深度学习

原理推导与代码实现

朱明超

Email: deityrayleigh@gmail.com

Github: github.com/MingchaoZhu/DeepLearning

目录

第一章　线性代数

1.1　标量, 向量, 矩阵, 张量

1. **标量** (`Scalar`): 表示一个单独的**数**，通常用斜体小写字母表示，如 $s \in \mathbb{R}, n \in \mathbb{N}$。

2. **向量** (`Vector`)：表示**一列数**，这些数有序排列的，可以通过下标获取对应值，通常用粗体小写字母表示：$\boldsymbol{x} \in \mathbb{R}^n$ ，它表示元素取实数，且有 n 个元素，第一个元素表示为：x_1。将向量写成列向量的形式：

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} \tag{1.1}$$

有时需要向量的子集，例如第 $1, 3, 6$ 个元素，那么我们可以令集合 $S = \{1, 3, 6\}$ ，然后用 $\boldsymbol{x}_S$ 来表示这个子集。另外，我们用符号 - 表示集合的补集：$\boldsymbol{x}_{-1}$ 表示除 x_1 外 $\boldsymbol{x}$ 中的所有元素，$\boldsymbol{x}_{-S}$ 表示除 x_1, x_3, x_6 外 $\boldsymbol{x}$ 中的所有元素。

3. **矩阵** (`Matrix`): 表示一个**二维数组**，每个元素的下标由两个数字确定，通常用大写粗体字母表示：$\boldsymbol{A} \in \mathbb{R}^{m \times n}$ ，它表示元素取实数的 m 行 n 列矩阵，其元素可以表示为：$A_{1,1}, A_{m,n}$。我们用 : 表示矩阵的一行或者一列：$\boldsymbol{A}_{i,:}$ 为第 i 行，$\boldsymbol{A}_{:,j}$ 为第 j 列。

矩阵可以写成这样的形式：

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \tag{1.2}$$

有时我们需要对矩阵进行**逐元素操作**，如将函数 f 应用到 $\boldsymbol{A}$ 的所有元素上，此时我们用 $f(\boldsymbol{A})_{i,j}$ 表示。

4. **张量** (`Tensor`): **超过二维的数组**，我们用 A 表示张量，$\mathsf{A}_{i,j,k}$ 表示其元素（三维张量情况下）。

[1]:
```
import numpy as np
```

[2]:
```
# 标量
s = 5
# 向量
v = np.array([1,2])
# 矩阵
m = np.array([[1,2], [3,4]])
# 张量
t = np.array([
  [[1,2,3],[4,5,6],[7,8,9]],
  [[11,12,13],[14,15,16],[17,18,19]],
  [[21,22,23],[24,25,26],[27,28,29]],
  ])
print("标量: " + str(s))
print("向量: " + str(v))
print("矩阵: " + str(m))
print("张量: " + str(t))
```

```
标量: 5
向量: [1 2]
矩阵: [[1 2]
 [3 4]]
张量: [[[ 1  2  3]
  [ 4  5  6]
  [ 7  8  9]]
 [[11 12 13]
  [14 15 16]
  [17 18 19]]
```

```
[[21 22 23]
 [24 25 26]
 [27 28 29]]]
```

1.2 矩阵转置

矩阵转置 (Transpose) 相当于**沿着对角线翻转**，定义如下：

$$A_{i,j}^{\top} = A_{i,j} \tag{1.3}$$

矩阵转置的转置等于矩阵本身：

$$\left(\boldsymbol{A}^{\top}\right)^{\top} = \boldsymbol{A} \tag{1.4}$$

转置将矩阵的形状从 $m \times n$ 变成了 $n \times m$ 。

向量可以看成是**只有一列的矩阵**，为了方便，我们可以使用行向量加转置的操作，如：$\boldsymbol{x} = [x_1, x_2, x_3]^{\top}$。

标量也可以看成是**一行一列的矩阵**，其转置等于它自身：$a^{\top} = a$ 。

[3]:
```
A = np.array([[1.0,2.0],[1.0,0.0],[2.0,3.0]])
A_t = A.transpose()
print("A:", A)
print("A 的转置:", A_t)
```

```
A: [[1. 2.]
 [1. 0.]
 [2. 3.]]
A 的转置: [[1. 1. 2.]
 [2. 0. 3.]]
```

1.3 矩阵加法

加法即**对应元素相加**，要求两个矩阵的形状一样：

$$\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}, C_{i,j} = A_{i,j} + B_{i,j} \tag{1.5}$$

数乘即**一个标量**与**矩阵每个元素相乘**：

$$\boldsymbol{D} = a \cdot \boldsymbol{B} + c, D_{i,j} = a \cdot B_{i,j} + c \tag{1.6}$$

有时我们允许矩阵和向量相加的，得到一个矩阵，把 $\boldsymbol{b}$ 加到了 A 的每一行上，本质上是构造了一个将 $\boldsymbol{b}$ 按行复制的一个新矩阵，这种机制叫做广播 (Broadcasting)：

$$\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{b}, C_{i,j} = A_{i,j} + b_j \tag{1.7}$$

[4]:
```
a = np.array([[1.0,2.0],[3.0,4.0]])
b = np.array([[6.0,7.0],[8.0,9.0]])
print("矩阵相加:", a + b)
```

```
矩阵相加: [[ 7.  9.]
 [11. 13.]]
```

1.4 矩阵乘法

两个矩阵相乘得到第三个矩阵，我们需要 $\boldsymbol{A}$ 的形状为 $m \times n$，$\boldsymbol{B}$ 的形状为 $n \times p$，得到的矩阵为 $\boldsymbol{C}$ 的形状为 $m \times p$：

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B} \tag{1.8}$$

具体定义为

$$C_{i,j} = \sum_{k} A_{i,k} B_{k,j} \tag{1.9}$$

注意矩阵乘法不是元素对应相乘，元素对应相乘又叫 Hadamard 乘积，记作 $\boldsymbol{A} \odot \boldsymbol{B}$。

向量可以看作是列为 1 的矩阵，两个相同维数的向量 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 的点乘（Dot Product）或者内积，可以表示为 $\boldsymbol{x}^{\top}\boldsymbol{y}$。

我们也可以把矩阵乘法理解为：$C_{i,j}$ 表示 $\boldsymbol{A}$ 的第 i 行与 $\boldsymbol{B}$ 的第 j 列的点积。

[5]:
```
m1 = np.array([[1.0,3.0],[1.0,0.0]])
m2 = np.array([[1.0,2.0],[5.0,0.0]])
print("按矩阵乘法规则：", np.dot(m1, m2))
print("按逐元素相乘：", np.multiply(m1, m2))
print("按逐元素相乘：", m1*m2)
v1 = np.array([1.0,2.0])
v2 = np.array([4.0,5.0])
print("向量内积：", np.dot(v1, v2))
```

```
按矩阵乘法规则： [[16.  2.]
 [ 1.  2.]]
按逐元素相乘： [[1. 6.]
 [5. 0.]]
按逐元素相乘： [[1. 6.]
 [5. 0.]]
向量内积： 14.0
```

1.5 单位矩阵

为了引入矩阵的逆，我们需要先定义单位矩阵 (Identity Matrix)：单位矩阵乘以任意一个向量等于这个向量本身。记 $\boldsymbol{I}_n$ 为保持 n 维向量不变的单位矩阵，即：

$$\boldsymbol{I}_n \in \mathbb{R}^{n\times n}, \forall \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{I}_n\boldsymbol{x} = \boldsymbol{x} \tag{1.10}$$

单位矩阵的结构十分简单，所有的对角元素都为 1 ，其他元素都为 0，如：

$$\boldsymbol{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.11}$$

[6]:
```
np.identity(3)
```

[6]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

1.6 矩阵的逆

矩阵 $\boldsymbol{A}$ 的逆 (Inversion) 记作 $\boldsymbol{A}^{-1}$，定义为一个矩阵使得

$$\boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}_n \tag{1.12}$$

如果 $\boldsymbol{A}^{-1}$ 存在，那么线性方程组 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 的解为：

$$\boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{I}_n\boldsymbol{x} = \boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{1.13}$$

[7]:
```
A = [[1.0,2.0],[3.0,4.0]]
A_inv = np.linalg.inv(A)
print("A 的逆矩阵", A_inv)
```

```
A 的逆矩阵 [[-2.   1. ]
 [ 1.5 -0.5]]
```

1.7 范数

通常我们用范数 (norm) 来衡量向量，向量的 L^p 范数定义为：

$$\|\boldsymbol{x}\|_p = \left(\sum_i |x_i|^p\right)^{\frac{1}{p}}, p \in \mathbb{R}, p \geq 1 \tag{1.14}$$

L^2 范数，也称欧几里得范数 (`Euclidean norm`)，是**向量 $\boldsymbol{x}$ 到原点的欧几里得距离**。有时也用 L^2 范数的平方来衡量向量：$\boldsymbol{x}^\top\boldsymbol{x}$ 。事实上，平方 L^2 范数在计算上更为便利，例如它的对 $\boldsymbol{x}$ 梯度的各个分量只依赖于 $\boldsymbol{x}$ 的对应的各个分量，而 L^2 范数对 $\boldsymbol{x}$ 梯度的各个分量要依赖于整个 $\boldsymbol{x}$ 向量。

L^1 范数：L^2 范数并不一定适用于所有的情况，它在原点附近的增长就十分缓慢，因此不适用于需要区别 0 和非常小但是非 0 值的情况。L^1 范数就是一个比较好的选择，它在所有方向上的增长速率都是一样的，定义为：

$$\|\boldsymbol{x}\|_1 = \sum_i |x_i| \tag{1.15}$$

它经常使用在需要**区分 0 和非 0 元素**的情形中。

L^0 范数：如果需要衡量向量中非 0 元素的个数，但**它并不是一个范数** (不满足三角不等式和数乘)，此时 L^1 范数可以作为它的一个替代。

L^∞ 范数：它在数学上是向量元素绝对值的最大值，因此也被叫做 (`Max norm`)：

$$\|\boldsymbol{x}\|_\infty = \max_i |x_i| \tag{1.16}$$

有时我们想衡量一个矩阵，机器学习中通常使用的是 `F` 范数 (`Frobenius norm`)，其定义为：

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{i,j} A_{i,j}^2} \tag{1.17}$$

[8]:
```
a = np.array([1.0,3.0])
print("向量 2 范数", np.linalg.norm(a,ord=2))
print("向量 1 范数", np.linalg.norm(a,ord=1))
print("向量无穷范数", np.linalg.norm(a,ord=np.inf))
```

```
向量 2 范数 3.1622776601683795
向量 1 范数 4.0
向量无穷范数 3.0
```

[9]:
```
a = np.array([[1.0,3.0],[2.0,1.0]])
print("矩阵 F 范数", np.linalg.norm(a,ord="fro"))
```

```
矩阵 F 范数 3.872983346207417
```

1.8 特征值分解

如果一个 $n \times n$ 矩阵 $\boldsymbol{A}$ 有 n 组线性无关的单位特征向量 $\{\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}\}$ ，以及对应的特征值 $\lambda_1, \ldots, \lambda_n$。将这些特征向量按列拼接成一个矩阵：$\boldsymbol{V} = [\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(n)}]$，并将对应的特征值拼接成一个向量：$\boldsymbol{\lambda} = [\lambda_1, \ldots, \lambda_n]$ 。

$\boldsymbol{A}$ 的特征值分解 (`Eigendecomposition`) 为：

$$\boldsymbol{A} = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1} \tag{1.18}$$

注意：

- 不是所有的矩阵都有特征值分解
- 在某些情况下，实矩阵的特征值分解可能会得到复矩阵

[10]:
```
A = np.array([[1.0,2.0,3.0],
        [4.0,5.0,6.0],
        [7.0,8.0,9.0]])
# 计算特征值
print("特征值:", np.linalg.eigvals(A))
# 计算特征值和特征向量
eigvals,eigvectors = np.linalg.eig(A)
print("特征值:", eigvals)
print("特征向量:", eigvectors)
```

```
特征值: [ 1.61168440e+01 -1.11684397e+00 -3.73313677e-16]
特征值: [ 1.61168440e+01 -1.11684397e+00 -3.73313677e-16]
特征向量: [[-0.23197069 -0.78583024  0.40824829]
 [-0.52532209 -0.08675134 -0.81649658]
 [-0.8186735   0.61232756  0.40824829]]
```

1.9 奇异值分解

奇异值分解 (Singular Value Decomposition, SVD) 提供了另一种分解矩阵的方式，将其分解为奇异向量和奇异值。

与特征值分解相比，奇异值分解更加通用，所有的实矩阵都可以进行奇异值分解，而特征值分解只对某些方阵可以。

奇异值分解的形式为：

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\top} \tag{1.19}$$

若 $\boldsymbol{A}$ 是 $m \times n$ 的，那么 $\boldsymbol{U}$ 是 $m \times m$ 的，其列向量称为左奇异向量，而 $\boldsymbol{V}$ 是 $n \times n$ 的，其列向量称为右奇异向量，而 $\boldsymbol{\Sigma}$ 是 $m \times n$ 的一个对角矩阵，其对角元素称为矩阵 $\boldsymbol{A}$ 的奇异值。

事实上，左奇异向量是 $\boldsymbol{A}\boldsymbol{A}^{\top}$ 的特征向量，而右奇异向量是 $\boldsymbol{A}^{\top}\boldsymbol{A}$ 的特征向量，非 0 奇异值的平方是 $\boldsymbol{A}^{\top}\boldsymbol{A}$ 的非 0 特征值。

[11]:
```
A = np.array([[1.0,2.0,3.0],
        [4.0,5.0,6.0]])
U,D,V = np.linalg.svd(A)
print("U:", U)
print("D:", D)
print("V:", V)
```

```
U: [[-0.3863177  -0.92236578]
 [-0.92236578  0.3863177 ]]
D: [9.508032   0.77286964]
V: [[-0.42866713 -0.56630692 -0.7039467 ]
 [ 0.80596391  0.11238241 -0.58119908]
 [ 0.40824829 -0.81649658  0.40824829]]
```

1.10 PCA (主成分分析)

假设我们有 m 个数据点 $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)} \in \mathbb{R}^n$，对于每个数据点 $\boldsymbol{x}^{(i)}$ ，我们希望找到一个对应的点 $\boldsymbol{c}^{(i)} \in \mathbb{R}^l, l < n$ 去表示它 (相当于对它进行降维)，并且让损失的信息量尽可能少。

我们可以将这个过程看作是一个编码解码的过程，设编码和解码函数分别为 f, g ，则有 $f(\boldsymbol{x}) = \boldsymbol{c}, \boldsymbol{x} \approx g(f(\boldsymbol{x}))$。考虑一个线性解码函数 $g(\boldsymbol{c}) = \boldsymbol{D}\boldsymbol{c}, \boldsymbol{D} \in \mathbb{R}^{n \times l}$，为了计算方便，我们将这个矩阵的列向量约束为相互正交的。另一方面，考虑到存在尺度放缩的问题，我们将这个矩阵的列向量约束为具有单位范数来获得唯一解。

对于给定的 $\boldsymbol{x}$ ，我们需要找到信息损失最小的 $\boldsymbol{c}^{\star}$，即求解：

$$\boldsymbol{c}^{\star} = \arg\min_{\boldsymbol{c}} \|\boldsymbol{x} - g(\boldsymbol{c})\|_2 = \arg\min_{\boldsymbol{c}} \|\boldsymbol{x} - g(\boldsymbol{c})\|_2^2 \tag{1.20}$$

这里我们用二范数来衡量信息的损失。展开之后我们有：

$$\|\boldsymbol{x} - g(\boldsymbol{c})\|_2^2 = (\boldsymbol{x} - g(\boldsymbol{c}))^{\top}(\boldsymbol{x} - g(\boldsymbol{c})) = \boldsymbol{x}^{\top}\boldsymbol{x} - 2\boldsymbol{x}^{\top}g(\boldsymbol{c}) + g(\boldsymbol{c})^{\top}g(\boldsymbol{c}) \tag{1.21}$$

结合 $g(\boldsymbol{c})$ 的表达式，忽略不依赖 $\boldsymbol{c}$ 的 $\boldsymbol{x}^{\top}\boldsymbol{x}$ 项，我们有：

$$\begin{aligned} \boldsymbol{c}^{\star} &= \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^{\top}\boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^{\top}\boldsymbol{D}^{\top}\boldsymbol{D}\boldsymbol{c} \\ &= \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^{\top}\boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^{\top}\boldsymbol{I}_l\boldsymbol{c} \\ &= \arg\min_{\boldsymbol{c}} -2\boldsymbol{x}^{\top}\boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^{\top}\boldsymbol{c} \end{aligned} \tag{1.22}$$

这里 $\boldsymbol{D}$ 具有单位正交性。

对 $\boldsymbol{c}$ 求梯度，并令其为零，我们有：

$$\begin{aligned} \nabla_{\boldsymbol{c}}(-2\boldsymbol{x}^{\top}\boldsymbol{D}\boldsymbol{c} + \boldsymbol{c}^{\top}\boldsymbol{c}) &= \boldsymbol{0} \\ -2\boldsymbol{D}^{\top}\boldsymbol{x} + 2\boldsymbol{c} &= \boldsymbol{0} \\ \boldsymbol{c} &= \boldsymbol{D}^{\top}\boldsymbol{x} \end{aligned} \tag{1.23}$$

因此，我们的编码函数为：

$$f(\boldsymbol{x}) = \boldsymbol{D}^{\top}\boldsymbol{x} \tag{1.24}$$

此时通过编码解码得到的重构为：

$$r(\boldsymbol{x}) = g(f(\boldsymbol{x})) = \boldsymbol{D}\boldsymbol{D}^{\top}\boldsymbol{x} \tag{1.25}$$

接下来求解最优的变换 $\boldsymbol{D}$ 。由于我们需要将 $\boldsymbol{D}$ 应用到所有的 $\boldsymbol{x}_i$ 上，所以我们需要最优化：

$$\begin{aligned} &\boldsymbol{D}^\star = \arg\min_{\boldsymbol{D}} \sqrt{\sum_{i,j}(\boldsymbol{x}_j^{(i)} - r(\boldsymbol{x}^{(i)})_j)^2} \\ &s.t.\ \boldsymbol{D}^\top \boldsymbol{D} = \boldsymbol{I}_l \end{aligned} \tag{1.26}$$

为了方便，我们考虑 $l = 1$ 的情况，此时问题简化为：

$$\begin{aligned} &\boldsymbol{d}^\star = \arg\min_{\boldsymbol{d}} \sum_{i}(\boldsymbol{x}_j^{(i)} - \boldsymbol{d}\boldsymbol{d}^\top \boldsymbol{x}^{(i)}))^2 \\ &s.t.\ \boldsymbol{d}^\top \boldsymbol{d} = 1 \end{aligned} \tag{1.27}$$

考虑 F 范数，并进一步的推导：

$$\begin{aligned} &\boldsymbol{d}^\star = \arg\max_{\boldsymbol{d}} \mathrm{Tr}(\boldsymbol{d}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{d}) \\ &s.t.\ \boldsymbol{d}^\top \boldsymbol{d} = 1 \end{aligned} \tag{1.28}$$

优化问题可以用特征分解来求解。但实际计算时，我们会采用如下方式计算：

PCA 将输入 $\boldsymbol{x}$ 投影表示成 $\boldsymbol{c}$。$\boldsymbol{c}$ 是比原始输入维数更低的表示，同时使得元素之间线性无关。假设有一个 $m \times n$ 的矩阵 $\boldsymbol{X}$，数据的均值为零，即 $\mathbb{E}[\boldsymbol{x}] = 0$，$\boldsymbol{X}$ 对应的无偏样本协方差矩阵：$\mathrm{Var}\,[\boldsymbol{x}] = \frac{1}{m-1}\boldsymbol{X}^\top \boldsymbol{X}$。

PCA 是**通过线性变换找到一个 $\mathrm{Var}\,[\boldsymbol{c}]$ 是对角矩阵**的表示 $\boldsymbol{c} = \boldsymbol{V}^\top \boldsymbol{x}$ ，**矩阵 $\boldsymbol{X}$ 的主成分可以通过奇异值分解 (SVD) 得到**，也就是说主成分是 $\boldsymbol{X}$ 的右奇异向量。假设 $\boldsymbol{V}$ 是 $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ 奇异值分解的右奇异向量，我们得到原来的特征向量方程：

$$\boldsymbol{X}^\top \boldsymbol{X} = (\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top)^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top = \boldsymbol{V}\boldsymbol{\Sigma}^\top \boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top = \boldsymbol{V}\boldsymbol{\Sigma}^2 \boldsymbol{V}^\top \tag{1.29}$$

因为根据奇异值的定义 $\boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{I}$ 。因此 $\boldsymbol{X}$ 的方差可以表示为：$\mathrm{Var}\,[\boldsymbol{x}] = \frac{1}{m-1}\boldsymbol{X}^\top \boldsymbol{X} = \frac{1}{m-1}\boldsymbol{V}\boldsymbol{\Sigma}^2 \boldsymbol{V}^\top$。

所以 $\boldsymbol{c}$ 的协方差满足：$\mathrm{Var}\,[\boldsymbol{c}] = \frac{1}{m-1}\boldsymbol{C}^\top \boldsymbol{C} = \frac{1}{m-1}\boldsymbol{V}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{V} = \frac{1}{m-1}\boldsymbol{V}^\top \boldsymbol{V}\boldsymbol{\Sigma}^2 \boldsymbol{V}^\top \boldsymbol{V} = \frac{1}{m-1}\boldsymbol{\Sigma}^2$，因为根据奇异值定义 $\boldsymbol{V}^\top \boldsymbol{V} = \boldsymbol{I}$ 。$\boldsymbol{c}$ 的协方差是对角的，$\boldsymbol{c}$ 中的元素是彼此无关的。

以 *iris* 数据为例，展示 PCA 的使用。

[12]:
```
import pandas as pd
import numpy as np
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
%matplotlib inline
```

[13]:
```
# 载入数据
iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['label'] = iris.target
df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
df.label.value_counts()
```

[13]:
```
2    50
1    50
0    50
Name: label, dtype: int64
```

[14]:
```
# 查看数据
df.tail()
```

[14]:

	sepal length	sepal width	petal length	petal width	label
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

[15]:
```
# 查看数据
X = df.iloc[:, 0:4]
y = df.iloc[:, 4]
```

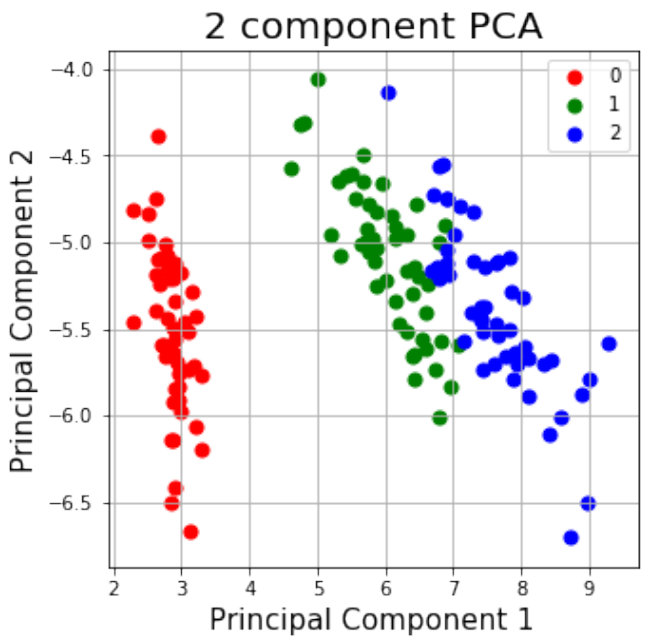
```
print("查看第一个数据： \n", X.iloc[0, 0:4])
print("查看第一个标签: \n", y.iloc[0])
```

```
查看第一个数据:
 sepal length    5.1
sepal width     3.5
petal length    1.4
petal width     0.2
Name: 0, dtype: float64
查看第一个标签:
 0
```

[16]:
```
class PCA():
    def __init__(self):
        pass

    def fit(self, X, n_components):
        n_samples = np.shape(X)[0]
        covariance_matrix = (1 / (n_samples-1)) * (X - X.mean(axis=0)).T.dot(X - X.mean(axis=0))
        # 对协方差矩阵进行特征值分解
        eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
        # 对特征值（特征向量）从大到小排序
        idx = eigenvalues.argsort()[::-1]
        eigenvalues = eigenvalues[idx][:n_components]
        eigenvectors = np.atleast_1d(eigenvectors[:, idx])[:, :n_components]
        # 得到低维表示
        X_transformed = X.dot(eigenvectors)
        return X_transformed
```

[17]:
```
model = PCA()
Y = model.fit(X, 2)
```

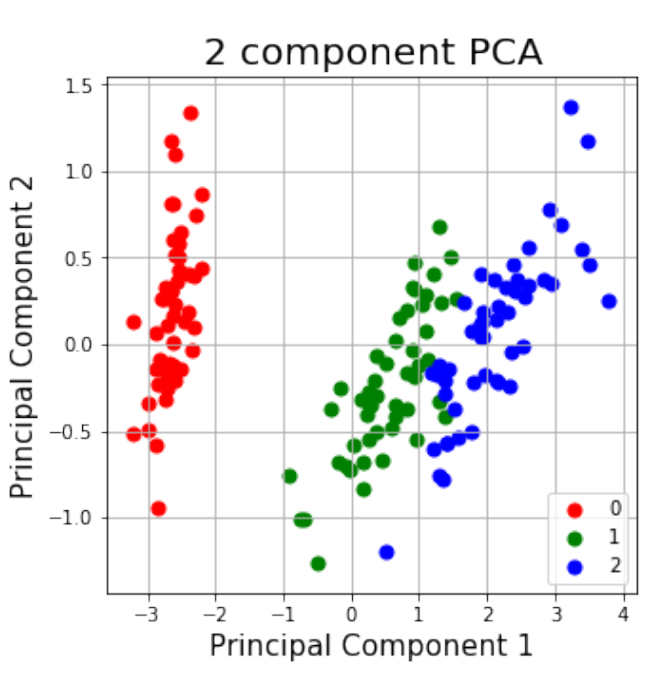
[18]:
```
principalDf = pd.DataFrame(np.array(Y),
                           columns=['principal component 1', 'principal component 2'])
Df = pd.concat([principalDf, y], axis = 1)
fig = plt.figure(figsize = (5,5))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA', fontsize = 20)

targets = [0, 1, 2]
# ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colors = ['r', 'g', 'b']
for target, color in zip(targets,colors):
    indicesToKeep = Df['label'] == target
    ax.scatter(Df.loc[indicesToKeep, 'principal component 1']
               , Df.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)
ax.legend(targets)
ax.grid()
```



使用 *sklearn* 包实现 PCA

[19]:
```
from sklearn.decomposition import PCA as sklearnPCA
sklearn_pca = sklearnPCA(n_components=2)
Y = sklearn_pca.fit_transform(X)
```

[20]:
```
principalDf = pd.DataFrame(data = np.array(Y), columns = ['principal component 1', 'principal component 2'])
Df = pd.concat([principalDf, y], axis = 1)
fig = plt.figure(figsize = (5,5))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA', fontsize = 20)

targets = [0, 1, 2]
# ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colors = ['r', 'g', 'b']
for target, color in zip(targets,colors):
    indicesToKeep = Df['label'] == target
    ax.scatter(Df.loc[indicesToKeep, 'principal component 1']
               , Df.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)
ax.legend(targets)
ax.grid()
```



[21]:
```
import numpy, pandas, matplotlib, sklearn
print("numpy:", numpy.__version__)
print("pandas:", pandas.__version__)
```

```
print("matplotlib:", matplotlib.__version__)
print("sklearn:", sklearn.__version__)
```

```
numpy: 1.14.5
pandas: 0.25.1
matplotlib: 3.1.1
sklearn: 0.21.3
```

第二章　概率与信息论

2.1　概率

2.1.1　概率与随机变量

- **频率学派概率** (`Frequentist Probability`)：认为概率和事件发生的频率相关。
- **贝叶斯学派概率** (`Bayesian Probability`)：认为概率是对某件事发生的确定程度，可以理解成是确信的程度。
- **随机变量** (`Random Variable`)：一个可能随机取不同值的变量。例如：抛掷一枚硬币，出现正面或者反面的结果。

2.1.2　概率分布

概率质量函数

概率质量函数 (`Probability Mass Function`)：对于离散型变量，我们先定义一个随机变量，然后用 ~ 符号来说明它遵循的分布：$\mathrm{x} \sim P(\mathrm{x})$ ，函数 P 是随机变量 x 的 PMF。

例如, 考虑一个离散型 x 有 k 个不同的值，我们可以假设 x 是均匀分布的 (也就是将它的每个值视为等可能的)，通过将它的 PMF 设为：

$$P(\mathrm{x} = x_i) = \frac{1}{k} \tag{2.1}$$

对于所有的 i 都成立。

概率密度函数

当研究的对象是连续型时，我们可以引入同样的概念。如果一个函数 p 是概率密度函数 (`Probability Density Function`)：

- 分布满足非负性条件：$\forall x \in \mathrm{x}, p(x) \geq 0$
- 分布满足归一化条件：$\int_{-\infty}^{\infty} p(x)\,dx = 1$

