propagation)，直到计算出初始的$h_\theta(x)$
  - 根据Cost Function计算此时的Cost
  - 使用反向计算(Backpropagation Algorithm)来计算偏导数（梯度），执行梯度递减算法
  - 循环执行

前向计算各层参数计算

1. 根据x和$\Theta^{(1)}$ 计算$a^{(2)}$ 的值：

   $a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3 + \Theta_{14}^{(1)}x_4)$

   $a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3 + \Theta_{24}^{(1)}x_4)$

   $a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3 + \Theta_{34}^{(1)}x_4)$

   g仍然代表sigmoid函数：$g(z) = \frac{1}{1+e^{-z}}$

   采用矩阵运算可表达为：$a^{(2)} = sigmoid(\Theta^{(1)} * a^{(1)})$

   - $a_i^{(j)}$表示第j层的第i个元素值。其中$a_i^{(1)}$ 代表输入的Features(X)中的第i个元素。$a^{(1)}$或X是一个具有N个元素的列向量
   - Θ表示权重矩阵(Matrix of Weights)。$\Theta^{(j)}$ 表示从第j层到第$j+1$层的权重矩阵。
   - $\Theta_{ik}^{(j)}$ 表示从第j层到第$j+1$层的权重矩阵中的第i行第k列权重值。i实际上对应着$j+1$层中的第i个元素；而k对应着第j层的第k个元素。
   - 第一个计算式$a_1^{(2)}$，可参见见上图中的红色箭头线，这5条线依次代表从第1层的第0、1、2、3、4号元素分别映射到第2层的第1号元素。第一条红线代表的权重矩阵元素就是：$\Theta_{10}^{(1)}$，第二条红线代表的是：$\Theta_{11}^{(1)}$，依次类推。

2. 根据$a^{(2)}$和$\Theta^{(1)}$计算$a^{(3)}$ 的值：

   $a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$

   $a_2^{(3)} = g(\Theta_{20}^{(2)}a_0^{(2)} + \Theta_{21}^{(2)}a_1^{(2)} + \Theta_{22}^{(2)}a_2^{(2)} + \Theta_{23}^{(2)}a_3^{(2)})$

其中，a_0为Bias Unit，一般赋值为1。

采用矩阵运算可表达为：$a^{(3)} = sigmoid(\Theta^{(2)} * a^{(2)})$

3. 根据$a^{(3)}$ 和$\Theta^{(2)}$ 计算$a^{(4)}$ 的值：

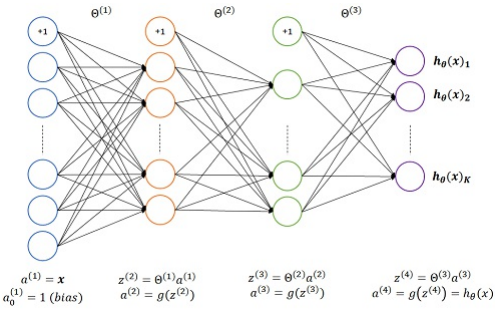
   $a_1^{(4)} = g(\Theta_{10}^{(3)} a_0^{(3)} + \Theta_{11}^{(3)} a_1^{(3)} + \Theta_{12}^{(3)} a_2^{(3)})$

   采用矩阵运算可表达为：$a^{(4)} = sigmoid(\Theta^{(3)} * a^{(3)})$

   最终可以计算出预测的Hypothesis值：$h_\theta(x) = a_1^{(4)}$
   上述结果就是针对x预测其属于其对应的y值的几率。

4. 通过向量/矩阵运算的维度来验证计算是否正确：
   - 如果第j层有M个有效元素(不含Unit Bias)，第$j+1$层有K个元素。则从第j层到第$j+1$层的矩阵$\Theta^{(j)}$ 大小为：$K \times (M+1)$
   - 上述过程可以看到，只要计算出Θ，就能根据输入的x预测出其对应的y值。
   - 请注意，上述计算仅仅还只是针对一组训练数据。事实上，针对每一组训练数据，都需要执行上述过程以便累积出该轮的Cost，然后再采用Back Propagation方法进行梯度递减计算

多个分类输出的神经网络



本模型中，$a^{(4)}$ 包括K个可能的类别输出。 例如，如果$K = 4$，则y的值可能为下列4个向量之一：

$$\begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\1\end{pmatrix}$$

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} [(-y_k^{(i)} log(h_\theta(x^{(i)})_k) - (1 - y_k^{(i)}) log(1 - h_\theta(x^{(i)})_k)] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

- K：$h\theta(x)$的可能结果数目（分类数)，也就是Output Layer的节点个数
- L：神经网络的层数（包括Input layer, Hidden layer, Output layer)

- s_l：第l层节点个数 (不包括bias unit)
- 在Regularization项中，不应包含Bias Unit项参数（也就是说，$\Theta_{j0}^{(l)}$ 不出现在Penalty项中）

使用**sklearn.neural_network.MLPClassifier**类实现手写数字图片识别

- 该类实现了前馈计算并且通过Back Propagation算法计算梯度
- 几个重要属性
  - alpha：代表Penalty项中的λ
  - hidden_layer_sizes：分别指定hidden layer中每一层的节点数(不包括Bias Unit)
  - activation：指定激活函数得类型
- 很多情况下，神经网络的计算需要花费较多时间，因此在训练完成后，需要保存模型数据
  - joblib.dump方法用于保存模型参数
  - joblib.load方法用于从文件种装载模型参数构造一个MLPClassifier对象
- score方法用于估算正确率
- 本节代码参考【13/01sklearn_mlp_classifier_digits.py】

第2节：反向传播算法

单组训练数据，无Regulization项的反向传播梯度推导

- 前向计算流
  考虑一组训练数据(x,y)，并且暂不考虑Regulization项，则成本函数如下：

  $J(\theta)=\sum_{k=1}^{K}[-y_k log(h_\theta(x)_k)-(1-y_k)log(1-h_\theta(x)_k)]$

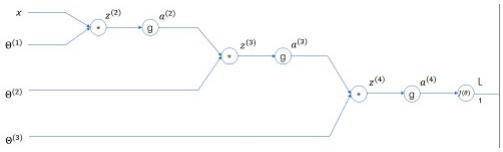
  $=\sum_{k=1}^{K}[-y_k log(a_k^{(4)})-(1-y_k)log(1-a_k^{(4)})]$

  假定x维度为：$(D,1)$，y维度为：$(K,1)$，也就是说，把它们视为列向量。则：正向计算时，各节点、权重的维度如下表格：

项	维度	项	维度	项	维度	项	维度
		$a^{(1)}$	(D+1, 1)	$\Theta^{(1)}$	(H1, D+1)	$\frac{\partial L}{\partial \Theta^{(1)}}$	(H1, D+1)
$z^{(2)}$	(H1, 1)	$a^{(2)}$	(H1+1, 1)	$\Theta^{(2)}$	(H2, H1+1)	$\frac{\partial L}{\partial \Theta^{(2)}}$	(H2, H1+1)
$z^{(3)}$	(H2, 1)	$a^{(3)}$	(H2+1, 1)	$\Theta^{(3)}$	(K, H2+1)	$\frac{\partial L}{\partial \Theta^{(3)}}$	(K, H2+1)
$z^{(4)}$	(K, 1)	$a^{(4)}$	(K, 1)				

  上表中，假设权重矩阵中与Bias项对应的参数放在第一列

  前向计算流程图如下图所示：



- 反向传播
  - 计算$\frac{\partial L}{\partial a^{(4)}}$，维度：$(K,1)$

    $\frac{\partial L}{\partial a^{(4)}}=\sum_{k=1}^{K}\frac{-y_k log(a_k^{(4)})-(1-y_k)log(1-a_k^{(4)})}{\partial a_i^{(4)}}$

    $=\frac{\partial[-y_i log(a_i^{(4)})-(1-y_i)log(1-a_i^{(4)})]}{\partial a_i^{(4)}}$

    $=-y_i*\frac{1}{a_i^{(4)}}-(1-y_i)*\frac{1}{1-a_i^{(4)}}*(-1)$

    $=\frac{-y_i*(1-a_i^{(4)})+(1-y_i)*a_i^{(4)}}{a_i^{(4)}*(1-a_i^{(4)})}$

    $=\frac{a_i^{(4)}-y_i}{a_i^{(4)}*(1-a_i^{(4)})}$

    上式中，用到了下列事实：

- 只有当$i=k$时，才会对$\frac{\partial L}{\partial a_i^{(4)}}$ 有贡献，所以上述求和公式就可以直接化简
- 对数函数的导数：$f(x)=log(x)$，则：$f'(x)=1/x$

向量形式：

$\frac{\partial L}{a^{(4)}}=\frac{a^{(4)}-y}{a^{(4)}*(1-a^{(4)})}$

上式中，分母部分，$a^{(4)}*(1-a^{(4)})$是两个向量的对应元素分别相乘，并且仍然保留向量形式，然后再与分子部分进行对应元素相除。

◦ 计算$\frac{\partial L}{\partial z^{(4)}}$ 维度：$(K,1)$

考虑前向计算公式：$a_i^{(4)}=g(z_i^{(4)})$

$\frac{\partial L}{\partial z_i^{(4)}}=\frac{\partial L}{\partial a_i^{(4)}}*\frac{\partial a_i^{(4)}}{\partial z_i^{(4)}}=\frac{\partial L}{\partial a_i^{(4)}}*\frac{g(z_i^{(4)})}{\partial z_i^{(4)}}=\frac{a_i^{(4)}-y_i}{a_i^{(4)}*(1-a_i^{(4)})}*g(z_i^{(4)})*(1-g(z_i^{(4)}))=a_i^{(4)}-y_i$

上式中，用到了下列事实：

Sigmoid函数的导数：$g(x)=\frac{1}{1+e^{-z}}$，则：$g'(x)=g(x)*(1-g(x))$

向量形式：

$\delta^{(4)}=\frac{\partial L}{\partial z^{(4)}}=a^{(4)}-y$

◦ 计算$\frac{\partial L}{\partial \Theta^{(3)}}$，$\frac{\partial L}{\partial a^{(3)}}$ 和$\frac{\partial L}{\partial z^{(3)}}$ 维度分别为：$(K,H2+1)$，$(H2+1,1)$，$(H2+1,1)$

考虑前向计算公式：$z^{(4)}=\Theta^{(3)}a^{(3)}$

考虑矩阵链式求导法则：若$Y=WX$，则$\frac{\partial L}{\partial W}=\frac{\partial L}{\partial Y}X^T$，$\frac{\partial L}{\partial X}=W^T\frac{\partial L}{\partial Y}$

可知：

$\frac{\partial L}{\partial \Theta^{(3)}}=\frac{\partial L}{\partial z^{(4)}}(a^{(3)})^T=\delta^{(4)}(a^{(3)})^T$

$\frac{\partial L}{\partial a^{(3)}}=(\Theta^{(3)})^T\frac{\partial L}{\partial z^{(4)}}=(\Theta^{(3)})^T\delta^{(4)}$

$\delta_{(3)}=\frac{\partial L}{\partial z^{(3)}}=\frac{\partial L}{\partial a^{(3)}}*\frac{\partial a^{(3)}}{\partial z^{(3)}}=(\Theta^{(3)})^T\delta^{(4)}*g(z^{(3)})*(1-g(z^{(3)}))$

在计算$\delta^{(3)}$ 时，一般要去掉$\Theta^{(3)}$ 中与Bias项对应的权重项，例如，$\Theta_{i0}^{(3)}$

◦ 计算$\frac{\partial L}{\partial \Theta^{(2)}}$，$\frac{\partial L}{\partial a^{(2)}}$ 和$\frac{\partial L}{\partial z^{(2)}}$ 维度分别为：$(H2,H1)$，$(H1,1)$，$(H1,1)$

回顾前向计算公式：$z^{(3)}=\Theta^{(2)}a^{(2)}$

可知：

$\frac{\partial L}{\partial \Theta^{(2)}} = \frac{\partial L}{\partial z^{(3)}}(a^{(2)})^T = \delta^{(3)}(a^{(2)})^T$

$\frac{\partial L}{\partial a^{(2)}} = (\Theta^{(2)})^T \frac{\partial L}{\partial z^{(3)}} = (\Theta^{(2)})^T \delta^{(3)}$

$\delta_{(2)} = \frac{\partial L}{\partial z^{(2)}} = \frac{\partial L}{\partial a^{(2)}} * \frac{\partial a^{(2)}}{\partial z^{(2)}} = (\Theta(2))^T \delta^{(3)} * g(z^{(2)}) * (1 - g(z^{(2)}))$

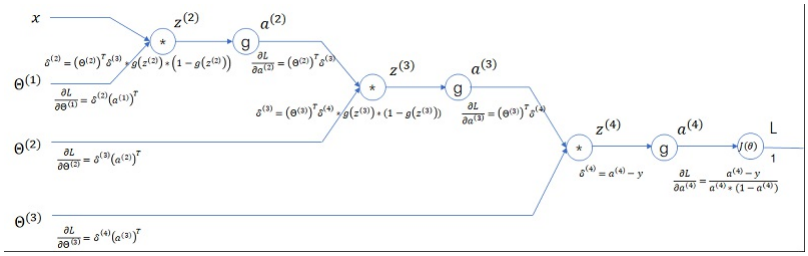
在计算$\delta^{(2)}$ 时，一般要去掉$\Theta^{(2)}$ 中与Bias项对应的权重项，例如，$\Theta_{i0}^{(2)}$

- 计算$\frac{\partial L}{\partial \Theta^{(1)}}$

  回顾正向计算公式：$z^{(1)} = \Theta^{(1)} a^{(1)}$

  $\frac{\partial L}{\partial \Theta^{(1)}} = \frac{\partial L}{\partial z^{(2)}}(a^{(1)})^T = \delta^{(2)}(a^{(1)})^T$

- 反向传播导数流图：



多组样本

在N组样本数据的情况下，所有的梯度应该进行累积求和，然后取平均值
使用Python实现时，各节点、权重的维度如下表格：

项	维度	项	维度	项	维度	项	维度
		X	(N, D)	y	(N, K)		
		$a^{(1)}$	(N, D+1)	$\Theta^{(1)}$	(H1, D+1)	$\frac{\partial L}{\partial \Theta^{(1)}}$	(H1, D+1)
$z^{(2)}$	(N, H1)	$a^{(2)}$	(N, H1+1)	$\Theta^{(2)}$	(H2, H1+1)	$\frac{\partial L}{\partial \Theta^{(2)}}$	(H2, H1+1)
$z^{(3)}$	(N, H2)	$a^{(3)}$	(N, H2+1)	$\Theta^{(3)}$	(K, H2+1)	$\frac{\partial L}{\partial \Theta^{(3)}}$	(K, H2+1)
$z^{(4)}$	(N, K)	$a^{(4)}$	(N, K)				

- 采用矩阵形式的前向计算

  $z^{(2)} = a^{(1)}(\Theta^{(1)})^T$

  $a^{(2)} = [1, g(z^{(2)})]$ 注意：在$g(z^{(2)})$矩阵中，插入Bias项列

  $z^{(3)} = a^{(2)}(\Theta^{(2)})^T$

  $a^{(3)} = [1, g(z^{(3)})]$ 注意：在$g(z^{(2)})$矩阵中，插入Bias项列

$z^{(4)} = a^{(3)}(\Theta^{(3)})^T$

$a^{(4)} = g(z^{(3)})$

- 采用矩阵形式的反向传播梯度计算

$\delta^{(4)} = a^{(4)} - y$

$\delta^{(3)} = \delta^{(4)}\Theta^{(3)} * g(z^{(3)}) * (1 - g(z^{(3)})$ 注意：$\Theta^{(3)}$中，Bias列$\Theta_{i0}^{(3)}$不参与$\delta^{(3)}$ 的计算。另外, $*$表示对应元素分别相乘(而不是矩阵运算)

$\delta^{(2)} = \delta^{(3)}\Theta^{(2)} * g(z^{(2)}) * (1 - g(z^{(2)}))$ 注意：$\Theta^{(2)}$中，Bias列$\Theta_{i0}^{(2)}$不参与$\delta^{(2)}$ 的计算。另外,$*$表示对应元素分别相乘(而不是矩阵运算)

$\frac{\partial L}{\partial \Theta^{(3)}} = \frac{(\delta^{(4)})^T a^{(3)}}{N}$

$\frac{\partial L}{\partial \Theta^{(2)}} = \frac{(\delta^{(3)})^T a^{(2)}}{N}$

$\frac{\partial L}{\partial \Theta^{(1)}} = \frac{(\delta^{(2)})^T a^{(1)}}{N}$

考虑**Regulization**项

$\frac{\partial L}{\partial \Theta^{(3)}} = \frac{(\delta^{(4)})^T a^{(3)}}{N} + \frac{\lambda * \Theta^{(3)}}{N}$

$\frac{\partial L}{\partial \Theta^{(2)}} = \frac{(\delta^{(3)})^T a^{(2)}}{N} + \frac{\lambda * \Theta^{(2)}}{N}$

$\frac{\partial L}{\partial \Theta^{(1)}} = \frac{(\delta^{(2)})^T a^{(1)}}{N} + \frac{\lambda * \Theta^{(1)}}{N}$

注意，上述所有计算的Regulization项，都不计入与Bias对应的权重项，例如$\Theta_{i0}^{(3)}$,$\Theta_{i0}^{(2)}$ 和$\Theta_{i0}^{(1)}$

Θ的随机初始化：

- 不能将Θ初始化为0或1，或者相同的数。否则将会导致计算出来的$a^{(l)}$ 中的每个节点都相同，从而使原本的n个Feature变成相当于只有1个Feature。
- 采用Symetric Breaking（random initialization) 方法可以解决上述问题。例如：

$\varepsilon_i\_init = 0.12$

$\Theta^{(3)} = rand(s_2, s_1 + 1) * 2 * \varepsilon_i\_init - \varepsilon_i\_init$

$\Theta^{(2)} = rand(s_3, s_2 + 1) * 2 * \varepsilon_i\_init - \varepsilon_i\_init$

$\Theta^{(1)} = rand(s_{l+1}, s_l + 1) * 2 * \varepsilon_i\_init - \varepsilon_i\_init$

其中，rand函数用于生成0~1中的随机数。s_i是网络第i层的节点个数(不包括Bias Unit)；

$s_i + 1$正好包含Bias Unit

Gradient Checking

- Back Propagation算法非常容易出错(程序设计上的错误)，有时甚至每次迭代$J(\Theta)$在不断减小，但仍然错误。因此需要有某种方式检查该算法编写是否正确
- Gradient Checking是通过数值方式模拟计算$J(\Theta)$在某个Θ处的导数：

  $f(\Theta_{ij}^{(l)}) = \frac{J(\Theta_{ij}^{(l)} + \varepsilon) - J(\Theta_{ij}^{(l)} - \varepsilon)}{2\varepsilon}$

  一般设置$\varepsilon = 0.0001$ 或更小

  可以随机初始化一组$\Theta_{ij}^{(l)}$数据，应包括每层、每行、每列的Θ参数（一组即可）

- 对比通过Back Propagation计算出来的某个导数$\frac{\partial}{\Theta_{ij}^{(l)}} J(\Theta)$与上述Gradient Checking计算出来的

  $f(\Theta_{ij}^{(l)})$进行对比，它们之间的差值应至少小于ε。这样才能说明Back Propagation的代码正确。

示例**1**：使用**Back Propagation**算法自行实现神经网络

- 使用了2个隐藏层，共需要3个Θ矩阵参数
- 定义了cost和gradient方法，提供给scipy.optimize.minimize来优化计算Θ
- 为了便于传递参数，将3个Θ矩阵展平成1个一维数组传递，然后在cost和gradient方法中还原成3个Θ矩阵
- 本案例限制了minimize方法的最大循环次数为1000次，计算的cost值较小，但是数值上并未收敛
- 调整λ(lmd)的值，观察其收敛性。本例中，λ越小，收敛越快，但是容易出现overfit
- 本节代码参考【13/02back_propagation_demo.py】

第三部分：神经网络与深度学习

本部分介绍多层神经网络和卷积神经网络，主要包括：

- 使用简单的模型进行图像分类
- 构建多层神经网络进行图像分类
- 构建卷积神经网络进行图像分离
- 使用Tensorflow实现前述模型并进行训练和推理

第十四章：图像分类基础

第1节：CIFAR10数据集

CIFAR10图片分类数据集

- 官方介绍和下载地址：http://www.cs.toronto.edu/~kriz/cifar.html
- 50000张训练图片，10000张测试图片。10个图像分类 airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck
- 每张图片32x32像素，每个像素包含RGB3个颜色通道 每个颜色通道用一个0~255的整数表示
- cifar-10数据目录
    - data_batch_x：共5个训练数据集，每个集中包含10000张图片数据
    - test_batch：测试数据集，10000张图片
    - 数据集文件中，每张图片都包含图像数据部分和标签部分
    - 图像数据部分有3072个元素，按照C(通道)xH(高)xW(宽）结构存放像素值
    - 标签部分有10个元素，其中第K个元素的值为1，其余值为0。K是该图片对应的分类索引编号(0~9)

示例**1**：从**cifar-10**数据目录中读取数据到**python**矩阵中

- 【14/dataset】下存放了cifar-10数据
- 代码参考【14/01demo_data.py】和【14/util_data.py】

第2节：KNN分类器

使用KNN对CIFAR10数据集进行分类

- 算法说明
  - 数组子采样
    - 50000个样本的KNN分类速度非常慢，故取其中5000个子样本作为训练集
    - 从验证样本中取500个作为验证集
    - 从测试样本中取500个作为测试集
  - 距离策略
    - 采用L2形式，即：对应元素值之差的平方和
  - 批量计算测试样本与训练样本之间的距离
    - 方法1：采用for循环，逐个测试样本分别计算，可以观察到速度非常慢
    - 方法2：采用矩阵操作，同时进行多个测试样本的计算，速度较快
  - 使用常规的验证方法优化超参数(Hyper-Parameter)k：
    - 变换不同的k值，始终采用TrainSet来训练，ValSet来验证
    - 选取验证正确率最高的k值，用TestSet测试最佳k值时的正确率
  - 使用Cross Validation优化超参数k
    - 将训练集分成5等份，每次取4个作为TrainSet, 1个作为ValSet。共5种组合
    - 分别设定k=1,3,5,...，针对每个k值，用上述5种组合训练和验证，记录5个Val正确率
    - 每个k值产生5个Val正确率，取平均值最高的那个作为最佳k值。用TestSet测试最佳k值时的正确率
  - KNN计算结果
    - 无论选取何种k值，测试的准确率仅仅在27%左右。说明这个算法是不合适的。

示例1：使用KNN进行图像分类，并求取最优的k值

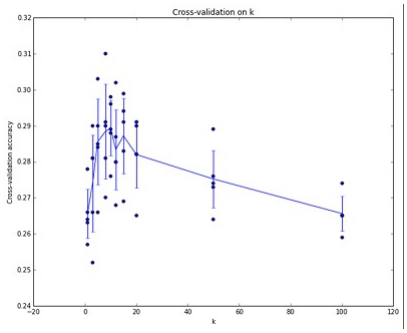
- 本实验代码位于【13/02demo_knn.py】和【13/k_nearest_neighbor.py】
  - 分析KNN实现代码并进行初步训练及分类

    对应03demo_knn.py中的代码段：【1】KNN训练
  - 使用常规的验证方法优化超参数k

    对应03demo_knn.py中的代码段：【2】常规验证
  - 使用cross validation优化超参数k

    对应03demo_knn.py中的代码段：【3】交叉验证

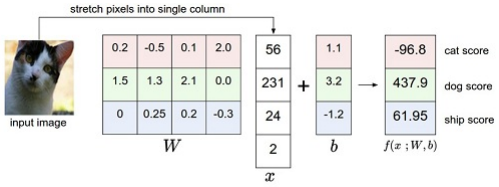
验证结果，可观察到下列k-accuracy的关系图(k=10左右达到最高的正确率)：



第3节：SVM线性分类器

基本概念

SVM线性分类器，是指成本函数使用多分类SVM损失函数。对一张图片的特征数据(D维)执行一个仿射变换(即：线性变换+偏移)，得到一个输出向量(C维，代表C个分类)，向量中每个元素代表着其对应的类别的得分；取得分最高者作为预测结果



判别式

- 一般形式：$f(x_i, W, b) = Wx_i + b$

  其中，x_i是D维列向量， W是(C, D)矩阵， b是C维列向量

- 矩阵形式：$f(x_i, W) = Wx_i$

  此时，x_i已经包含了Bias项， W也已经添加一个了Bias权重列。计算的便捷性上考虑，应尽可能采用矩阵形式。本节下面的内容都采用矩阵形式

成本函数

- 单个训练样本

  设某个样本为x_i，其对应的实际分类结果为y_i，其中，y_i是$0\ C-1$中的某个分类编号值

  定义判别结果向量中，分类编号为j的得分为：$s_{ij} = f(x_i, W)_j = w_j x_i$，则该样本的成本定义为：

  $$L_i = \sum_{j \neq y_i} \begin{cases} 0, & s_{iy_i} \geq s_{ij} + \Delta \\ s_{ij} - s_{iy_i} + \Delta, & \text{其它情况} \end{cases} = \sum_{j \neq y_i} max(0, s_{ij} - s_{iy_i} + \Delta)$$

  - w_j代表W中的第j行(其实就是表示编号为 j 的分类的权重系数)
  - s_{iy_i} 表示根据判别式计算出来的样本x_i的所有分类编号得分中，其实际分类编号所对应的得分

- $s_{iy_i} \geq s_{ij} + \Delta$：实际分类编号对应的得分，比其它分类编号对应的得分要至少多Δ分。Δ可以看成SVM中的分类边界距离。
- 一般情况下，取SVM的边界距离Δ为1，但有些情况下，也可以取大于1的值。边界距离Δ保证了：实际分类编号的得分，不仅要超过其它分类得分，而且还需要超出一定的范围。
- 如果满足$s_{iy_i} \geq s_ij + \Delta$，则我们认为这个计算结果是令人满意的，因此不必计入到样本x_i的成本结果中；否则，则认为这个计算结果有误差，需要把误差$s_{ij} - s_{iy_i} + \Delta$计入到样本$x_i$的成本结果中
- 需要将s_{iy_i}与其它每一个s_{ij}分别比较，并将误差累计到成本结果中；但是s_{iy_i}无需与自身进行比较，也就是说，无需累计$j = y_i$时的误差项
- 这种成本函数的定义方式称为Hinge loss

- 所有训练样本

  $L = \frac{1}{N}\sum_i L_i + \lambda R(W) = \frac{1}{N}\sum_i \sum_{jney_i}[max(0, f(x_i, W)_j - f(x_i, W)_{y_i} + \Delta)] + \lambda \sum_k \sum_l W_{kl}^2$

  式中，Δ为SVM边界距离，一般设置为1

梯度计算

定义操作符号：

$$D(expression) = \begin{cases} 1, & expression\ is\ true \\ 0, & expression\ is\ false) \end{cases}$$

- 单个训练样本
  根据单个样本的成本函数：

  $L_i = \sum_{j \neq y_i} max(0, s_{ij} - s_{y_i} + \Delta) = \sum_{j \neq y_i} max(0, w_j x_i - w_{y_i} x_i + \Delta)$

  - w_j 代表 W中的第 j 行元素(其实就是表示编号为 j 的分类的权重系数)，$w_j x_i$ 是这两个向量的内积和(也就是编号为 j 的分类的得分）
  - 当$j \neq y_i$ 时