例如在 (a, b) 上的均匀分布：

$$U(x; a, b) = \frac{\mathbf{1}_{ab}(x)}{b-a}$$

这里 $\mathbf{1}_{ab}(x)$ 表示在 (a, b) 内为 1，否则为 0。

累积分布函数

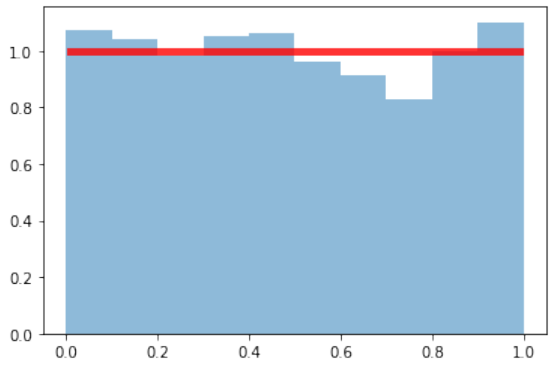
累积分布函数 (`Cummulative Distribution Function`) 表示对小于 x 的概率的积分：

$$\mathrm{CDF}(x) = \int_{-\infty}^{x} p(t)\,dt \tag{2.2}$$

[21]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import uniform
%matplotlib inline
```

[22]:
```
# 生成样本
fig, ax = plt.subplots(1, 1)
r = uniform.rvs(loc=0, scale=1, size=1000)
ax.hist(r, density=True, histtype='stepfilled', alpha=0.5)
# 均匀分布 pdf
x = np.linspace(uniform.ppf(0.01), uniform.ppf(0.99), 100)
ax.plot(x, uniform.pdf(x), 'r-', lw=5, alpha=0.8, label='uniform pdf')
```

[22]:
```
[<matplotlib.lines.Line2D at 0x118521b38>]
```



2.1.3 条件概率与条件独立

边缘概率 (Marginal Probability)：如果我们知道了一组变量的联合概率分布，但想要了解其中一个子集的概率分布。这种定义在子集上的概率分布被称为边缘概率分布。

$$\forall x \in \mathrm{x},\ P\left(\mathrm{x}=x\right)=\sum_{y} P\left(\mathrm{x}=x, \mathrm{y}=y\right) \tag{2.3}$$

条件概率 (Conditional Probability)：在很多情况下，我们感兴趣的是某个事件，在给定其他事件发生时出现的概率。这种概率叫做条件概率。我们将给定 $\mathrm{x}=x$，$\mathrm{y}=y$ 发生的条件概率记为 $P(\mathrm{y}=y \mid \mathrm{x}=x)$。这个条件概率可以通过下面的公式计算:

$$P\left(\mathrm{y}=y \mid \mathrm{x}=x\right)=\frac{P\left(\mathrm{y}=y, \mathrm{x}=x\right)}{P\left(\mathrm{x}=x\right)} \tag{2.4}$$

条件概率的链式法则 (Chain Rule of Conditional Probability)：任何多维随机变量的联合概率分布，都可以分解成只有一个变量的条件概率相乘的形式:

$$P\left(x_{1}, ..., x_{n}\right)=P\left(x_{1}\right) \prod_{i=2}^{n} P(x_{i} \mid x_{1}, ..., x_{i-1}) \tag{2.5}$$

独立性 (Independence)：两个随机变量 x 和 y，如果它们的概率分布可以表示成两个因子的乘积形式，并且一个因子只包含 x 另一个因子只包含 y，我们就称这两个随机变量是相互独立的:

$$\forall x \in \mathrm{x}, y \in \mathrm{y},\ p(\mathrm{x}=x, \mathrm{y}=y)=p(\mathrm{x}=x) p(\mathrm{y}=y) \tag{2.6}$$

条件独立性 (Conditional Independence)：如果关于 x 和 y 的条件概率分布对于 z 的每一个值都可以写成乘积的形式，那么这两个随机变量 x 和 y 在给定随机变量 z 时是条件独立的。

$$\forall x \in \mathrm{x}, y \in \mathrm{y}, z \in \mathrm{z},\ p(\mathrm{x}=x, \mathrm{y}=y \mid \mathrm{z}=z)=p(\mathrm{x}=x \mid \mathrm{z}=z) p(\mathrm{y}=y \mid \mathrm{z}=z) \tag{2.7}$$

2.1.4 随机变量的度量

期望 (Expectation)：函数 f 关于**概率分布** $P(\mathrm{x})$ 或 $p(\mathrm{x})$ 的期望表示为由概率分布产生 x，再计算 f 作用到 x 上后 $f(x)$ 的平均值。对于离散型随机变量，这可以通过求和得到：

$$\mathbb{E}_{\mathrm{x} \sim P}[f(x)]=\sum_{x} \mathrm{P}(x) f(x) \tag{2.8}$$

对于连续型随机变量可以通过求积分得到：

$$\mathbb{E}_{\mathrm{x} \sim p}[f(x)]=\int P(x) f(x) d x \tag{2.9}$$

另外，期望是线性的：

$$\mathbb{E}_{\mathrm{x}}[\alpha f(x)+\beta g(x)]=\alpha \mathbb{E}_{\mathrm{x}}[f(x)]+\beta \mathbb{E}_{\mathrm{x}}[f(x)] \tag{2.10}$$

方差 (Variance)：衡量的是当我们对 x 依据它的概率分布进行采样时，随机变量 x 的函数值会呈现多大的差异，描述采样得到的函数值在期望上下的波动程度：

$$\operatorname{Var}\left(f(x)\right)=\mathbb{E}[(f(x)-\mathbb{E}[f(x)])^{2}] \tag{2.11}$$

将方差开平方即为**标准差** (Standard Deviation)。

协方差 (Covariance)：用于衡量两组值之间的**线性相关程度**：

$$\operatorname{Cov}\left(f(x), g(y)\right)=\mathbb{E}[(f(x)-\mathbb{E}[f(x)])(g(y)-\mathbb{E}[g(y)])] \tag{2.12}$$

注意，独立比零协方差要求更强，因为**独立还排除了非线性的相关**。

[23]:
```
x = np.array([1,2,3,4,5,6,7,8,9])
y = np.array([9,8,7,6,5,4,3,2,1])
Mean = np.mean(x)
```

```
Var = np.var(x) # 默认总体方差
Var_unbias = np.var(x, ddof=1)  # 样本方差（无偏方差）
Cov = np.cov(x,y)
Mean, Var, Var_unbias, Cov
```

[23]:
```
(5.0, 6.666666666666667, 7.5, array([[ 7.5, -7.5],
        [-7.5,  7.5]]))
```

2.1.5 常用概率分布

伯努利分布 (两点分布)

伯努利分布 (Bernoulli Distribution) 是**单个二值随机变量**的分布，随机变量只有两种可能。它由一个参数 $\phi \in [0,1]$ 控制，ϕ 给出了随机变量等于 1 的概率：

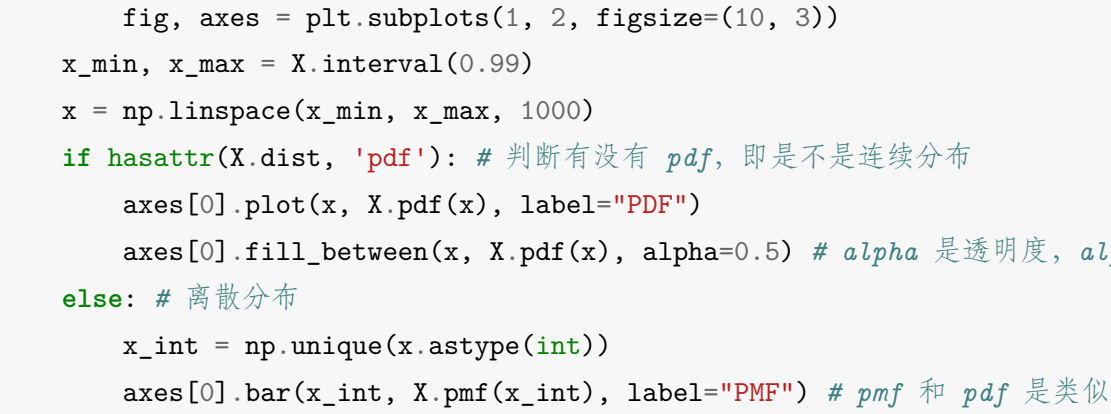
$$
\begin{aligned}
P(\mathrm{x}=1) &= \phi \\
P(\mathrm{x}=0) &= 1-\phi \\
P(\mathrm{x}=x) &= \phi^{x}(1-\phi)^{1-x}
\end{aligned}
\tag{2.13}
$$

表示一次试验的结果要么成功要么失败。

[24]:
```
def plot_distribution(X, axes=None):
    """ 给定随机变量，绘制 PDF，PMF，CDF"""
    if axes is None:
        fig, axes = plt.subplots(1, 2, figsize=(10, 3))
    x_min, x_max = X.interval(0.99)
    x = np.linspace(x_min, x_max, 1000)
    if hasattr(X.dist, 'pdf'): # 判断有没有 pdf，即是不是连续分布
        axes[0].plot(x, X.pdf(x), label="PDF")
        axes[0].fill_between(x, X.pdf(x), alpha=0.5) # alpha 是透明度，alpha=0 表示 100% 透明，alpha=100 表示完全不透明
    else: # 离散分布
        x_int = np.unique(x.astype(int))
        axes[0].bar(x_int, X.pmf(x_int), label="PMF") # pmf 和 pdf 是类似的
    axes[1].plot(x, X.cdf(x), label="CDF")
    for ax in axes:
        ax.legend()
    return axes
```

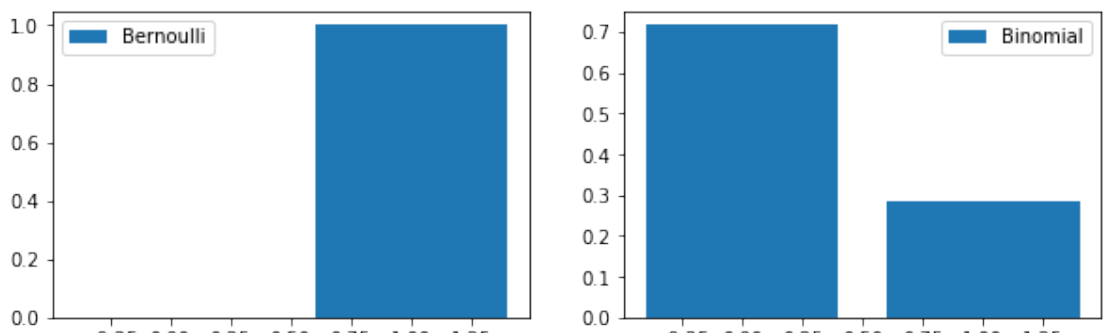
[25]:
```
from scipy.stats import bernoulli
fig, axes = plt.subplots(1, 2, figsize=(10, 3)) # 画布
p = 0.3
X = bernoulli(p) # 伯努利分布
plot_distribution(X, axes=axes)
```

[25]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x1187d3d30>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x11885ccf8>],
      dtype=object)
```



[26]:
```
# 产生成功的概率
possibility = 0.3
def trials(n_samples):
    samples = np.random.binomial(n_samples, possibility)  # 成功的次数
```

```
    proba_zero = (n_samples-samples)/n_samples
    proba_one = samples/n_samples
    return [proba_zero, proba_one]

fig, axes = plt.subplots(1, 2, figsize=(10, 3))
# 一次试验， 伯努利分布
n_samples = 1
axes[0].bar([0, 1], trials(n_samples), label="Bernoulli")
# n 次试验， 二项分布
n_samples = 1000
axes[1].bar([0, 1], trials(n_samples), label="Binomial")
for ax in axes:
    ax.legend()
```



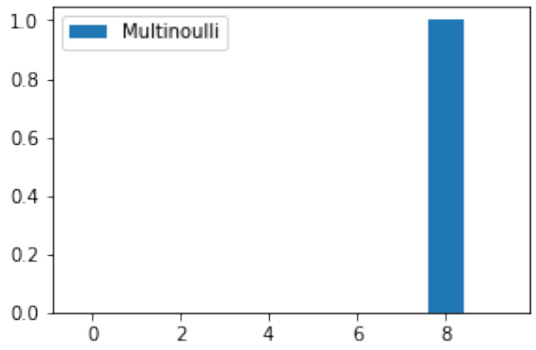
范畴分布 (分类分布)

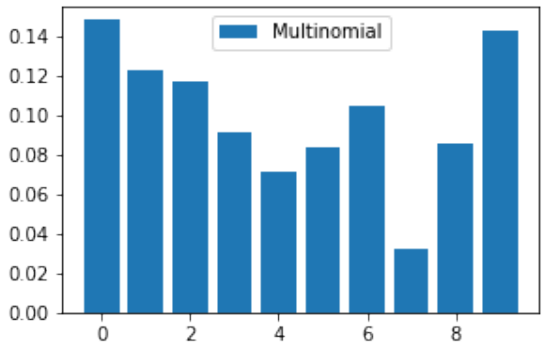
范畴分布 (Multinoulli Distribution) 是指在**具有 k 个不同值**的**单个离散型随机变量**上的分布:

$$p(\mathrm{x}=x)=\prod_i \phi_i^{x_i} \tag{2.14}$$

例如每次试验的结果就可以记为一个 k 维的向量，只有此次试验的结果对应的维度记为 1，其他记为 0。

[27]:
```
def k_possibilities(k):
    """
    随机产生一组 10 维概率向量
    """
    res = np.random.rand(k)
    _sum = sum(res)
    for i, x in enumerate(res):
        res[i] = x / _sum
    return res

fig, axes = plt.subplots(1, 2, figsize=(10, 3))
# 一次试验， 范畴分布
k, n_samples = 10, 1
samples = np.random.multinomial(n_samples, k_possibilities(k))  # 各维度“成功”的次数
axes[0].bar(range(len(samples)), samples/n_samples, label="Multinoulli")
# n 次试验， 多项分布
n_samples = 1000
samples = np.random.multinomial(n_samples, k_possibilities(k))
axes[1].bar(range(len(samples)), samples/n_samples, label="Multinomial")
for ax in axes:
    ax.legend()
```





高斯分布 (正态分布)

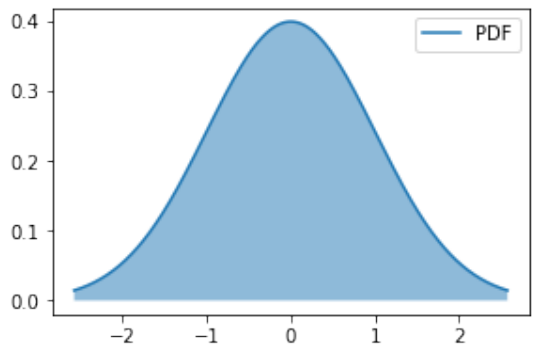
高斯分布 (`Gaussian Distribution`) 或正态分布 (`Normal Distribution`) 形式如下：

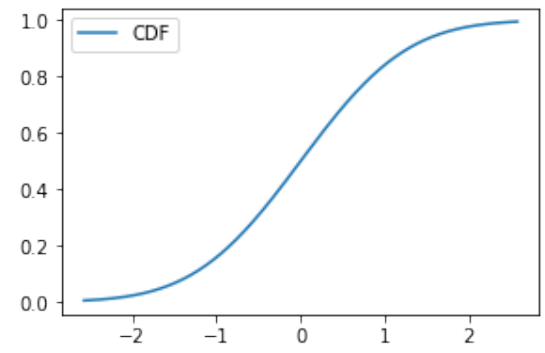
$$N(x;\mu,\sigma^2)=\sqrt{\frac{1}{2\pi\sigma^2}}\exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) \tag{2.15}$$

有时也会用 $\beta=\frac{1}{\sigma^2}$ 表示分布的精度 (`precision`)。中心极限定理 (`Central Limit Theorem`) 认为，大量的独立随机变量的和近似于一个正态分布，因此可以认为噪声是属于正态分布的。

[28]:
```
from scipy.stats import norm
fig, axes = plt.subplots(1, 2, figsize=(10, 3)) # 画布
mu, sigma = 0, 1
X = norm(mu, sigma) # 标准正态分布
plot_distribution(X, axes=axes)
```

[28]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x118b76710>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x118c54978>],
      dtype=object)
```





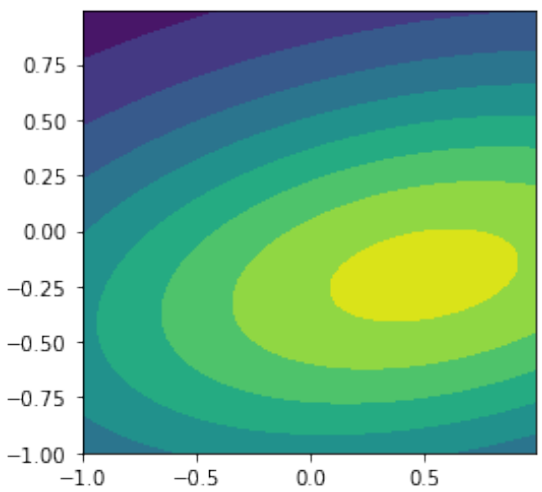
多元高斯分布 (多元正态分布)

多元正态分布 (`Multivariate Normal Distribution`) 形式如下：

$$N(x;\mu,\Sigma)=\sqrt{\frac{1}{(2\pi)^n\det(\Sigma)}}\exp\left(-\frac{1}{2}(x-\mu)^\top\Sigma^{-1}(x-\mu)\right) \tag{2.16}$$

[29]:
```
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
x, y = np.mgrid[-1:1:.01, -1:1:.01]
pos = np.dstack((x, y))
fig = plt.figure(figsize=(4,4))
axes = fig.add_subplot(111)
mu = [0.5, -0.2]  # 均值
sigma = [[2.0, 0.3], [0.3, 0.5]]  # 协方差矩阵
X = multivariate_normal(mu, sigma)
axes.contourf(x, y, X.pdf(pos))
```

[29]:
```
<matplotlib.contour.QuadContourSet at 0x118cd4438>
```



指数分布

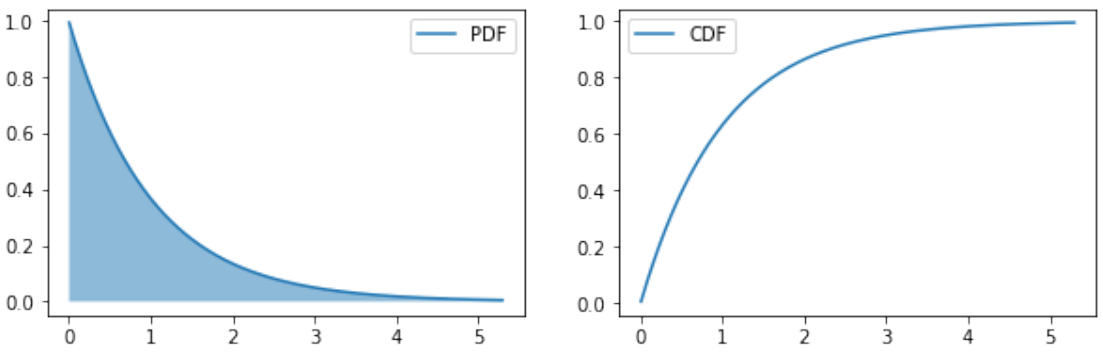
指数分布 (Exponential Distribution) 形式如下：

$$p(x;\lambda) = \lambda 1_{x\geq 0}\exp(-\lambda x) \tag{2.17}$$

是用于在 $x=0$ 处获得最高的概率的分布，其中 $\lambda > 0$ 是分布的一个参数，常被称为率参数 (Rate Parameter)。

[30]:
```
from scipy.stats import expon
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
# 定义 scale = 1 / lambda
X = expon(scale=1)
plot_distribution(X, axes=axes)
```

[30]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x11900f438>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x1190956d8>],
      dtype=object)
```



拉普拉斯分布

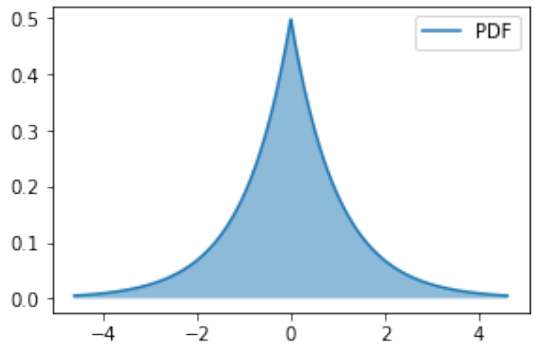
拉普拉斯分布（Laplace Distribution）形式如下：

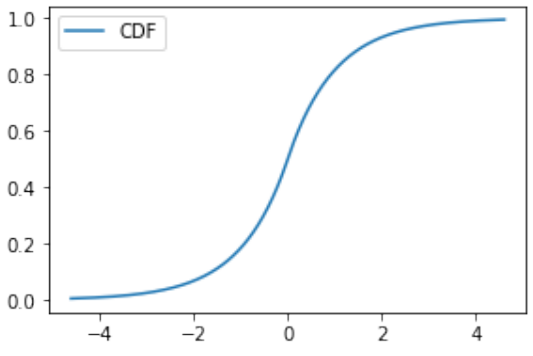
$$\text{Laplace}(x;\mu,\gamma) = \frac{1}{2\gamma}\exp\left(-\frac{|x-\mu|}{\gamma}\right) \tag{2.18}$$

这也是可以在一个点获得比较高的概率的分布。

[31]:
```
from scipy.stats import laplace
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
mu, gamma = 0, 1
X = laplace(loc=mu, scale=gamma)
plot_distribution(X, axes=axes)
```

[31]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x11913eb00>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x118a3b2b0>],
      dtype=object)
```





Dirac 分布

Dirac delta 函数定义为 $p(x) = \delta(x - \mu)$，这是一个泛函数。它常被用于组成**经验分布** (`Empirical Distribution`)：

$$\hat{p}(x) = \frac{1}{m}\sum_{i=1}^{m}\delta(x - x^{(i)}) \tag{2.19}$$

2.1.6 常用函数的有用性质

logistic sigmoid 函数

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{2.20}$$

logistic sigmoid 函数通常用来产生伯努利分布中的参数 ϕ ，因为它的范围是 $(0, 1)$，处在 ϕ 的有效取值范围内。sigmoid 函数在变量取绝对值非常大的正值或负值时会出现饱和 (`Saturate`) 现象，意味着函数会变得很平，并且对输入的微小改变会变得不敏感。

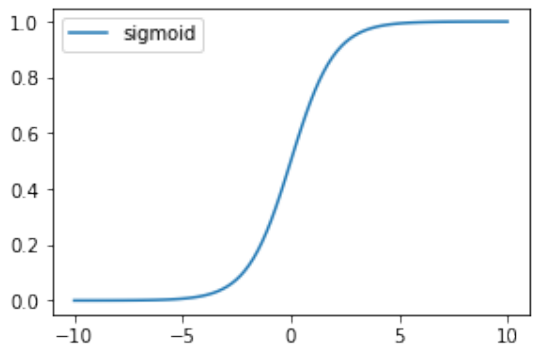
softplus 函数

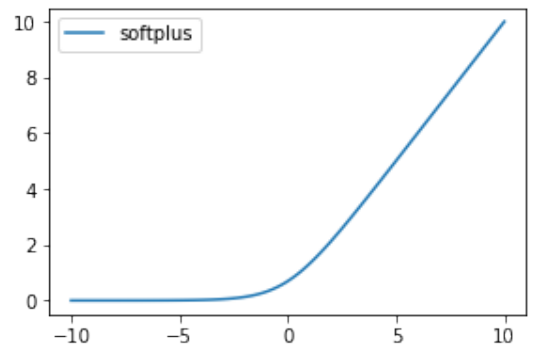
$$\zeta(x) = \log(1 + \exp(x)) \tag{2.21}$$

softplus 函数可以用来产生正态分布的 β 和 σ 参数，因为它的范围是 $(0, \infty)$。当处理包含 sigmoid 函数的表达式时它也经常出现。softplus 函数名来源于它是另外一个函数的平滑 (或 “软化”) 形式，这个函数是:

$$x^{+} = \max(0, x) \tag{2.22}$$

[32]:
```
x = np.linspace(-10, 10, 100)
sigmoid = 1/(1 + np.exp(-x))
softplus = np.log(1 + np.exp(x))
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
axes[0].plot(x, sigmoid, label='sigmoid')
axes[1].plot(x, softplus, label='softplus')
for ax in axes:
    ax.legend()
```





2.2 信息论

信息论背后的思想：一件不太可能的事件比一件比较可能的事件更有信息量。

信息 (`Information`) 需要满足的三个条件：

- 比较可能发生的事件的信息量要少。
- 比较不可能发生的事件的信息量要大。
- 独立发生的事件之间的信息量应该是可以叠加的。例如，投掷的硬币两次正面朝上传递的信息量，应该是投掷一次硬币正面朝上的信息量的两倍。

自信息 (`Self-Information`)：对事件 $x = x$，我们定义：

$$I(x) = -\log\ P(x) \tag{2.23}$$

自信息满足上面三个条件，单位是奈特 (nats) （底为 e）

香农熵 (`Shannon Entropy`)：上述的自信息只包含一个事件的信息，而**对于整个概率分布** P，不确定性可以这样衡量：

$$\mathbb{E}_{x\sim P}[\mathrm{I}\left(x\right)] = -\mathbb{E}_{x\sim P}[\log P\left(x\right)] \tag{2.24}$$

也可以表示成 H(P)。香农熵是编码原理中最优编码长度。

多个随机变量：

- **联合熵** (`Joint Entropy`)：表示同时考虑多个事件的条件下（即考虑联合分布概率）的熵。

$$H(X,Y) = -\sum_{x,y} P(x,y)\log\left(P(x,y)\right) \tag{2.25}$$

- **条件熵** (`Conditional Entropy`)：表示某件事情已经发生的情况下，另外一件事情的熵。

$$H(X|Y) = -\sum_{y} P(y)\sum_{x} P(x|y)\log\left(P(x|y)\right) \tag{2.26}$$

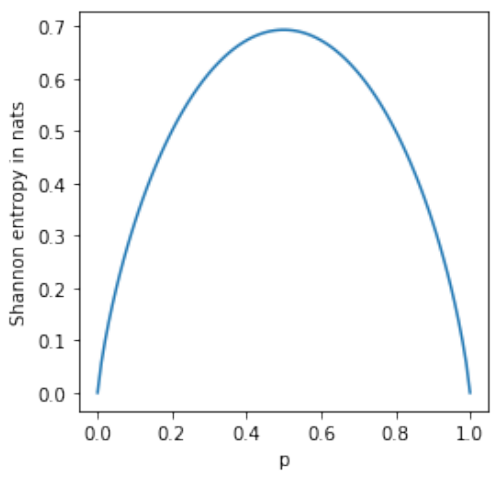
- **互信息** (`Mutual Information`)：表示两个事件的信息**相交**的部分。

$$I(X,Y) = H(X) + H(Y) - H(X,Y) \tag{2.27}$$

- **信息变差** (`Variation of Information`)：表示两个事件的信息**不相交**的部分。

$$V(X,Y) = H(X,Y) - I(X,Y) \tag{2.28}$$

[33]:
```
p = np.linspace(1e-6,1-1e-6,100)
entropy = (p-1)*np.log(1-p)-p*np.log(p)
plt.figure(figsize=(4,4))
plt.plot(p,entropy)
plt.xlabel('p')
plt.ylabel('Shannon entropy in nats')
plt.show()
```



[34]:
```
def H(sentence):
    """
    最优编码长度
    """
    entropy = 0
    # 这里有 256 个可能的 ASCII 符号
    for character_i in range(256):
        Px = sentence.count(chr(character_i))/len(sentence)
        if Px > 0:
```

```
            entropy += -Px * math.log(Px,2) # 注:log 以 2 为底
    return entropy
```

[35]:
```
import random
import math
# 只用 64 个字符
simple_message = "".join([chr(random.randint(0,64)) for i in range(500)])
print(simple_message)
H(simple_message)
```

```
-.218?
        2>  -=?$'    <$<
:&5<>=>
/'+#)>@((931":< > 5)
4$%79@
)$6#63(?1.,&4'%%<
7 )1,
<%.)*4 :57+ ? 3#
```

[35]:
```
5.939286791378741
```

KL 散度 (Kullback-Leibler Divergence) 用于衡量两个分布 $P(\mathrm{x})$ 和 $Q(\mathrm{x})$ 之间的差距：

$$\mathrm{D}_{\mathrm{KL}}(P||Q) = \mathbb{E}_{x\sim P}[\log \frac{P(x)}{Q(x)}] = \mathbb{E}_{x\sim P}[\log P(x) - \log Q(x)] \tag{2.29}$$

注意 $\mathrm{D}_{\mathrm{KL}}(\mathrm{P}||\mathrm{Q}) \neq \mathrm{D}_{\mathrm{KL}}(\mathrm{Q}||\mathrm{P})$，不满足对称性。

交叉熵 (Cross Entropy)：

$$H(P,Q) = H(P) + \mathrm{D}_{\mathrm{KL}}(P||Q) = -\mathbb{E}_{x\sim P}[\log Q(x)] \tag{2.30}$$

假设 P 是真实分布，Q 是模型分布，那么最小化交叉熵 $\mathrm{H}(\mathrm{P},\mathrm{Q})$ 可以让模型分布逼近真实分布。

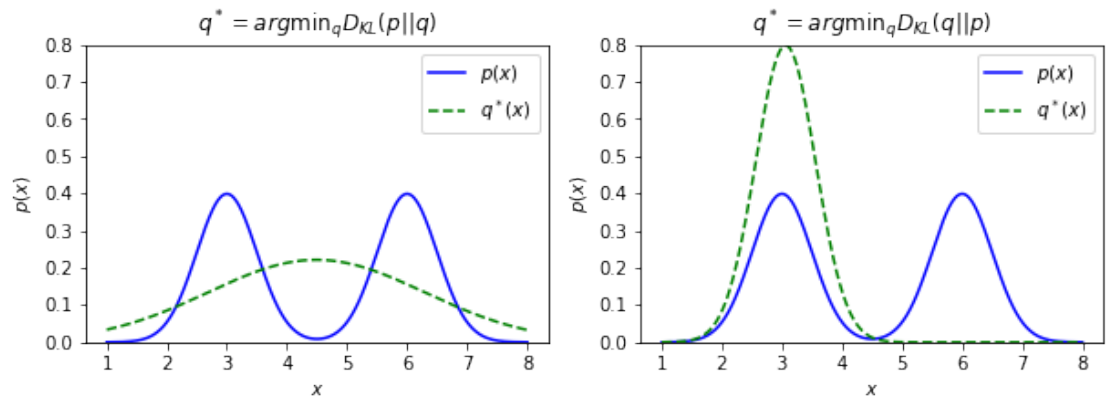
[36]:
```
# KL 定义
from scipy.stats import entropy  # 内置 kl
def kl(p, q):
    """
    D(P || Q)
    """
    p = np.asarray(p, dtype=np.float)
    q = np.asarray(q, dtype=np.float)
    return np.sum(np.where(p != 0, p * np.log(p / q), 0))
```

[37]:
```
# 测试
p = [0.1, 0.9]
q = [0.1, 0.9]
print(entropy(p, q) == kl(p, q))
```

```
True
```

[38]:
```
# D(P||Q) 与 D(Q||P) 比较
x = np.linspace(1, 8, 500)
y1 = norm.pdf(x, 3, 0.5)
y2 = norm.pdf(x, 6, 0.5)
p = y1 + y2  # 构造 p(x)
KL_pq, KL_qp = [], []
q_list = []
for mu in np.linspace(0, 10, 50):
    for sigma in np.linspace(0.1, 5, 50):  # 寻找最优 q(x)
        q = norm.pdf(x, mu, sigma)
        q_list.append(q)
        KL_pq.append(entropy(p, q))
        KL_qp.append(entropy(q, p))
```