    $\frac{\partial L_i}{\partial w_{jk}} =$
    $[max(0, w_1 x_i - w_{y_i} x_i + \Delta) + \cdots + max(0, w_{y_i - 1} x_i - w_{y_i} x_i + \Delta) +$
    $max(0, w_{y_i + 1} x_i - w_{y_i} x_i + \Delta) + \cdots + max(0, w_c x_i - w_{y_i} x_i + \Delta)]$
    $/ \partial w_{jk}$

    上式中，只有$max(0, w_j x_i - w_{y_i} x_i + \Delta)$与$w_j$有关(进而与$w_{jk}$有关)，其余项均不参与对$w_j$的偏导计算，因此：

$\frac{\partial L_i}{\partial w_{jk}} = \frac{max(0, w_j x_i - w_{y_i} x_i + \Delta)}{\partial w_{jk}}$

进一步的：

- 如果$w_j x_i - w_{y_i} x_i + \Delta \leq 0$，则$\frac{\partial L_i}{\partial w_{jk}} = 0$
- 如果$w_j x_i - w_{y_i} x_i + \Delta > 0$, 则

  $\frac{\partial L_i}{\partial w_{jk}} = \frac{\partial (w_j x_i - w_{y_i} x_i + \Delta)}{\partial w_{jk}} = \frac{\partial w_j x_i}{\partial w_{jk}} = \frac{\partial (w_{j1} x_{i1} + w_{j2} x_{i2} + \cdots + w_{jD} x_{iD})}{\partial w_{jk}} = x_k$

  上式中，(w_j和x_i都具有D个元素，x_{ik}代表x_i中第k个元素)

综合上面两种情况，可得：

$\frac{\partial L_i}{\partial w_{jk}} = D(w_j x_i - w_{y_i} x_i + \Delta > 0) x_{ik}$

故而：

$\frac{\partial L_i}{\partial w_j} =$
$[D(w_j x_i - w_{y_i} x_i + \Delta > 0) x_{i1}, D(w_j x_i - w_{y_i} x_i + \Delta > 0) x_{12}, \cdots,$
$D(w_j x_i - w_{y_i} x_i + \Delta > 0) x_{iD}] =$
$D(w_j x_i - w_{y_i} x_i + \Delta > 0)(x_i)^T$

- 当$j = y_i$ 时

$\frac{\partial L_i}{\partial w_{y_i k}}$
$= [max(0, w_1 x_i - w_{y_i} x_i + \Delta) + \cdots + max(0, w_{y_i - 1} x_i - w_{y_i} x_i + \Delta)$
$+ max(0, w_{y_i + 1} x_i - w_{y_i} x_i + \Delta) + \cdots + max(0, w_c x_i - w_{y_i} x_i + \Delta)]$
$/ \partial w_{jk}$
$= \frac{max(0, w_1 x_i - w_{y_i} x_i + \Delta)}{\partial w_{jk}} + \cdots + \frac{max(0, w_{y_i - 1} x_i - w_{y_i} x_i + \Delta)}{\partial w_{jk}} +$
$\frac{max(0, w_{y_i + 1} x_i - w_{y_i} x_i + \Delta)}{\partial w_{jk}} + \cdots + \frac{max(0, w_c x_i - w_{y_i} x_i + \Delta)}{\partial w_{jk}}$
$= D(w_1 x_i - w_{y_i} x_i + \Delta)(-x_{ik}) + \cdots + D(w_{y_i - 1} x_i - w_{y_i} x_i + \Delta)(-x_{ik}) +$
$D(w_{y_i + 1} x_i - w_i x_i + \Delta)(-x_{ik}) + \cdots + D(w_c x_i - w_{y_i} x_i + \Delta)(-x_{ik})$

即：$\frac{\partial L_i}{\partial w_{y_i k}} = (\sum_{j \neq y_i} D(w_j x_i - w_{y_i} x_i + \Delta > 0)) x_{ik}$

进而得出：

$\frac{\partial L_i}{\partial w_{y_i}} = -(\sum_{j \neq y_i} D(w_j x_i - w_{y_i} x_i + \Delta \ > 0))(x_i)^T$

- 上述两个梯度计算式子可以这样理解：
  - 一次性计算W中一行元素(每行有D个元素)的梯度
  - 对于W中与当前样本x实际所属类别不同的那些列($j \neq y_i$)，只要

    $w_j x_i - w_{y_i} x_i + \Delta > 0$，其梯度就是$(x_i)^T$；否则梯度为0

        - 对于W中与当前样本x实际所属类别相同的那一列，则需要先判断该列得分是否大于其它列得分(含边界Δ)。如果大于，则记为1；否则记为0。然后累计1的个数。
- 所有训练样本的梯度

$$\frac{\partial L}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L_i}{\partial w_j} + \lambda w_j$$

---

mini-batch随机梯度下降算法

- 在通过多次迭代执行批量梯度下降算法时，如果每次迭代选择所有的样本来计算成本函数值以及梯度，计算开销会比较大，在某些情况下(例如使用sigmoid激活函数)甚至会出现计算溢出的情况。
- mini-batch随机梯度下降法指的是：每次随机选取m个样本，按照批量梯度下降算法进行计算。这是一种常见的计算方式

示例**1**：使用**SVM**线性分类器进行图像分类

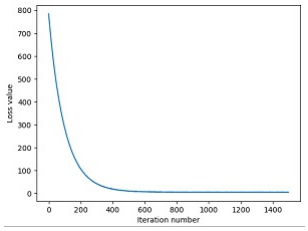
本实验源代码位于【14/03demo_svm.py】【14/util_svm.py】【14/linear_classifier.py】【14/gradient_check.py】

1. 关于特征的正则化(Normalizing)
    - 本例采用均值化方法对特征值进行预处理，计算训练样本各列均值，然后Train/Val/Test中各元素分别减去对应列均值
    - 均值化处理后，能够使输入的特征数据值以0为中心左右分布，一般来说具有更好的计算收敛性
2. 矩阵形式的成本函数和梯度计算对比
    - 对应03demo_svm.py中的代码段：【1】计算耗时对比
    - 矩阵运算速度会比使用循环计算快很多，应尽可能采用
    - 在矩阵形式下，X维度为(N, D)，W的维度为(D, C)，从而可通过XW一次性计算处所有样本的的得分值
    - 请结合前述算法说明及代码注释理解如何实现成本函数和梯度矩阵的计算：util_svm.py中函数：svm_loss_naive和svm_loss_vectorized
3. 梯度检验
    - 对应03demo_svm.py中的代码段：【2】梯度检验
    - util_svm.py中，使用分析方法(就是使用前述梯度计算公式)计算出来的梯度(导数)，比较容易出错，所以一般需要使用数值方法进行检验。
    - 数值方法求梯度或导数，一般采用：$\nabla f(x) = \frac{f(x+\Delta h) - f(x-\Delta h)}{2\Delta h}$，其中$\Delta h$是一个非常小的步长值(例如0.00001）
    - 数值方法计算梯度，虽然比较精确，但是计算开销很大，因此无法作为正常计算中的梯度计算方案；而是用于检验使用分析方法求出的梯度的是否正确。同样因为开销问题，无法对大量数据点进行检验，只能随机取出若干个点进行检验。

- 本实验使用了斯坦福大学CS231N课程中的数值梯度计算及检验的代码文件gradient_check.py

4. 执行SVM分类，并查看计算过程中成本值的收敛情况

   - 对应03demo_svm.py中的代码段：【3】训练并查看成本变化过程
   - 可以观察到成本值随迭代进行而变化的曲线图：



   这种情况表明，计算得到了很好的收敛

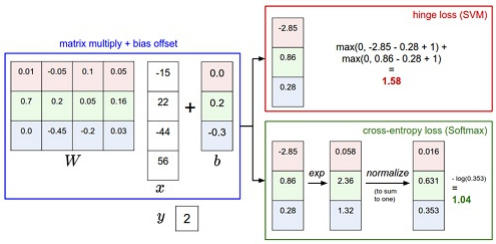
5. 优化超参数：learning rate以及regularization strength

   - 对应03demo_svm.py中的代码段：【4】选取hyper-parameters
   - 给定一批learning rate和regularization strength的候选值，并且交叉组合；然后针对所有训练数据训练，针对验证数据来遴选最优的组合

第4节：Softmax线性分类器

基本概念

- Softmax线性分类器，是指使用使用跨熵(cross entropy)作为成本函数
- 先对数据做SVM线性变换，对变换后得到的结果向量：
  - 向量中每个元素取e的指数
  - 每个元素值除以元素和(归一化)，得到每个分类的分值(可视为判别为该分类的概率)



判别式

- 一般形式：$f(x_i, W, b) = Wx_i + b$

  其中，x_i是D维列向量， W是(C, D)矩阵， b是C维列向量

- 矩阵形式：$f(x_i, W) = Wx_i$

  此时，x_i已经包含了Bias项， W也已经添加一个了Bias权重列。计算的便捷性上考虑，应尽可能采用矩阵形式。本节下面的内容都采用矩阵形式

成本函数：

- 单个训练样本的成本定义

  设样本为x_i，其对应的实际分类为y_i(0 ~ C-1中的某个分类编号值)，f_{ij} 代表线性变换后向量中第j个元素值，也就是分类编号为j的得分值

  该样本的成本定义为：

  $$L_i = \frac{-log(e^{f_{iy_i}})}{\sum_j e^{f_{ij}}} = -f_{iy_i} + log\sum_j e^{f_{ij}}$$

  - f_{iy_i} 表示根据判别式计算出来的样本x_i的所有分类编号得分中，其实际分类编号所对应的

得分

- $\sum_j e^{f_{ij}}$ 表示，将线性变换得分向量中所有元素取指数后求和
- L_i实际上就是从归一化以后的结果向量中取出实际分类对应的那个元素，然后取$-log$操作

- 所有训练样本的成本定义

$L = \frac{1}{N}\sum_i L_i + \lambda R(W)$

梯度计算

- 单个样本的梯度
  - 当$j \neq y_i$时

    由$f_{iy_i} = w_{y_i}x_i$ 可知，它与w_j无关(从而与所有的w_{jk} 无关)，因此：

    $\frac{\partial L_i}{\partial w_{jk}} = 0 + log\sum_j e^{f_{ij}} = log(e^{f_{11}} + e^{f_{12}} + \cdots + e^{f_{1}D})$

    由$(logx)' = 1/x$；$(e^x)' = e^x$，推算：

    $\frac{\partial L_i}{\partial w_{jk}} = \frac{1}{e^{f_{11}} + e^{f_{12}} + \cdots + e^{f_{1D}}} * (\frac{\partial e^{w_1 x_i}}{\partial w_{jk}} + \frac{\partial e^{w_2 x_i}}{\partial w_{jk}} + \cdots + \frac{\partial e^{w_D x_i}}{\partial w_{jk}}) = \frac{1}{\sum_j e^{f_{ij}}} * \sum_d^D (e^{w_d x_i} * \frac{\partial w_d x_i}{\partial w_{jk}})$

    显然，当$d \neq j$，则$w_d x_i$ 与∂w_{jk} 无关，此时$\frac{\partial w_d x_i}{\partial w_{jk}} = 0$，推算：

    $\sum_i^D (e^{w_d x_i} * \frac{\partial w_d x_i}{\partial w_{jk}}) = e^{w_j x_i} * \frac{\partial w_j x_i}{\partial w_{jk}}$

    由$w_j x_i = w_{j1}x_{i1} + w_{j2}x_{i2} + \cdots + w_{jk}x_{ik} + \cdots + w_{jD}x_{iD}$，可知：$\frac{\partial w_j x_i}{\partial w_{jk}} = x_{ik}$

    因此：

    $\frac{\partial L_i}{\partial w_{jk}} = \frac{1}{\sum_j e^{f_{ij}}} * \sum_d^D (e^{w_i x_i} * \frac{\partial w_i x_i}{\partial w_{jk}}) = \frac{1}{\sum_j e^{f_{ij}}} * e^{w_j x_i} * \frac{\partial w_j x_i}{\partial w_{jk}} = \frac{e^{f_{ij}}}{\sum_j e^{f_{ij}}} * x_{ik}$

    进而：

    $\frac{\partial L_i}{\partial w_j} = \frac{e^{f_{ij}}}{\sum_j e^{f_{ij}}} (x_i)^T$

  - 当$j = y_i$ 时

    $\frac{\partial L_i}{\partial w_{y_i}} = \frac{\partial(-f_{iy_i} + log\sum_j e^{f_{ij}})}{\partial w_{y_i}} = -\frac{\partial w_{y_i} x_i}{\partial w_{y_i}} + \frac{e^{f_{iy_j}}}{\sum_j e^{f_{ij}}} x_i^T = -x_i^T + \frac{e^{f_{iy_j}}}{\sum_j e^{f_{ij}}} x_i^T$

- 所有训练样本的梯度

$\frac{\partial L}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L_i}{\partial w_j} + \lambda w_j$

- 改善数值计算稳定性

  在计算$\frac{e^{f_{iy_j}}}{\sum_j e^{f_{ij}}}$的过程中，如果$f_{ij}$的值稍大，则会造成$e^{f_{ij}}$溢出。为此，采用下列方法进行变形计算：

  $$\frac{e^{f_{iy_j}}}{\sum_j e^{f_{ij}}} = \frac{C*e^{f_{iy_j}}}{C*\sum_j e^{f_{ij}}} = \frac{e^{f_{iy_j}+logC}}{\sum_j e^{f_{ij}+logC}}$$

  上式中，一般取$logC = -max f_{ij}$，也就是取类别分类分值中最大的一个并取负。这样，

  $f_{iy_j} + logC$将小于0，从而保证$e^{f_{iy_j}} + logC$不会溢出，同时计算结果与原式相同

示例**1**：使用**Softmax**线性分类器进行图像分类

本实验源代码位于【14/04demo_softmax.py】【14/util_softmax.py】【14/linear_classifier.py】【14/gradient_check.py】

1. 成本函数和梯度计算
   - util_softmax.py中函数：softmax_loss_naive通过循环的方式来计算成本函数值和梯度；函数softmax_loss_vectorized通过矩阵方式计算。
   - 请结合前述算法说明及代码注释理解如何实现成本函数和梯度矩阵的计算
2. 成本值和梯度的初步检验
   - 对应04demo_softmax.py中的代码段：【1】初步检测成本值和梯度
   - 一般通过初步检验，快速排除一些明显的错误
3. 梯度值的详细检验
   - 分别对regularization strength=0.0和50.0时的梯度进行数值检测
   - 差值应该在1e-7级别
4. 优化超参数：learning rate以及regularization strength
   - 对应04demo_softmax.py中的代码段：【3】选取hyper-parameters
   - 将优化出来的权重矩阵W，转化成32x32x3的图像形式展现出来：

可以看到，W保留了原始图像分类的某些基本特征

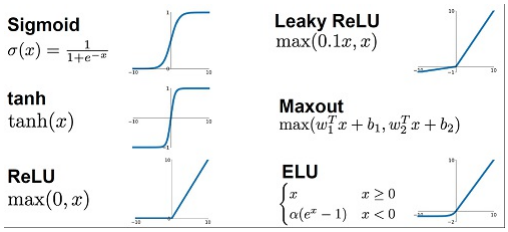
第5节：简单神经网络

两层神经网络

通过仅有一个隐藏层的神经网络(2层神经网络)来进行图像分类。该神经网络采用如下结构：
输入层 -> 线性变换 -> ReLU激活函数 -> 隐藏层 -> 线性变换 -> Softmax -> 输出层
在此过程中，需要进行两次线性变换，因此需要分别设定两次变换的权重和偏置系数：$W1$, $b1$以及$W2$, $b2$

激活函数

常用激活函数：

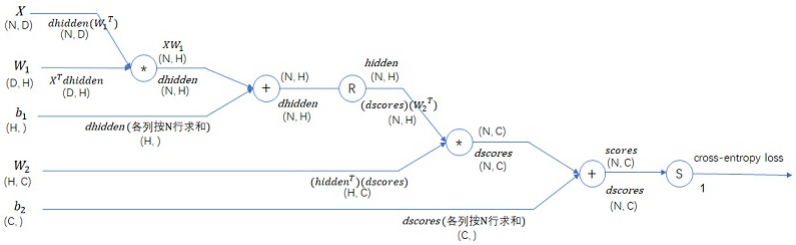


激活函数是非线性的，它有助于对上一个线性变换的计算结果进行调整(例如：限定值范围、去除某些值域等)
常用激活函数的特点如下表：

激活函数	描述	优点	缺点
sigmoid	将任意值域范围限定到0~1范围。当x值小于-4，输出值已经趋于0；当x值大于4，输出值已趋于1	容易被理解为0~1之间的"概率"	仅当x在很小的范围内时，其梯度才比较大；否则梯度将趋于0，将会导致梯度下降算法失效；输出值不是基于0左右分布
tanh	将任意值域范围限定到-1~1范围。当x值小于-2，输出值已经趋于-1；当x值大于2，输出值已趋于1	输出值基于0对称分布	仅当x在很小的范围内时，其梯度才比较大；否则梯度将趋于0，将会导致梯度下降算法失效；
ReLU	所有小于0的输入，全都输出0；所有大于0的输入，直接输出	对于大于0的输入，能够确保梯度下降算法不会失效；计算简单	对于小于0的输入，梯度为0；输出值不是基于0对称分布

梯度计算

本网络需要分别计算$W1$, $b1$以及$W2$, $b2$的梯度。反向传播梯度计算流程图如下



迭代

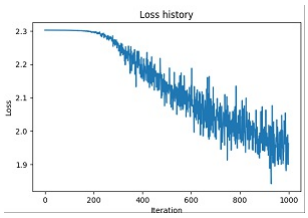
在采用mini-batch随机梯度下降后，每次从N个样本中随机选出M个进行计算。因此，把N个样本全部计算一次，需要执行N/M次。

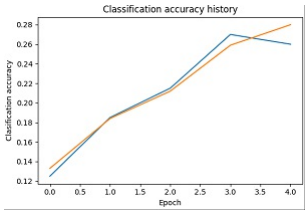
- 每针对选出的M个样本执行一次mini-batch梯度下降计算，称为一个iteration
- 将N个样本全部计算一次，称为一个epoch
- 可以设定一个总iteration数，然后根据M的大小，自动确定需要执行的epoch次数。可根据收敛情况调整iteration数。

示例**1**：使用简单神经网络进行图像分类

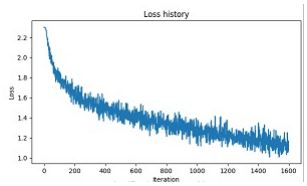
本实验源代码位于【14/05demo_two_layer_neral_net.py】【14/two_layer_neural_net.py】

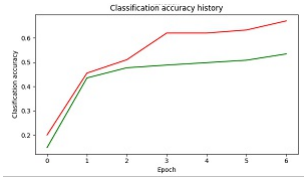
1. 试训练，并查看收敛情况
    - 对应05demo_two_layer_neral_net.py中的代码段：【1】执行训练并监视训练过程
    - 在这一过程中，设置较低的iteration值，主要通过观察loss和accuracy的趋势，初步判断神经网络是否正常工作
    - 观察结果如下





2. 优选训练速率和惩罚权重系数
    - 对应05demo_two_layer_neral_net.py中的代码段：【2】调整hyper-parameters
    - 通过指定learning-rate和regularization-strength组合，通过验证数据集确定最佳的学习速率和惩罚权重系数
    - 在当前模型下，可以获得的最佳测试正确率在0.5以上

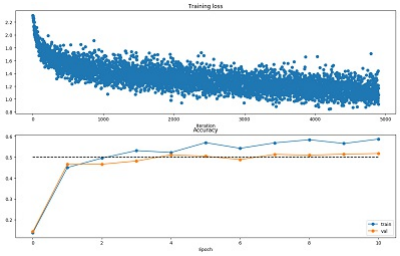

Loss history
Loss
Iteration


Classification accuracy history
Classification accuracy
Epoch

第6节：模块化神经网络

示例**1**：模块化的**2**层神经网络实现

- 任务：使用模块化的方式实现一个2层神经网络(1个隐藏层)
- 神经网络架构：输入层 -> 线性变换 -> ReLU激活 -> 线性变换 -> Softmax -> 输出层
- 模块化组件：
    - 工具类：【14/util_common_layer.py】，实现一些通用的计算，包括：
        - 针对线性变换的正向计算和反向传播梯度计算代码
        - 针对ReLU激活函数的正向计算和反向传播梯度计算代码
        - 针对Softmax的正向计算、成本值计算以及反向传播梯度计算代码
    - 模型类：【14/two_layer_neural_net_modular.py】
        - 存储两层神经网络的的各级权重、偏置系数
        - 组合调用工具类中的方法，实现本神经网络完整的正向计算流、成本值计算以及各级权重、偏置的反向梯度计算
    - 优化求解器类：【14/solver.py】
        - 根据传入的模型和训练数据，执行训练的epoch和iteration过程
        - 根据传入的配置参数(例如learning_rate，regularization strength等)，在训练过程中应用这些参数
        - 根据传入的配置参数，选择合适的梯度更新策略，调用梯度更新策略类中的相应函数，在训练过程中执行梯度更新
        - 根据需要存储或打印计算过程中的中间结果
    - 梯度更新策略类：【14/optim.py】
        - 定义梯度更新策略
        - 目前仅定义了sgd策略，未来可以添加更多梯度更新策略
    - 测试程序：【14/06test_two_layer_neural_net_modular.py】
        - 所有代码汇总：util_common_layer.py，two_layer_neural_net_modular.py，solver.py，optim.py，06test_two_layer_neural_net_modular.py 注：上述大部分源代码来自斯坦福大学CS231N课程中提供的样例代码
- 计算结果：
  与上一节"简单神经网络"的性能接近：



示例**2**：模块化的任意层次神经网络实现

代码位置：【14/full_connected_neural_net.py】【14/util_common_layer.py】【14/optim.py】【14/solver.py】【14/07test_full_connected_neural_net.py】

第十五章：多层神经网络

第1节：数据预处理

预备知识

- 矢量的点积：

  假设有平面上的坐标向量u和v，二者的夹角为θ，则：$u*v=|u|*|v|*cos\theta$

  - $u*v$表示向量的点积，其结果是一个实数；
  - $|u|$表示向量模(长度)
  - 如果$\theta=0$(两个向量同一方向)，则：$u*v=|u|*|v|$
  - 如果$\theta=90^\circ$ (两个向量垂直)，则：$u*v=0$
- 向量的投影：

  向量u在向量v上的投影长度为：$u*(v)'$

  - $(v)'$表示v的单位向量(也就是v的方向)，$u*(v)'$表示向量的点积
- 对称矩阵的性质：
  对称矩阵的特征向量，彼此正交。这就意味着任意两个特征向量的点积为θ
  协方差矩阵：给定(N,M)的矩阵A，希望计算该矩阵中M个列之间的协方差矩阵，计算方法如下：
  - 样本中心化，将矩阵A中的每个元素，减去该列的平均值，得到中心化的矩阵B
  - 进行下列矩阵运算获得协方差矩阵：$Acov=\frac{1}{n}(B^TB)$
  - 注意，这是计算M列之间的协方差矩阵。也可以计算N行之间的协方差矩阵，这时样本中心化需要减去行的平均值，且$Acov=\frac{1}{m}(BB^T)$
  - 协方差矩阵是对称矩阵，因此其所有的特征向量互相正交

训练样本的问题

假设训练样本共N个，每个样本包含M个特征，这些特征可能存在如下问题：

- M个特征中的2个或更多个互相可替代。例如，一个特征是以公里为单位，另一个特征使用英里为单位。显然，这两个特征中有一个是多余的，徒增计算量。
- 某个特征与训练的结果没有关联。例如，学生努力程度(特征)与学生成绩(结果)之间有一定的关联，而学生姓名(特征)与学生的成绩(结果)应该没有关联。那么学生姓名这个特征，对于学生成绩这个结果来说就是完全无用的；该特征的存在反而模糊了其它真正有用的特征。
- 某个特征的值，在各个样本中基本相同。例如，在N个样本中，某特征的值都是10，或者在

10附近很小的区间。这就意味着，通过该特征根本不能很好的区分各个样本，这样的特征价值很小，在一定程度上也是多余的。
- 过拟合。当M很大，而N又比较小时，容易造成过拟合

因此，在进行机器学习前，有必要对样本数据进行一些处理，使之：

- 去掉多余的或者没有贡献的特征
- 提取出最具辨识度(或最有价值贡献)的特征
- 减少特征的数量

但是，如何找到最具/最不具辨识度的特征，以便精简特征数量(从而达到特征降维)呢？主成分分析(Principal Component Analysis, PCA)提供了一种常用的方法。

主成分分析

给定N个训练样本，每个样本包含M个特征，形成$A(N, M)$矩阵
实现步骤：

- 样本中心化：对A中的每个元素，减去其列的平均值，得到矩阵B
- 获得B的各列协方差矩阵$Bcov(M, M)$
- 对协方差矩阵进行特征值分解，选取最大的K个特征值所对应的特征向量，形成投影矩阵$T(M, K)$。具体K值的选取，可参考对应得特征值。如果特征值接近θ，则可认为该维度没有价值，从而可以去掉。有时候可以这样选取K的最小值，使得下式成立：

  $\frac{\sum_{i=1}^{K}\lambda_i}{\sum_{i=1}^{M}\lambda_i} \geq 0.9$
- 使用投影矩阵对样本B进行投影，得到降维的训练样本矩阵：$C = AT$，维度为：(N, K)
- 对于测试数据、验证数据矩阵，先中心化，然后与T进行矩阵运算，得到降维后的测试数据和验证数据矩阵

示例**1**：使用**PCA**对手写图片数据降维

本例代码位于【15/01_01pca_demo.py】【15/dataset/digits_trainging.csv】
【15/dataset/digits_testing.csv】

示例**2**：使用**sklearn**的**PCA**对手写图片数据降维

本例代码位于【15/01_02sklearn_pca_demo.py】【15/dataset/digits_trainging.csv】
【15/dataset/digits_testing.csv】

PCA原理分析

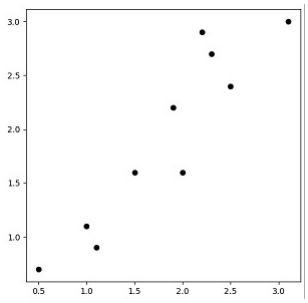
- PCA的主导思想：

- 要实现降维，可以视为空间变换，把训练矩阵从M维特征空间映射到K维特征空间($K < M$)
- 但是，并不能简单的删除某些特征维度来达到降维目的，而是通过设定新的坐标系(新坐标系的坐标轴少于原始坐标系的坐标轴数量)，使得原始数据在新坐标系下，沿着坐标轴的方向具有最大的价值
- 数据具有最大的"价值"，是说数据的辨识度很高，或者说，坐标系中的数据点能够清楚的区分彼此。这就要求：这些数据点比较分散！试想一下，如果所有点都集中在一起，它们之间就很难区分了。用定量的方法来描述，就是：这些数据点的方差越大(越离散)越好!
- 因此，PCA需要找到若干个关键坐标轴，使得原始数据点映射到由这些关键坐标轴组成的空间后，在每个坐标轴方向上都具有大的方差
- 这些坐标轴也并不是同等效果的。一般选择前K个最大方差的坐标轴构成新坐标系，将原始数据映射到新的坐标系中。此时特征数量从M将为K