```
KL_pq_min = np.argmin(KL_pq)
KL_qp_min = np.argmin(KL_qp)

fig, axes = plt.subplots(1, 2, figsize=(10, 3))
axes[0].set_ylim(0, 0.8)
axes[0].plot(x, p/2, 'b', label='$p(x)$')
axes[0].plot(x, q_list[KL_pq_min], 'g--', label='$q^*(x)$')
axes[0].set_xlabel('$x$')
axes[0].set_ylabel('$p(x)$')
axes[0].set_title('$q^*= {arg\min}_ q D_{KL}(p||q)$')
axes[1].set_ylim(0, 0.8)
axes[1].plot(x, p/2, 'b', label='$p(x)$')
axes[1].plot(x, q_list[KL_qp_min], 'g--', label='$q^*(x)$')
axes[1].set_xlabel('$x$')
axes[1].set_ylabel('$p(x)$')
axes[1].set_title('$q^*= {arg\min}_ q D_{KL}(q||p)$')
for ax in axes:
    ax.legend(loc='upper right')
```



2.3 图模型

机器学习算法会涉及到非常多的随机变量上的概率分布。利用分解可以减少表示联合分布的成本，于是用图来表示概率分布的分解，这称为结构化概率模型 (`Structured Probabilistic Model`) 或者图模型 (`Graphical Model`)。

2.3.1 有向图模型

有向图模型 (`Directed Model`) 的概率可以因子分解 $P(x) = P(x_1, ...x_i, ...) = \prod_i P(x_i \mid \mathrm{PA}(x_i))$，其中 $\mathrm{PA}(x_i)$ 是 x_i 的父节点，单个因子 $P(x_i|\mathrm{PA}(x_i))$ 称为条件概率分布 (CPD)。示例如下图所示，有：

$$P(a, b, c, d, e) = P(a)P(b \mid a)P(c \mid a, b)P(d \mid b)P(e \mid c)$$

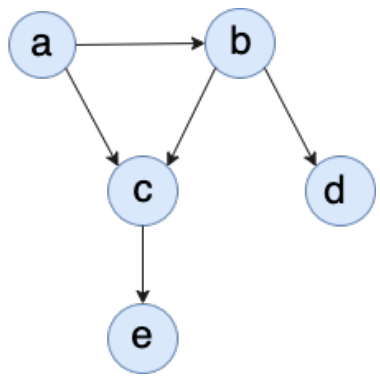


图 2.1. 有向图示例

有向图的代表是**贝叶斯网**。

贝叶斯网与朴素贝叶斯模型建立在相同的直观假设上：**通过利用分布的条件独立性来获得紧凑而自然的表示**。贝叶斯网核心是一个有向无环图 (DAG)，其节点为论域中的随机变量，节点间的有向箭头表示这两个节点的依赖关系。

贝叶斯网可以看作是**各特征节点间的依赖关系图** (有向无环图表示) 和**各特征节点相对其依赖节点的条件概率表**。

有向无环图可以由如下 3 种元结构构成：

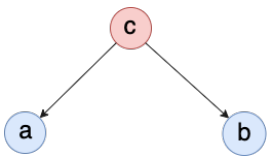
- 同父结构。例如在上图中，若不考虑节点 a ，则 c 和 d 有同一父节点 b，于是 $P(b,c,d) = P(b)P(c \mid b)P(d \mid b)$。
- V 型结构。例如在上图中，若不考虑节点 a 到 b 的依赖关系，则 $P(a,b,c) = P(a)P(b)P(c \mid a,b)$。
- 顺序结构。例如在上图中，若仅考虑节点 a、b、d，则有 $P(a,b,d) = P(a)P(b \mid a)P(d \mid b)$。

基于此，我们对简化的条件概率分布 (如 $P(c \mid a,b)$) 获取条件概率表。同时，也可以求得联合概率分布 $P(a,b,c,d,e) = P(a)P(b \mid a)P(c \mid a,b)P(d \mid b)P(e \mid c)$。

贝叶斯网的独立性

贝叶斯网的基本独立性体现在

- **局部独立性**：给定父节点条件下，每个节点都独立于它的非后代节点。例如，给定父节点 c 时，e 与网中其他节点条件独立 $\Longrightarrow (e \perp a,b,d \mid c)$
- **全局独立性** (d - 分离)：d - 分离是用来判断变量是否条件独立的图形化方法。它常见于以下三种条件独立的情况：
  1. tail-to-tail

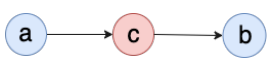


$\Longrightarrow$ 可以得知 $P(a,b,c) = P(a \mid c)P(b \mid c)P(c)$。

$\Longrightarrow$ 若 c 不作为观察点，可得 $P(a,b) = \sum_c P(a \mid c)P(b \mid c)P(c) \neq P(a)P(b)$，于是 a 和 b 不是 c 条件独立的。

$\Longrightarrow$ 若 c 作为观察点，可得 $P(a,b|c) = \frac{P(a,b,c)}{P(c)} = P(a \mid c)P(b \mid c)$，于是 a 和 b 是 c 条件下独立的。

  2. head-to-tail

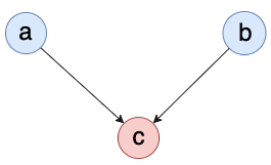


$\Longrightarrow$ 可以得知 $P(a,b,c) = P(b \mid c)P(c \mid a)P(a)$。

$\Longrightarrow$ 若 c 不作为观察点，可得 $P(a,b) = P(a)\sum_c P(c \mid a)P(b|c) = P(a)P(b \mid a)$，于是 a 和 b 不是 c 条件独立的。

$\Longrightarrow$ 若 c 作为观察点，可得 $P(a,b \mid c) = \frac{P(a,b,c)}{P(c)} = \frac{P(a)P(c|a)P(b|c)}{P(c)} = P(a \mid c)P(b \mid c)$，于是 a 和 b 是 c 条件下独立的。

  3. head-to-head



$\Longrightarrow$ 可以得知 $P(a,b,c) = P(a)P(b)P(c \mid a,b)$。

$\Longrightarrow$ 若 c 不作为观察点，可得 $P(a,b) = P(a)P(b)\sum_c P(c \mid a,b) = P(a)P(b)$，于是 a 和 b 是 c 条件独立的。

$\Longrightarrow$ 若 c 作为观察点，可得 $P(a,b|c) = \frac{P(a,b,c)}{P(c)} = \frac{P(a)P(b)P(c|a,b)}{P(c)} \neq P(a \mid c)P(b \mid c)$，于是 a 和 b 不是 c 条件下独立的。

从而我们考虑复杂的有向无环图 (DAG)，如果 A,B,C 是三个集合 (可以是单独的节点或者是节点的集合)。为了判断 A 和 B 是否是 C 条件独立的，我们考虑图中所有 A 和 B 之间的路径。对于其中的一条路径，如果它满足以下两个条件中的任意一条，则称这条路径是阻塞 (`block`) 的：

1. 路径中存在某个节点 X 是 head-to-tail 或者 tail-to-tail 节点，并且 X 是包含在 C 中的；
2. 路径中存在某个节点 X 是 head-to-head 节点，并且 X 或 X 的儿子是不包含在 C 中的。

如果 A,B 间所有的路径都是阻塞的，那么 A,B 就是关于 C 条件独立的；否则 A,B 不是关于 C 条件独立的。

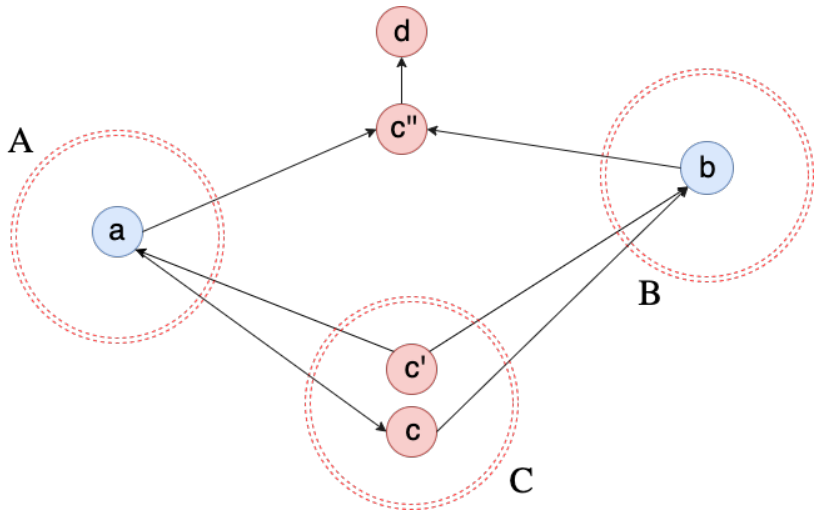


图 2.2. 考虑集合下的有向无环图

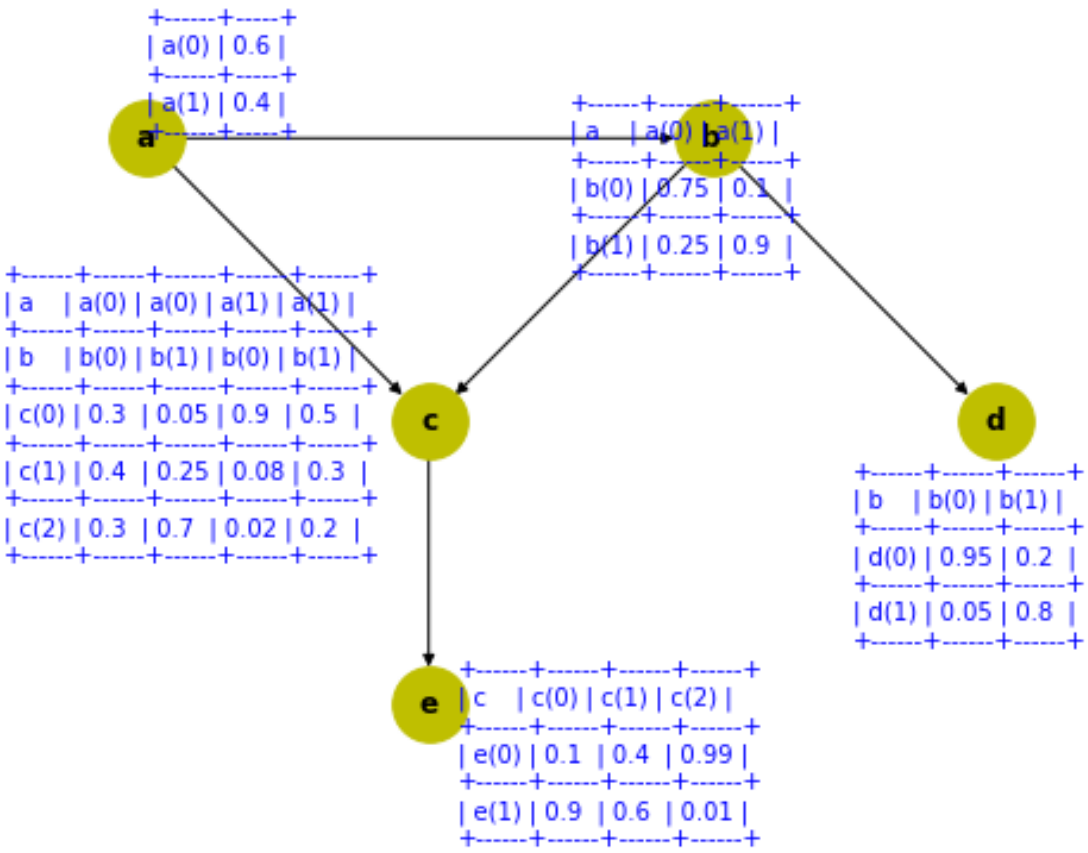
我们形象化阐述上面的结论：如图 2.2 所示，集合 x_{A}, x_{B} 和 x_{C}，我们希望 x_{A} 和 x_{B} 在给定 x_{C} 下条件独立或所有路径阻塞，即 $x_{\mathrm{A}} \perp x_{\mathrm{B}} \mid x_{\mathrm{C}}$。对

于集合中的 a，b 和 c，如果我们希望 $a \to c \to b$ 的路径阻塞，当路径为 tail-to-tail 或者 head-to-tail，则 c 应当在集合 C 中，这便是条件 1；当路径为 head-to-head，则 c 以及其后代 d 不能在集合 C 中，这便是条件 2。

例如，我们再考虑图 2.1 中的例子，判断 a 和 d 是否是 e 下条件独立的：可以看出 a 到 d 有两条路径 $a \to c \to b \to d$ 以及 $a \to b \to d$。考虑路径 $a \to b \to d$，b 是 head-to-tail 类型的，但不在条件集合中，因此该路径不阻塞。考虑路径 $a \to c \to b \to d$，c 是 head-to-head 类型的，且它的儿子 e 在条件集合中，因此该路径也不阻塞。因此，得到结论：a 和 d 不是 e 下条件独立的。

[39]:
```
import networkx as nx
from pgmpy.models import BayesianModel
from pgmpy.factors.discrete import TabularCPD
import matplotlib.pyplot as plt
%matplotlib inline

# 建立一个简单贝叶斯模型框架
model = BayesianModel([('a', 'b'), ('a', 'c'), ('b', 'c'), ('b', 'd'), ('c', 'e')])
# 最顶层的父节点的概率分布表
cpd_a = TabularCPD(variable='a', variable_card=2, values=[[0.6, 0.4]])  # a: (0,1)
# 其它各节点的条件概率分布表（行对应当前节点索引，列对应父节点索引）
cpd_b = TabularCPD(variable='b', variable_card=2,       # b: (0,1)
                   values=[[0.75, 0.1],
                           [0.25, 0.9]],
                   evidence=['a'],
                   evidence_card=[2])
cpd_c = TabularCPD(variable='c', variable_card=3,       # c: (0,1,2)
                   values=[[0.3, 0.05, 0.9,  0.5],
                           [0.4, 0.25, 0.08, 0.3],
                           [0.3, 0.7,  0.02, 0.2]],
                   evidence=['a', 'b'],
                   evidence_card=[2, 2])
cpd_d = TabularCPD(variable='d', variable_card=2,       # d: (0,1)
                   values=[[0.95, 0.2],
                           [0.05, 0.8]],
                   evidence=['b'],
                   evidence_card=[2])
cpd_e = TabularCPD(variable='e', variable_card=2,       # e: (0,1)
                   values=[[0.1, 0.4, 0.99],
                           [0.9, 0.6, 0.01]],
                   evidence=['c'],
                   evidence_card=[3])
# 将各节点的概率分布表加入网络
model.add_cpds(cpd_a, cpd_b, cpd_c, cpd_d, cpd_e)
# 验证模型数据的正确性
print(u"验证模型数据的正确性:",model.check_model())
# 绘制贝叶斯图（节点 + 依赖关系）
nx.draw(model, with_labels=True, node_size=1000, font_weight='bold', node_color='y', \
        pos={"e":[4,3],"c":[4,5],"d":[8,5],"a":[2,7],"b":[6,7]})
plt.text(2,7,model.get_cpds("a"), fontsize=10, color='b')
plt.text(5,6,model.get_cpds("b"), fontsize=10, color='b')
plt.text(1,4,model.get_cpds("c"), fontsize=10, color='b')
plt.text(4.2,2,model.get_cpds("e"), fontsize=10, color='b')
plt.text(7,3.4,model.get_cpds("d"), fontsize=10, color='b')
plt.show()
```

```
验证模型数据的正确性: True
```



2.3.2 无向图模型

无向图模型 (`Undirected Model`) 的概率可以记作 $P(\boldsymbol{x}) = \frac{1}{\mathrm{Z}} \prod_{\mathrm{C} \in \mathbf{Q}} \Phi_{\mathrm{C}} \boldsymbol{x}_{\mathrm{C}}$。其中，我们将所有节点都彼此联通的集合称作团 (`Clique`，C)，Φ 称作因子 (`factor`)，每个因子和一个团 C 相对应，Z 是归一化常数。示例如下图所示，有 $P(a, b, c, d, e) = \frac{1}{\mathrm{Z}} \Phi^{(1)}(a, b, c) \Phi^{(2)}(b, d) \Phi^{(3)}(c, e)$。

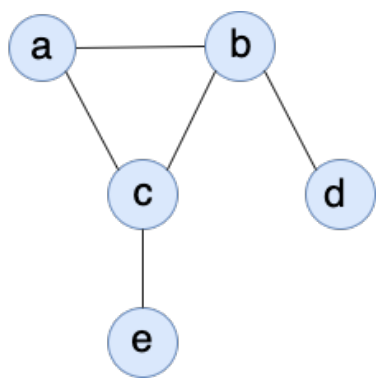


图 2.3. 无向图示例

有向图的代表是**马尔可夫网**。

贝叶斯网是根据节点依赖关系构成有向无环图，进而引申出每个节点的条件概率分布来表征其对父节点的依赖。但**马尔可夫网节点间的依赖关系是无向的**（相互平等的关系），无法用条件概率分布来表示，为此为引入**极大团**概念，进而为每个极大团引入一个**势函数**作为因子，然后将联合概率分布表示成这些因子的乘积再归一化，归一化常数被称作**配分函数**。

团: 假设一个特征集的任何两个特征都相互关联，那么这个特征集的联合概率分布是无法简化的，我们称这样的特征集为团。

极大团: 如果一个团不能被其他团包含，那么我们称这个团为极大团。

对于具有 n 个特征变量 $\boldsymbol{x} = (x_1, ..., x_n)$ 的马尔可夫网的所有极大团构成的集合 $\mathbf{Q}$，与极大团 $\mathrm{C} \in \mathbf{Q}$ 对应的属性变量集合记作 $\boldsymbol{x}_{\mathrm{C}}$，那么马尔可夫网 $\mathrm{P}(\boldsymbol{x})$ 可以写成因子分解的形式：

$$P(\boldsymbol{x}) = \frac{1}{\mathrm{Z}} \prod_{\mathrm{C} \in \mathbf{Q}} \Phi_{\mathrm{C}}(\boldsymbol{x}_{\mathrm{C}}) \qquad \mathrm{Z} = \sum_{\boldsymbol{x}} \prod_{\mathrm{C} \in \mathbf{Q}} \Phi_{\mathrm{C}}(\boldsymbol{x}_{\mathrm{C}}) \tag{2.31}$$

其中 Φ_{C} 就是极大团 C 对应的势函数 (因子)，用于对极大团 C 内的特征变量关系进行建模，必须为正。Z 为归一化因子 (配分函数)，就是对势函数乘积的所有属性变量求和求积分，使 P 成为概率。此时势函数可以写作 $\Phi(\mathbf{x}_{\mathrm{C}}) = \exp(-E(\mathbf{x}_{\mathrm{C}}))$，其中 E 为能量函数，我们也称 $P(\boldsymbol{x})$ 是由因子集 $\{\Phi_{\mathrm{C}} \mid \mathrm{C} \in \mathbf{Q}\}$ 参数化的吉布斯分布 (`Gibbs Distribution`) 或玻尔兹曼分布 (`Boltzmann Distribution`)。于是，示例的马尔可夫网的联合概率分布可写成：

$$P(a, b, c, d, e) = \frac{1}{\mathrm{Z}} \Phi_{a,b,c}(a, b, c) \Phi_{b,d}(b, d) \Phi_{c,e}(c, e)$$

马尔可夫网的条件独立性

前面介绍了贝叶斯网的元结构及其条件独立性，而马尔可夫网的有向分离，同样能够引出条件独立性 (相对分离集)，这就是全局马尔可夫性。

如果将 $\{a,b,c\}$ 视作团 A，$\{b,d\}$ 视作团 B，$\{c,e\}$ 视作团 C。特别地，我们可以验证最简单情况下的全局马尔可夫性：

$$
\begin{aligned}
\mathrm{P}(x_A, x_B \mid x_C) &= \frac{\mathrm{P}(x_A, x_B, x_C)}{\mathrm{P}(x_C)} && \text{条件概率的定义} \\
&= \frac{\frac{1}{Z}\Phi_{A,C}(x_A, x_C)\Phi_{B,C}(x_B, x_C)}{\sum_{x'_A, x'_B} \mathrm{P}(x'_A, x'_B, x_C)} && \text{分子极大团分解} \\
& && \text{分母概率分布边缘求和 (积分)} \\
&= \frac{\frac{1}{Z}\Phi_{A,C}(x_A, x_C)\Phi_{B,C}(x_B, x_C)}{\sum_{x'_A, x'_B} \frac{1}{Z}\Phi_{A,C}(x'_A, x_C)\Phi_{B,C}(x'_B, x_C)} && \text{分母局部做极大团分解} \\
&= \frac{\Phi_{A,C}(x_A, x_C)}{\sum_{x'_A} \Phi_{A,C}(x'_A, x_C)} \frac{\Phi_{B,C}(x_B, x_C)}{\sum_{x'_B} \Phi_{B,C}(x'_B, x_C)} && \text{简单的代数操作} \\
&= \mathrm{P}(x_A \mid x_C)\mathrm{P}(x_B \mid x_C) && \text{利用边缘求和和极大团分解} \\
& && \text{而}P(x_A, x_B, x_C)\text{作为中间分布}
\end{aligned}
\tag{2.32}
$$

由全局马尔可夫性可以容易推导出两个推论：

- 局部马尔可夫性：将节点 $v \in \mathrm{V}$ 的所有邻接节点集作为分离集 $\mathrm{N}(v) \subset \mathrm{V}$，于是该节点 v 与被邻接变量集分离的剩余变量集是条件独立的 (相对 $\mathrm{N}(v)$ 而言)。

$$x_v \perp \boldsymbol{x}_{V \setminus N^*(v)} \mid \boldsymbol{x}_{N(v)}, \quad N^*(v) = N(v) \cup \{v\} \tag{2.33}$$

- 成对马尔可夫性：两个非邻接节点 $u, v \in \mathrm{V}$，必然可以被其他所有节点构成的集 $\boldsymbol{x}_{\mathrm{V}\setminus\{u,v\}}$ 分离，进而 u, v 也具有条件独立性 (相对前面指定的节点集)。

$$x_u \perp x_v \mid \boldsymbol{x}_{\mathrm{V}\setminus\{u,v\}}, \quad \{u,v\} \notin E, \quad E\text{是边集} \tag{2.34}$$

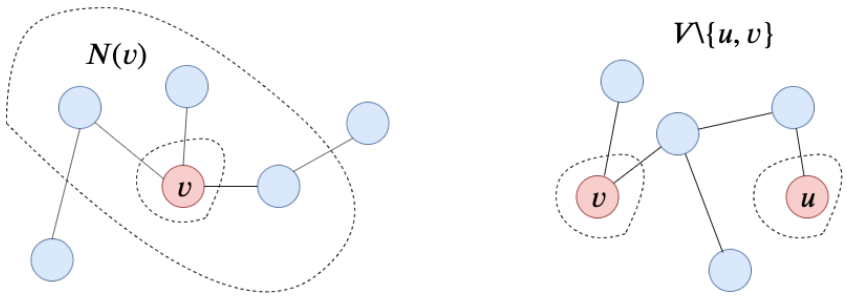
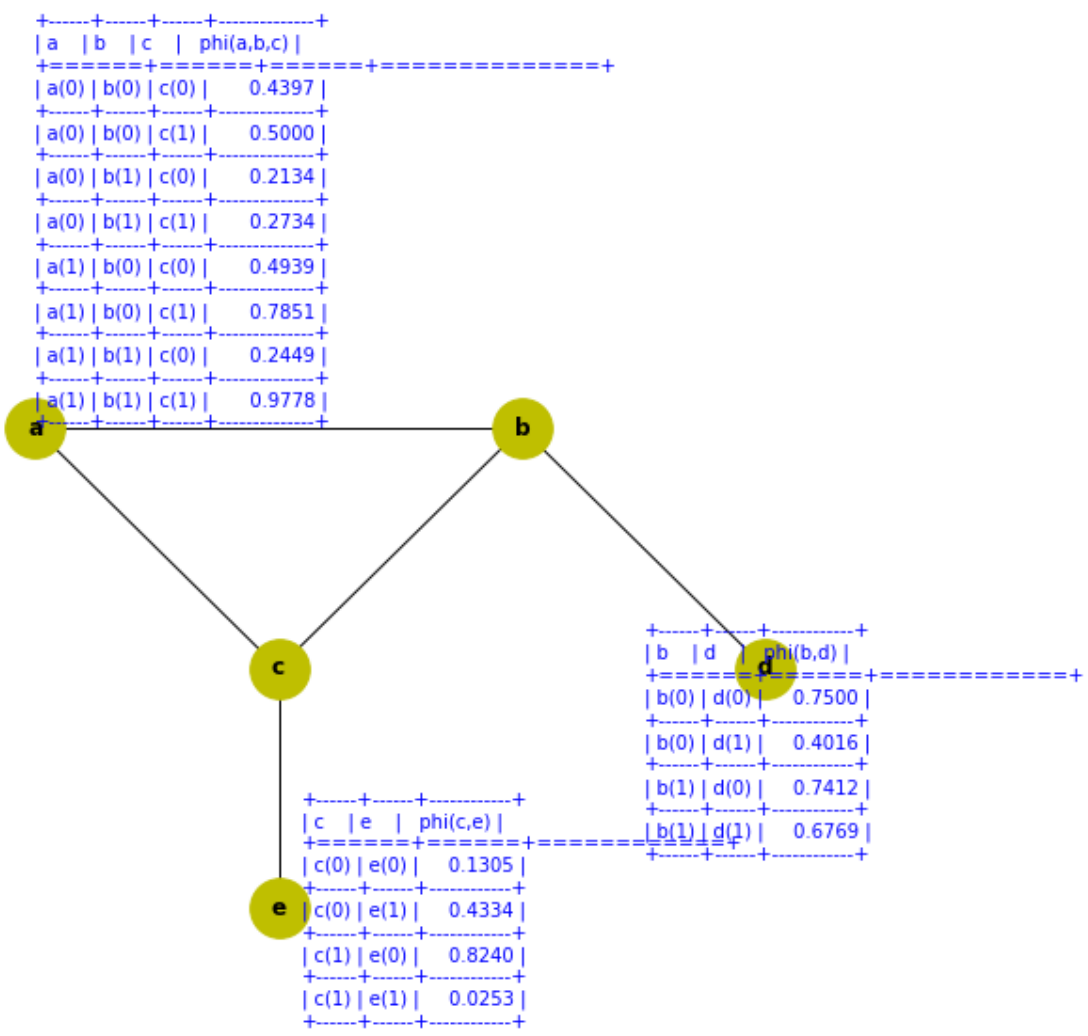


图 2.4. 局部马尔可夫性与成对马尔可夫性

[40]:
```
import networkx as nx
from pgmpy.models import MarkovModel
from pgmpy.factors.discrete import DiscreteFactor
import matplotlib.pyplot as plt
%matplotlib inline
# 建立一个简单马尔科夫网
model = MarkovModel([('a', 'b'), ('a', 'c'), ('b', 'c'),('b', 'd'), ('c', 'e')])
# 各团因子 (参数随机选择)
factor_abc = DiscreteFactor(['a', 'b', 'c'], cardinality=[2,2,2], values=np.random.rand(8))
factor_bd = DiscreteFactor(['b', 'd'], cardinality=[2,2], values=np.random.rand(4))
factor_ce = DiscreteFactor(['c', 'e'], cardinality=[2,2], values=np.random.rand(4))
# 将各团因子加入网络
model.add_factors(factor_abc,factor_bd,factor_ce)
# 验证模型数据的正确性
print(u"验证模型数据的正确性:",model.check_model())
# # 绘制贝叶斯图 (节点 + 依赖关系)
nx.draw(model, with_labels=True, node_size=1000, font_weight='bold', node_color='y', \
        pos={"e":[4,3],"c":[4,5],"d":[8,5],"a":[2,7],"b":[6,7]})
plt.text(2,7,model.get_factors()[0], fontsize=10, color='b')
plt.text(7,3.4,model.get_factors()[1], fontsize=10, color='b')
plt.text(4.2,2,model.get_factors()[2], fontsize=10, color='b')
plt.show()
```

```
验证模型数据的正确性: True
```



有关于图模型的更多内容会在第十六章呈现。

[41]:
```
import numpy, scipy, matplotlib, networkx, pgmpy
print("numpy:", numpy.__version__)
print("scipy:", scipy.__version__)
print("matplotlib:", matplotlib.__version__)
print("networkx:", networkx.__version__)
print("pgmpy:", pgmpy.__version__)
```

```
numpy: 1.14.5
scipy: 1.3.1
matplotlib: 3.1.1
networkx: 2.4
pgmpy: 0.1.10
```

第三章　数值计算

3.1　上溢和下溢

下溢（Underflow）：当接近零的数被四舍五入为零时发生下溢。

上溢（Overflow）：当大量级的数被近似为 ∞ 或 $-\infty$ 时发生上溢。

必须对上溢和下溢进行数值稳定的一个例子是 softmax 函数。softmax 函数经常用于预测与范畴分布相关联的概率，定义为:

$$\text{softmax}(\boldsymbol{x})_i = \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)} \tag{3.1}$$

[1]:
```
import numpy as np
import numpy.linalg as la
```

[2]:
```
x = np.array([1e7, 1e8, 2e5, 2e7])
y = np.exp(x)/sum(np.exp(x))
print("上溢: ", y)
x = x - np.max(x)  # 减去最大值
y = np.exp(x)/sum(np.exp(x))
print("上溢处理: ", y)
```

```
上溢:  [nan nan nan nan]
上溢处理:  [0. 1. 0. 0.]
```

[3]:
```
x = np.array([-1e10, -1e9, -2e10, -1e10])
y = np.exp(x)/sum(np.exp(x))
print("下溢: ", y)
x = x - np.max(x)  # 减去最大值
y = np.exp(x)/sum(np.exp(x))
print("下溢处理: ", y)
print("log softmax(x):", np.log(y))
# 对 log softmax 下溢的处理:
def logsoftmax(x):
    y = x - np.log(sum(np.exp(x)))
    return y

print("logsoftmax(x):", logsoftmax(x))
```

```
下溢:  [nan nan nan nan]
下溢处理:  [0. 1. 0. 0.]
log softmax(x): [-inf   0. -inf -inf]
logsoftmax(x): [-9.0e+09  0.0e+00 -1.9e+10 -9.0e+09]
```

3.2　优化方法

3.2.1　梯度下降法

梯度下降法 (Gradient Descent) 或最速下降法 (Method of Steepest Descent) 的目标函数是最小化具有多维输入的函数：$f : \mathbb{R}^n \to \mathbb{R}$。梯度下降法建议新的点为:

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{3.2}$$

其中 ϵ 为学习率 (learning rate)，是一个确定步长大小的正标量。

这里引入实例 (线性最小二乘)：

$$f(\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2 \tag{3.3}$$

假设我们希望找到最小化该式的 x 值。

可以计算梯度得到：

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \boldsymbol{A}^\top(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) = \boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top\boldsymbol{b} \tag{3.4}$$

实例：线性最小二乘

[4]:
```
x0 = np.array([1.0, 1.0, 1.0])
A = np.array([[1.0, -2.0, 1.0], [0.0, 2.0, -8.0], [-4.0, 5.0, 9.0]])
b = np.array([0.0, 8.0, -9.0])
epsilon = 0.001
delta = 1e-3
# 给定 A, b, 真正的解 x 为 [29, 16, 3]
```

[5]:
```
"""
梯度下降法
"""
def matmul_chain(*args):
    if len(args) == 0: return np.nan
    result = args[0]
    for x in args[1:]:
        result = result@x
    return result

def gradient_decent(x, A, b, epsilon, delta):
    while la.norm(matmul_chain(A.T, A, x)-matmul_chain(A.T, b)) > delta:
        x -= epsilon*(matmul_chain(A.T, A, x)-matmul_chain(A.T, b))
    return x

gradient_decent(x0, A, b, epsilon, delta)
```

[5]:
```
array([27.82277014, 15.34731055,  2.83848939])
```

3.2.2 牛顿法

牛顿法 (Newton's Method) 基于一个二阶泰勒展开来近似 $\boldsymbol{x}^{(0)}$ 附近的 $f(\boldsymbol{x})$：

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \mathrm{H}(\boldsymbol{x}^{(0)})(\boldsymbol{x} - \boldsymbol{x}^{(0)}) \tag{3.5}$$

接着通过计算，我们可以得到这个函数的临界点：

$$\boldsymbol{x}^* = \boldsymbol{x}^{(0)} - \mathrm{H}(\boldsymbol{x}^{(0)})^{-1}\nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)}) \tag{3.6}$$

牛顿法迭代地更新近似函数和跳到近似函数的最小点可以比梯度下降法更快地到达临界点。这在接近全局极小时是一个特别有用的性质，但是在鞍点附近是有害的。

针对上述实例，计算得到：$\mathrm{H} = \boldsymbol{A}^\top\boldsymbol{A}$

进一步计算得到最优解：

$$\boldsymbol{x}^* = \boldsymbol{x}^{(0)} - (\boldsymbol{A}^\top\boldsymbol{A})^{-1}(\boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x}^{(0)} - \boldsymbol{A}^\top\boldsymbol{b}) = (\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top\boldsymbol{b} \tag{3.7}$$

[6]:
```
"""
牛顿法
"""
def matmul_chain(*args):
    if len(args) == 0: return np.nan
    result = args[0]
    for x in args[1:]:
        result = result@x
    return result
```

```
def newton(x, A, b, delta):
    x = matmul_chain(np.linalg.inv(matmul_chain(A.T, A)), A.T, b)
    return x

newton(x0, A, b, delta)
```

[6]: array([29., 16., 3.])

3.2.3 约束优化

我们希望通过 m 个函数 $g^{(i)}$ 和 n 个函数 $h^{(j)}$ 描述 S ，那么 S 可以表示为 $\mathrm{S} = \{\boldsymbol{x} \mid \forall i, g^{(i)}(\boldsymbol{x}) = 0 \text{ and } \forall j, h^{(j)}(\boldsymbol{x}) \leq 0\}$。其中涉及 $g^{(i)}$ 的等式称为等式约束，涉及 $h^{(j)}$ 的不等式称为不等式约束。

我们为每个约束引入新的变量 λ_i 和 α_j，这些新变量被称为 KKT 乘子。广义拉格朗日式可以如下定义：

$$L(\boldsymbol{x}, \lambda, \alpha) = f(\boldsymbol{x}) + \sum_i \lambda_i g^{(i)}(\boldsymbol{x}) + \sum_j \alpha_j h^{(j)}(\boldsymbol{x}) \tag{3.8}$$

可以通过优化无约束的广义拉格朗日式解决约束最小化问题：

$$\min_{\boldsymbol{x}} \max_{\lambda} \max_{\alpha, \alpha \geq 0} L(\boldsymbol{x}, \lambda, \alpha) \tag{3.9}$$

优化该式与下式等价：

$$\min_{\boldsymbol{x} \in \mathrm{S}} f(\boldsymbol{x}) \tag{3.10}$$

针对上述实例，约束优化: $\boldsymbol{x}^\top \boldsymbol{x} \leq 1$

引入广义拉格朗日式：

$$L(\boldsymbol{x}, \lambda) = f(\boldsymbol{x}) + \lambda(\boldsymbol{x}^\top \boldsymbol{x} - 1) \tag{3.11}$$

解决以下问题：

$$\min_{\boldsymbol{x}} \max_{\lambda, \lambda \geq 0} L(\boldsymbol{x}, \lambda) \tag{3.12}$$

关于 $\boldsymbol{x}$ 对 Lagrangian 微分，我们得到方程:

$$\boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x} - \boldsymbol{A}^\top \boldsymbol{b} + 2\lambda \boldsymbol{x} = \boldsymbol{0} \tag{3.13}$$

得到解的形式是：

$$\boldsymbol{x} = (\boldsymbol{A}^\top \boldsymbol{A} + 2\lambda \boldsymbol{I})^{-1} \boldsymbol{A}^\top \boldsymbol{b} \tag{3.14}$$

λ 的选择必须使结果服从约束，可以对 λ 梯度上升找到这个值：

$$\frac{\partial}{\partial \lambda} L(\boldsymbol{x}, \lambda) = \boldsymbol{x}^\top \boldsymbol{x} - 1 \tag{3.15}$$

[7]:
```
"""
约束优化，约束解的大小
"""
def matmul_chain(*args):
    if len(args) == 0: return np.nan
    result = args[0]
    for x in args[1:]:
        result = result@x
    return result

def constrain_opti(x, A, b, delta):
    k = len(x)
    lamb = 0
    while np.abs(np.dot(x.T, x)-1) > 5e-2:  # delta 设为 5e-2, 最优设为 0
        x = matmul_chain(np.linalg.inv(matmul_chain(A.T, A)+2*lamb*np.identity(k)), A.T, b)
        lamb += np.dot(x.T, x)-1
    return x

constrain_opti(x0, A, b, delta)
```

[7]:
```
array([ 0.23637902,  0.05135858, -0.94463626])
```

[8]:
```
import numpy
print("numpy:", numpy.__version__)
```

```
numpy: 1.14.5
```

第四章　机器学习基础

4.1　学习算法

机器学习算法描述一种能够**从数据中学习的算法**。学习指对于某类任务 T，为其定义性能度量 P，一个计算机程序被认为可以从经验 E 中学习是指：**通过经验 E 改进后，它在任务 T 上的性能度量 P 有所提高**。

任务 T：机器学习**任务**定义为**机器学习系统应该如何处理样本**（Example）。例如，识别手写体数字识别的任务为：通过将输入的图片处理后，输出该图片对应的数字（分类）。**样本**是**量化的特征（Feature）的集合**，用向量 $\boldsymbol{x} \in \mathbb{R}^n$ 表示，其中向量的每个元素 x_i 是一个特征。例如一张图片的特征就是这张图片里的像素点的值。

性能度量 P：为了**评估机器学习的优劣**，需要对算法的输出结果进行**定量的衡量分析**，这就需要合适的性能度量指标。

	指标	说明
True Positive	TP	将正样本预测为正例数目
True Negative	TN	将负样本预测为负例数目
False Positive	FP	将负样本预测为正例数目
False Negative	FN	将正样本预测为负例数目

- 针对**分类**任务 (详细描述见第十一章)：
  - 准确率 (Accuracy)：$acc = \frac{\mathrm{TP+TN}}{\mathrm{TP+TN+FP+FN}}$。
  - 错误率 (Error-rate)：$err = 1 - acc$
  - 精度 (Precision)：$\mathrm{P} = \frac{\mathrm{TP}}{\mathrm{TP+FP}}$
  - 召回率 (Recall)：$\mathrm{R} = \frac{\mathrm{TP}}{\mathrm{TP+FN}}$
  - F_1 值：$\mathrm{F}_1 = \frac{2\mathrm{PR}}{\mathrm{P+R}}$
- 针对**回归**任务：距离误差

经验 E：根据经验 E 的不同，机器学习算法可以分为：无监督 (Unsupervised) 算法和监督 (Supervised) 算法。

- 监督学习算法 (Supervised Learning)：训练集的数据中**包含样本特征和标签值**，常见的分类和回归算法都是有监督的学习算法。
- 无监督学习算法 (Unsupervised Learning)：训练集的数据中**只包含样本特征**，算法需要从中学习出特征中隐藏的结构化特征，聚类、密度估计等都是无监督的学习算法。

4.1.1　举例：线性回归

线性回归（Linear Regression）的目标：获得一个函数 f，满足 $f(\boldsymbol{x}) = \hat{y}$，其中 $\boldsymbol{x} \in \mathbb{R}^n, \hat{y} \in \mathbb{R}$，使得 $\hat{y}$ 接近于真实的标签 y。

我们定义线性回归的输出为：

$$f(x) = \boldsymbol{w}^\top \boldsymbol{x} \tag{4.1}$$

其中 $\boldsymbol{w} \in \mathbb{R}^n$ 是我们需要学习的参数 (Parameter)。

在线性回归中，对任务 T 的定义：通过输出 $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$，从 $\boldsymbol{x}$ 预测 y。

性能度量 P 的定义：假设测试集的特征和标签分别用 $\boldsymbol{X}^{(test)}$ 和 $\boldsymbol{y}^{(test)}$ 表示。可以采用的性能度量方式是均方误差（Mean Squared Error），如果 $\hat{\boldsymbol{y}}^{(test)}$ 表示模型在测试集上的预测值，那么均方误差公式为：

$$\mathrm{MSE}_{test} = \frac{1}{m}\sum_i (\hat{\boldsymbol{y}}^{(test)} - \boldsymbol{y}^{(test)})_i^2 = \frac{1}{m}||\hat{\boldsymbol{y}}^{(test)} - \boldsymbol{y}^{(test)}||_2^2 \tag{4.2}$$

为了构建一个机器学习算法，需要设计一个算法，通过观察训练集 $(\boldsymbol{X}^{(train)}, \boldsymbol{y}^{(train)})$ 获得经验，改进权重 $\boldsymbol{w}$ 以减少 MSE_{test}。一种直观的方式是

最小化训练集上的均方误差，即 MSE_{train}。最小化 MSE_{train}，我们可以简单地求解其导数为 0 的情况：

$$\begin{aligned}\nabla_{\boldsymbol{w}}\text{MSE}_{train} &= 0 \\ \Longrightarrow\ & \nabla_{\boldsymbol{w}}\frac{1}{m}\ ||\ \hat{\boldsymbol{y}}^{(train)} - \boldsymbol{y}^{(train)}\ ||_2^2 = 0 \\ \Longrightarrow\ & \nabla_{\boldsymbol{w}}\frac{1}{m}\ ||\ \boldsymbol{X}^{(train)}\boldsymbol{w} - \boldsymbol{y}^{(train)}\ ||_2^2 = 0 \\ \Longrightarrow\ & \boldsymbol{w} = (\boldsymbol{X}^{(train)\top}\boldsymbol{X}^{(train)})^{-1}\boldsymbol{X}^{(train)\top}\boldsymbol{y}^{(train)}\end{aligned} \tag{4.3}$$

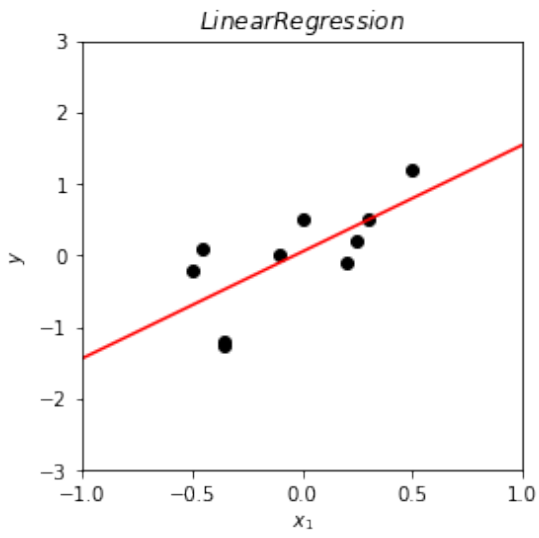
方程的解：$\boldsymbol{w} = (\boldsymbol{X}^{(train)\top}\boldsymbol{X}^{(train)})^{-1}\boldsymbol{X}^{(train)\top}\boldsymbol{y}^{(train)}$ 被称为正规方程。

函数 $f(\boldsymbol{x}) = a\boldsymbol{x} + b$ 称为仿射函数，其中，当 $b = 0$ 时，变为 $f(\boldsymbol{x}) = a\boldsymbol{x}$，称为线性函数，即线性函数是仿射函数的一个特例。

[1]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
```

[2]:
```
X = np.hstack((np.array([[-0.5,-0.45,-0.35,-0.35,-0.1,0,0.2,0.25,0.3,0.5]]).reshape(-1, 1), np.ones((10,1))*1))
y = np.array([-0.2,0.1,-1.25,-1.2,0,0.5,-0.1,0.2,0.5,1.2]).reshape(-1,1)
# 用公式求权重
w = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
hat_y = X.dot(w)
print("Weight:{}".format{list(w)})
x = np.linspace(-1, 1, 50)
hat_y = x * w[0] + w[1]
plt.figure(figsize=(4,4))
plt.xlim(-1.0, 1.0)
plt.xticks(np.linspace(-1.0, 1.0, 5))
plt.ylim(-3, 3)
plt.plot(x, hat_y, color='red')
plt.scatter(X[:,0], y[:,0], color='black')
plt.xlabel('$x_1$')
plt.ylabel('$y$')
plt.title('$Linear Regression$')
plt.show()
```

```
Weight:[array([1.49333333]), array([0.04966667])]
```



4.2 容量、过拟合、欠拟合

4.2.1 泛化问题

机器学习的主要挑战在于**在未见过的数据输入上表现良好**，这个能力称为泛化能力 (Generalization)。我们量化一下模型在训练集和测试集上的表现，将其分别称为训练误差 (Training Error) 和测试误差 (Test Error)，后者也经常称为泛化误差 (Generalization Error)。可以说，**理想的模型就是在最小化训练误差的同时，最小化泛化误差**，具有良好泛化能力的算法才是符合需求的。

在实际的应用过程中，会采样两个数据集，减小训练误差得到参数后，再在测试集中验证。这个过程中，就会发生测试误差的期望大于训练误差的期望的情况。以下是决定机器学习算法效果是否好的因素：

- 降低训练误差。
- 缩小训练误差与测试误差之间的差距。

这俩个因素分别对应了机器学习的两个大挑战：欠拟合 (Underfitting) 和过拟合 (Overfitting)。**欠拟合**指的是**模型在训练集上的误差较大**，这通常是由于训练不充分或者模型不合适导致；**过拟合**指的是**模型在训练集和测试集上的误差差距过大**，通常由于模型过分拟合了训练集中的随机噪音，导致泛化能力较差。采用**正则化**，可以**降低泛化误差**，我们会在第七章进一步的介绍。

4.2.2 容量

通过调节机器学习模型的容量，可以控制模型是否偏于过拟合还是欠拟合，容量 (Capacity) 是描述了**整个模型拟合各种函数的能力**。如果容量不足，模型将不能够很好地表示数据，表现为欠拟合；如果容量太大，那么模型就很容易过分拟合数据，因为其记住了不适合于测试集的训练集特性，表现为过拟合。容量的控制可以通过多种方法控制，包括：

- 控制模型的假设空间。
- 添加正则项对模型进行偏好排除。

当机器学习算法的容量适合于所执行任务的复杂度和所提供训练数据的数量时，算法效果通常会最佳。统计学习方法理论提供了量化模型的容量的不同方法，其中最为出名的是 Vapnik-Chervonenkis 维度 (Vapnik-Chervonenkis dimension)。统计学习理论中最重要的结论阐述了**训练误差和泛化误差之间差异的上界随着模型容量增长而增长，但随着训练样本增多而下降**。

通常，当模型容量上升时，训练误差会下降，直到其渐近最小可能误差（假设误差度量有最小值），而泛化误差会是一个关于模型容量的 U 形曲线函数。

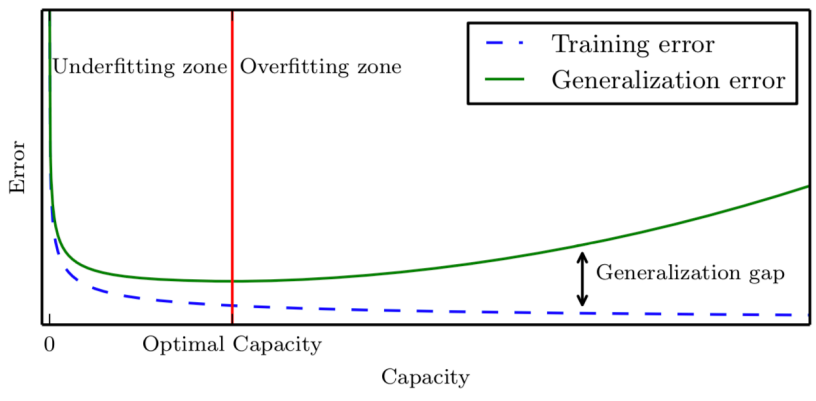


图 4.1. 容量和误差之间的典型关系

4.3 超参数与验证集

超参数：用来**控制学习算法的参数**而非学习算法本身学出来的参数。例如，进行曲线的回归拟合时，曲线的次数就是一个超参数；在构建模型对一些参数的分布假设也是超参数。

验证集 (Validation Set)：通常在需要选取超参数时，将训练集再划分为训练和验证集两部分，使用新的训练集训练模型，**验证集用来进行测试和调整超参**。通常，80% 的训练数据用于训练学习参数，20% 用于验证。

k 折交叉验证：将数据集均分为**不相交的 k 份**，每次选取其中的一份作为测试集，其他的为训练集，**训练误差为 k 次的平均误差**。

[3]:
```python
def KFoldCV(D, A, k):
    """
    k-fold 交叉验证

    参数说明:
    D: 给定数据集
    A: 学习函数
    k: 折数
    """
    np.random.shuffle(D)
    dataset = np.split(D, k)
    acc_rate = 0
    for i in range(k):
        train_set = dataset.copy()
        test_set = train_set.pop(i)
        train_set = np.vstack(train_set)
```

```
        A.train(train_set[:,:-1], train_set[:,-1])    # 每次的训练集
        labels = A.fit(test_set[:,:-1])  # 每次的测试集
        acc_rate += np.mean(labels==test_set[:,-1])  # 计算平均误差
    return acc_rate/k
```

4.4 偏差和方差

4.4.1 偏差

估计的偏差 (Bias) 被定义为：

$$bias(\hat{\theta}_m) = \mathbb{E}(\hat{\theta}_m) - \theta \tag{4.4}$$

其中期望作用在所有数据上，θ 是用于定义数据生成分布的真实值。偏差反映的是模型在样本上的输出与真实值之间的误差，即模型本身的精准度，或者说**算法本身的拟合能力**。

- 如果 $bias(\hat{\theta}_m) = 0$，那么估计量 $\hat{\theta}_m$ 被称为是无偏 (Unbiased)。
- 如果 $\lim_{m\to\infty} bias(\hat{\theta}_m) = 0$，那么估计量 $\hat{\theta}_m$ 被称为是渐进无偏 (Asymptotically Unbiased)。

4.4.2 方差

估计的方差 (Variance) 被定义为：

$$\mathrm{Var}\,(\hat{\theta}) \tag{4.5}$$

方差反映的是模型每一次输出结果与模型输出期望之间的误差，即**模型的稳定性**。

标准差被记为

$$\mathrm{SE}(\hat{\mu}_m) = \sqrt{\mathrm{Var}\left[\frac{1}{m}\sum_{i=1}^{m}\boldsymbol{x}^{(i)}\right]} = \frac{\sigma}{\sqrt{m}} \tag{4.6}$$

其中，σ^2 是样本 $\{\boldsymbol{x}^{(i)}\}$ 的真实方差，标准差通常被标记为 σ。

4.4.3 误差与偏差和方差的关系

一个复杂的模型并不总是能在测试集上表现出更好的性能，那么误差源于哪?

以**回归**为例，对测试样本 $\boldsymbol{x}$，令 y_D 为 $\boldsymbol{x}$ 在数据集上的标记，y 为 $\boldsymbol{x}$ 的真实标记。由于噪声的存在，有可能 $y_D \neq y$，$f(\boldsymbol{x}; D)$ 为在训练集 D 上学得函数 f 对 $\boldsymbol{x}$ 的预测输出。因此，算法的期望预测可以表示为:

$$\bar{f}(\boldsymbol{x}) = \mathbb{E}_D[f(\boldsymbol{x}; D)] \tag{4.7}$$

不同训练集学得的函数 f 的预测输出的方差 (Variance) 为:

$$var(\boldsymbol{x}) = \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] \tag{4.8}$$

期望输出与真实标记之间的差距称为偏差 (Bias) 为:

$$bias^2(\boldsymbol{x}) = (\bar{f}(\boldsymbol{x}) - y)^2 \tag{4.9}$$

噪声 (真实标记与数据集中的实际标记间的偏差) 为:

$$\varepsilon^2 = \mathbb{E}_D[(y_D - y)^2] \tag{4.10}$$

假定噪声期望为零，即 $\mathbb{E}_D[y_D - y] = 0$。算法的期望泛化误差为:

$$\begin{aligned}
\mathbb{E}(f; D) &= \mathbb{E}_D[(f(\boldsymbol{x}; D) - y_D)^2] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}) + \bar{f}(\boldsymbol{x}) - y_D)^2] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] + \mathbb{E}_D[(\bar{f}(\boldsymbol{x}) - y_D)^2] + \mathbb{E}_D[2(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))(\bar{f}(\boldsymbol{x}) - y_D)] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] + \mathbb{E}_D[(\bar{f}(\boldsymbol{x}) - y_D)^2] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] + \mathbb{E}_D[(\bar{f}(\boldsymbol{x}) - y + y - y_D)^2] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] + \mathbb{E}_D[(\bar{f}(\boldsymbol{x}) - y)^2] + \mathbb{E}_D[(y - y_D)^2] + \mathbb{E}_D[2(\bar{f}(\boldsymbol{x}) - y)(y - y_D)] \\
&= \mathbb{E}_D[(f(\boldsymbol{x}; D) - \bar{f}(\boldsymbol{x}))^2] + (\bar{f}(\boldsymbol{x}) - y)^2 + \mathbb{E}_D[(y - y_D)^2]
\end{aligned} \tag{4.11}$$

式中，第一个加红公式等于 0，因为 $(f(\boldsymbol{x};D)-\bar{f}(\boldsymbol{x}))$ 与 $(\bar{f}(\boldsymbol{x})-y_D)$ 相互独立，所以 $\mathbb{E}_D[2(f(\boldsymbol{x};D)-\bar{f}(\boldsymbol{x}))(\bar{f}(\boldsymbol{x})-y_D)]=2\mathbb{E}_D[(f(\boldsymbol{x};D)-\bar{f}(\boldsymbol{x}))]\mathbb{E}_D[\bar{f}(\boldsymbol{x})-y_D)]$。根据期望预测公式 $\bar{f}(\boldsymbol{x})=\mathbb{E}_D[f(\boldsymbol{x};D)]$ 有 $\mathbb{E}_D[(f(\boldsymbol{x};D)-\bar{f}(\boldsymbol{x}))]=0$。同理第二个加红公式等于 0，因为噪声期望为 0。于是：

$$\mathbb{E}(f;D)=bias^2(\boldsymbol{x})+var(\boldsymbol{x})+\varepsilon^2 \tag{4.12}$$

也就是说，泛化误差可分解为偏差、方差与噪声之和。噪声无法人为控制，所以通常我们认为:

$$\mathbb{E}(f;D)=bias^2(\boldsymbol{x})+var(\boldsymbol{x}) \tag{4.13}$$

我们需要在模型复杂度之间权衡，使偏差和方差得以均衡 (`trade-off`)，这样模型的整体误差才会最小。

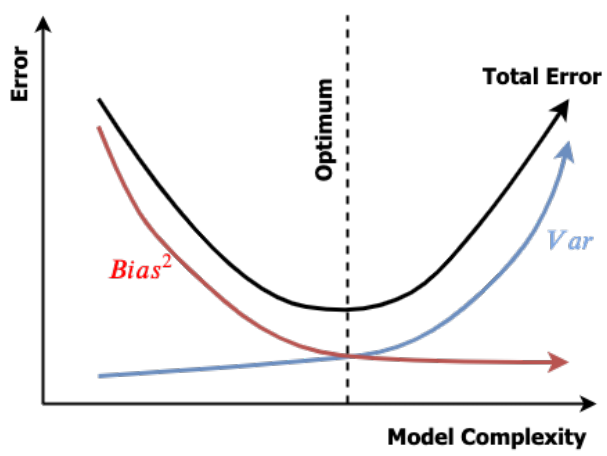


图 4.2. 当容量增大 (x 轴) 时，偏差 (红线) 随之减小，而方差 (蓝线) 随之增大，使得泛化误差 (黑线) 产生了另一种 U 形。

4.5 最大似然估计

最大似然估计 (`Maximum Likelihood Estimation`，MLE) 是一种最为常见的估计准则，其思想是在**已知分布产生的一些样本**而**未知分布具体参数**的情况下**根据样本值推断最有可能产生样本的参数值**。将数据的真实分布记为 $P_{data}(\boldsymbol{x})$ ，为了使用 MLE，需要先假设样本服从某一簇有参数确定的分布 $P_{model}(\boldsymbol{x};\theta)$，现在的目标就是使用估计的 P_{model} 来拟合真实的 P_{data} (**条件一：“模型已定，参数未知”**)。

对于一组由 m 个样本组成的数据集 $\boldsymbol{X}=\{\boldsymbol{x}^{(1)},\cdots,\boldsymbol{x}^{(m)}\}$，假设数据独立且由未知的真实数据分布 $\boldsymbol{P}_{data}(\boldsymbol{x})$ 生成 (**条件二：独立同分布采样的数据**)，可以通过最大似然估计：

$$\begin{aligned}\theta_{ML}&=\arg\max_{\theta}P_{model}(\boldsymbol{X};\theta)\\&=\arg\max_{\theta}\prod_{i=1}^{m}P_{model}(\boldsymbol{x}^{(i)};\theta)\end{aligned} \tag{4.14}$$

获得真实分布的参数。

通常为了计算方便，会对 MLE 加上 log，将乘积转化为求和然后将求和变为期望：$\theta_{ML}=\arg\max_{\theta}\Sigma_{i=1}^{m}\log P_{model}(\boldsymbol{x}^{(i)};\theta)$。

使用训练数据经验分布 $\hat{P}_{data}$ 相关的期望进行计算：$\theta_{ML}=\arg\max_{\theta}\mathbb{E}_{\boldsymbol{x}\sim\hat{P}_{data}}\log P_{model}(\boldsymbol{x};\theta)$。该式是许多监督学习算法的基础假设。

最大似然估计的一种解释是使 P_{model} 与 P_{data} 之间的差异性尽可能的小，形式化的描述为最小化两者的 KL 散度。

4.6 贝叶斯统计

最大似然估计属于典型的频率学派统计方法，它假设数据是**由单一的最优参数值 θ 生成**，并在此基础上对参数进行估计。而另一种方法是考虑到所有的的参数值以及这些参数的先验概率分布，通过贝叶斯准则来估计参数的后验分布情况，**贝叶斯统计 (`Bayesian Statistics`) 认为训练数据是确定的，而参数是随机且不唯一的，每个参数都有相应的概率**。

在观察到数据前，将 θ 的已知知识称为先验概率分布 (`Prior Probability Distribution`) $p(\theta)$。现在有一组数据样本 $\{\boldsymbol{x}^{(1)},\cdots,\boldsymbol{x}^{(m)}\}$，将数据似然及先验代入贝叶斯规则，得到：

$$p(\theta\mid\boldsymbol{x}^{(1)},\cdots,\boldsymbol{x}^{(m)})=\frac{p(\boldsymbol{x}^{(1)},\cdots,\boldsymbol{x}^{(m)}\mid\theta)p(\theta)}{p(\boldsymbol{x}^{(1)},\cdots,\boldsymbol{x}^{(m)})} \tag{4.15}$$

在贝叶斯常用情景下，**先验是相对均匀的分布或高熵的高斯分布**，观测数据通常会使后验的熵下降，并集中在几个可能性很高的值。当训练数据有限时，贝叶斯方法通常泛化得更好。

贝叶斯估计**假设已知的是参数的先验分布情况和模型的类簇，之后利用数据集的样本点根据贝叶斯准则来对参数的分布情况进行修正**，它得到的结果不是一个单一的参数值，而是根据参数先验分布和真实样本得到的修正过后的参数分布，即参数的后验分布 (`Posterior Distribution`)。根据贝叶斯估计，在已知 m 个样本后，估计第 $m+1$ 的样本分布的公式如下：

$$p(\boldsymbol{x}^{(m+1)}\mid\boldsymbol{x}^{(1)},\boldsymbol{x}^{(2)}...,\boldsymbol{x}^{(m)})=\int p(\boldsymbol{x}^{(m+1)}\mid\theta)p(\theta|\boldsymbol{x}^{(1)}\boldsymbol{x}^{(2)}...,\boldsymbol{x}^{(m)})d\theta \tag{4.16}$$

可以看到，不同于频率学派单点估计的方法，贝叶斯估计在对未知数据预测时，将所有的参数分布都进行了考虑，同时按照参数的概率密度情况进行加权。(贝叶斯方法用于推断和预测可以参考第十一章)

4.7 最大后验估计

完整的贝叶斯估计需要使用参数的完整分布进行预测，然而对绝大多数的机器学习任务而言，这将会导致十分繁重的计算。一种合理的方式是利用最大后验估计 (`Maximum A Posteriori`, MAP) 来**选取一个计算可行的单点估计参数作为贝叶斯估计的近似解**，公式如下：

$$\theta_{MAP} = \arg\max_{\theta} \log p(\theta \mid \boldsymbol{x}) = \arg\max_{\theta} \log p(\boldsymbol{x} \mid \theta) + \log p(\theta) \tag{4.17}$$

可以看到 MAP 的估计实际上就是对数似然加上参数的先验分布。实际上，在参数服从高斯分布的情况下，上式的右边就对应着 L2 正则项；在 Laplace 的情况下，对应着 L1 的正则项；在均匀分布的情况下则为 0，等价于 MLE。

4.7.1 举例：线性回归

线性回归 (`Linear Regression`) 的目标：获得一个函数 f，满足 $f(\boldsymbol{x}) = \hat{y}$，其中 $\boldsymbol{x} \in \mathbb{R}^n, \hat{y} \in \mathbb{R}$，使得 $\hat{y}$ 接近于真实的标签 y。

我们定义线性回归的输出为：

$$f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$$

其中 $\boldsymbol{w} \in \mathbb{R}^n$ 是我们需要学习的参数。

在线性回归中，对任务 `T` 的定义：通过输出 $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$，从 $\boldsymbol{x}$ 预测 y。假设一共 m 个样本。

现在通过 MLE 解释为什么性能度量方式是均方误差（`Mean Squared Error`）。

我们将模型预测结果和真实标签的差值定义为残差 (`residual`)：$\epsilon = y - f(\boldsymbol{x})$。如果每一次的观测都属于独立事件，所有观测误差的期望和方差应该都一致：这符合中心极限定理，应该构成正态分布，并且误差的期望值应该是 0。所以大多数情况下，可以认为这个误差服从高斯分布，如下：

$$\epsilon \sim N(0, \sigma^2) \tag{4.18}$$

于是可以得到我们的观测到的标签服从如下高斯分布：$y \sim N(f(\boldsymbol{x}), \sigma^2)$。此时，我们定义了产出观测数据的模型，处于 “模型已定，参数未知” 的情况，找到一组参数使我们观测到一系列 y 的概率最大 (最大似然估计的思路)。观察到结果 y 的概率密度函数如下：

$$p(y \mid \boldsymbol{x}, \boldsymbol{w}, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y - f(\boldsymbol{x}))^2}{2\sigma^2}) \tag{4.19}$$

将似然函数记作：$\mathcal{L}(\boldsymbol{w}, \boldsymbol{X}, \sigma) = \prod_{i=1}^{m} p(y^{(i)} \mid \boldsymbol{x}^{(i)}, \boldsymbol{w}, \sigma)$

对其取对数并化简：

$$\begin{aligned}
\ln \mathcal{L}(\boldsymbol{w}, \boldsymbol{X}, \sigma) &= \ln \prod_{i=1}^{m} p(y^{(i)} \mid \boldsymbol{x}^{(i)}, \boldsymbol{w}, \sigma) \\
&= \sum_{i=1}^{m} \ln\, p(y^{(i)} \mid \boldsymbol{x}^{(i)}, \boldsymbol{w}, \sigma) \\
&= \sum_{i=1}^{m} \ln\{\frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)})^2}{2\sigma^2})\} \\
&= m\, \ln\{\frac{1}{\sqrt{2\pi\sigma^2}}\} - \frac{1}{2\sigma^2} \sum_{i=1}^{m} (y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)})^2
\end{aligned} \tag{4.20}$$

σ 可以假设为任意大于 0 的常数，优化问题化为剩下部分最大化 (因为负号变成最小化)：

$$\boldsymbol{w}_{\text{MLE}} = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{m} (y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)})^2 \tag{4.21}$$

这与 MSE 一致。

接下来考虑参数具有先验知识的情况

如果对 MLE 加入**高斯先验分布**，假设我们要求解的参数 $\boldsymbol{w}$ 本身服从一个先验分布：$\boldsymbol{w} \sim N(\boldsymbol{0}, \boldsymbol{\Sigma})$。这里我们就简单化 $\boldsymbol{w} \sim N(0, \gamma^2)$。此时，MAP 最大化的目标函数如下：

$$\begin{aligned}
\mathcal{L}(\boldsymbol{w}) &= p(y \mid \boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w}) \\
&= \prod_{i=1}^{m} \{\frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)})^2}{2\sigma^2})\} \prod_{j=1}^{n} \{\frac{1}{\sqrt{2\pi\gamma^2}} \exp(-\frac{(\boldsymbol{w}_j)^2}{2\gamma^2})\}
\end{aligned} \tag{4.22}$$