- 使用PCA查看特征映射效果
  - 原始数据点
    假设有一批平面数据点集合$A(N = 10, M = 2)$：

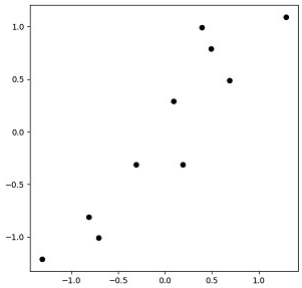
    X坐标值分别为：[2.5, 0.5, 2.2, 1.9, 3.1, 2.3, 2.0, 1.0, 1.5, 1.1]
    Y坐标值分别为：[2.4, 0.7, 2.9, 2.2, 3.0, 2.7, 1.6, 1.1, 1.6, 0.9]

    在原始坐标系下如图所示：



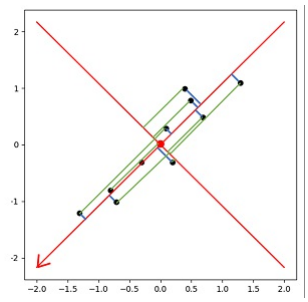
  - 中心化 此时这些点并不是基于0中心化的。将其进行了中心化，也就是，将每个X值都减去X向量的均值，Y值减去Y向量的均值，得到B：

    X中心化坐标值$X1$：[0.69, -1.31, 0.39, 0.09, 1.29, 0.49, 0.19, -0.81, -0.31, -0.71] Y中心化坐标值$Y1$：[0.49, -1.21, 0.99, 0.29, 1.09, 0.79, -0.31, -0.81, -0.31, -1.01]



    请注意，中心化以后，$X1$的平均值就是0， $Y1$的平均值也是0

- 映射 现在需要把这些点映射到某一个坐标系中，假设新坐标系的坐标轴如下图红线所示：



具体计算如下：

1. 根据PCA提供的方法，针对B的各列求得其协方差矩阵$Bcov$：

$$\begin{pmatrix} 0.5549 & 0.5539 \\ 0.5539 & 0.6449 \end{pmatrix}$$

2. 求$Bcov$的特征值和特征向量，并且按照特征值从大到小排列，对应的特征向量也要按照这个顺序：

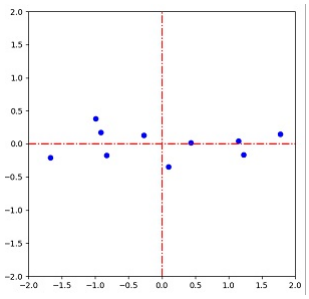
$$\lambda = (1.15562494, 0.04417506)$$

$$V = \begin{pmatrix} -0.6778734 & -0.73517866 \\ -0.73517866 & 0.6778734 \end{pmatrix}$$

3. 首先选取$K = 2$作为映射后的特征数量，将B映射到新的坐标空间中，得到$C = BV$：

$$\begin{pmatrix} -0.82797019 & -0.17511531 \\ 1.77758033 & 0.14285723 \\ -0.99219749 & 0.38437499 \\ -0.27421042 & 0.13041721 \\ -1.67580142 & -0.20949846 \\ -0.9129491 & 0.17528244 \\ 0.09910944 & -0.3498247 \\ 1.14457216 & 0.04641726 \\ 0.43804614 & 0.01776463 \\ 1.22382056 & -0.16267529 \end{pmatrix}$$

4. V中的两个特征向量是正交的，其中第一个特征向量，成为了新坐标空间中的水平轴方向，而第二个特征向量成为了新坐标空间中的垂直轴方向，作图如下：

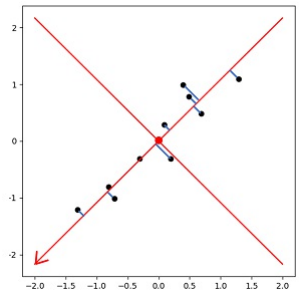


5. 新的坐标系中，水平方向分布显然更容易区分各个数据点。也观察到：各数据点的水平坐标值离散程度较大，或者说方差较大，这个方向的坐标轴可以被视为关键坐标轴

- 如何选取K
  - 新坐标系中，可以根据需要采用一维或二维坐标。如果仅选取$K=1$作为映射后的特征数量，则$C=BV[:,0]$，将只有水平方向的坐标轴：
  - 在本例中，因为水平方向的离散度远大于垂直方向的，所以原来二维数据点，近似可以用水平方向一维的映射点来表示

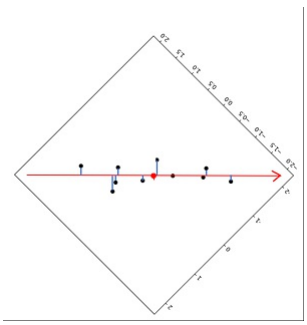
坐标映射的原理

考虑将数据点集合B投影到V中第一个特征向量指示的新坐标轴上；并且新坐标轴的原点与原始坐标轴原点一致。 设单位向量$(v)'=V[:,0]$，表示新坐标轴在原始坐标系中的方向。在本例中是：(-0.6778, -0.7351)，可以看出，该向量方向是朝向坐标系左下角的。 根据预备知识中关于向量的投影描述，$X1, Y1$数据点在新坐标轴上的长度(相对原点而言)为：$u*(v)'$，其中u是B中的$X1, Y1$数据点在原始坐标系中的向量表示，例如(0.69, 0.49)

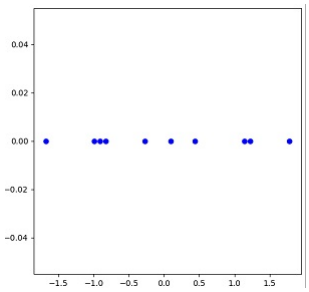


图中，$X1,Y1$数据点像新坐标轴做垂线，交点就是投影点，各投影点到原点(中间红色点)的长度，就是$u*(v)'$

将新坐标轴旋转至正向向右：



假设该新坐标轴，从原点开始，按照个投影点的长度分别计算新的水平坐标，即之前计算出来的： [-0.82797019 1.77758033 -0.99219749 -0.27421042 -1.67580142 -0.9129491 0.09910944 1.14457216 0.43804614 1.22382056] 就构成了上节中的新坐标系图：



示例**2**：查看二维坐标映射图像

- 本例代码位于【15/01_03point_project.py】

为什么协方差矩阵的特征向量能将原始特征映射到方差大的坐标轴

或者说，为什么协方差矩阵的特征向量代表着新坐标轴的方向？

假设有已经完成中心化的N个样本点$x^{(1)}, x^{(2)}, \cdots, x^{(n)}$，每个样本点都包含$M$个特征。例如样本点$x^{(i)}$ 的特征分别记为：$x_1^{(i)}, x_2^{(i)}, \cdots, x_m^{(i)}$ 。它们形成(N, M)矩阵

现在要映射到一个新的坐标轴，使得在新坐标轴方向上的各个投影点方差最大。

设新坐标轴的单位向量为u : $(u_1, u_2, \cdots, u_m)^T$ 。根据坐标投影计算法则，$x^{(i)}$在u上的投影为：$x^{(i)}u$，也可以写为：$(u)^T(x^{(i)})^T$

因为原有样本已经完成中心化，因此投影后的各个样本的均值也为0，而方差可以定义为：

$$\frac{1}{N}\sum_{i=1}^{N}(x^{(i)}u)^2 = \frac{1}{N}\sum_{i=1}^{N}(u^T(x^{(i)})^T x^{(i)}u) = u^T(\frac{1}{N}\sum_{i=1}^{N}(x^{(i)})^T x^{(i)})u$$

可以看出来，上式中大括号的部分$(\frac{1}{N}\sum_{i=1}^{N}(x^{(i)})^T x^{(i)})$ 就是协方差矩阵！

设：

$$\lambda = \frac{1}{N}\sum_{i=1}^{N}(x^{(i)}u)^2$$

$$Cov = \frac{1}{N}\sum_{i=1}^{N}(x^{(i)})^T x^{(i)}$$

则上式可写为：

$$\lambda = u^T Cov u => u\lambda = uu^T Cov u => \lambda u = Cov u$$ (因为u是单位向量，所以$uu^T = 1$)

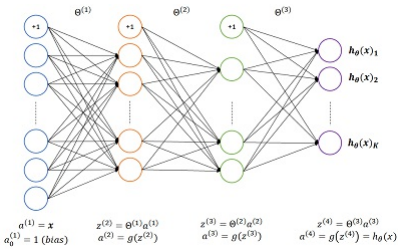
对比上式以及矩阵特征分解：$A\nu = \lambda\nu$

可知：u就是矩阵Cov的一个特征向量，而λ是Cov的一个特征值。 这就是说：只要计算出协方差矩阵Cov，那么它的特征向量就是新坐标轴的方向！

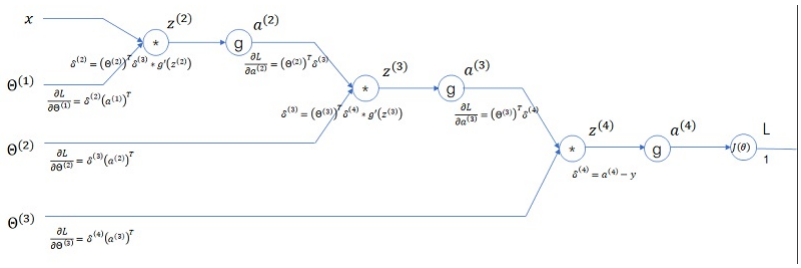
第2节：权重初始化

提出问题

观察下列神经网络：



以及对应的反向回归梯度流：



考虑$\Theta^{(1)}$，$\Theta^{(2)}$ 和$\Theta^{(3)}$ 要分别如何初始化？

全都初始化为相同的实数

考虑一个样本x

- 第1轮正向计算，$a^{(2)}$ 中所有元素都相同，$a^{(3)}$ 中所有元素都相同，$a^{(4)}$ 中所有元素也都相同；$z^{(1)}$，$z^{(2)}$，$z^{(3)}$ 中的元素也分别相同
- 第1轮反向传播：
  - $\delta^{(4)}$：各元素可能不同
  - $\frac{\partial L}{\partial a^{(3)}}$：所有元素都相同。$\frac{\partial L}{\partial \theta^{(3)}}$：每行行内各元素相同。$\delta^{(3)}$：所有元素都相同
  - $\frac{\partial L}{\partial a^{(2)}}$：所有元素都相同。$\frac{\partial L}{\partial \theta^{(2)}}$：所有元素都相同。$\delta^{(2)}$：所有元素都相同
  - $\frac{\partial L}{\partial \theta^{(1)}}$：每列列内各元素相同(或者说，每行都相同)
- 第2轮正向计算(已完成一次梯度下降)

    - $\theta^{(1)}$：每列列内各元素相同(或者说，每行都相同)
    - $z^{(2)}$：所有元素都相同；$a^{(2)}$：所有元素都相同
    - $\theta^{(2)}$：所有元素都相同
    - $z^{(3)}$：所有元素都相同；$a^{(3)}$：所有元素都相同
    - $\theta^{(3)}$：每行行内各元素相同(或者说，每列都相同)
    - $z^{(4)}$，$a^{(4)}$：各元素可能不同
- 第2轮反向计算
    - $\delta^{(4)}$：各元素可能不同
    - $\frac{\partial L}{\partial a^{(3)}}$：所有元素都相同。$\frac{\partial L}{\partial \theta^{(3)}}$：每行行内各元素相同。$\delta^{(3)}$：所有元素都相同
    - $\frac{\partial L}{\partial a^{(2)}}$：所有元素都相同。$\frac{\partial L}{\partial \theta^{(2)}}$：所有元素都相同。$\delta^{(2)}$：所有元素都相同
    - $\frac{\partial L}{\partial \theta^{(1)}}$：每列列内各元素相同(或者说，每行都相同)
- 可以得出规律，每轮计算后得到：
    - $a^{(2)}$：所有元素都相同
    - $a^{(3)}$：所有元素都相同。
    - 推广可知：隐藏层每层都只相当于有1个神经元
- 因此，不能全部初始化为相同的实数！

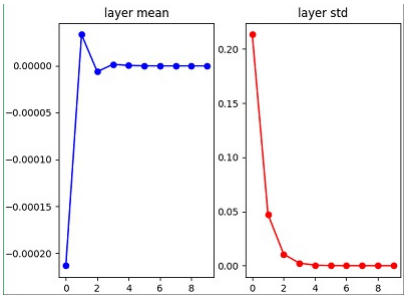
示例**1**：观察神经网络中权重初始化为相同实数时的问题

- 代码参考【15/02_01weight_init_same_value.py】

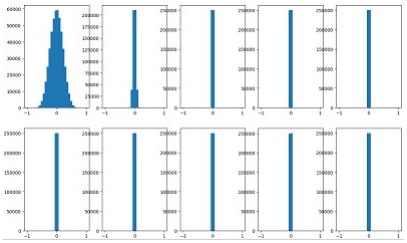
初始化为较小的随机数

- 条件
    - 考虑N个样本，使用10个隐藏层神经网络进行前向计算
    - 采用标准正态分布来生成随机数并乘以一个较小的系数(0.01)来初始化各级权重
    - 使用tanh作为激活函数
    - 统计每层节点值的均值和标准差
- 查看问题
    - 每层节点的均值和标准差
        - 输入层，均值：-0.000597，标准差：0.999271
        - 隐藏层1，均值：-0.000213，标准差：0.213266
        - 隐藏层2，均值：0.000033，标准差：0.047405