取对数后，化简整理得如下结果：

$$\ln\mathcal{L}(\boldsymbol{w}) = n \ln\frac{1}{\sqrt{2\pi\sigma^2}} + m \ln\frac{1}{\sqrt{2\pi\gamma^2}} - \frac{1}{2\sigma^2} \sum_{i=1}^{m} (y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)})^2 - \frac{1}{2\gamma^2} \boldsymbol{w}^\top \boldsymbol{w} \tag{4.23}$$

其中，σ 和 γ 均看作是常数，它们的取值会影响两个目标 (likelihood prior) 的权重，所以引入超参数 λ 来表示先验的权重，最终有：

$$\boldsymbol{w}_{\text{MAP}} = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{m} (y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)})^2 + \lambda ||\boldsymbol{w}||^2 \tag{4.24}$$

上式就是标准的岭回归 (Ridge Regression) 公式，就是在最小二乘法的基础上增加的参数本身的先验分布，并认为参数本身服从高斯分布。

如果对 MLE 加入**拉普拉斯分布**，同样的步骤，通过 MAP 可以化简得到 (LASSO)：

$$\boldsymbol{w}_{\text{MAP}} = \arg\min_{\boldsymbol{w}} \sum_{i=1}^{m} (y^{(i)} - \boldsymbol{w}^{\top}\boldsymbol{x}^{(i)})^2 + \lambda||\boldsymbol{w}||^1 \tag{4.25}$$

如果对 MLE 加入**均匀分布**，最终 MAP 得到的和 MLE 一样。

自定义实现

[4]:
```python
class NaiveBayes():

    def __init__(self):
        self.parameters = [] # 保存每个特征针对每个类的均值和方差
        self.y = None
        self.classes = None

    def fit(self, X, y):
        self.y = y
        self.classes = np.unique(y) # 类别
        # 计算每个特征针对每个类的均值和方差
        for i, c in enumerate(self.classes):
            # 选择类别为 c 的 X
            X_where_c = X[np.where(self.y == c)]
            self.parameters.append([])
            # 添加均值与方差
            for col in X_where_c.T:
                parameters = {"mean": col.mean(), "var": col.var()}
                self.parameters[i].append(parameters)

    def _calculate_prior(self, c):
        """
        先验函数。
        """
        frequency = np.mean(self.y == c)
        return frequency

    def _calculate_likelihood(self, mean, var, X):
        """
        似然函数。
        """
        # 高斯概率
        eps = 1e-4 # 防止除数为 0
        coeff = 1.0 / math.sqrt(2.0 * math.pi * var + eps)
        exponent = math.exp(-(math.pow(X - mean, 2) / (2 * var + eps)))
        return coeff * exponent

    def _calculate_probabilities(self, X):
        posteriors = []
        for i, c in enumerate(self.classes):
            posterior = self._calculate_prior(c)
            for feature_value, params in zip(X, self.parameters[i]):
                # 独立性假设
                # P(x1,x2|Y) = P(x1|Y)*P(x2|Y)
                likelihood = self._calculate_likelihood(params["mean"], params["var"], feature_value)
                posterior *= likelihood
            posteriors.append(posterior)
        # 返回具有最大后验概率的类别
        return self.classes[np.argmax(posteriors)]

    def predict(self, X):
```

```
        y_pred = [self._calculate_probabilities(sample) for sample in X]
        return y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

用自定义贝叶斯估计，*iris* 数据集测试

[5]:
```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

[6]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
```

```
[5.8 2.6 4.  1.2] 1.0
```

[7]:
```
model = NaiveBayes()
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
1.0
```

用 sklearn 实现贝叶斯估计，*iris* 数据集测试

[8]:
```
# 从 sklearn 包中调用 GaussianNB 实现贝叶斯估计
from sklearn.naive_bayes import GaussianNB
skl_model = GaussianNB()
skl_model.fit(X_train, y_train)
print(skl_model.score(X_test, y_test))
```

```
1.0
```

4.8 监督学习方法

4.8.1 概率监督学习

通过定义一族不同的概率分布，可以将线性回归扩展到分类情况中：$p(y \mid \boldsymbol{x}; \theta) = N(y; \theta^\top \boldsymbol{x}, I)$。

由于二元变量上的分布中，均值必须始终在 0 和 1 之间，为解决这个问题，可以使用 logistic sigmoid 函数将线性函数的输出压缩到区间 (0, 1) 上，则概率为：

$$p(y = 1 \mid \boldsymbol{x}; \theta) = \sigma(\theta^\top \boldsymbol{x}) \tag{4.26}$$

这个方法称为**逻辑回归** (`Logistic Regression`)。其中 logistic sigmoid 函数描述为 $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$。其导数 $\sigma'(x) = \sigma(x)(1 - \sigma(x))$。

具体展开为：

$$\begin{aligned} p(y = 1 \mid \boldsymbol{x}; \theta_0, \theta_1) &= \frac{\exp(\theta_0 + \theta_1 \boldsymbol{x})}{1 + \exp(\theta_0 + \theta_1 \boldsymbol{x})} = \pi(\boldsymbol{x}; \theta_0, \theta_1) \\ p(y = 0 \mid \boldsymbol{x}; \theta_0, \theta_1) &= \frac{1}{1 + \exp(\theta_0 + \theta_1 \boldsymbol{x})} = 1 - \pi(\boldsymbol{x}; \theta_0, \theta_1) \end{aligned} \tag{4.27}$$

对于二分类问题训练目标, 训练目标为最大似然:

$$\prod_{i}^{m}[\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)]^{y^{(i)}}[1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)]^{1-y^{(i)}} \tag{4.28}$$

最大化似然相当于最小化其负对数似然形式：

$$\mathcal{L}(\theta_0,\theta_1)=-\sum_{i=1}^{m}\left[y^{(i)}\log\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)+(1-y^{(i)})\log(1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1))\right] \tag{4.29}$$

梯度下降法更新参数：

$$\begin{aligned}\frac{\partial\mathcal{L}(\theta)}{\partial\theta_j}&=-\sum_{i=1}^{m}(y^{(i)}\frac{1}{\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)}-(1-y^{(i)})\frac{1}{1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)})\frac{\partial\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)}{\partial\theta_j}\\&=-\sum_{i=1}^{m}(y^{(i)}\frac{1}{\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)}-(1-y^{(i)})\frac{1}{1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)})\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1)(1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1))\frac{\partial\theta^{\top}\boldsymbol{x}^{(i)}}{\partial\theta_j}\\&=-\sum_{i=1}^{m}(y^{(i)}(1-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1))-(1-y^{(i)})\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1))x_j^{(i)}\\&=-\sum_{i=1}^{m}(y^{(i)}-\pi(\boldsymbol{x}^{(i)};\theta_0,\theta_1))x_j^{(i)}\end{aligned} \tag{4.30}$$

然后梯度更新 $\theta\leftarrow\theta-\epsilon\frac{\partial\mathcal{L}(\theta)}{\partial\theta}$。$\epsilon$ 为学习率，表示更新的步长。

预测过程：用最优 $\widehat{\theta}$ 预测新的输入 $\boldsymbol{x}$。

$$\widehat{\pi}(x)=\frac{\exp\left(\widehat{\theta}_0+\widehat{\theta}_1\boldsymbol{x}\right)}{1+\exp\left(\widehat{\theta}_0+\widehat{\theta}_1\boldsymbol{x}\right)} \tag{4.31}$$

逻辑回归的另一种描述：逻辑回归实质上是用线性模型拟合**对数几率** (`log odds`)：

$$\log\left(\frac{p(y=1)}{1-p(y=1)}\right)=\theta^{\top}\boldsymbol{x} \tag{4.32}$$

自定义实现

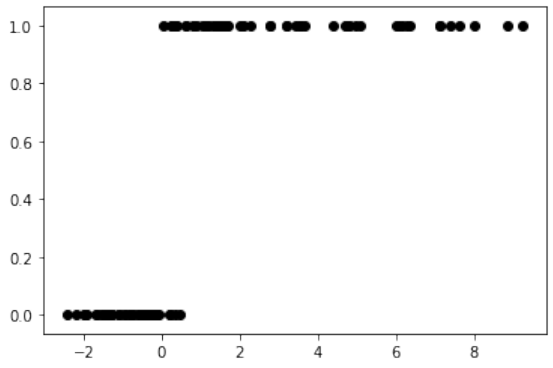
[9]:
```python
def Sigmoid(x):
    return 1/(1 + np.exp(-x))

class LogisticRegression():

    def __init__(self, learning_rate=.1):
        self.param = None
        self.learning_rate = learning_rate
        self.sigmoid = Sigmoid

    def _initialize_parameters(self, X):
        n_features = np.shape(X)[1]
        # 初始化参数 theta,  [-1/sqrt(N), 1/sqrt(N)]
        limit = 1 / math.sqrt(n_features)
        self.param = np.random.uniform(-limit, limit, (n_features,))

    def fit(self, X, y, n_iterations=4000):
        self._initialize_parameters(X)
        # 参数 theta 的迭代更新
        for i in range(n_iterations):
            # 求预测
            y_pred = self.sigmoid(X.dot(self.param))
            # 最小化损失函数，参数更新公式
            self.param -= self.learning_rate * -(y - y_pred).dot(X)

    def predict(self, X):
        y_pred = self.sigmoid(X.dot(self.param))
        return y_pred

    def score(self, X, y):
```

```
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

用自定义逻辑回归，合成数据集测试

[10]:
```
# 二分类问题，生成数据
n_samples = 100
np.random.seed(0)
x = np.random.normal(size=n_samples)
y = (x > 0).astype(np.float)
x[x > 0] *= 4
x += .3 * np.random.normal(size=n_samples)
x = x[:, np.newaxis]  # 输入增加一维，用于与 theta_0 结合
plt.scatter(x, y, color='black')
```

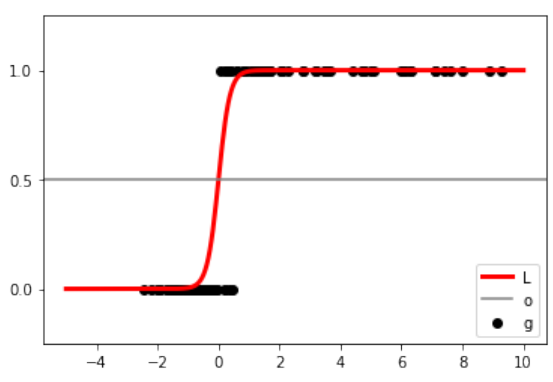
[10]:
```
<matplotlib.collections.PathCollection at 0x113db1ac8>
```



[11]:
```
# 训练一个逻辑回归模型
model = LogisticRegression()
model.fit(x, y)
```

[12]:
```
# 预测
x_test = np.linspace(-5, 10, 300)
x_test = x_test[:, np.newaxis]
prob = model.predict(x_test).ravel()

plt.plot(x_test, prob, color='red', linewidth=3)
plt.scatter(x, y, color='black');
plt.axhline(0.5, color='0.5');
plt.ylim(-0.25, 1.25);
plt.yticks([0, 0.5, 1]);
plt.legend(('Logistic Regression Model'), loc='lower right')
```

[12]:
```
<matplotlib.legend.Legend at 0x113e21198>
```

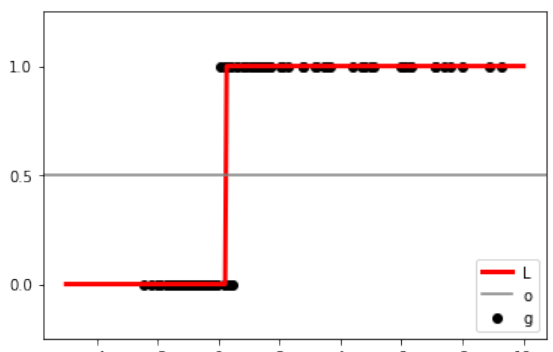


用 sklearn 实现逻辑回归，合成数据集测试

[13]:
```
from sklearn.linear_model import LogisticRegression
# C 表示正则化系数  的倒数， solver 表示优化算法选择参数
skl_model = LogisticRegression()
skl_model.fit(x, y)
```

```
# 预测
x_test = np.linspace(-5, 10, 300)
x_test = x_test[:, np.newaxis]
prob = skl_model.predict(x_test).ravel()

plt.plot(x_test, prob, color='red', linewidth=3)
plt.scatter(x, y, color='black');
plt.axhline(0.5, color='0.5');
plt.ylim(-0.25, 1.25);
plt.yticks([0, 0.5, 1]);
plt.legend(('Logistic Regression Model'), loc='lower right')
```

[13]: <matplotlib.legend.Legend at 0x1143a2f60>



用自定义逻辑回归，*iris* 数据集测试

[14]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
```

```
[5.  3.5 1.6 0.6] 0.0
```

[15]:
```
model = LogisticRegression()
model.fit(X_train, y_train)
# 测试数据
print(model.score(X_test, y_test))
```

```
1.0
```

用 sklearn 实现逻辑回归，*iris* 数据集测试

[16]:
```
# 从 sklearn 包中调用 LogisticRegression 测试
from sklearn.linear_model import LogisticRegression
skl_model = LogisticRegression()
skl_model.fit(X_train, y_train)
# 测试数据
print(skl_model.score(X_test, y_test))
```

```
1.0
```

4.8.2 支持向量机

支持向量机 (Support Vector Machine，SVM) 同样基于线性函数 $\boldsymbol{w}^\top \boldsymbol{x} + b$，不同于逻辑回归，SVM 不输出概率，只输出类别。

SVM 的思想是**找到一个分割线 (或分割平面)，把两个类别的点分得越开越好。**

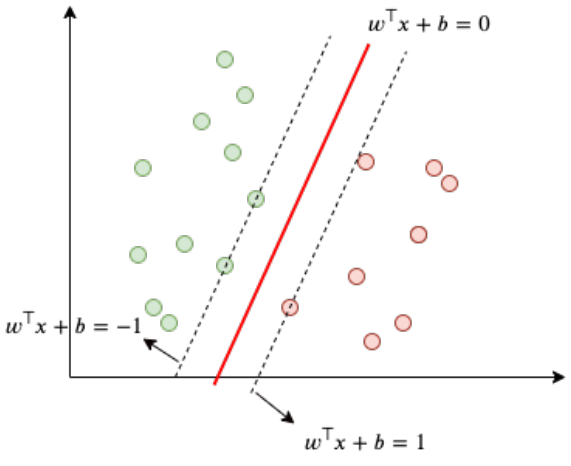


图 4.3. 支持向量机

在超平面 $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$ 确定的情况下，$|\boldsymbol{w}^\top \boldsymbol{x} + b|$ 能够表示点 $\boldsymbol{x}$ 到超平面的距离远近，而通过观察 $\boldsymbol{w}^\top \boldsymbol{x} + b$ 的符号与类型标记 y 符号是否一致，可以判断分类是否正确。于是定义函数间隔 (`Functional Margin`) 的概念：$\hat{\gamma} = y(\boldsymbol{w}^\top \boldsymbol{x} + b) = yf(\boldsymbol{x})$。**超平面 $(\boldsymbol{w}, b)$ 关于训练数据集中所有样本点 $(\boldsymbol{x}^{(i)}, y^{(i)})$ 的函数间隔最小值，便成为超平面 $(\boldsymbol{w}, b)$ 关于数据集的的函数间隔**：$\hat{\gamma} = \min \hat{\gamma}_i (i = 1, \dots m)$。

但成比例的改变 $\boldsymbol{w}, b$ (如都变成原来的 2 倍)，则函数间隔 $f(\boldsymbol{x})$ 的值变成了原来的 2 倍，但此时超平面却没有改变。因此引入真正定义**点到超平面的距离–几何间隔** (`Geometrical Margin`) 的概念：$\widetilde{\gamma} = \frac{\hat{\gamma}}{\|\boldsymbol{w}\|}$。

因此，目标函数可以写作：$\max \widetilde{\gamma}$。

同时，限定约束条件：$s.t., y^{(i)}(\boldsymbol{w}^\top \boldsymbol{x}^{(i)} + b) = \hat{\gamma}_i \geq \hat{\gamma}, \qquad i = 1, ..., m$

下面为了和传统写法统一以及方便理解，在本章的剩余内容中，我会将 $(\boldsymbol{x}^{(i)}, y^{(i)})$ 写作 $(\boldsymbol{x}_i, y_i)$，将样本 $\boldsymbol{x}$ 的第 j 个特征写作 x^j。如果令函数间隔 $\hat{\gamma} = 1$，则有 $\widetilde{\gamma} = \frac{1}{\|\boldsymbol{w}\|}$。上述目标函数便转化成：

$$\begin{cases} \max \dfrac{1}{\|\omega\|} \\ s.t., y_i(\omega^\top \boldsymbol{x}_i + b) = \hat{\gamma}_i \geq 1, i = 1, ..., m \end{cases} \tag{4.33}$$

该式等价于求解：

$$\begin{cases} \min \dfrac{1}{2}\|\boldsymbol{w}\|^2 \\ s.t., y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) = \hat{\gamma}_i \geq 1, i = 1, ..., m \end{cases} \tag{4.34}$$

由于现在的目标函数是二次的，约束条件是线性的，所以它是一个凸二次规划问题。这个问题可以用现成的 QP (`Quadratic Programming`) 优化包进行求解。由于这个问题的特殊结构，还可以通过拉格朗日对偶性 (`Lagrange Duality`) 变换到对偶变量 (`dual variable`) 的优化问题，即通过求解与原问题等价的对偶问题 (`Dual Problem`) 得到原始问题的最优解，这就是线性可分条件下支持向量机的对偶算法。其优点在于：

- 对偶问题往往更容易求解。
- 可以自然的引入核函数，进而推广到非线性分类问题。

定义拉格朗日函数如下式：

$$\mathcal{L}(\boldsymbol{w}, b, \alpha) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^{m} \alpha_i [y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) - 1] \tag{4.35}$$

求解这个对偶学习问题，分为三个步骤 (参考第四章 KKT 条件)：

- 令 $\mathcal{L}(\boldsymbol{w}, b, \alpha)$ 关于 $\boldsymbol{w}$ 和 b 最小化。
- 利用 SMO 算法求解对偶问题中的拉格朗日乘子，求对 α 的极大。
- 求参数 $\boldsymbol{w}$，b 。

首先固定 α，令 $\mathcal{L}$ 关于 $\boldsymbol{w}$,b 最小化

分别对 $\boldsymbol{w}$，b 求偏导数:

$$\begin{cases} \dfrac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 \Rightarrow \boldsymbol{w} = \sum\limits_{i=1}^{m} \alpha_i y_i \boldsymbol{x}_i \\ \dfrac{\partial \mathcal{L}}{\partial b} = 0 \Rightarrow \sum\limits_{i=1}^{m} \alpha_i y_i = 0 \end{cases} \tag{4.36}$$

代入原式中得到：

$$\begin{aligned}\mathcal{L}(\boldsymbol{w},b,\alpha) &= \frac{1}{2}\boldsymbol{w}^\top\cdot\sum_{i=1}^{m}\alpha_i\boldsymbol{x}_i y_i-\boldsymbol{w}^\top\sum_{i=1}^{m}\alpha_i y_i\boldsymbol{x}_i-b\sum_{i=1}^{m}\alpha_i y_i+\sum_{i=1}^{m}\alpha_i\\ &= -\frac{1}{2}\boldsymbol{w}^\top\sum_{i=1}^{m}\alpha_i y_i\boldsymbol{x}_i-b\sum_{i=1}^{m}\alpha_i y_i+\sum_{i=1}^{m}\alpha_i\\ &= -\frac{1}{2}[\sum_{i=1}^{m}\alpha_i y_i\boldsymbol{x}_i]^\top\sum_{i=1}^{m}\alpha_i y_i\boldsymbol{x}_i-b\sum_{i=1}^{m}\alpha_i y_i+\sum_{i=1}^{m}\alpha_i\\ &= -\frac{1}{2}\sum_{i=1}^{m}\alpha_i y_i(\boldsymbol{x}_i)^\top\cdot\sum_{i=1}^{m}\alpha_i y_i\boldsymbol{x}_i-b\sum_{i=1}^{m}\alpha_i y_i+\sum_{i=1}^{m}\alpha_i\\ &= -\frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j\boldsymbol{x}_i^\top\boldsymbol{x}_j+\sum_{i=1}^{m}\alpha_i\end{aligned}$$

利用 SMO 算法求解对偶问题中的拉格朗日乘子

经过上一步后得到此时的目标函数：

$$\begin{aligned}&\max_{\alpha}\sum_{i=1}^{m}\alpha_i-\frac{1}{2}\sum_{i=1}^{m}\alpha_i\alpha_j y_i y_j\boldsymbol{x}_i^\top\boldsymbol{x}_j\\ &s.t.,\alpha_i\geq 0,i=1,...,m\\ &\sum_{i=1}^{m}\alpha_i y_i=0\end{aligned}\tag{4.38}$$

通过 SMO 算法可以求解对偶问题中的拉格朗日乘子 α。

求参数 $\boldsymbol{w}$, b

上面一步求出了拉格朗日乘子 α，可以计算出：$\boldsymbol{w}^*=\sum_{i=1}^{m}\alpha_i y_i\boldsymbol{x}_i$。

因为支持向量处于边界点，满足：

$$\begin{cases}\max\limits_{y_i=-1}\boldsymbol{w}^\top\boldsymbol{x}_i+b=-1\\ \min\limits_{y_i=1}\boldsymbol{w}^\top\boldsymbol{x}_i+b=1\end{cases}\tag{4.39}$$

由于对于边界上的支持向量有 $y(\boldsymbol{w}^\top\boldsymbol{x}+b)=1$，可以用支持向量求 b 值：$b^*=y_j-\sum_{i=1}^{m}\alpha_i y_i\langle\boldsymbol{x}_i,\boldsymbol{x}_j\rangle$。可以看出，$\boldsymbol{w}^*$ 和 b^* 只依赖于数据中 $\alpha>0$ 的样本点。

分类函数：

$$\begin{aligned}f(\boldsymbol{x}) &= (\sum_{i=1}^{m}\alpha_i y_i\boldsymbol{x}_i)^\top\boldsymbol{x}+b\\ &= \sum_{i=1}^{m}\alpha_i y_i\langle\boldsymbol{x}_i,\boldsymbol{x}\rangle+b\end{aligned}\tag{4.40}$$

核技巧

对于非线性的情况，SVM 的处理方法是选择一个核函数 $k(\cdot,\cdot)$，通过将数据映射到高维空间，来解决在原始空间中线性不可分的问题。其思想是**通过核函数将输入空间映射到高维特征空间，最终在高维特征空间中构造出最优分离超平面，从而把平面上本身不好分的非线性数据分开**。

计算两个向量在隐式映射过后的空间中的内积的函数叫做核函数 (`Kernel Function`)。核函数相当于将原来的分类函数 $f(\boldsymbol{x})=\sum_{i=1}^{m}\alpha_i y_i\langle\boldsymbol{x}_i,\boldsymbol{x}\rangle+b$ 映射成了 $f(\boldsymbol{x})=\sum_{i=1}^{m}\alpha_i y_i\langle\phi(\boldsymbol{x}_i),\phi(\boldsymbol{x})\rangle+b$。

其中的 α 值是可以通过求解由映射变形来的对偶问题得到：

$$\begin{aligned}&\max_{\alpha}\sum_{i=1}^{m}\alpha_i-\frac{1}{2}\sum_{i=1}^{m}\alpha_i\alpha_j y_i y_j\langle\phi(\boldsymbol{x}_i),\phi(\boldsymbol{x}_j)\rangle\\ &s.t.,\alpha_i\geq 0,i=1,...,m\\ &\sum_{i=1}^{m}\alpha_i y_i=0\end{aligned}\tag{4.41}$$

这里的核函数描述为：对于所有的数据点 $\boldsymbol{x},\boldsymbol{z}$，满足 $k(\boldsymbol{x},\boldsymbol{z})=\langle\phi(\boldsymbol{x})\cdot\phi(\boldsymbol{z})\rangle$。**核函数的作用等同于用特征函数 $\phi(\boldsymbol{x})$ 预处理所有输入，然后在新的转换空间学习线性模型**。

常用核函数

线性核 (`Linear kernel`)：$k(u,v)=\langle u,v\rangle$

多项式核 (`Polynomial kernel`)：$k(u,v)=(1+\langle u,v\rangle)^d$

高斯核 (`RBF kernel`)：$k(u,v)=N(u-v;0,\sigma^2 I)=\exp\left(-\frac{||u-v||^2}{2\sigma^2}\right)$

其中 $N(x;\mu,\Sigma)$ 是标准正态密度，也被称为径向基函数。

自定义实现

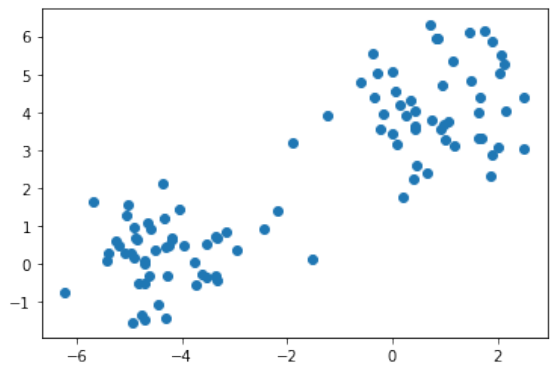
[17]:
```
import cvxopt

# 隐藏 cvxopt 输出
cvxopt.solvers.options['show_progress'] = False

def linear_kernel(**kwargs):
    """
    线性核
    """
    def f(x1, x2):
        return np.inner(x1, x2)
    return f

def polynomial_kernel(power, coef, **kwargs):
    """
    多项式核
    """
    def f(x1, x2):
        return (np.inner(x1, x2) + coef)**power
    return f

def rbf_kernel(gamma, **kwargs):
    """
    高斯核
    """
    def f(x1, x2):
        distance = np.linalg.norm(x1 - x2) ** 2
        return np.exp(-gamma * distance)
    return f

class SupportVectorMachine():

    def __init__(self, kernel=linear_kernel, power=4, gamma=None, coef=4):
        self.kernel = kernel
        self.power = power
        self.gamma = gamma
        self.coef = coef
        self.lagr_multipliers = None
        self.support_vectors = None
        self.support_vector_labels = None
        self.intercept = None

    def fit(self, X, y):
        n_samples, n_features = np.shape(X)
        # gamma 默认设置为 1 / n_features
        if not self.gamma:
            self.gamma = 1 / n_features
        # 定义核函数
        self.kernel = self.kernel(
            power=self.power,
            gamma=self.gamma,
            coef=self.coef)
        # 计算 Gram 矩阵
        kernel_matrix = np.zeros((n_samples, n_samples))
        for i in range(n_samples):
            for j in range(n_samples):
```

```
            kernel_matrix[i, j] = self.kernel(X[i], X[j])
        # 构造二次规划问题
        # 形式为 min (1/2)x.T*P*x+q.T*x, s.t. G*x<=h, A*x=b
        P = cvxopt.matrix(np.outer(y, y) * kernel_matrix, tc='d')
        q = cvxopt.matrix(np.ones(n_samples) * -1)
        A = cvxopt.matrix(y, (1, n_samples), tc='d')
        b = cvxopt.matrix(0, tc='d')
        G = cvxopt.matrix(np.identity(n_samples) * -1)
        h = cvxopt.matrix(np.zeros(n_samples))
        # 用 cvxopt 求解二次规划问题
        minimization = cvxopt.solvers.qp(P, q, G, h, A, b)
        lagr_mult = np.ravel(minimization['x'])
        # 非 0 的 alpha 值
        idx = lagr_mult > 1e-7
        # alpha 值
        self.lagr_multipliers = lagr_mult[idx]
        # 支持向量
        self.support_vectors = X[idx]
        # 支持向量的标签
        self.support_vector_labels = y[idx]
        # 通过第一个支持向量计算 b
        self.intercept = self.support_vector_labels[0]
        for i in range(len(self.lagr_multipliers)):
            self.intercept -= self.lagr_multipliers[i] * self.support_vector_labels[
                i] * self.kernel(self.support_vectors[i], self.support_vectors[0])

    def predict(self, X):
        y_pred = []
        for sample in X:
            # 对于输入的 x, 计算 f(x)
            prediction = 0
            for i in range(len(self.lagr_multipliers)):
                prediction += self.lagr_multipliers[i] * self.support_vector_labels[
                    i] * self.kernel(self.support_vectors[i], sample)
            prediction += self.intercept
            y_pred.append(np.sign(prediction))
        return np.array(y_pred)

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

用自定义的支持向量机，合成数据集测试

[18]:
```
from sklearn import datasets
# 测试用例
X, y = datasets.make_blobs(n_samples=100, centers=2, random_state=3)
y[y == 0] = -1
y[y == 1] = 1
plt.scatter(X[:,0], X[:,1])
```

[18]:
```
<matplotlib.collections.PathCollection at 0x114594a58>
```



[19]:
```
model = SupportVectorMachine()
model.fit(X, y)
print(model.predict([np.array([-0.4, -0.5])]))
print(model.predict([np.array([2.6, 5.3])]))
```

```
[1.]
[-1.]
```

用 sklearn 实现支持向量机，合成数据集测试

[20]:
```
from sklearn import svm
skl_model = svm.SVC(kernel='linear', C=1000)
skl_model.fit(X, y)
# 测试数据
print(skl_model.predict([[-0.4, -0.5]]))
print(skl_model.predict([[2.6, 5.3]]))
```

```
[1]
[-1]
```

用自定义的支持向量机，*iris* 数据集测试

[21]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]


X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
y_train[y_train == 0] = -1  # 标签设为 1 和-1
```

```
[6.1 2.9 4.7 1.4] 1.0
[6.1 2.9 4.7 1.4] 1.0
```

[22]:
```
model = SupportVectorMachine()
model.fit(X_train, y_train)
# 测试数据
print(model.score(X_test, y_test))
```

```
0.43333333333333335
```

用 sklearn 实现支持向量机，*iris* 数据集测试

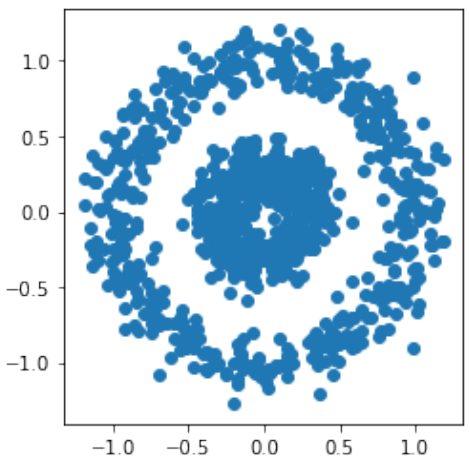
[23]:
```
from sklearn import svm
skl_model = svm.SVC(kernel='linear', C=1000)
skl_model.fit(X_train, y_train)
# 测试数据
print(skl_model.score(X_test, y_test))
```

```
0.43333333333333335
```

用 sklearn 展示核函数

[24]:
```
from sklearn import svm
from sklearn.model_selection import cross_val_score
X, y = datasets.make_circles(n_samples=1000, factor=0.3, noise=0.1, random_state=2019)
plt.subplot(111, aspect='equal')
plt.scatter(X[:,0], X[:,1])
```

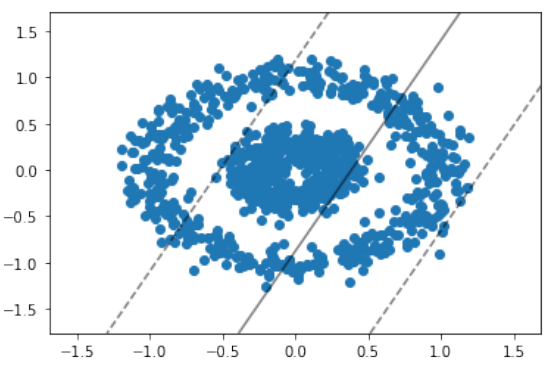
[24]:
```
<matplotlib.collections.PathCollection at 0x1146513c8>
```



[25]:
```
xx = np.linspace(X[:,0].min()-0.5, X[:,0].max()+0.5, 30)
yy = np.linspace(X[:,1].min()-0.5, X[:,1].max()+0.5, 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
```

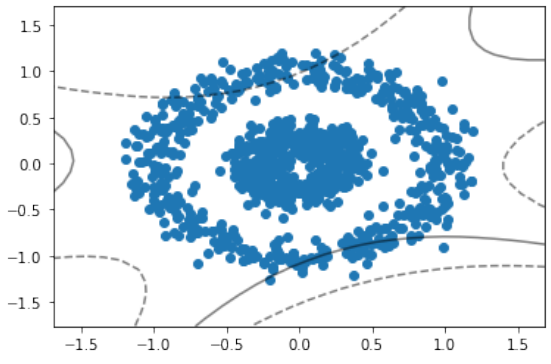
[26]:
```
# 线性核
clf = svm.SVC(kernel='linear', C=1000)
clf.fit(X,y)
Z = clf.decision_function(xy).reshape(XX.shape)
# 画决策边界和间隔
plt.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--']);
plt.scatter(X[:,0], X[:,1]);
print('10-fold cv scores with linear kernel: ', np.mean(cross_val_score(clf, X, y, cv=10)))
```

```
10-fold cv scores with linear kernel:  0.638
```



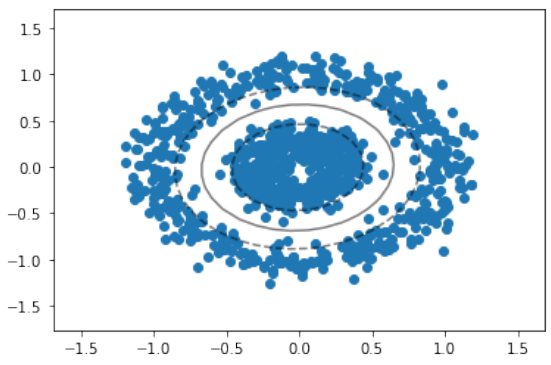
[27]:
```
# 多项式核
clf = svm.SVC(kernel='poly', gamma='auto')
clf.fit(X,y)
Z = clf.decision_function(xy).reshape(XX.shape)
# 画决策边界和间隔
plt.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
plt.scatter(X[:,0], X[:,1]);
print('10-fold cv scores with Polynomial kernel: ', np.mean(cross_val_score(clf, X, y, cv=10)))
```

```
10-fold cv scores with Polynomial kernel:  0.5290000000000001
```



[28]:
```
# RBF 高斯核
clf = svm.SVC(kernel='rbf', gamma='auto')
clf.fit(X,y)
Z = clf.decision_function(xy).reshape(XX.shape)
# 画决策边界和间隔
plt.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])
plt.scatter(X[:,0], X[:,1]);
print('10-fold cv scores with RBF kernel: ', np.mean(cross_val_score(clf, X, y, cv=10)))
```

```
10-fold cv scores with RBF kernel:  1.0
```



4.8.3 k-近邻

k-近邻 (k-Nearest Neighbors) 的思想是给定测试样本，基于某种距离度量 (一般使用欧几里德距离) 找出训练集中与其最靠近的 k 个训练样本，然后基于这 k 个 “邻居” 的信息来进行预测 (**“物以类聚”**)。

算法步骤:

- 根据给定的距离度量，在训练集中找出与 $\boldsymbol{x}$ 最近邻的 k 个点，涵盖这 k 个点的 $\boldsymbol{x}$ 的邻域记作 $\mathcal{N}_k(\boldsymbol{x})$。
- 在 $\mathcal{N}_k(\boldsymbol{x})$ 中根据分类决策规则，如多数表决决定 $\boldsymbol{x}$ 的类别 y。

可见，决定了 k 近邻模型的三个基本要素——距离度量、k 值的选择、分类决策规则。

距离度量

- 一般使用欧几里德距离
- 相关度 (如皮尔逊相关系数)
- 曼哈顿距离 (Manhattan Distance)

k 值的选择

- 当选择比较小的 k 值的时候，表示使用较小领域中的样本进行预测，训练误差会减小，但是会导致模型变得复杂，容易导致过拟合。
- 当选择较大的 k 值的时候，表示使用较大领域中的样本进行预测，训练误差会增大，同时会使模型变得简单，容易导致欠拟合。

分类决策规则

- **多数表决法**：每个邻近样本的权重是一样的，最终预测的结果为出现类别最多的那个类。
- **加权多数表决法**：每个邻近样本的权重是不一样的，一般情况下采用权重和距离成反比的方式来计算，最终预测结果是出现权重最大的那个类别。

自定义实现

[29]:
```
class KNN():


    def __init__(self, k=10):
```

```
        self._k = k

    def fit(self, X, y):
        self._unique_labels = np.unique(y)
        self._class_num = len(self._unique_labels)
        self._datas = X
        self._labels = y.astype(np.int32)

    def predict(self, X):
        # 欧式距离计算
        dist = np.sum(np.square(X), axis=1, keepdims=True) - 2 * np.dot(X, self._datas.T)
        dist = dist + np.sum(np.square(self._datas), axis=1, keepdims=True).T
        dist = np.argsort(dist)[:,:self._k]
        return np.array([np.argmax(np.bincount(self._labels[dist][i])) for i in range(len(X))])

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

用自定义的 k-近邻，$iris$ 数据集测试

[30]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
```

```
[7.  3.2 4.7 1.4] 1.0
```

[31]:
```
model = KNN()
model.fit(X_train, y_train)
# 测试数据
print(model.score(X_test, y_test))
```

```
1.0
```

用 sklearn 实现 k-近邻，$iris$ 数据集测试

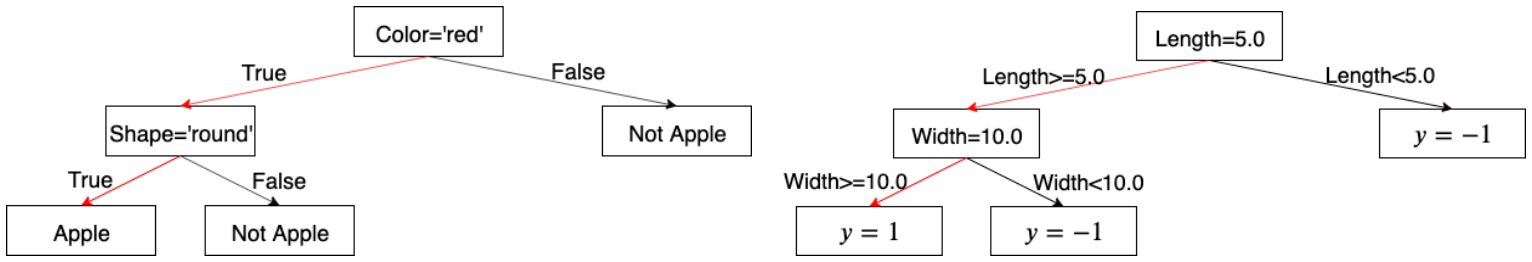
[32]:
```
# 从 sklearn 包中调用 neighbors 测试
from sklearn import neighbors
skl_model = neighbors.KNeighborsClassifier()
# 训练数据集
skl_model.fit(X_train, y_train)
# 测试数据
print(skl_model.score(X_test, y_test))
```

```
1.0
```

4.8.4 决策树

决策树 (Decision Tree) 由节点和有向边组成，一般一棵决策树包含一个根节点、若干内部节点和若干叶节点。**决策树的决策过程需要从决策树的根节点开始，待测数据与决策树中的特征节点进行比较，并按照比较结果选择选择下一比较分支，直到叶子节点作为最终的决策结果**。

目标变量采用一组离散值的决策树称为**分类树** (Classification Tree)(如下图左，常用的分类树算法有 ID3、C4.5、CART)，而目标变量采用连续值 (通常是实数) 的决策树被称为**回归树** (Regression Tree)(如下图右，常用的回归树算法有 CART)。



(i) 分类树 (ii) 回归树

图 4.4. 决策树

假设先不考虑决策树剪枝，决策树算法的核心问题在于特征选择和决策树生成。

特征选择

特征选择即**选择最优划分属性**，从当前数据的特征中选择一个特征作为当前节点的划分标准。我们希望在不断划分的过程中，决策树的分支节点所包含的样本尽可能属于同一类，即节点的 “纯度” (`Impurity`) 越来越高。而选择最优划分特征的标准不同，也导致了决策树算法的不同。常见的方法基于以下几种：

- 信息增益 (`Information Gain`)
- 信息增益率 (`Information Gain Ratio`)
- 基尼指数 (`Gini Index`)

信息增益:

划分标准主要基于信息论。在一个有 K 个类别的训练数据 D 中，假设输出类别的概率分布为：

$$P(y = k) = p_k \tag{4.42}$$

那么具有 K 个类别的样本的信息熵为：

$$H(D) = -\sum_{k=1}^{K} p_k \log p_k \tag{4.43}$$

可以看出，当整个训练数据集只有一个类别时，熵最低，为 $H_{min}(D) = -1 \cdot log1 = 0$；当训练数据集各类别为均匀分布时，熵最大，为 $H_{max}(D) = -K \cdot \frac{1}{K} \cdot \log \frac{1}{K} = -\log K$。现在假设某一离散特征 $\boldsymbol{X}_j$ (表示取所有样本 $\boldsymbol{X}$ 的第 j 个特征) 有 V 个不同的的取值，那么当以特征 x_j 来划分时可以将数据集 D 分为 V 个子集：

$$D = \sum_{v=1}^{V} D_v \tag{4.44}$$

那么划分之后的 V 个子数据集的加权熵为：

$$H(D \mid \boldsymbol{X}_j) = \sum_{v=1}^{V} \frac{|D_v|}{|D|} H(D_v) \tag{4.45}$$

当按特征 $\boldsymbol{X}_j$ 来划分数据集时的信息增益 (`Information Gain`) 定义为：

$$G(D, \boldsymbol{X}_j) = H(D) - H(D \mid \boldsymbol{X}_j) \tag{4.46}$$

信息增益表示得知特征 $\boldsymbol{X}_j$ 的信息情况下使得预测类别的信息的不确定性减少程度。

信息增益有个缺点，就是会倾向于选择类别数多的特征来做划分。假设有一列特征 (例如样本 ID) 类别数与样本数相等，如果以该特征来进行划分数据集，则数据集被划分成了单样本节点，每个节点的熵均为 0，总熵也为 0。如此一来，就得到了最大的信息增益，但是这种划分显然是不合理的。

信息增益率:

为解决信息增益的缺点，而是使用信息增益率 (`Information Gain Ratio`) 来决定使用哪个特征来划分数据集。其定义为：

$$GR(D, \boldsymbol{X}_j) = \frac{G(D, \boldsymbol{X}_j)}{IV(\boldsymbol{X}_j)} \tag{4.47}$$

其中 $IV(\boldsymbol{X}_j)$ 被称为固有值 (`Intrinsic Value`)，它等价于这个特征在数据集中的熵。

具体来说，假设考虑数据集 D 中的特征 $\boldsymbol{X}_j$，它有 V 个不同的取值 $s_1, \cdots, s_V$，那么特征 $\boldsymbol{X}_j$ 在数据集中的概率分布为：

$$P(\boldsymbol{X}_j = s_v) = \frac{|D_v|}{|D|} = p_v \tag{4.48}$$

那么数据集中该特征的固有值 (熵) 为：

$$IV(\boldsymbol{X}_j) = -\sum_{v=1}^{V} p_v \log p_v \tag{4.49}$$

基尼指数：

对于有 K 个类别的数据集 D，某一样本属于类别 k 的概率等于该类别的分布概率：

$$P(y=k)=p_k \tag{4.50}$$

那么数据集 D 的基尼指数定义如下：

$$Gini(D)=1-\sum_{k=1}^{K}p_k^2 \tag{4.51}$$

由公式可以看出，基尼指数表示的是从数据集中随机取两个样本类别不同的概率，其值越小则数据集纯度越高。

如对一个数据集 D 以特征 $\boldsymbol{X}_j$ 的一个取值 s_v 来划分，那么数据集会被划分成 $D_{\boldsymbol{X}_j=s_v}$ 和 $D_{\boldsymbol{X}_j\neq s_v}$，那么数据集 D 依照特征 $\boldsymbol{X}_j=s_v$ 划分之后的加权基尼指数为：

$$Gini(D\mid \boldsymbol{X}_j=s_v)=\frac{|D_{\boldsymbol{X}_j=s_v}|}{|D|}Gini(D_{\boldsymbol{X}_j=s_v})+\frac{|D_{\boldsymbol{X}_j\neq s_v}|}{|D|}Gini(D_{\boldsymbol{X}_j\neq s_v}) \tag{4.52}$$

决策树生成

生成算法	划分标准
ID3	信息增益
C4.5	信息增益率
CART	基尼指数

ID3 算法：

ID3 算法的核心是**在决策树各个节点上根据信息增益来选择进行划分的特征，然后递归地构建决策树**。

算法思路：首先，树的根节点中包含整个数据集，ID3 算法会遍历所有特征分别来计算划分后的信息增益，选择信息增益最大的特征；然后由该特征的不同取值建立子节点，将数据集划分到新一层的各个子节点中；对各个子节点中的数据递归进行这个过程，直到信息增益足够小或者无特征可用，最终得到决策树。

CART 算法：

分类树

CART 分类树选取特征的依据是基尼指数，在划分时会遍历所有的特征与其所有可能的取值，再全局考量选取一个最佳特征与最佳划分点。若数据集有 n 个特征，每个特征 $\boldsymbol{X}_j$ 有 V_j 种不同取值，CART 分类树可以用下式来表述：

$$(\boldsymbol{X}_*,s_*)=\arg\min\left(G(D\mid \boldsymbol{X}_j=s_v)\right),\qquad j\,from\,1\to n, v\,from\,1\to V_j \tag{4.53}$$

回归树

首先需要说明的是前面分类树的输出是叶子节点所有样本目标值的多数表决，回归树的输出是叶子节点中所有样本目标值的均值。那么对于回归任务，如何生成树？可以采用一种直观的方式来对数据集进行划分，假设某一时刻数据集 (数据子集) D 被决策树以特征 $\boldsymbol{X}_j$ 按取值 s 划分成了两部分 (或两个叶子节点)：

$$\begin{aligned}\mathrm{R}_1(\boldsymbol{X}_j,s)&=\{\boldsymbol{X}\mid \boldsymbol{X}_j<s\}\\ \mathrm{R}_2(\boldsymbol{X}_j,s)&=\{\boldsymbol{X}\mid \boldsymbol{X}_j\geq s\}\end{aligned} \tag{4.54}$$

则此次划分的优劣可以用 MSE 判断，用真实值和划分区域的预测值的最小二乘来衡量：

$$\mathrm{MSE}(\boldsymbol{X}_j,s)=\sum_{\mathrm{R}_1}(y_i-\bar{y}_{\mathrm{R}_1})^2+\sum_{\mathrm{R}_2}(y_i-\bar{y}_{\mathrm{R}_2})^2 \tag{4.55}$$

其中，$\hat{y}_{\mathrm{R}_1}$ 为 R_1 区域所有样本目标值的均值，$\hat{y}_{\mathrm{R}_2}$ 为 R_2 区域所有样本目标值的均值。

在生成回归树时，使用贪心策略，遍历所有特征下所有可能的取值，找到一个最优分割点，然后以此类推。决策树在做预测时，首先将测试样本输入决策树进行判定，判定该测试样本属于哪一个叶子节点，然后把该叶子结点内所有训练样本的目标均值作为测试样本的预测值。

除了使用 MSE 作为分裂依据之外，还有一个指标可以用作回归树的分裂：方差 (`Variance`)。当使用方差作为分裂依据时，生成树的目的是希望子节点内的值越稳定越好。

回归同样是考虑分割前后使得指标变化最大的特征 $\boldsymbol{X}_j$ 及特征取值 s。

决策树正则化

决策树算法的缺点在于极易过拟合，所以控制决策树的模型复杂度以防止过拟合是很有必要的。

首先可以设定几个参数抑制树的生长：最大的树深、分割的最少样本数、分割的最小纯度。除此之外，也可以对树的生长不做限制，然后再对树进行剪枝。CART 便使用 Cost-Complexity 剪枝方法。

我们用 T 表示一棵树，$\mathcal{L}(T)$ 表示树 T 的分类 (回归) 误差。设 α 为正则化系数，$\Omega(T)$ 是能表示树结构复杂度的函数。于是 Cost-Complexity 函数：

$$J_\alpha(T) = \mathcal{L}(T) + \alpha\Omega(T) \tag{4.56}$$

令树 T 的结构复杂度函数等于树的叶节点数：

$$\Omega(T) = |T| \tag{4.57}$$

以及树 T 的误差函数为：

$$\mathcal{L}(T) = \sum_{i=1}^{|T|} err(t_i)p(t_i) \tag{4.58}$$

其中 t_i 为树 T 中的第 i 个叶节点，$err(t_i)$ 为该叶节点的分类误差率，$p(t_i)$ 为该叶节点的样本比例。所以这实质上是一个加权误差率。对一个确定的 α 值，一定会有一颗最小化 J_α 的树 T_α。为了找到这颗最优剪枝树 T_α，使用最弱连接剪枝 (`Weakest Link Pruning`) 策略，自底向上地对非叶节点进行剪枝并查看效果，然后选取一个表现最好的 T_α。

假设某一时刻以节点 t 进行剪枝，那么剪枝后与剪枝前的 Cost-Complexity 函数差为：

$$\begin{aligned}\Delta J_\alpha(t) &= J_\alpha(T - T_t) - J_\alpha(T) \\ &= \mathcal{L}(T - T_t) - \mathcal{L}(T) + \alpha(|T - T_t| - |T|) \\ &= (-\mathcal{L}(T_t) + err(t)) + \alpha(-|T_t| + 1) \\ &= err(t) - \mathcal{L}(T_t) + \alpha(1 - |T_t|)\end{aligned} \tag{4.59}$$

其中，T_t 为树 T 中以节点 t 为根节点的子树。令 $\Delta J_\alpha(t) = 0$，得到 $g(t) = \alpha\text{'} = \frac{err(t)-\mathcal{L}(T_t)}{|T_t-1|}$。

于是算法流程如下：

1. 生成一颗完整树 T^0，对所有的非叶节点都进行剪枝尝试，找到一个最小化 $g(t_1)$ 的剪枝节点 t_1，令 $\alpha^1 = g(t_1)$，$T^1 = T^0 - T_{t_1}$。
2. 对 T^1 所有的非叶节点都进行剪枝尝试，找到一个最小化 $g(t_2)$ 的剪枝节点 t_2，令 $\alpha^2 = g(t_2)$，$T^2 = T^1 - T_{t_2}$。
3. 依次进行下去，直到只剩下一个根节点为止，那么可以得到一个子树序列 $[T^0, T^1, ..., root]$ 和一系列参数 $[\alpha^1, \alpha^2, ...]$，然后在所有子树上使用交叉验证来选取一个最佳参数 $\hat{\alpha}$ 与最佳剪枝树 $T_{\hat{\alpha}}$。

自定义实现 (主要基于 CART)

[33]:
```
class DecisionNode():


    def __init__(self, feature_i=None, threshold=None,
                 value=None, true_branch=None, false_branch=None):
        self.feature_i = feature_i          # 当前结点测试的特征的索引
        self.threshold = threshold          # 当前结点测试的特征的阈值
        self.value = value                  # 结点值（如果结点为叶子结点）
        self.true_branch = true_branch      # 左子树（满足阈值， 将特征值大于等于切分点值的数据划分为左子树）
        self.false_branch = false_branch    # 右子树（未满足阈值， 将特征值小于切分点值的数据划分为右子树）


def divide_on_feature(X, feature_i, threshold):
    """
    依据切分变量和切分点，将数据集分为两个子区域
    """
    split_func = None
    if isinstance(threshold, int) or isinstance(threshold, float):
        split_func = lambda sample: sample[feature_i] >= threshold
    else:
        split_func = lambda sample: sample[feature_i] == threshold
    X_1 = np.array([sample for sample in X if split_func(sample)])
    X_2 = np.array([sample for sample in X if not split_func(sample)])
    return np.array([X_1, X_2])


class DecisionTree(object):

    def __init__(self, min_samples_split=2, min_impurity=1e-7,
                 max_depth=float("inf"), loss=None):
```

```
        self.root = None  # 根结点
        self.min_samples_split = min_samples_split  # 满足切分的最少样本数
        self.min_impurity = min_impurity  # 满足切分的最小纯度
        self.max_depth = max_depth  # 树的最大深度
        self._impurity_calculation = None  # 计算纯度的函数，如对于分类树采用信息增益
        self._leaf_value_calculation = None  # 计算 y 在叶子结点值的函数
        self.one_dim = None  # y 是否为 one-hot 编码

    def fit(self, X, y):
        self.one_dim = len(np.shape(y)) == 1
        self.root = self._build_tree(X, y)

    def _build_tree(self, X, y, current_depth=0):
        """
        递归方法建立决策树
        """
        largest_impurity = 0
        best_criteria = None    # 当前最优分类的特征索引和阈值
        best_sets = None        # 数据子集
        if len(np.shape(y)) == 1:
            y = np.expand_dims(y, axis=1)
        Xy = np.concatenate((X, y), axis=1)
        n_samples, n_features = np.shape(X)
        if n_samples >= self.min_samples_split and current_depth <= self.max_depth:
            # 对每个特征计算纯度
            for feature_i in range(n_features):
                feature_values = np.expand_dims(X[:, feature_i], axis=1)
                unique_values = np.unique(feature_values)
                # 遍历特征 i 所有的可能值找到最优纯度
                for threshold in unique_values:
                    # 基于 X 在特征 i 处是否满足阈值来划分 X 和 y， Xy1 为满足阈值的子集
                    Xy1, Xy2 = divide_on_feature(Xy, feature_i, threshold)
                    if len(Xy1) > 0 and len(Xy2) > 0:
                        # 取出 Xy 中 y 的集合
                        y1 = Xy1[:, n_features:]
                        y2 = Xy2[:, n_features:]
                        # 计算纯度
                        impurity = self._impurity_calculation(y, y1, y2)
                        # 如果纯度更高，则更新
                        if impurity > largest_impurity:
                            largest_impurity = impurity
                            best_criteria = {"feature_i": feature_i, "threshold": threshold}
                            best_sets = {
                                "leftX": Xy1[:, :n_features],   # X 的左子树
                                "lefty": Xy1[:, n_features:],   # y 的左子树
                                "rightX": Xy2[:, :n_features],  # X 的右子树
                                "righty": Xy2[:, n_features:]   # y 的右子树
                                }

        if largest_impurity > self.min_impurity:
            # 建立左子树和右子树
            true_branch = self._build_tree(best_sets["leftX"], best_sets["lefty"], current_depth + 1)
            false_branch = self._build_tree(best_sets["rightX"], best_sets["righty"], current_depth + 1)
            return DecisionNode(feature_i=best_criteria["feature_i"], threshold=best_criteria[
                                "threshold"], true_branch=true_branch, false_branch=false_branch)
        # 如果是叶结点则计算值
        leaf_value = self._leaf_value_calculation(y)
        return DecisionNode(value=leaf_value)
```

```
    def predict_value(self, x, tree=None):
        """
        预测样本，沿着树递归搜索
        """
        # 根结点
        if tree is None:
            tree = self.root
        # 递归出口
        if tree.value is not None:
            return tree.value
        # 选择当前结点的特征
        feature_value = x[tree.feature_i]
        branch = tree.false_branch
        if isinstance(feature_value, int) or isinstance(feature_value, float):
            if feature_value >= tree.threshold:
                branch = tree.true_branch
        elif feature_value == tree.threshold:
            branch = tree.true_branch
        return self.predict_value(x, branch)

    def predict(self, X):
        y_pred = [self.predict_value(sample) for sample in X]
        return y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy

    def print_tree(self, tree=None, indent=" "):
        """
        输出树
        """
        if not tree:
            tree = self.root
        if tree.value is not None:
            print(tree.value)
        else:
            print("feature|threshold -> %s | %s" % (tree.feature_i, tree.threshold))
            print("%sT->" % (indent), end="")
            self.print_tree(tree.true_branch, indent + indent)
            print("%sF->" % (indent), end="")
            self.print_tree(tree.false_branch, indent + indent)
```

[34]:
```
def calculate_entropy(y):
    log2 = lambda x: math.log(x) / math.log(2)
    unique_labels = np.unique(y)
    entropy = 0
    for label in unique_labels:
        count = len(y[y == label])
        p = count / len(y)
        entropy += -p * log2(p)
    return entropy

def calculate_gini(y):
    unique_labels = np.unique(y)
    var = 0
    for label in unique_labels:
        count = len(y[y == label])
```

```
            p = count / len(y)
            var += p ** 2
        return 1 - var

class ClassificationTree(DecisionTree):
    """
    分类树，在决策树节点选择计算信息增益/基尼指数，在叶子节点选择多数表决。
    """
    def _calculate_gini_index(self, y, y1, y2):
        """
        计算基尼指数
        """
        p = len(y1) / len(y)
        gini = calculate_gini(y)
        gini_index = gini - p * \
            calculate_gini(y1) - (1 - p) * \
            calculate_gini(y2)
        return gini_index

    def _calculate_information_gain(self, y, y1, y2):
        """
        计算信息增益
        """
        p = len(y1) / len(y)
        entropy = calculate_entropy(y)
        info_gain = entropy - p * \
            calculate_entropy(y1) - (1 - p) * \
            calculate_entropy(y2)
        return info_gain

    def _majority_vote(self, y):
        """
        多数表决
        """
        most_common = None
        max_count = 0
        for label in np.unique(y):
            count = len(y[y == label])
            if count > max_count:
                most_common = label
                max_count = count
        return most_common

    def fit(self, X, y):
        self._impurity_calculation = self._calculate_gini_index
        self._leaf_value_calculation = self._majority_vote
        super(ClassificationTree, self).fit(X, y)

def calculate_mse(y):
    return np.mean((y - np.mean(y)) ** 2)

def calculate_variance(y):
    n_samples = np.shape(y)[0]
    variance = (1 / n_samples) * np.diag((y - np.mean(y)).T.dot(y - np.mean(y)))
    return variance


class RegressionTree(DecisionTree):
    """
```

```
    回归树，在决策树节点选择计算 MSE/方差降低，在叶子节点选择均值。
    """
    def _calculate_mse(self, y, y1, y2):
        """
        计算 MSE 降低
        """
        mse_tot = calculate_mse(y)
        mse_1 = calculate_mse(y1)
        mse_2 = calculate_mse(y2)
        frac_1 = len(y1) / len(y)
        frac_2 = len(y2) / len(y)
        mse_reduction = mse_tot - (frac_1 * mse_1 + frac_2 * mse_2)
        return mse_reduction

    def _calculate_variance_reduction(self, y, y1, y2):
        """
        计算方差降低
        """
        var_tot = calculate_variance(y)
        var_1 = calculate_variance(y1)
        var_2 = calculate_variance(y2)
        frac_1 = len(y1) / len(y)
        frac_2 = len(y2) / len(y)
        variance_reduction = var_tot - (frac_1 * var_1 + frac_2 * var_2)
        return sum(variance_reduction)

    def _mean_of_y(self, y):
        """
        计算均值
        """
        value = np.mean(y, axis=0)
        return value if len(value) > 1 else value[0]

    def fit(self, X, y):
        self._impurity_calculation = self._calculate_mse
        self._leaf_value_calculation = self._mean_of_y
        super(RegressionTree, self).fit(X, y)
```

用自定义的分类树，$iris$ 数据集测试

[36]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
```

```
[5.  3.6 1.4 0.2] 0.0
```

[37]:
```
# 分类树
model = ClassificationTree()
model.fit(X_train, y_train)
# 测试数据
print(model.score(X_test, y_test))
```

```
1.0
```

[38]:
```
model.print_tree()
```

```
feature|threshold -> 2 | 3.0
 T->1.0
 F->0.0
```

用自定义的回归树，*iris* 数据集测试

[39]:
```
# 回归树
model = RegressionTree()
model.fit(X_train, y_train)
# 测试数据
print(model.score(X_test, y_test))
```

```
1.0
```

[40]:
```
model.print_tree()
```

```
feature|threshold -> 2 | 3.0
 T->1.0
 F->0.0
```

用 sklearn 实现分类树，*iris* 数据集测试

[41]:
```
from sklearn import tree

skl_model = tree.DecisionTreeClassifier()
skl_model.fit(X_train, y_train)
# 测试数据
print(skl_model.score(X_test, y_test))
```

```
1.0
```

用 sklearn 实现回归树，*iris* 数据集测试

[42]:
```
from sklearn import tree

skl_model = tree.DecisionTreeRegressor()
skl_model.fit(X_train, y_train)
# 测试数据
print(skl_model.score(X_test, y_test))
```

```
1.0
```

4.9 无监督学习方法

无监督学习的任务是找到数据更简单的表示：低维表示、稀疏表示和独立表示。低维表示将数据压缩到维度更少的空间中；稀疏表示将数据扩展到更多维度，但数据倾向于表示在坐标轴上；独立表示将数据拆分到能独立统计的维度上。

4.9.1 主成分分析法

主成分分析法，亦名 PCA。在第二章我们已经介绍过，为方便阅读，这里重述一下。PCA 将输入 $\boldsymbol{x}$ 投影表示成 $\boldsymbol{c}$。$\boldsymbol{c}$ 是比原始输入维数更低的表示，同时使得元素之间线性无关。假设有一个 $m \times n$ 的矩阵 $\boldsymbol{X}$，数据的均值为零，即 $\mathbb{E}[\boldsymbol{x}] = 0$，$\boldsymbol{X}$ 对应的无偏样本协方差矩阵：$\text{Var}\left[\boldsymbol{x}\right] = \frac{1}{m-1}\boldsymbol{X}^\top \boldsymbol{X}$。

PCA 是**通过线性变换找到一个 $\text{Var}\left[\boldsymbol{c}\right]$ 是对角矩阵**的表示 $\boldsymbol{c} = \boldsymbol{V}^\top \boldsymbol{x}$ ，**矩阵 $\boldsymbol{X}$ 的主成分可以通过奇异值分解 (SVD) 得到**，也就是说主成分是 $\boldsymbol{X}$ 的右奇异向量。假设 $\boldsymbol{V}$ 是 $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ 奇异值分解的右奇异向量，我们得到原来的特征向量方程：

$$\boldsymbol{X}^\top \boldsymbol{X} = (\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top)^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top = \boldsymbol{V}\boldsymbol{\Sigma}^\top \boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top = \boldsymbol{V}\boldsymbol{\Sigma}^2 \boldsymbol{V}^\top \tag{4.60}$$