        - 隐藏层3，均值：-0.000006，标准差：0.010633
        - 隐藏层4，均值：0.000002，标准差：0.002390
        - 隐藏层5，均值：0.000000，标准差：0.000533
        - 隐藏层6，均值：0.000000，标准差：0.000119
        - 隐藏层7，均值：-0.000000，标准差：0.000027
        - 隐藏层8，均值：-0.000000，标准差：0.000006
        - 隐藏层9，均值：0.000000，标准差：0.000001
        - 隐藏层10，均值：-0.000000，标准差：0.000000
    - 图像
        - 均值和标准差：



        - 各层节点值分布直方图：



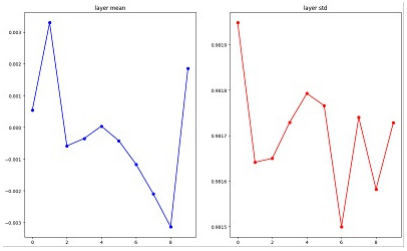
    - 结果讨论
        - 前向计算时，第1个隐藏层的节点计算结果也能很好的符合正态分布
        - 但是从第2层开始，各级节点的计算结果的均值越来越趋于0，而标准差也趋于0。这就是说：各级节点值都趋于0。从上面的直方图也可以看出来，后面几个层级几乎所有节点都集中在0附近
        - 如果某层级所有节点值a趋于0，从而使得$\frac{\partial L}{\partial \theta} = \delta a^T$ 趋于0，同样导致"梯度消失"
        - 一旦出现梯度消失现象，那就意味着每个计算循环基本没有什么效果，导致计算很难收敛
        - 若使用relu作为激活函数，也有类似的结果
        - 层数越多，后续的层级越容易出现梯度消失

示例**2**：观察采用较小的随机值初始化权重时的问题

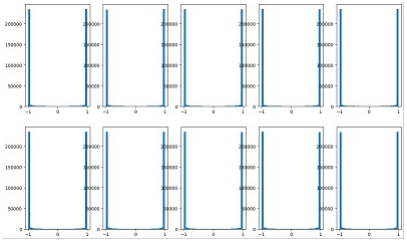
- 代码参考【15/02_02weight_init_small_random.py】

初始化为较大的随机数

- 条件
  - 考虑N个样本，使用10个隐藏层神经网络进行前向计算
  - 采用标准正态分布来生成随机数并乘以一个较小的系数(1.0)来初始化各级权重
  - 使用tanh作为激活函数
  - 统计每层节点值的均值和标准差
- 图像
  - 均值和标准差：



  - 各层节点值分布直方图：



- 结果讨论
  - 各层节点均值接近0，而标准差接近1。这说明节点值集中在-1或1附近
  - 回顾：本级节点(设为$a^{(2)}$)值是上级节点(设为$a^{(1)}$)与权重矩阵(设为W)进行线性变换，再经过激活函数(设为g)而得：

$$z^{(2)} = Wa^{(1)}$$

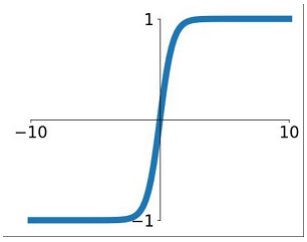
$$a^{(2)} = g(z^{(2)})$$

  - 在反向传播时，假设已经求得：$\frac{\partial L}{\partial a^{(2)}}$，如果要计算权重矩阵$W$的梯度$\frac{\partial L}{\partial W}$，那么必须先计算$\frac{\partial L}{\partial z^{(2)}}$

$$\frac{\partial L}{\partial z^{(2)}} = \frac{\partial L}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}}$$

$$\frac{\partial a^{(2)}}{\partial z^{(2)}} = tanh'(z^{(2)})$$

- 从tanh函数的曲线图可以看到，当$a^{(2)}$ 值(图中纵坐标)趋于-1或1时，其针对$z^{(2)}$ (图中横坐标)的梯度(曲线斜率)趋于0。这种情况可以叫做"激活值饱和"：



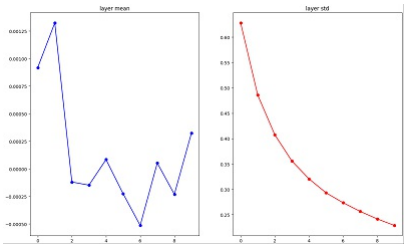
- 因此，将会导致权重矩阵的梯度为0，再次造成"梯度消失"
- 可自行尝试relu激活函数

示例**3**：观察采用较大的随机值初始化权重时的问题

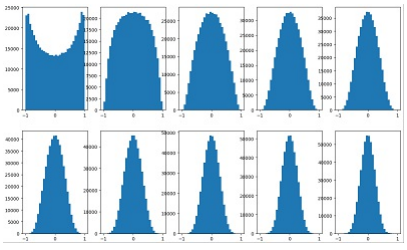
- 代码参考【15/02_03weight_init_big_random.py】

Xavier初始化

- 条件
  - 考虑N个样本，使用10个隐藏层神经网络进行前向计算
  - 采用下列公式计算初始权重：`np.random.randn(fan_in, fan_out) / np.sqrt(fan_in)`
    - 使用tanh作为激活函数
    - 统计每层节点值的均值和标准差
- 图像
  - 均值和标准差：

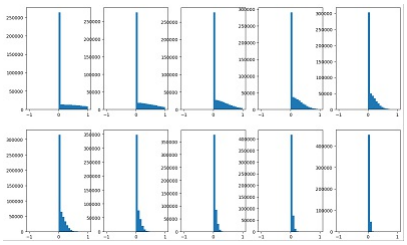


  - 各层节点值分布直方图：



- 结果讨论

    - 可以看到，各层节点值，并不像之前一样几乎全部集中于某个特定数值上，而是有较好的分布效果。从而可以作为一种不错的初始化方法
    - 但是，如果使用relu激活函数，将会看到下列直方图：



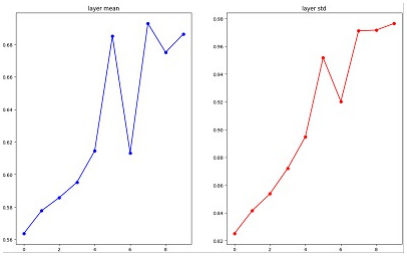
    - 这说明，在relu激活函数下，在后续几层，绝大部分节点值都在0附近。考虑到，relu函数的梯度，在$y=0$附近时就是0。这将导致巨大部分节点值对应的权重梯度都是0，同样产生"梯度消失"

示例**4**：观察**Xavier**初始化权重时的问题

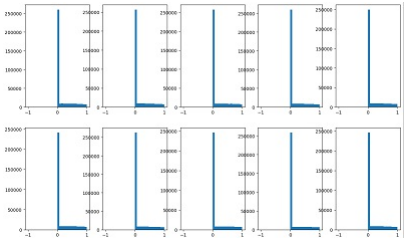
- 代码参考【15/02_04weight_init_xavier.py】

He et al初始化

- 条件
    - 考虑N个样本，使用10个隐藏层神经网络进行前向计算
    - 采用下列公式计算初始权重：`np.random.randn(fan_in, fan_out) / np.sqrt(fan_in / 2)`
    - 使用relu作为激活函数
    - 统计每层节点值的均值和标准差
- 图像
    - 均值和标准差：



    - 各层节点值分布直方图：



- 结果讨论
    - 虽然有大量节点值为0，但是仍有一半值(250000)是非0的
    - 在深度神经网络种，建议可以采用这种初始化的方式

示例**5**：观察**He et al**初始化权重时的效果

- 代码参考【15/02_05weight_init_he.py】

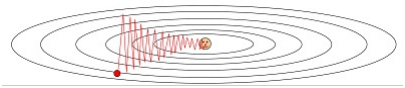
第3节：梯度更新策略

梯度更新策略指的是：在迭代训练过程中，已经计算出了权重w的梯度值∇w以后，如后使用该梯度值更新权重值

SGD标准更新

$w_t = w_{t-1} - \alpha * \nabla w$

- w的更新值仅与当前计算出的梯度∇w，以及学习速率α有关
- ∇w在每个分量方向上，都将乘以相同的速率α，这就可能导致：在某个方向上变化值过大，而在另一个方向上变化值过小。
- 如果某个方向上变化值过大，可能造成偏离最优点(来回震荡)；变化值过小，可能造成收敛速度缓慢
- 可能产生如下图所示打得的震荡收敛效果（水平方向收敛缓慢，垂直方向来回震荡）：



- 在碰到鞍点或拐点时，因为计算出的$\nabla w = 0$，将导致计算立即"收敛"，却并没有达到最优值

SGD+Momentum(动量)

$v_t = \rho * \nu_{t-1} - \alpha * \nabla w$

$w_t = w_{t-1} + v_t$

式中，ρ取值一般为0.5, 0.9, 0.99中的一个，v_0=0

- 从物理角度解释
    - 假设某个物体，沿着某个斜坡(斜坡各处梯度不同)下滑。将v视为速度，ρ可以视为摩擦系数；∇w视为加速度，w视为下滑距离；并且假设每单位时间(每秒)更新一次距离
    - 设最开始v的速度为0(也就是静止)，在加速度(重力沿坡面的分量)作用下，不断加速；与此同时，受到摩擦力制约，速度要衰减
    - 当加速度∇w逐渐趋于0后，速度v还会持续一段时间，w也仍会继续增大，直到在摩擦力ρ的作用下，速度逐渐变为0
- 这种方式考虑了历史速度对当前速度的影响，或者说，不仅考虑当前计算的梯度值，也考虑了历史变化速度。因此：
    - 即使当前计算出来的梯度趋于0，计算也不会立即停止，而是沿着之前的梯度下降方向还

要继续进行若干个循环。这就有利于越过拐点、鞍点，尤其是比较差的局部最优点
    - 不同方向(维度)的梯度，将会结合各自历史变化速率，在一定程度上有助于减少震荡，加速收敛

Nesterov Momentum

$\tilde{w_t} = w_{t-1} + \rho * v_{t-1}$

$\nabla w = f'(\tilde{w})$

$v_t = \rho * v_{t-1} - \alpha * \nabla w$

$w_t = w_{t-1} + v_t$

• 解释

- 不是计算当前位置的梯度，而是每次都向前看一步，计算前一步的梯度值
- $x_{t-1} + \rho * v_{t-1}$ 就代表了在当前位置基础上，增加一个单位时间内的$\rho * v_{t-1}$从而获得(以当前速度为基础的)下一个位置值$\tilde{w_t}$
- 然后计算出下一个位置的梯度：$\nabla w = f'(\tilde{w_t})$
- 再利用$\rho * v_{t-1} - \alpha * \nabla w$，更新当前速度$v_t$
- 最后再求出当前距离(也就是更新后的权重)w_t
- 公式变形： 不计算前一步的梯度值，而是计算当前的梯度值，然后再进行后续变形。变形后的公式一般更易于计算。 $\nabla w = f'(w_{t-1})$

  $v_t = \rho * v_{t-1} - \alpha * \nabla w$

  $w_t = w_{t-1} - \rho * v_{t-1} + (1 + \rho) * v_t$
- 有证据表明该种方法更易于收敛

AdaGrad

$cache+ = (\nabla w)^2$

$w_t = w_{t-1} - \frac{\alpha * \nabla w}{\sqrt{cache} + \varepsilon}$

式中，ε可设置为1e-7，用以防止分母为0的情况。

- 自适应权重更新
  - 前述三种算法中，α在每次更新时不变，因此在一定程度上造成每次更新权重时步长的变化不大；总体来说收敛速度都比较慢
  - 实际应用时，我们希望在循环初期阶段可以采用较大的步长，以便尽快到达最优值附近；然后改用较低的步长精确到达最优值。
  - 这种自动调整权重步长的方法称为自适应算法。AdaGrad是第一种自适应算法
- 关于AdaGrad的说明
  - α保持不变，根据w的更新表达式，可知每次更新，cache的值越累积越大，w的步长将越来越小。达到了逐步降低权重步长的目的。
  - 在各次循环过程中，cache的值单调增加，而w的增量绝对值将单调减小。
  - 但是这样有可能使得学习率过早的降低，最后甚至可能停留在离最优点较远的位置

RMSProp

$cache = decayrate * cache + (1 - decayrate) * (\nabla w)^2$

$w_t = w_{t-1} - \frac{\alpha * \nabla w}{\sqrt{cache} + \varepsilon}$

- 式中，decayrate是一个超参数，取值可以尝试0.9, 0.99, 0.999
- 引入了decayrate后，每次循环，cache都会损失一部分值，同时又会从$(\nabla w)^2$ 中获得一部分值。这使得cache不是单调递增，而w的增量也不会单调减少
- 可以认为，会根据∇w来进行加速或减速

Adam

设置初值：w_0 (随机数)，$m_0 = 0$(第一个动量)，$v_0 = 0$(第二个动量)，$t = 0$(循环计数) 循环过程：$t = t + 1$

$\nabla w = f'(w_{t-1})$

$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \nabla w$

$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * (\nabla w)^2$

$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$

$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$

$w_t = w_{t-1} - \frac{\alpha * \hat{u}_t \varepsilon}{\sqrt{\hat{v}_t}}$

上式中，β_1一般可取经验值0.9，β_2 一般可取0.995

Adam可视为结合了Momentum和RMSProp, 是目前比较推荐的算法

调整α

- 显然，调整α将影响所有算法的步长
- 一般有下列三种显式调整法：
  - step decay：每隔若干个epoch，将α调整为$k\alpha$。这是比较推荐的算法。例如，每5个epoch，α调整为0.5α ；或者每个epoch，α调整为0.9α
  - exponential decay：$\alpha = \alpha_0 e^{-kt}$ 。其中t为iteration次数，k是超参数，需要选择优化
  - 1/t decay：$\alpha = \frac{\alpha_0}{1+kt}$