因为根据奇异值的定义 $\boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{I}$ 。因此 $\boldsymbol{X}$ 的方差可以表示为：$\text{Var}\left[\boldsymbol{x}\right] = \frac{1}{m-1}\boldsymbol{X}^\top \boldsymbol{X} = \frac{1}{m-1}\boldsymbol{V}\boldsymbol{\Sigma}^2 \boldsymbol{V}^\top$。

所以 $\boldsymbol{c}$ 的协方差满足：$\text{Var}[\boldsymbol{c}] = \frac{1}{m-1}\boldsymbol{C}^\top\boldsymbol{C} = \frac{1}{m-1}\boldsymbol{V}^\top\boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{V} = \frac{1}{m-1}\boldsymbol{V}^\top\boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^\top\boldsymbol{V} = \frac{1}{m-1}\boldsymbol{\Sigma}^2$，因为根据奇异值定义 $\boldsymbol{V}^\top\boldsymbol{V} = \boldsymbol{I}$ 。$\boldsymbol{c}$ 的协方差是对角的，$\boldsymbol{c}$ 中的元素是彼此无关的。

自定义实现

[44]:
```
class PCA():

    def __init__(self):
        pass

    def fit(self, X, n_components):
        n_samples = np.shape(X)[0]
        covariance_matrix = (1 / (n_samples-1)) * (X - X.mean(axis=0)).T.dot(X - X.mean(axis=0))
        # 对协方差矩阵进行特征值分解
        eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
        # 对特征值（特征向量）从大到小排序
        idx = eigenvalues.argsort()[::-1]
        eigenvalues = eigenvalues[idx][:n_components]
        eigenvectors = np.atleast_1d(eigenvectors[:, idx])[:, :n_components]
        # 得到低维表示
        X_transformed = X.dot(eigenvectors)
        return X_transformed
```

用自定义的 PCA，*iris* 数据集测试

[45]:
```
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, :])
    return data[:,:-1], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
print(X_train[0], y_train[0])
```

```
[6.4 3.2 4.5 1.5] 1.0
```

[46]:
```
model = PCA()
model.fit(X_train[:20], 2)  # 取前 20 个数据测试
```

[46]:
```
array([[-5.75535057,  6.35428159],
       [-2.20479768,  5.95454742],
       [-2.26520769,  5.87699147],
       [-5.75594076,  6.00577621],
       [-2.30122891,  6.298297  ],
       [-2.5234036 ,  6.07896967],
       [-6.28590718,  5.67677524],
       [-4.04494949,  5.07447986],
       [-5.72955335,  6.51557733],
       [-5.98904726,  5.94901496],
       [-2.0822997 ,  5.87694555],
       [-1.95898348,  5.11327258],
       [-5.71754628,  6.37514216],
       [-5.10531342,  5.27893081],
       [-5.27416173,  5.68555962],
       [-2.18424456,  5.46422474],
       [-2.18386992,  6.24395656],
       [-2.51767442,  6.40421362],
       [-5.27762276,  5.47610728],
       [-2.14860047,  6.44145051]])
```

4.9.2 k-均值聚类

k-均值聚类 (k-means) 算法的基本思想是**初始随机给定 k 个簇中心，按照最邻近原则把待分类样本点分到各个簇。然后按平均法重新计算各个簇的质心，从而确定新的簇心。一直迭代，直到簇心的移动距离小于某个给定的值**。k 是我们事先需要给定的聚类数目。

算法步骤：

- 初始化 k 个不同的中心点 $\{\mu_1, \cdots, \mu_k\}$, 然后迭代交换以下两个步骤直至收敛。
- 第一步：每个训练样本分配到最近的中心点 μ_i 所代表的聚类 i。
- 第二步：每一个中心点 μ_i 更新为聚类 i 中所有训练样本 $\boldsymbol{x}_j$ 的均值。

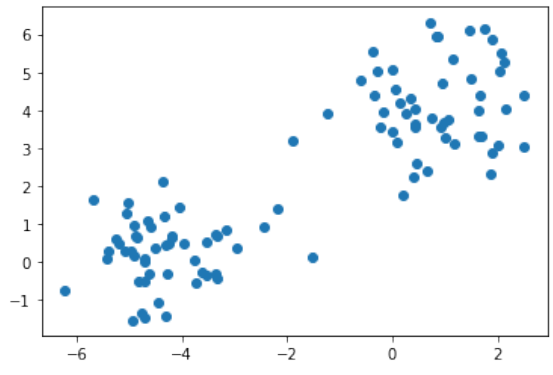
自定义实现

[47]:
```
def distEclud(x,y):
    """
    计算欧氏距离
    """
    return np.sqrt(np.sum((x-y)**2))

def randomCent(dataSet,k):
    """
    为数据集构建一个包含 K 个随机质心的集合
    """
    m,n = dataSet.shape
    centroids = np.zeros((k,n))
    for i in range(k):
        index = int(np.random.uniform(0,m))
        centroids[i,:] = dataSet[index,:]
    return centroids

class KMeans():

    def __init__(self):
        self.dataSet = None
        self.k = None

    def fit(self, dataSet, k):
        self.dataSet = dataSet
        self.k = k
        m = np.shape(dataSet)[0]
        # 第一列存样本属于哪一簇
        # 第二列存样本的到簇的中心点的误差
        clusterAssment = np.mat(np.zeros((m,2)))
        clusterChange = True
        centroids = randomCent(self.dataSet,k)
        while clusterChange:
            clusterChange = False
            for i in range(m):
                minDist = 1e6
                minIndex = -1
                # 遍历所有的质心，找出最近的质心
                for j in range(k):
                    distance = distEclud(centroids[j,:], self.dataSet[i,:])
                    if distance < minDist:
                        minDist = distance
                        minIndex = j
                # 更新每一行样本所属的簇
                if clusterAssment[i,0] != minIndex:
                    clusterChange = True
                    clusterAssment[i,:] = minIndex, minDist**2
            # 更新质心
```

```
            for j in range(k):
                pointsInCluster = dataSet[np.nonzero(clusterAssment[:,0].A == j)[0]]  # 获取簇类所有的点
                centroids[j,:] = np.mean(pointsInCluster,axis=0)    # 对矩阵的行求均值

        return centroids,clusterAssment
```

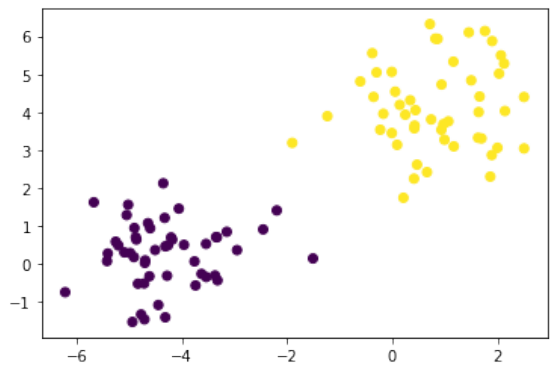
用自定义的 k-均值聚类，合成数据集测试

[48]:
```
X, y = datasets.make_blobs(n_samples=100, centers=2, random_state=3)
plt.scatter(X[:,0], X[:,1]);
```



[49]:
```
model = KMeans()
center, clusterAssment = model.fit(X, k=2)
y_pred = np.squeeze(np.array(clusterAssment[:,0]))
```

[50]:
```
plt.scatter(X[:,0], X[:,1], c=y_pred);
```



[51]:
```
import matplotlib
import numpy
import sklearn
import pandas
import cvxopt

print("numpy:", numpy.__version__)
print("matplotlib:", matplotlib.__version__)
print("sklearn:", sklearn.__version__)
print("pandas:", pandas.__version__)
print("cvxopt:", cvxopt.__version__)
```

```
numpy: 1.14.5
matplotlib: 3.1.1
sklearn: 0.21.3
pandas: 0.25.1
cvxopt: 1.2.4
```

第五章　深度前馈网络

5.1　深度前馈网络

深度前馈网络 (Deep Feedforward Network，DFN) 也叫前馈神经网络 (Feedforward Neural Network，FNN) 或多层感知机 (Multilayer Perceptron，MLP)，是最典型的深度学习模型。目标是拟合一个函数，如有一个分类器 $y = f^*(x)$ 将输入 x 映射到输出类别 y。深度前馈网将这个映射定义为 $f(x, \theta)$ 并学习这个参数 θ 的值来得到最好的函数拟合。

深度前馈网络中信息从 x 流入，通过中间 f 的计算，最后到达输出 y。如图 5.1 所示，假设有 $f^{(1)}, f^{(2)}, f^{(3)}$ 这三个函数链式连接，这个链式连接可以表示为 $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$，这种链式结构是神经网络最为常用的结构。$f^{(1)}, f^{(2)}$ 被称为神经网络的第一层，第二层，也为网络的隐藏层 (Hidden Layer)，深度前馈网络最后一层 $f^{(3)}$ 就是输出层 (Output Layer)。这个链的长度就是神经网络的深度，输入向量的每个元素均视作一个神经元。

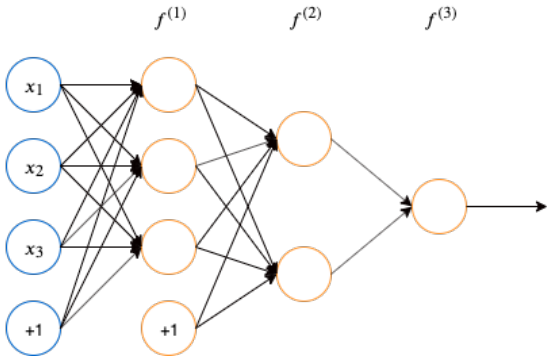


图 5.1.　深度前馈网络示意图

5.2　DFN 相关设计

DFN 内部的神经网络层可以分为三类，输入层，隐藏层和输出层。为了构造一个可训练的 DFN，我们需要考虑几个方面，包括隐藏单元，输出单元，和代价函数 (损失函数)。

5.2.1　隐藏单元

层与层之间是全连接的，也就是说，第 i 层的任意一个神经元一定与第 $i+1$ 层的任意一个神经元相连。如下图所示，大多数隐藏单元 (Hidden Unit) 都可以描述为接受输入向量 $\boldsymbol{x}$，计算仿射变换 $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$，然后使用一个逐元素的非线性函数 $g(\boldsymbol{z})$ 得到隐藏单元的输出 $\boldsymbol{a}$。而大多数隐藏单元的区别仅仅在于激活函数 $g(\boldsymbol{z})$ 的形式。(**这里增加激活函数，是为了提高模型的表达能力；如果不引入激活函数，可以验证，无论多少层神经网络，输出都是输入的线性组合**)

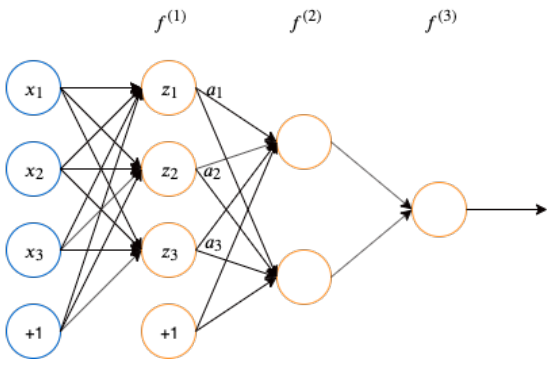


图 5.2.　深度前馈网络及第一层隐藏单元示意图

如图 5.2 所示，假设激活函数 $g(\boldsymbol{z})$ 为 σ，于是 $f^{(1)}$ 层的隐藏单元可以描述为：

$$\begin{cases} a_1 = \sigma(z_1) = \sigma(w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1) \\ a_2 = \sigma(z_2) = \sigma(w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2) \\ a_3 = \sigma(z_3) = \sigma(w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + b_3) \end{cases} \tag{5.1}$$

选择隐藏单元实际上就是要选择一个合适的激活函数。常见的有如下几种激活函数：

- 整流线性单元 (ReLU)：$g(z) = \max\{0, z\}$。优点是易于优化，二阶导数几乎处处为 0，**处于激活状态时一阶导数处处为 1**，也就是其相比于引入二阶效应的激活函数，梯度方向对学习更有用。如果使用 ReLU，第一步做线性变换 $\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$ 时的 $\boldsymbol{b}$ 一般设置成小的正值。缺陷是不能通过基于梯度的方法学习那些使单元激活为 0 的样本。

  ReLU 函数的梯度为 $g'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$

- sigmoid 函数 (常写作 σ) 或双曲正切函数 tanh。两者之间有一个联系：$\tanh(z) = 2\sigma(2z) - 1$。两者都比较容易饱和，仅当 z 接近 0 时才对输入强烈敏感，因此使得基于梯度的学习变得非常困难，不适合做前馈网络中的隐藏单元。**如果必须要使用这两种中的一个，那么 tanh 通常表现更好**，因为在 0 附近其类似于单位函数。即，如果网络的激活能一直很小，训练 $\hat{y} = w^\top \tanh(\mathrm{U}^\top \tanh(\mathrm{V}^\top x))$ 类似于训练一个线性模型 $\hat{y} = w^\top \mathrm{U}^\top \mathrm{V}^\top x$。RNN 和一些自编码器有一些额外的要求，因此不能使用分段激活函数，此时这种类 sigmoid 单元更合适。

  sigmoid 函数写作 $g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$，梯度为 $g'(z) = \sigma(z)(1 - \sigma(z))$

  双曲正切函数写作 $g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$，梯度为 $g'(z) = 1 - \tanh^2(z)$

[1]:
```
from abc import ABC, abstractmethod
import numpy as np
import time
import re
from collections import OrderedDict
```

[2]:
```
class ActivationBase(ABC):

    def __init__(self, **kwargs):
        super().__init__()

    def __call__(self, z):
        if z.ndim == 1:
            z = z.reshape(1, -1)
        return self.forward(z)

    @abstractmethod
    def forward(self, z):
        """
        函数作用：前向传播，通过激活函数获得 a
        """
        raise NotImplementedError

    @abstractmethod
    def grad(self, x, **kwargs):
        """
        函数作用：反向传播，获得梯度
        """
        raise NotImplementedError

class ReLU(ActivationBase):
    """
    整流线性单元。
    """
    def __init__(self):
        super().__init__()

    def __str__(self):
        return "ReLU"

    def forward(self, z):
        return np.clip(z, 0, np.inf)

    def grad(self, x):
        return (x > 0).astype(int)
```

```python
class Sigmoid(ActivationBase):
    """
    sigmoid 激活函数。更多激活函数见 ../method 文件夹。
    """
    def __init__(self):
        super().__init__()

    def __str__(self):
        return "Sigmoid"

    def forward(self, z):
        return 1 / (1 + np.exp(-z))

    def grad(self, x):
        return self.forward(x) * (1 - self.forward(x))

class Tanh(ActivationBase):
    """
    双曲正切函数。
    """
    def __init__(self):
        super().__init__()

    def __str__(self):
        return "Tanh"

    def forward(self, z):
        return np.tanh(z)

    def grad(self, x):
        return 1 - np.tanh(x) ** 2

class Affine(ActivationBase):
    """
    affine 激活函数， 即仿射变换。 输出 slope*z + intercept。当 slope=1 且 intercept=0 表示不做变换。
    """
    def __init__(self, slope=1, intercept=0):
        self.slope = slope
        self.intercept = intercept
        super().__init__()

    def __str__(self):
        return "Affine(slope={}, intercept={})".format(self.slope, self.intercept)

    def forward(self, z):
        return self.slope * z + self.intercept

    def grad(self, x):
        return self.slope * np.ones_like(x)

class ActivationInitializer(object):

    def __init__(self, acti_name='sigmoid'):
        self.acti_name = acti_name

    def __call__(self):
        if self.acti_name.lower() == 'sigmoid':
            acti_fn = Sigmoid()
        elif self.acti_name.lower() == 'relu':
```

```
            acti_fn = ReLU()
        elif "affine" in self.acti_name.lower():
            r = r"affine\(slope=(.*), intercept=(.*)\)"
            slope, intercept = re.match(r, self.acti_name.lower()).groups()
            acti_fn = Affine(float(slope), float(intercept))
        return acti_fn
```

5.2.2 输出单元

假设前面已经使用若干隐藏层提供了一组隐藏特征 $\boldsymbol{h}$，输出层是要对这些特征做一些额外变换来完成任务。

常见的有如下几种输出单元：

- 用于高斯输出分布的线性单元，即对隐藏特征不做非线性变换，直接产生 $\hat{\boldsymbol{y}} = \boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$。其用来产生条件高斯分布的均值：$p(\boldsymbol{y} \mid \boldsymbol{x}) = N(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$
- 用于伯努利输出分布的 sigmoid 单元，即对隐藏特征先用线性层求 $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$，然后对这个值做一个 sigmoid 变换 $\sigma(z)$ 将其映射到 $[0, 1]$ 区间，转化成一个**概率值**，即 $\hat{\boldsymbol{y}} = \sigma(\boldsymbol{z}) = \sigma(\boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b})$
- 用于多元伯努利输出分布的 softmax 单元，可以看作是伯努利输出分布的 sigmoid 单元在多分类问题上的推广。此时**输出标签空间是一个离散的，多类别的集合**。假如一共有 K 个类别，则标签空间 $\mathcal{Y} = \{0, 1, 2, \ldots, \mathrm{K}-1\}$。对隐藏特征做线性变换 $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$ 后，通过 softmax 函数得到一个向量 $\mathrm{softmax}(\boldsymbol{z})$，这个向量的每个维度可以看做是输入样本属于对应类别标签的概率，因此有 $\forall i \in \{0, 1, \ldots, \mathrm{K}-1\}, z_i \in [0, 1] \wedge \sum_i z_i = 1$。softmax 函数的具体形式为：

$$\mathrm{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{5.2}$$

[3]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def softmax(x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return e_x / e_x.sum(axis=-1, keepdims=True)
```

5.2.3 代价函数

任何能够衡量模型预测值与真实值之间的差异的函数都可以叫做代价函数。如果有多个样本，则可以将所有代价函数的取值求均值，记作 $J(\theta)$。当我们确定了模型后，再要做的是训练模型的参数 θ (如 $\boldsymbol{W}$，$\boldsymbol{b}$)。**训练参数的过程就是误差反向传递，不断调整 θ，从而得到更小的 $J(\theta)$ 的过程**。理想情况下，当我们取到代价函数 J 的最小值时，就得到了最优的参数 θ ，记为：$\min_\theta J(\theta)$ 。常见的代价函数主要有：

- 二次代价函数，具体形式为：

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)})^2 \tag{5.3}$$

梯度的计算：以单个样本为例，假设输出单元 $\hat{\boldsymbol{y}} = g(\boldsymbol{z})$，$g$ 为输出单元的激活函数，$\boldsymbol{z}$ 为 θ 的函数，则此时代价函数为 $\frac{1}{2}(g(\boldsymbol{z}) - \boldsymbol{y})^2$ 。可以计算梯度：

$$\frac{\partial J(\theta)}{\partial \boldsymbol{z}} = (g(\boldsymbol{z}) - \boldsymbol{y}) g'(\boldsymbol{z}) \tag{5.4}$$

可以验证，当输出单元激活函数采用 sigmoid 或 tanh 的 S 型激活函数，二次代价函数在梯度收敛至 0 时 (如果真实值为 0)，存在收敛速度慢而导致的训练速度慢的问题。

- 最大 (对数) 似然代价函数或者最小负 (对数) 似然代价函数，具体形式为：

$$\underset{\theta}{\mathrm{argmin}} \; -\mathcal{L}(\theta \mid \boldsymbol{y}, \hat{\boldsymbol{y}}) \tag{5.5}$$

似然 $\mathcal{L}(\theta \mid \boldsymbol{y}, \hat{\boldsymbol{y}})$ 可以表示成联合概率：

$$P(t, \boldsymbol{z} \mid \theta) = P(t \mid \boldsymbol{z}, \theta) P(\boldsymbol{z} \mid \theta) \tag{5.6}$$

  - 如果输出单元是用于伯努利输出分布的 sigmoid 单元，可以得到对应的代价函数：

$$J(\theta) = \underset{\theta}{\mathrm{argmin}} \; -\mathcal{L}(\theta \mid \boldsymbol{y}, \hat{\boldsymbol{y}}) = -\frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{y}^{(i)} \log \hat{\boldsymbol{y}}^{(i)} + (1 - \boldsymbol{y}^{(i)}) \log(1 - \hat{\boldsymbol{y}}^{(i)}) \right) \tag{5.7}$$

梯度的计算：以单个样本为例，假设输出单元 $\hat{\boldsymbol{y}} = g(\boldsymbol{z})$ ，g 为输出单元的激活函数，$\boldsymbol{z}$ 为 θ 的函数。可以计算梯度：

$$\frac{\partial J(\theta)}{\partial \boldsymbol{z}} = -\left(\frac{\boldsymbol{y}}{g(\boldsymbol{z})} - \frac{(1 - \boldsymbol{y})}{1 - g(\boldsymbol{z})} \right) g'(\boldsymbol{z}) \tag{5.8}$$

如果 g 为 sigmoid 函数，可以进一步化简：

$$J(\theta) = -\boldsymbol{y}\boldsymbol{z} + \log(1 + e^{\boldsymbol{z}}) \tag{5.9}$$

$$\frac{\partial \mathrm{J}(\theta)}{\partial \boldsymbol{z}} = \sigma(\boldsymbol{z}) - \boldsymbol{y} \tag{5.10}$$

$\sigma(\boldsymbol{z}) - \boldsymbol{y}$ 表示真实值与输出值之间的误差。当误差越大时，梯度就越大，θ 的调整就越快，训练速度就越快。**当输出神经元的激活函数是线性时 (如 ReLU 函数)，二次代价函数是一种合适的选择；当输出神经元的激活函数是 S 型函数时 (如 sigmoid 、tanh 函数)，选择交叉熵代价函数则比较合理**。

– 如果输出单元是用于多元伯努利输出分布的 softmax 单元，可以得到对应的代价函数：

$$J(\theta) = \underset{\theta}{\operatorname{argmin}} \ -\mathcal{L}(\theta \mid \boldsymbol{y}, \hat{\boldsymbol{y}}) = -\frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{y}^{(i)} \log \hat{\boldsymbol{y}}^{(i)}) \tag{5.11}$$

其中对每一个 $\hat{\boldsymbol{y}}$ 有 $\hat{y}_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$。因此可以进一步展开为：

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=0}^{\mathrm{K}-1}(y_k^{(i)} \log \hat{y}_k^{(i)}) \tag{5.12}$$

梯度的计算：以单个样本为例，假设输出单元 $\hat{\boldsymbol{y}} = g(\boldsymbol{z})$ ，g 为输出单元的激活函数，则此时代价函数为 $J(\theta) = -\sum_{k=0}^{\mathrm{K}-1}(y_k \log \hat{y}_k)$。
首先，对 softmax 函数求导：

1. 当 $j = i$ 时：

$$\begin{aligned}
\frac{\partial \hat{y}_j}{\partial z_i} &= \frac{\partial}{\partial z_i}\left(\frac{e^{z_j}}{\sum_k e^{z_k}}\right) \\
&= \frac{(e^{z_j})' \cdot \sum_k e^{z_k} - e^{z_j} \cdot e^{z_j}}{(\sum_k e^{z_k})^2} \\
&= \frac{e^{z_j}}{\sum_k e^{z_k}} - \frac{e^{z_j}}{\sum_k e^{z_k}} \cdot \frac{e^{z_j}}{\sum_k e^{z_k}} \\
&= \hat{y}_j(1 - \hat{y}_j) \\
&= \hat{y}_j - \hat{y}_j\hat{y}_j
\end{aligned} \tag{5.13}$$

2. 当 $j \neq i$ 时：

$$\begin{aligned}
\frac{\partial \hat{y}_j}{\partial z_i} &= \frac{\partial}{\partial z_i}\left(\frac{e^{z_j}}{\sum_k e^{z_k}}\right) \\
&= \frac{0 \cdot \sum_k e^{z_k} - e^{z_j} \cdot e^{z_i}}{(\sum_k e^{z_k})^2} \\
&= -\frac{e^{z_j}}{\sum_k e^{z_k}} \cdot \frac{e^{z_i}}{\sum_k e^{z_k}} \\
&= -\hat{y}_j\hat{y}_i \\
&= 0 - \hat{y}_j\hat{y}_i
\end{aligned} \tag{5.14}$$

然后，进一步计算：

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial z_j} &= -\sum_k y_k \frac{\partial}{\partial z_j}(\log \hat{y}_k) \\
&= -\sum_k y_k \frac{1}{\hat{y}_k}\frac{\partial \hat{y}_k}{\partial z_j}(\text{拆分求导}) \\
&= -y_j \frac{1}{\hat{y}_j} \cdot \hat{y}_j(1 - \hat{y}_j) - \sum_{k \neq j} y_k \frac{1}{\hat{y}_k} \cdot (-\hat{y}_j\hat{y}_k) \\
&= -y_j(1 - \hat{y}_j) + \hat{y}_j \sum_{k \neq j} y_k(\text{合并}) \\
&= \hat{y}_j - y_j
\end{aligned} \tag{5.15}$$

5.2.4 架构设计

架构 (`Architecture`) 一词指网络的整体结构：它应该具有多少单元，以及这些单元应该如何连接。
在实践中，神经网络具有多样性。
用于计算机视觉的卷积神经网络的特殊架构在第九章中介绍。前馈网络也可以推广到序列处理的循环神经网络，但也有它们自己的架构考虑，将在第十章中介绍。

5.3 反向传播算法

5.3.1 单个神经元的训练

单个神经元的结构如图 5.3 所示，假设训练样本 $(\boldsymbol{x}, y)$，在图 5.3 左，$\boldsymbol{x}$ 是输入向量，通过一个激励函数 $h_{\boldsymbol{w},b(\boldsymbol{x})}$ 得到一个输出 a，a 再通过代价函数得到 J。激励函数以使用 sigmoid 为例：

$$\begin{aligned}
f(\boldsymbol{W}, b, \boldsymbol{x}) = a &= \text{sigmoid}(\sum_i x_i w_i + b) \\
J(\boldsymbol{W}, b, \boldsymbol{x}, y) &= \frac{1}{2}\|y - a\|^2
\end{aligned} \tag{5.16}$$

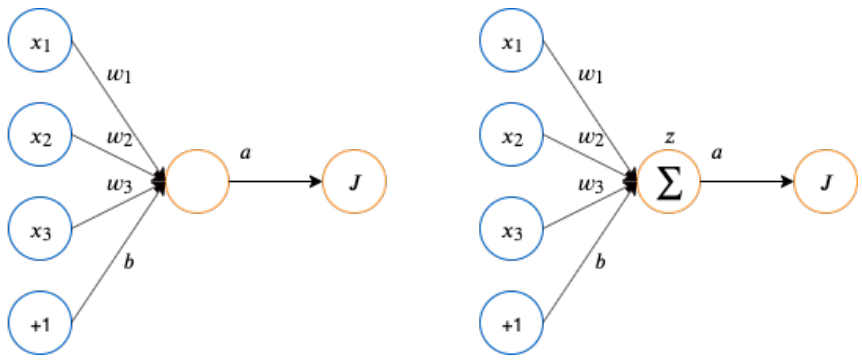


图 5.3. 单个神经元用于训练的示意图

将激励函数拆成两部分，第一部分通过仿射求和得到 $z=\sum_i x_i w_i + b = \sum_i w_i x_i + b$，第二部分通过 sigmoid 得到 a。

1. 训练过程中，要求代价函数 J 关于 $\boldsymbol{w}$ 和 b 的偏导数。先求 J 关于中间变量 a 和 z 的偏导：

$$\begin{aligned}\delta^{(a)} &= \frac{\partial}{\partial a} J(\boldsymbol{w}, b, \boldsymbol{x}, y) = -(y-a) \\ \delta^{(z)} &= \frac{\partial}{\partial z} J(\boldsymbol{w}, b, \boldsymbol{x}, y) = \frac{\partial J}{\partial a}\frac{\partial a}{\partial z} = \delta^{(a)} a(1-a)\end{aligned} \tag{5.17}$$

其中 sigmoid 的导数为 $a(1-a)$；

2. 再根据链导法则，可以求得 J 关于 $\boldsymbol{w}$ 和 b 的偏导数，即得 $\boldsymbol{w}$ 和 b 的梯度。

$$\begin{aligned}\nabla_{\boldsymbol{w}} J(\boldsymbol{w}, b, \boldsymbol{x}, y) &= \frac{\partial}{\partial w} J = \frac{\partial J}{\partial z}\frac{\partial z}{\partial \boldsymbol{w}} = \boldsymbol{x}\delta^{(z)} \\ \nabla_b J(\boldsymbol{w}, b, \boldsymbol{x}, y) &= \frac{\partial}{\partial b} J = \frac{\partial J}{\partial z}\frac{\partial z}{\partial b} = \delta^{(z)}\end{aligned} \tag{5.18}$$

在这个过程中，先求 $\frac{\partial J}{\partial a}$，进一步求 $\frac{\partial J}{\partial z}$，最后求得 $\frac{\partial J}{\partial \boldsymbol{w}}$ 和 $\frac{\partial J}{\partial b}$。结合上图及链导法则，可以看出这是一个将代价函数的增量 ∂J 自后向前传播的过程，因此称为反向传播 (`Back Propagation`)。

5.3.2 多层神经网络的训练

在描述多层神经网络时，我们不再仔细描述某一层内部节点，而是描述层与层之间的关系。如图 5.4，为方便标记，在描述多层网络时，我们将第 i 层记作下标 $_i$。在本书中，第 i 层用 $\boldsymbol{a}_l$ 和 a^l 两种方式表示，视情况而用。假设第 $l+1$ 层的输入和输出分别是 $\boldsymbol{a}_l$ 和 $\boldsymbol{a}_{l+1}$，参数为 $\boldsymbol{W}_l$ 和 b_l，其中 $\boldsymbol{W}_l$ 的维数为 (n_{in}, n_{out})。仿射结果为中间变量 $\boldsymbol{z}_{l+1}$。其中第一层的输出 $\boldsymbol{a}_1 = \boldsymbol{x}$，为整个网络的输入，最后一层的输出 $\boldsymbol{a}_L$ 是代价函数的输入。

$$\begin{aligned}\boldsymbol{z}_{l+1} &= \boldsymbol{W}_l^\top \boldsymbol{a}_l + \boldsymbol{b}_l \\ \boldsymbol{a}_{l+1} &= \text{sigmoid}(\boldsymbol{z}_{l+1})\end{aligned} \tag{5.19}$$

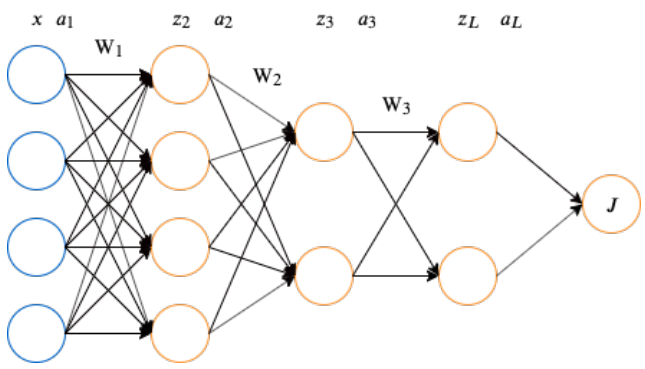


图 5.4. 多层神经网络训练的示意图

在实际神经网络中，网络层的输入 $\boldsymbol{a}_l$ 的维数通常表示为 (n_{samp}, n_{in})，其中第一维 n_{samp} 为输入样本的数量。于是，计算公式可以写作：

$$\boldsymbol{z}_{l+1} = \boldsymbol{a}_l \boldsymbol{W}_l + \boldsymbol{b}_l \tag{5.20}$$

对多层网络的训练需要计算代价函数 J 对每一层的参数求偏导。后向传播过程变为：

1. 求 J 对 $\boldsymbol{a}_L$ 的偏导 $\delta_L^{(a)}$；

2. 在第 $l+1$ 层，将误差信号从 $\boldsymbol{a}_{l+1}$ 传递到 $\boldsymbol{z}_{l+1}$；

$$\frac{\partial \boldsymbol{a}_{l+1}}{\partial \boldsymbol{z}_{l+1}} = \boldsymbol{a}_{l+1}(1-\boldsymbol{a}_{l+1}) \tag{5.21}$$

3. 将误差信号从第 $l+1$ 层向第 l 层传播；

4. 对第 l 层计算得到 J 对 $\boldsymbol{a}_l$ 和 $\boldsymbol{z}_l$ 的偏导数；

$$\begin{aligned}\delta_l^{(a)} = \frac{\partial J}{\partial \boldsymbol{a}_l} &= \begin{cases} -(\boldsymbol{y}-\boldsymbol{a}_l), & \text{if } l = L \\ \frac{\partial J}{\partial \boldsymbol{z}_{l+1}}\frac{\partial \boldsymbol{z}_{l+1}}{\partial \boldsymbol{a}_l} = \delta_{l+1}^{(z)}(\boldsymbol{W}_l)^\top, & \text{otherwise} \end{cases} \\ \delta_l^{(z)} = \frac{\partial J}{\partial \boldsymbol{z}_l} &= \frac{\partial J}{\partial \boldsymbol{a}_l}\frac{\partial \boldsymbol{a}_l}{\partial \boldsymbol{z}_l} = \delta_l^{(a)} \boldsymbol{a}_{l+1}(1-\boldsymbol{a}_{l+1})\end{aligned} \tag{5.22}$$

5. 对第 l 层计算得到 J 对参数 $\boldsymbol{W}_l$ 和 $\boldsymbol{b}_l$ 的梯度。

$$\begin{aligned}\nabla_{\boldsymbol{W}_l} J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y}) &= \frac{\partial}{\partial \boldsymbol{W}_l} J = \frac{\partial J}{\partial \boldsymbol{z}_{l+1}}\frac{\partial \boldsymbol{z}_{l+1}}{\partial \boldsymbol{W}_l} = (\boldsymbol{a}_l)^\top \delta_{l+1}^{(z)} \\ \nabla_{\boldsymbol{b}_l} J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y}) &= \frac{\partial}{\partial \boldsymbol{b}_l} J = \frac{\partial J}{\partial \boldsymbol{z}_{l+1}}\frac{\partial \boldsymbol{z}_{l+1}}{\partial \boldsymbol{b}_l} = \delta_{l+1}^{(z)}\end{aligned} \tag{5.23}$$

为具体实现深度前馈网络，需要首先定义以下模块：

定义权重初始化方法

[4]:
```
class std_normal:
    """
    标准正态初始化
    """
    def __init__(self, gain=0.01):
        self.gain = gain