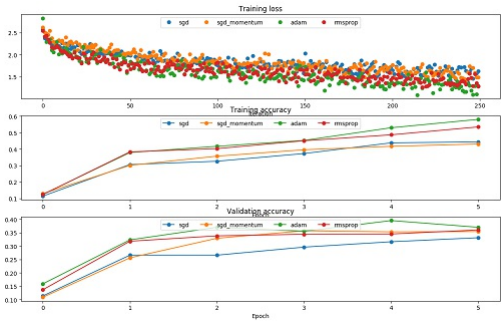
Newton法

$w_t = w_{t-1} - [Hf(w)]^{-1}\nabla f(w)$ 式中，$Hf(w)$称为Hessian matrix，是$f(w)$对w的二阶偏导数构成的方阵，$\nabla f(w)$仍然是梯度

- Hessian matrix记录了成本函数$f(w)$的曲率，在取逆矩阵后，将能够使得在曲率比较平缓的方向上选取较大的步长，在曲率比较陡峭的方向上选取较短的步长，从而更高效的收敛
- 但是Hessian matrix往往比较巨大，无法存储，更无法方便的求逆。因此一些近似Newton法诞生了。它们主要就是要近似模拟Hessian matrix的逆。
- L-BFGS是一种最流行的近似Newton法。但是目前的研究表明：它对于一次性运算所有样本数据是比较合适的，但是对mini-batch却不太合适 因此L-BFGS可能适于样本数据量较小的情形，但不太适大规模深度学习或卷积神经网络

示例**1**：各种梯度更新策略的比较

- 代码位于【15/full_connected_neural_net_simple.py】【15/optim.py】【15/solver.py】【15/03_01update_policy_compare.py】
- 可看到随着训练的进行，成本的下降趋势、训练数据的正确率如下图：



相对来说，adam(图中绿色)具有较好的收敛效果

Batch Normalization

使用目的

- 使用Batch Normalization对输入的数据(可能是样本的特征值，也可能是神经网络某一级的节点值)进行一定的处理后，使之符合某种正态分布
- 如果某一层的数据数据经过BN后，具有一定的正态分布特性，那么再使用激活函数时，就能够获得不错的反向传播梯度计算结果(可参考权重初始化中的介绍)

工作模式

- 在某个全连接层线性变换函数之后、激活函数之前，加入一个Batch Normalization(BN)层
- 训练数据时
  - 针对传入的数据值(线性变换后的计算结果，假设有N个样本/节点数据，每个样本/节点具有M个特征)，执行下列操作：
    1. 计算每个特征维度的均值，其中，第m个特征维度的均值为：

       $\mu_m = \frac{1}{N}\sum_{i=1}^{N} x_m^{(i)}$

       式中，$x_m^{(i)}$ 表示第i个样本的第m个维度
    2. 计算每个特征维度的方差

       $\sigma_m^2 = \frac{1}{N}\sum_{i=1}^{N}(x_m^{(i)} - \mu_m)^2$
    3. 执行Normalization，得到归一化结果：

       $\hat{x}_m^{(i)} = \frac{x_m^{(i)} - \mu_m}{\sigma_m + \varepsilon}$

       式中，ε为一个很小的数(例如1e-5)，用于防止σ_m^2为0而导致除数为0
    4. 应用Scale和Shift，得到最终输出结果

       $y^{(i)} = \gamma\hat{x}^{(i)} + \beta$

       式中，γ和β都是向量，维度为M
  - 关于BN处理后的输出结果y
    - 各个维度的均值与β中对应的元素接近，即：

$(\mu y)_m = \frac{1}{N}\sum_{i=1}^{N} y_m^{(i)} \approx \beta_m$

- 各个维度的方差与γ中对应的元素接近，即：

$(\sigma y)_m^2 = \frac{1}{N}\sum_{i=1}^{N}(y_m^{(i)} - (\mu y)_m)^2 \approx \gamma_m$

- 因此，通过给定合适的γ和β，就能够获得所期望的正态分布输出

- 在对数据进行预测、验证或测试时
  - 不应在待验证/测试的数据上求均值、方差，而是应该使用训练数据中的均值和方差，代入到前述公式中计算$\hat{x}_m^{(i)}$和$y^{(i)}$

BN结合Mini-Batch

- 过程分析
  - 在使用神经网络训练数据时，一般会将样本数据分成若干个batch，每个batch分别训练。
  - 如果插入BN层，那么BN层处理的就只是单个batch中的数据
  - 待训练完成，需要对测试数据进行计算时，根据前述说明，必须使用训练数据中的均值和方差。但是之前训练数据中原本只保存了一个batch的均值和方差，这是不够的，需要将所有样本数据的均值、方差考虑进去
  - 因此，在BN层计算单个batch中的数据时，还需要将该batch对应的均值、方差缓存起来；每个batch进行累加
  - 当最后一个batch计算完后，最终累加的均值和方差，就是所有样本数据的均值、方差
- 均值和方差累加策略：
  - 因为BN层本身并不知道总的样本数量(它只知道单个batch的样本数量)，所以并不方便精确计算出多个batch累计下来的均值和方差。但是可以根据下列办法近似计算：

    1. 计算第一个batch时，按照前述BN计算的基本方法进行，将得出的均值sample_mean和方差sample_var，保存到一个缓存中，命名为：running_mean, running_var

    2. 在计算后续batch时，先计算出该batch本身的sample_mean和sample_var，然后按照下列公式累加：

       ```
       running_mean = momentum * running_mean + (1 - momentum) * sample_mean
       running_var = momentum * running_var + (1 - momentum) * sample_var
       ```

       上式中，momentum是一个0~1之间的小数(例如0.9)，用于设定已完成计算的样本和正在计算的样本之间的比例关系。但这只是一个近似的估计值。

    3. 将更新后的running_mean, running_var放回到缓存中

4. 每个batch重复上述过程，最终得到的running_mean, running_var就可以视为总样本的均值和方差

- 在对验证、测试数据进行计算时，使用上述最终得到的running_mean和running_var

使用**Batch Normalization**的优点

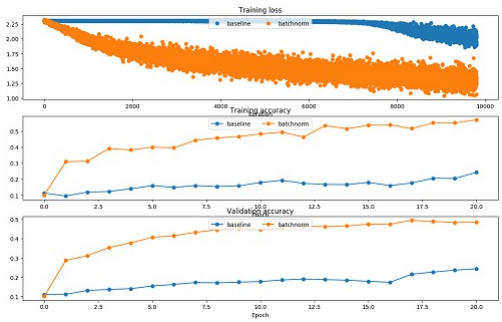
- 对深度神经网络的梯度计算有所改善，能够在一定程度上减少"梯度消失"
- 降低了对权重初始化的要求。允许更大一些的权重初始值存在
- 有可能允许更大的learning rate
- 有可能在一定程度上可以作为某种regularization的手段

示例**1**：查看**Batch Normalization**的计算结果

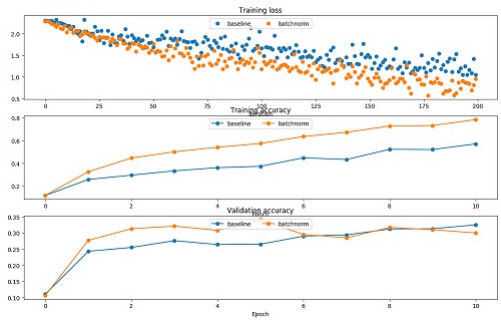
- 代码位于【15/util_batch_norm_layer.py】【15/04_01batch_norm_forward_backward.py】

示例**2:** 观察启用和不启用**Batch Normalization**的对比

- 代码位于【15/full_connected_neural_net_with_batch_norm.py】【15/util_batch_norm_layer.py】【15/04_02batch_norm_compare.py】
- 对于一个深度神经网络，在使用sgd梯度更新策略时，如果不采用Batch Norm，则有可能根本不收敛：

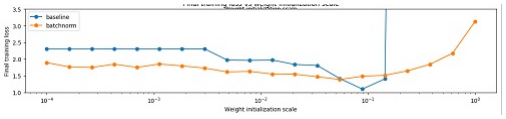


- 即使采用收敛效果更好的adam梯度更新策略，也可以看出，应用了batch norm后，成本下降的更快一些，而训练正确率提升得更快一些：



示例**3**：观察**Batch Norm**在不同权重初始化值情况下的表现

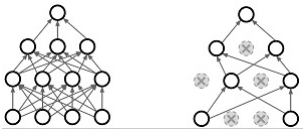
- 代码位于【15/full_connected_neural_net_with_batch_norm.py】【15/util_batch_norm_layer.py】【15/04_03batch_norm_for_weight_init.py】
- 仅选取1000个样本进行计算。
- 一般情况下，权重初始化值越大，越不容易收敛。但是在应用了Batch Norm后，它可以允许更大的初始化值。



如上图所示，随着初始化缩放系数越来越大到一定程度，不采用Batch Norm的训练会突然发散；而采用了Batch Norm的训练，会随着系数的进一步增大而逐渐发散

第5节：Dropout

Dropout是指：在训练时，冻结某些神经元节点，用剩余的节点进行计算；在预测时，使用所有的神经元节点进行计算。这种做法在一定程度上可起到Regularization的作用。下图列出了训练时的情形：

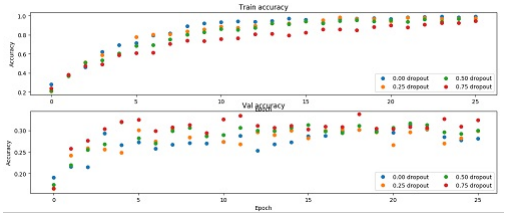


工作模式：

- 设置drop out的比例系数p。p指定了在每层网络中，被冻结的节点占该层总节点的比例。一般可设置为0.25, 0.5等。如果设为0，表明不采用dropout
- 训练的前向计算环节，选出p比例(p在0~1之间)的节点，将其节点值设置为0(相当于冻结了这些神经元)；剩下$1-p$比例的节点，按照原有节点值计算。因为每层网络都相当于只有$(1-p)$比例的节点贡献了最终的结果，因此相当于这个训练结果(例如，可视为每个分类结果的概率值)也只有实际的$(1-p)$倍
- 在预测环节，要使用所有的神经元进行计算。这时因为使用了相当于训练时$\frac{1}{1-p}$倍的节点，因此其计算结果也相当于实际结果的$\frac{1}{1-p}$倍。
- 为了协调训练和预测时的结果倍差，往往采用下列办法：
  - 在训练时，将选取出来的激活节点的节点值均除以$\frac{1}{1-p}$，这样就相当于得到了实际的结果
  - 在测试时，无需再做任何代码变动

示例**1**：观察**drop out**对训练结果的影响

- 代码位于【15/util_dropout_layer.py】【15/full_connected_neural_net_dropout.py】【15/05_01dropout_compare.py】
- 例题中分别使用p=0, 0.25, 0.5, 0.75，并观察其训练正确率和验证正确率。如下图：



- 可以看到，drop out越大，其训练准确率会下降(不容易过拟合），但验证准确率提高(说明起到了regularization的作用)
- 本例中选择的神经元数量较大(500个)，这时候drop out会有一定的效果。但如果神经元数量

本身就较少，可能drop out不会起到良好作用

- 如果训练样本数量很多，那么过拟合的可能性本来就会降低，此时drop out的效果可能也不太好

第**6**节：完整的多层神经网络代码实现

示例**1**：在上一章模块化神经网络代码的实现的基础上，应用梯度更新策略、**Batch Normalization**和**Dropout**，实现完整功能的神经网络

- 代码位于【15/full_connected_neural_net_all.py】【15/06_01test_full_connected_neural_net_all.py】
- 验证正确率可达：0.561，测试正确率可达：0.536。但这可能已经是多层神经网络的极限能力了。如果想进一步提升正确率，需要使用卷积神经网络

第十六章：卷积神经网络

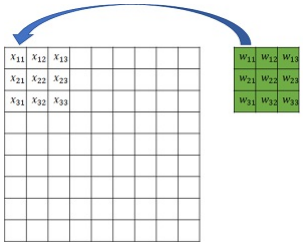
第1节：图像卷积操作

二维卷积操作

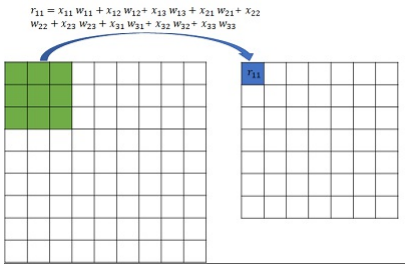
- 计算过程

  设某灰度图片X有(H,W)像素(例如9x9)，每个像素值在0~255之间。现在使用一个HHxWW矩阵W(例如3x3)与该图片进行卷积操作。矩阵W称为卷积核，有时候也被称为Filter

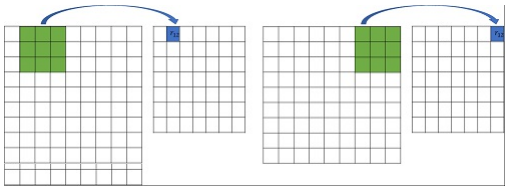
  1. 将W的左上角点元素与X中的坐上角点元素对齐

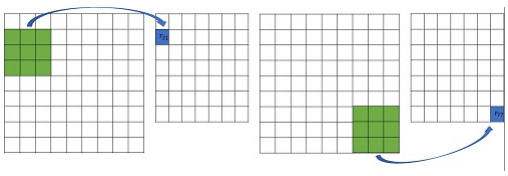
  2. W分别与图片中的HHxWW区块进行"点积和"操作，然后作为左上角点元素放入到卷积结果矩阵R中。在本例中，R维度为(7, 7)

  3. 将W向右移动一个元素，与X的第1行第2个元素对齐，使用点积和计算R中的第2个结果元素。依次类推，直到与X最右边3x3区域重叠

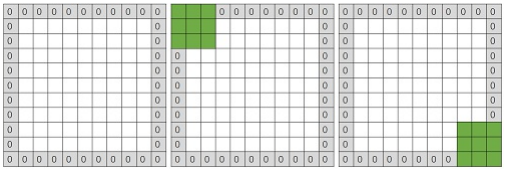
     

  4. W向下移动一个元素，再次从左向右与X中的各个3x3区域进行点积和操作，直到X最右下角的3x3区域

- 图像及卷积核大小的选取

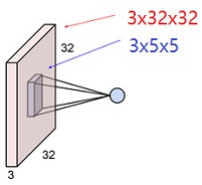
    - 一般情况下，图像应为方阵，即H=W
    - 一般情况下，HH和WW取相同的数
    - 当卷积核覆盖的区域已经超出了图像区域时，一般不再计算
- 计算卷积结果矩阵的大小
    - 步长(stride)：卷积核每次水平/垂直移动的元素个数。例如，stride=1，表示每次移动1个元素
    - 补全(pad)：在X的外围填补的行或列数。填补行或列的目的一般是为了使产生的结果矩阵具有特定的维度。填补行或列时，补入的元素一般赋值为0。
    - 输出结果矩阵大小计算公式
        - 垂直方向：OH = (H + pad*2 -HH)/stride +1
        - 水平方向：OW = (W + pad*2 -WW)/stride +1
        - 例如，9x9的图像，使用3x3的卷积核，stride=1，pad=0，此时输出的矩阵大小为：7x7
        - w 9x9的图像，使用3x3卷积核，stride=2，pad=0，此时输出的矩阵大小为：4x4
        - 9x9的图像，使用3x3卷积核，stride=1，pad=1，此时输出的矩阵大小为：9x9
        - 9x9的图像，使用3x3卷积核，stride=1，pad=2，此时输出的矩阵大小为：11x11



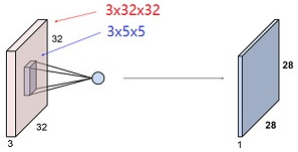
三维卷积运算

- 如果图片有RGB三个颜色通道，那么图片X构成(3, H, W)三维矩阵。此时卷积核W大小必须是(3, HH ,WW)。
- 在计算点积和时，是两个三维矩阵之间计算。例如，第一个卷积的计算结果为：

$$
\begin{aligned}
r_{11} = & x_{111}w_{111} + x_{112}w_{112} + x_{113}w_{113} + x_{121}w_{121} + x_{122}w_{122} + x_{123}w_{123} + x_{131}w_{131} + x_{132}w_{132} + \\
& x_{211}w_{211} + x_{212}w_{212} + x_{213}w_{213} + x_{221}w_{221} + x_{222}w_{222} + x_{223}w_{223} + x_{231}w_{231} + x_{232}w_{232} + \\
& x_{311}w_{311} + x_{312}w_{312} + x_{313}w_{313} + x_{321}w_{321} + x_{322}w_{322} + x_{323}w_{323} + x_{331}w_{331} + x_{332}w_{332} +
\end{aligned}
$$

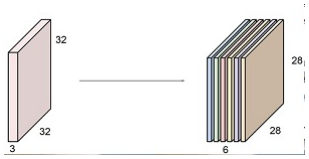


- 输出的卷积结果矩阵R仍为二维矩阵，它不再包含3个颜色通道。可以视为，这个卷积操作将3个颜色通道整合到了1个颜色通道中



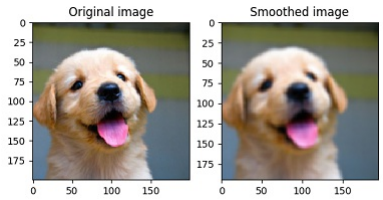
使用多个卷积核

- 针对一张图片，可以使用多个卷积核对其进行卷积运算。
- 因为每个卷积核对图片的卷积运算都会产生一个二维矩阵，故而多个卷积核的运算会产生多个二维矩阵，从而构成一个三维矩阵
- 采用下列表达式表示进行三维卷积运算的多个卷积核：W(F, C, HH, WW)。其中，F表示卷积核(Filter)的数量，C表示颜色通道数量，HH表示卷积核垂直方向元素数，WW表示卷积核水平方向元素数
- 每个卷积核的参数值是不同的。例如，采用(6, 3, 5, 5)形式的多个卷积核，就有6x3x5x5=450个卷积核的权重参数需要优化计算。



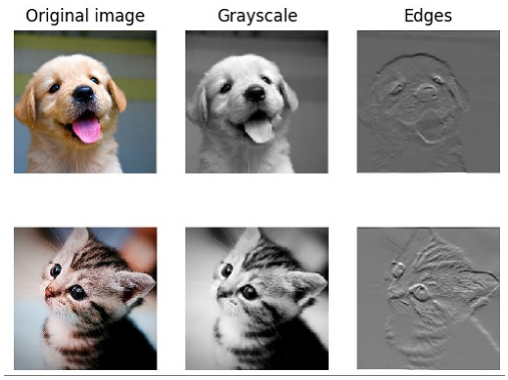
示例**1**：使用卷积实现图像的平滑**(**模糊**)**处理

- 代码位于【16/01_01conv_for_smooth.py】



示例**2**：使用卷积计算实现图像的灰度，以及进行边缘检测

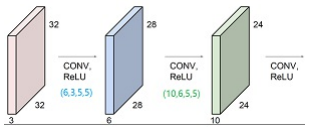
- 代码位于【16/01_02conv_for_gray_and_border.py】【16/util_conv_layer.py】



第2节：定义网络结构

卷积层

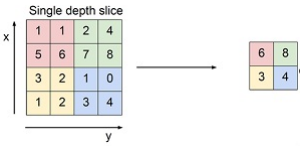
- 在每个卷积后，使用一个激活函数，一般可选择ReLU
- 可以将多个卷积层(含激活函数)连接起来。其中，每个卷积层又分别包含各自的多个卷积核。
  - 下图中第一层包含6个卷积核，每个卷积核为3x5x5，在stride=1，pad=0情况下，输出6个卷积结果，每个结果为28x28。然后对卷积结果进行ReLU
  - 第二层包含10个卷积核，每个卷积核为6x5x5。注意，因为上一级卷积结果相当于"颜色通道为6"，因此卷积核的通道数(或厚度)必须也为6
- 一般情况下，不应使卷积结果过快收缩！如下图，卷积结果的长和宽之比卷积之前减少4个元素。有时候，需要通过pad，使输出尺寸与卷积之前一致



- 需要先正向计算每层卷积结果及ReLU，在反向传播时，需要计算每层ReLU及卷积的梯度

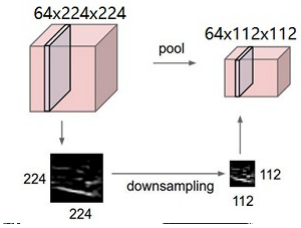
池化层

- 在某个卷积层之后，使用池化层，将卷积结果尺寸变小。例如将10x24x24的卷积结果缩小为10x12x12。注意，"厚度"是不变的，只缩小长和宽
- 池化的方法，是将卷积结果分成若干个小区块，对每个区块分别操作，得到一个计算结果；所有区块的计算结果合起来形成一个(变小了的)池化结果：



  上图中，将卷积结果划分为2x2小块依次处理；stride设为2。每个小块中取最大元素值作为池化结果。可以看到，池化结果尺寸是卷积结果的一半

- 常见的池化方法有Max-Pooling或Mean-Pooling。前者取区块中的最大值，后者取区块中的平均值
- 可以认为池化的主要作用是是卷积结果缩小，为下一层的输入作准备。注意：缩小矩阵大小，主要依靠池化，而不是卷积



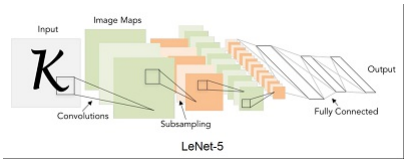
- 池化层同样需要正向计算以及反向传播的时的梯度计算。但是因为池化层无需参数权重，所以反向传播梯度只需要计算数据x的即可

全连接层

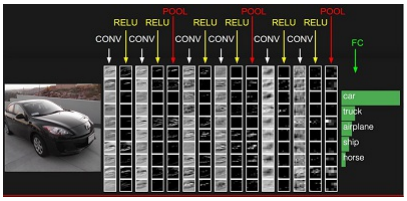
整个卷积网络的最后，放置一个全连接神经网络层。该全连接神经网络可以包含多个隐藏层

完整的卷积神经网络结构示意

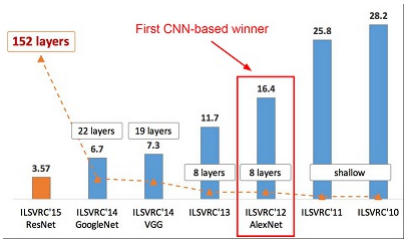
- LeNet-5卷积神经网络结构示意图



- 典型图像分类卷积神经网络结构示意图



- 在图像识别领域著名的神经网络结构



第3节：实现卷积神经网络

示例**1**：**cnn**的快速计算模块

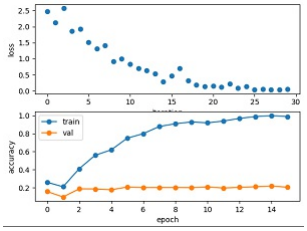
- 原始的CNN算法计算速度较慢，而CNN网络规模一般较大，因此非常耗时。斯坦福大学CS231N课程提供了一个快速计算模块。包括下列文件：
  - 【16/fast_layers.py】：这是主要的接口文件，供CNN网络调用以执行前向计算和反向传播梯度计算
  - 【16/im2col.py】【16/im2col_cython.py】：底层实现
  - 【16/setup.py】：编译文件
- 编译方法
  - Windows下，要安装Visual C++ 2017版本，以确保存在C++的编译器；
  - Ubuntu下：sudo apt-get install build-essentials
  - 安装好Python3及Pip3
  - 安装Cython： pip3 install Cython
  - 进入【16】目录，执行：python setup.py build_ext --inplace

示例**2**：对比**fastcnn**和原始**cnn**计算的效率

- 代码位于：【16/util_common_layer.py】【16/util_conv_layer.py】【16/fast_layer.py】【16/02_01cnn_impl_compare.py】
- fastcnn的速度是原始cnn实现的100倍以上

示例**3**：构造一个基本的**CNN**网络执行**CIFAR-10**图像分类

- 代码位于：【16/util_common_layer.py】【16/util_conv_layer.py】【16/fast_layer.py】【16/cnn.py】【16/02_02cnn_for_cifar10.py】
- 网络结构为：conv - relu - 2x2 max pool - affine - relu - affine - softmax
- 选取100个样本试算，观察loss下降情况，以确保网络初步计算的合理性



- 计算完整的计算，大约训练集:0.459，验证集：0.476。如果要进一步提升性能，需要更精细的设计卷积神经网络的结构

第十七章：Tensorflow

第1节：基本概念

Tensorflow程序执行过程

- Tensorflow程序分为构建阶段和执行阶段
- 在构建阶段，主要使定义各类张量(Tensor)、Tensor的每个计算步骤(Operation)，并将这些步骤组合成一个计算流(Graph)。
- 在执行阶段，可通过编程来指定依次或多次执行上述计算流
- 注意，只有才执行阶段，各个张量才会真正被计算出结果来

Tensor的类型

- tf.constant：直接初始化，后续计算不会修改之
- tf.Variable：定义变量，在计算过程中将修改之。在定义时需要给定一个初始值
- tf.placeholder：定义占位符。例如，一个实数、一个向量或者一个矩阵等。在定义时要指定占位符的长度(或维度)，在执行阶段，通过feed_dict参数将实际的数据值传入

示例**1**：**tensorflow**编程元素

- 代码位于【17/quickstart.py】

第2节：典型机器学习算法的Tensorflow实现

示例1：tensorflow线性回归

- 代码位于：【17/02linear_regression.py】

示例2：tensorflow逻辑回归

- 代码位于：【17/03logistic_regression.py】

示例3：KNN方法对MNIST手写图片进行识别

- 代码位于：【17/04knn_classify.py】

示例4：softmax线性分类器对MNIST手写图片进行识别

- 代码位于：【17/05softmax.py】

示例5：神经网络对MNIST手写图片进行识别

- 代码位于：【17/06neural_network.py】

第3节：卷积神经网络的Tensorflow实现

示例**1**：使用简单的卷积神经网络对**MNIST**手写数字图片识别

- 网络结构：Conv -> ReLU -> Max Pooling -> Conv -> ReLU -> Max Pooling -> FC1(1024) -> ReLU -> Dropout -> Affine -> Softmax
- 代码位于：【17/07conv_mnist.py】
- 能获得99.2%的验证数据识别率

示例**2**：使用最简单的卷积神经网络对**CIFAR10**图片识别

- 网络结构：Conv -> ReLU -> Affine -> SVM
- 代码位于：【17/08simple_conv_cifar10.py】
- 作为演示，本例没有使用池化层

示例**3**：使用较复杂的卷积神经网络对**CIFAR10**图片识别

- 网络结构：Conv -> ReLU -> Batch Norm -> Max Pooling -> FC(1024) -> ReLUAffine -> SVM
- 代码位于：【17/09complex_conv_cifar10.py】
- 验证数据的识别正确率大约为0.65

示例**4**：使用较复杂的卷积神经网络对**CIFAR10**图片识别

- 网络结构： [conv->relu->batch norm->conv-relu->batch norm->-maxing pool] ->

  [conv->relu->batch norm->conv-relu->batch norm->-maxing pool] ->

  [affine-relu->batch norm->dropout] ->

  [affine-relu->batch norm->dropout]->affine

- 代码位于：【17/10complex_conv_by_layers.py】
- 本例还演示了如何将优化的模型保存到磁盘文件中
- 验证数据的识别正确率大约0.82，测试数据的识别正确率大约0.81

示例**5**：从磁盘文件装载优化模型进行计算

- 代码位于：【17/11restore_and_test_conv.py】