    def __call__(self, weight_shape):
        return self.gain * np.random.randn(*weight_shape)

class he_uniform:
    """
    He 初始化， 通过 Uniform(-b, b) 初始化权重矩阵 W， 这里的 b=sqrt(6 / n_in)
    （更多实现会在第 8 章展开）
    """
    def __init__(self):
        pass

    def __call__(self, weight_shape):
        n_in, n_out = weight_shape
        b = np.sqrt(6 / n_in)
        return np.random.uniform(-b, b, size=weight_shape)

class WeightInitializer(object):

    def __init__(self, mode="he_uniform"):
        self.mode = mode  # 更多初始化方法在第 8 章展开
        r = r"([a-zA-Z]*)=([^,)]*)"
        mode_str = self.mode.lower()
        kwargs = dict([(i, eval(j)) for (i, j) in re.findall(r, mode_str)])
        if "std_normal" in mode_str:
            self.init_fn = std_normal(**kwargs)
        elif "he_uniform" in mode_str:
            self.init_fn = he_uniform(**kwargs)

    def __call__(self, weight_shape):
        W = self.init_fn(weight_shape)
        return W
```

定义激活函数 (见前文描述)

定义优化方法

[5]:
```
"""
这里采用 sgd 优化方法，更多实现细节和优化方法见第 8 章
"""
class OptimizerBase(ABC):

    def __init__(self):
        pass

    def __call__(self, params, params_grad, params_name):
        """
        参数说明:
        params: 待更新参数， 如权重矩阵 W;
        params_grad: 待更新参数的梯度;
        params_name: 待更新参数名;
```

```
        """
        return self.update(params, params_grad, params_name)

    @abstractmethod
    def update(self, params, params_grad, params_name):
        raise NotImplementedError

class SGD(OptimizerBase):
    """
    sgd 优化方法
    """
    def __init__(self, lr=0.01):
        super().__init__()
        self.lr = lr

    def __str__(self):
        return "SGD(lr={})".format(self.hyperparams["lr"])

    def update(self, params, params_grad, params_name):
        update_value = self.lr * params_grad
        return params - update_value

    @property
    def hyperparams(self):
        return {
            "op": "SGD",
            "lr": self.lr
        }

class OptimizerInitializer(ABC):

    def __init__(self, opti_name="sgd"):
        self.opti_name = opti_name

    def __call__(self):
        r = r"([a-zA-Z]*)=([^,)]*)"
        opti_str = self.opti_name.lower()
        kwargs = dict([(i, eval(j)) for (i, j) in re.findall(r, opti_str)])
        if "sgd" in opti_str:
            optimizer = SGD(**kwargs)
        return optimizer
```

定义网络层的框架

[6]:
```
class LayerBase(ABC):

    def __init__(self, optimizer=None):
        self.X = []   # 网络层输入
        self.gradients = {}  # 网络层待梯度更新变量
        self.params = {}  # 网络层参数变量
        self.acti_fn = None   # 网络层激活函数
        self.optimizer = OptimizerInitializer(optimizer)()  # 网络层优化方法

    @abstractmethod
    def _init_params(self, **kwargs):
        """
        函数作用：初始化参数
        """
        raise NotImplementedError
```

```
    @abstractmethod
    def forward(self, X, **kwargs):
        """
        函数作用：前向传播
        """
        raise NotImplementedError

    @abstractmethod
    def backward(self, out, **kwargs):
        """
        函数作用：反向传播
        """
        raise NotImplementedError

    def flush_gradients(self):
        """
        函数作用：重置更新参数列表
        """
        self.X = []
        for k, v in self.gradients.items():
            self.gradients[k] = np.zeros_like(v)

    def update(self):
        """
        函数作用：更新参数
        """
        for k, v in self.gradients.items():
            if k in self.params:
                self.params[k] = self.optimizer(self.params[k], v, k)
```

[7]:
```
class FullyConnected(LayerBase):
    """
    定义全连接层，实现 a=g(x*W+b)，前向传播输入 x，返回 a；反向传播输入
    """
    def __init__(self, n_out, acti_fn, init_w, optimizer=None):
        """
        参数说明：
        acti_fn：激活函数， str 型
        init_w：权重初始化方法， str 型
        n_out：隐藏层输出维数
        optimizer：优化方法
        """
        super().__init__(optimizer)
        self.n_in = None  # 隐藏层输入维数， int 型
        self.n_out = n_out  # 隐藏层输出维数， int 型
        self.acti_fn = ActivationInitializer(acti_fn)()
        self.init_w = init_w
        self.init_weights = WeightInitializer(mode=init_w)
        self.is_initialized = False  # 是否初始化， bool 型变量

    def _init_params(self):
        b = np.zeros((1, self.n_out))
        W = self.init_weights((self.n_in, self.n_out))
        self.params = {"W": W, "b": b}
        self.gradients = {"W": np.zeros_like(W), "b": np.zeros_like(b)}
        self.is_initialized = True

    def forward(self, X, retain_derived=True):
```

```
        """
        全连接网络的前向传播，原理见上文 反向传播算法 部分。

        参数说明：
        X：输入数组，为 (n_samples, n_in)，float 型
        retain_derived：是否保留中间变量，以便反向传播时再次使用，bool 型
        """
        if not self.is_initialized:  # 如果参数未初始化，先初始化参数
            self.n_in = X.shape[1]
            self._init_params()
        W = self.params["W"]
        b = self.params["b"]
        z = X @ W + b
        a = self.acti_fn.forward(z)
        if retain_derived:
            self.X.append(X)
        return a

    def backward(self, dLda, retain_grads=True):
        """
        全连接网络的反向传播，原理见上文 反向传播算法 部分。

        参数说明：
        dLda：关于损失的梯度，为 (n_samples, n_out)，float 型
        retain_grads：是否计算中间变量的参数梯度，bool 型
        """
        if not isinstance(dLda, list):
            dLda = [dLda]
        dX = []
        X = self.X
        for da, x in zip(dLda, X):
            dx, dw, db = self._bwd(da, x)
            dX.append(dx)
            if retain_grads:
                self.gradients["W"] += dw
                self.gradients["b"] += db
        return dX[0] if len(X) == 1 else dX

    def _bwd(self, dLda, X):
        W = self.params["W"]
        b = self.params["b"]
        Z = X @ W + b
        dZ = dLda * self.acti_fn.grad(Z)
        dX = dZ @ W.T
        dW = X.T @ dZ
        db = dZ.sum(axis=0, keepdims=True)
        return dX, dW, db

    @property
    def hyperparams(self):
        return {
            "layer": "FullyConnected",
            "init_w": self.init_w,
            "n_in": self.n_in,
            "n_out": self.n_out,
            "acti_fn": str(self.acti_fn),
            "optimizer": {
                "hyperparams": self.optimizer.hyperparams,
            },
```

```
            "components": {
                k: v for k, v in self.params.items()
            }
        }
```

[8]:
```
class Softmax(LayerBase):
    """
    定义 Softmax 层
    """
    def __init__(self, dim=-1, optimizer=None):
        super().__init__(optimizer)
        self.dim = dim
        self.n_in = None
        self.is_initialized = False

    def _init_params(self):
        self.params = {}
        self.gradients = {}
        self.is_initialized = True

def forward(self, X, retain_derived=True):
    """
    Softmax 的前向传播， 原理见上文 代价函数 部分。
    """
    if not self.is_initialized:
        self.n_in = X.shape[1]
        self._init_params()
    Y = self._fwd(X)
    if retain_derived:
        self.X.append(X)
    return Y

def _fwd(self, X):
    e_X = np.exp(X - np.max(X, axis=self.dim, keepdims=True))
    return e_X / e_X.sum(axis=self.dim, keepdims=True)

def backward(self, dLdy):
    """
    Softmax 的反向传播，原理见上文 代价函数 部分。
    """
    if not isinstance(dLdy, list):
        dLdy = [dLdy]
    dX = []
    X = self.X
    for dy, x in zip(dLdy, X):
        dx = self._bwd(dy, x)
        dX.append(dx)
    return dX[0] if len(X) == 1 else dX

def _bwd(self, dLdy, X):
    dX = []
    for dy, x in zip(dLdy, X):
        dxi = []
        for dyi, xi in zip(*np.atleast_2d(dy, x)):
            yi = self._fwd(xi.reshape(1, -1)).reshape(-1, 1)
            dyidxi = np.diagflat(yi) - yi @ yi.T
            dxi.append(dyi @ dyidxi)
        dX.append(dxi)
    return np.array(dX).reshape(*X.shape)
```

```
    @property
    def hyperparams(self):
        return {
            "layer": "SoftmaxLayer",
            "n_in": self.n_in,
            "n_out": self.n_in,
            "optimizer": {
                "hyperparams": self.optimizer.hyperparams,
            },
        }
```

定义代价函数

[9]:
```
class ObjectiveBase(ABC):

    def __init__(self):
        super().__init__()

    @abstractmethod
    def loss(self, y_true, y_pred):
        """
        函数作用：计算损失
        """
        raise NotImplementedError

    @abstractmethod
    def grad(self, y_true, y_pred, **kwargs):
        """
        函数作用：计算代价函数的梯度
        """
        raise NotImplementedError


class SquaredError(ObjectiveBase):
    """
    二次代价函数。
    """
    def __init__(self):
        super().__init__()

    def __call__(self, y_true, y_pred):
        return self.loss(y_true, y_pred)

    def __str__(self):
        return "SquaredError"

    @staticmethod
    def loss(y_true, y_pred):
        """
        参数说明：
        y_true：训练的 n 个样本的真实值， 形状为 (n,m) 数组；
        y_pred：训练的 n 个样本的预测值， 形状为 (n,m) 数组；
        """
        (n, _) = y_true.shape
        return 0.5 * np.linalg.norm(y_pred - y_true) ** 2 / n

    @staticmethod
    def grad(y_true, y_pred, z, acti_fn):
```

```
        (n, _) = y_true.shape
        return (y_pred - y_true) * acti_fn.grad(z) / n


class CrossEntropy(ObjectiveBase):
    """
    交叉熵代价函数。
    """
    def __init__(self):
        super().__init__()

    def __call__(self, y_true, y_pred):
        return self.loss(y_true, y_pred)

    def __str__(self):
        return "CrossEntropy"

    @staticmethod
    def loss(y_true, y_pred):
        """
        参数说明:
        y_true: 训练的 n 个样本的真实值， 要求形状为 (n,m) 二进制（每个样本均为 one-hot 编码);
        y_pred: 训练的 n 个样本的预测值， 形状为 (n,m);
        """
        (n, _) = y_true.shape
        eps = np.finfo(float).eps  # 防止 np.log(0)
        cross_entropy = -np.sum(y_true * np.log(y_pred + eps)) / n
        return cross_entropy

    @staticmethod
    def grad(y_true, y_pred):
        (n, _) = y_true.shape
        grad = (y_pred - y_true) / n
        return grad
```

定义深度前馈网络

[10]:
```
def minibatch(X, batchsize=256, shuffle=True):
    """
    函数作用：将数据集分割成 batch， 基于 mini batch 训练，具体可见第 8 章。
    """
    N = X.shape[0]
    idx = np.arange(N)
    n_batches = int(np.ceil(N / batchsize))
    if shuffle:
        np.random.shuffle(idx)
    def mb_generator():
        for i in range(n_batches):
            yield idx[i * batchsize : (i + 1) * batchsize]
    return mb_generator(), n_batches
```

[11]:
```
class DFN(object):

    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=0.01)",
        init_w="std_normal",
        loss=CrossEntropy()
```

```
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用：模型初始化
        FC1 -> Sigmoid -> FC2 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["FC1"] = FullyConnected(
            n_out=self.hidden_dims_1,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC2"] = FullyConnected(
            n_out=self.hidden_dims_2,
            acti_fn="affine(slope=1, intercept=0)",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.is_initialized = True

    def forward(self, X_train):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()
```

```
    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False):
        """
        参数说明:
        X_train: 训练数据
        y_train: 训练数据标签
        n_epochs: epoch 次数
        batch_size: 每次 epoch 的 batch size
        verbose: 是否每个 batch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch)
                y_pred_batch = softmax(out)
                batch_loss = self.loss(y_batch, y_pred_batch)
                grad = self.loss.grad(y_batch, y_pred_batch)
                _, _ = self.backward(grad)
                self.update()
                loss += batch_loss
                if self.verbose:
                    fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                    print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))
            loss /= n_batch
            fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
            print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
            prev_loss = loss

    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

用自定义 DFN 实现，测试 MNIST 数据集

[12]:
```
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1]*X_train.shape[2]).astype('float32')
X_test = X_test.reshape(-1, X_test.shape[1]*X_test.shape[2]).astype('float32')
print(X_train.shape, y_train.shape)
N = 20000   # 取 20000 条数据用以训练
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 784) (60000, 10)
(20000, 784) (20000, 10)
```

[13]:
```
model = DFN(hidden_dims_1=200, hidden_dims_2=10)
```

[14]:
```
model.fit(X_train, y_train, n_epochs=20)
```

```
[Epoch 1] Avg. loss: 2.283  Delta: inf (0.02m/epoch)
[Epoch 2] Avg. loss: 2.204  Delta: 0.078 (0.02m/epoch)
[Epoch 3] Avg. loss: 1.986  Delta: 0.219 (0.02m/epoch)
[Epoch 4] Avg. loss: 1.628  Delta: 0.357 (0.02m/epoch)
[Epoch 5] Avg. loss: 1.292  Delta: 0.336 (0.02m/epoch)
[Epoch 6] Avg. loss: 1.051  Delta: 0.242 (0.02m/epoch)
[Epoch 7] Avg. loss: 0.886  Delta: 0.165 (0.02m/epoch)
[Epoch 8] Avg. loss: 0.772  Delta: 0.114 (0.02m/epoch)
[Epoch 9] Avg. loss: 0.691  Delta: 0.081 (0.02m/epoch)
[Epoch 10] Avg. loss: 0.631  Delta: 0.060 (0.02m/epoch)
[Epoch 11] Avg. loss: 0.585  Delta: 0.046 (0.02m/epoch)
[Epoch 12] Avg. loss: 0.548  Delta: 0.037 (0.02m/epoch)
[Epoch 13] Avg. loss: 0.519  Delta: 0.029 (0.02m/epoch)
[Epoch 14] Avg. loss: 0.494  Delta: 0.025 (0.02m/epoch)
[Epoch 15] Avg. loss: 0.474  Delta: 0.020 (0.02m/epoch)
[Epoch 16] Avg. loss: 0.456  Delta: 0.018 (0.02m/epoch)
[Epoch 17] Avg. loss: 0.441  Delta: 0.015 (0.02m/epoch)
[Epoch 18] Avg. loss: 0.428  Delta: 0.013 (0.02m/epoch)
[Epoch 19] Avg. loss: 0.417  Delta: 0.011 (0.02m/epoch)
[Epoch 20] Avg. loss: 0.406  Delta: 0.011 (0.02m/epoch)
```

[15]:
```
print(model.evaluate(X_test, y_test))
```

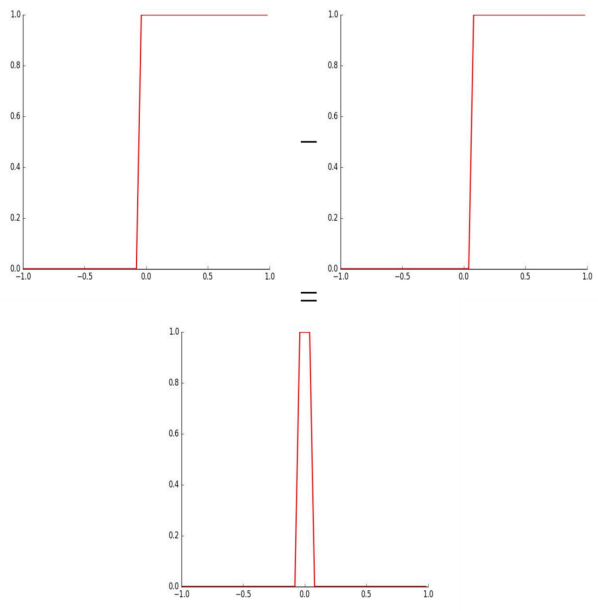
```
0.894
```

5.4 神经网络的万能近似定理

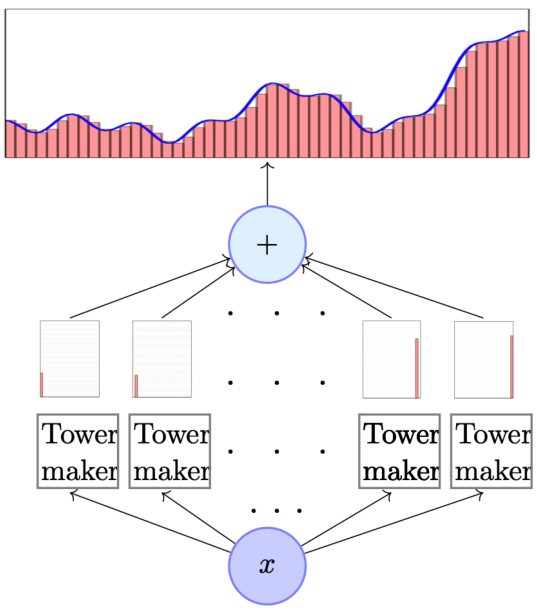
万能近似定理：一个前馈神经网络如果具有线性层和至少一层具有“挤压”性质的激活函数（如 sigmoid 等），给定网络足够数量的隐藏单元，它可以以任意精度来近似任何从一个有限维空间到另一个有限维空间的 borel 可测函数。

sigmoid 函数的万能近似可视化证明 (以一元函数为例)：

1. 我们可以通过两个 sigmoid 函数 ($\boldsymbol{y} = \text{sigmoid}(\boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{b})$) 生成一个 *tower*，如图：



2. 我们构造多个这样的 *tower* 近似任意函数：



可视化证明可参考：http://neuralnetworksanddeeplearning.com/chap4.html

或者 https://www.cse.iitm.ac.in/~miteshk/CS7015/Slides/Teaching/pdf/Lecture3.pdf

5.5 实例：学习 XOR

我们的网络是 $f(\boldsymbol{x}; \boldsymbol{W}, c, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{\boldsymbol{0}, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b$

[16]:
```
class XOR(object):

    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=1.0)",
        init_w="std_normal(gain=1.0)",
        loss=SquaredError()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用: 模型初始化
        FC1 -> Relu -> FC2
        """
```

```
        self.layers = OrderedDict()
        self.layers["FC1"] = FullyConnected(
            n_out=self.hidden_dims_1,
            acti_fn="relu",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC2"] = FullyConnected(
            n_out=self.hidden_dims_2,
            acti_fn="affine(slope=1, intercept=0)",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.is_initialized = True

    def forward(self, X_train):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20001, batch_size=4):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        """
        self.n_epochs = n_epochs
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
```

```
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            out, _ = self.forward(X_train)
            anti_fn = Affine()
            y_pred = anti_fn.forward(out)
            loss = self.loss(y_train, y_pred)
            grad = self.loss.grad(y_train, y_pred, out, anti_fn)
            _, _ = self.backward(grad)
            self.update()
            if not i%5000:
                fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
                print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
                prev_loss = loss

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

[17]:
```
X_train = np.array([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
y_train = np.array([[0.0], [1.0], [1.0], [0.0]])
```

[20]:
```
model = XOR(hidden_dims_1=2, hidden_dims_2=1)
```

[21]:
```
model.fit(X_train, y_train)
```

```
[Epoch 1] Avg. loss: 0.086  Delta: inf (0.00m/epoch)
[Epoch 5001] Avg. loss: 0.000  Delta: 0.086 (0.00m/epoch)
[Epoch 10001] Avg. loss: 0.000  Delta: 0.000 (0.00m/epoch)
[Epoch 15001] Avg. loss: 0.000  Delta: 0.000 (0.00m/epoch)
[Epoch 20001] Avg. loss: 0.000  Delta: 0.000 (0.00m/epoch)
```

[22]:
```
print(model.hyperparams)
```

```
{'init_w': 'std_normal(gain=1.0)', 'loss': 'SquaredError', 'optimizer':
'sgd(lr=1.0)', 'hidden_dims_1': 2, 'hidden_dims_2': 1, 'components': {'FC1':
{'W': array([[-1.35300239,  0.95949967],
       [ 0.88153178, -0.95209678]]), 'b': array([[ 3.76300323e-17,
-7.40288739e-03]])}, 'FC2': {'W': array([[1.13438905],
       [1.0503134 ]]), 'b': array([[-1.1389201e-17]])}}}
```

[23]:
```
import numpy
import re

print("numpy:", numpy.__version__)
print("re:", re.__version__)
```

```
numpy: 1.14.5
re: 2.2.1
```

第六章　深度学习中的正则化

正则化的目标是减少模型泛化误差，为此提出了各种方法。本篇提出的正则化方法主要是考虑**当训练误差较小，但泛化误差较大**的情况下。

6.1　参数范数惩罚

许多正则化方法 (如神经网络、线性回归、逻辑回归) 通过对目标函数 J 添加一个参数范数惩罚 $\Omega(\theta)$，限制模型的学习能力。将正则化后的目标函数记为 $\tilde{J}$：

$$\tilde{J}(\theta;\boldsymbol{X},\boldsymbol{y}) = J(\theta;\boldsymbol{X},\boldsymbol{y}) + \alpha\Omega(\theta) \tag{6.1}$$

其中 $\alpha \in [0,+\infty)$ 是衡量参数范数惩罚程度的超参数。$\alpha = 0$ 表示没有正则化，α 越大对应正则化惩罚越大。

在神经网络中，参数包括每层线性变换的权重和偏置，我们**通常只对权重做惩罚而不对偏置做正则惩罚**；使用向量 $\boldsymbol{w}$ 表示应受惩罚影响的权重，用向量 θ 表示所有参数。

6.1.1　L^2 正则化

L^2 参数正则化 (也称为岭回归、Tikhonov 正则) 通常被称为权重衰减 (`weight decay`)，是通过向目标函数添加一个正则项 $\Omega(\theta) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$ ，使权重更加接近原点。

目标函数：

$$\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = J(\boldsymbol{w};\boldsymbol{X},y) + \frac{\alpha}{2}\boldsymbol{w}^\top\boldsymbol{w} \tag{6.2}$$

计算梯度：

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) + \alpha\boldsymbol{w} \tag{6.3}$$

更新权重：

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon(\alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y})) = (1-\epsilon\alpha)\boldsymbol{w} - \epsilon\nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) \tag{6.4}$$

从上式可以看出，加入权重衰减后会导致学习规则的修改，即在每步执行梯度更新前先收缩权重 (乘以 $(1-\epsilon\alpha)$)。

以第六章介绍的代价函数 $J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{y}^{(i)}\log\hat{\boldsymbol{y}}^{(i)} + (1-\boldsymbol{y}^{(i)})\log(1-\hat{\boldsymbol{y}}^{(i)})\right)$ 为例，在增加 L^2 正则化后，代价函数变为：

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{y}^{(i)}\log\left(\hat{\boldsymbol{y}}^{(i)} + (1-\boldsymbol{y}^{(i)})\log\left(1-\hat{\boldsymbol{y}}^{(i)}\right)\right)\right.}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}} \tag{6.5}$$

而在反向传播的时候，须加上正则化项的梯度：

$$\frac{d}{d\boldsymbol{W}}(\frac{1}{2}\frac{\lambda}{m}\boldsymbol{W}^2) = \frac{\lambda}{m}\boldsymbol{W} \tag{6.6}$$

6.1.2　L^1 正则化

将参数惩罚项 $\Omega(\theta)$ 由权重衰减项修改为各个参数的绝对值之和，可以得到 L^1 正则化：

$$\Omega(\theta) = \|\boldsymbol{w}\|_1 = \sum_i |w_i| \tag{6.7}$$

目标函数：

$$\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) + \alpha\|\boldsymbol{w}\|_1 \tag{6.8}$$

计算梯度：

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) + \alpha sgn(\boldsymbol{w}) \tag{6.9}$$

其中 $sgn(x)$ 为符号函数，取各个元素的正负号。

更新权重：

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon(\alpha sgn(\boldsymbol{w}) + \nabla_{\boldsymbol{w}}J(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y})) \tag{6.10}$$

同样，在增加 L^1 正则化后，代价函数变为：

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{y}^{(i)}\log\hat{\boldsymbol{y}}^{(i)} + (1-\boldsymbol{y}^{(i)})\log(1-\hat{\boldsymbol{y}}^{(i)})\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}|W_{k,j}^{[l]}|}_{\text{L1 regularization cost}} \tag{6.11}$$

而在反向传播的时候，须加上正则化项的梯度：

$$\frac{d}{d\boldsymbol{W}}(\frac{\lambda}{m}\|\boldsymbol{W}\|) = \frac{\lambda}{m}sgn(\boldsymbol{W}) \tag{6.12}$$

[1]:
```
from abc import ABC, abstractmethod
import numpy as np
from PIL import Image
%matplotlib inline
import matplotlib.pyplot as plt
import math
import re
import time
import progressbar
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

[2]:
```
class RegularizerBase(ABC):

    def __init__(self, **kwargs):
        super().__init__()

    @abstractmethod
    def loss(self, **kwargs):
        raise NotImplementedError

    @abstractmethod
    def grad(self, **kwargs):
        raise NotImplementedError

class L1Regularizer(RegularizerBase):

    def __init__(self, lambd=0.001):
        super().__init__()
        self.lambd = lambd

    def loss(self, params):
        loss = 0
        pattern = re.compile(r'^W\d+')
        for key, val in params.items():
            if pattern.match(key):
                loss +=  0.5 * np.sum(np.abs(val)) * self.lambd
        return loss

    def grad(self, params):
        for key, val in params.items():
            grad = self.lambd * np.sign(val)
        return grad

class L2Regularizer(RegularizerBase):

    def __init__(self, lambd=0.001):
        super().__init__()
        self.lambd = lambd

    def loss(self, params):
        loss = 0
```

```
        for key, val in params.items():
            loss +=  0.5 * np.sum(np.square(val)) * self.lambd
        return loss

    def grad(self, params):
        for key, val in params.items():
            grad = self.lambd * val
        return grad

class RegularizerInitializer(object):

    def __init__(self, regular_name="l2"):
        self.regular_name = regular_name

    def __call__(self):
        r = r"([a-zA-Z]*)=([^,)]*)"
        regular_str = self.regular_name.lower()
        kwargs = dict([(i, eval(j)) for (i, j) in re.findall(r, regular_str)])
        if  "l1" in regular_str.lower():
            regular = L1Regularizer(**kwargs)
        elif "l2" in regular_str.lower():
            regular = L2Regularizer(**kwargs)
        else:
            raise ValueError("Unrecognized regular: {}".format(regular_str))
        return regular
```

我们对第六章介绍的 DFN 引入正则项，我们将第六章中介绍的函数存储在 chapter6.py 中。

[3]:
```
from chapter6 import WeightInitializer, ActivationInitializer, LayerBase, CrossEntropy, OrderedDict, softmax
```

[4]:
```
class FullyConnected(LayerBase):
    """
    定义全连接层，实现 a=g(x*W+b)，前向传播输入 x，返回 a；反向传播输入
    """
    def __init__(self, n_out, acti_fn, init_w, optimizer=None):
        """
        参数说明：
        acti_fn：激活函数， str 型
        init_w：权重初始化方法， str 型
        n_out：隐藏层输出维数
        optimizer：优化方法
        """
        super().__init__(optimizer)
        self.n_in = None  # 隐藏层输入维数， int 型
        self.n_out = n_out  # 隐藏层输出维数， int 型
        self.acti_fn = ActivationInitializer(acti_fn)()
        self.init_w = init_w
        self.init_weights = WeightInitializer(mode=init_w)
        self.is_initialized = False  # 是否初始化， bool 型变量

    def _init_params(self):
        b = np.zeros((1, self.n_out))
        W = self.init_weights((self.n_in, self.n_out))
        self.params = {"W": W, "b": b}
        self.gradients = {"W": np.zeros_like(W), "b": np.zeros_like(b)}
        self.is_initialized = True

    def forward(self, X, retain_derived=True):
        """
```

```
        全连接网络的前向传播，原理见上文 反向传播算法 部分。

        参数说明：
        X：输入数组，为 (n_samples, n_in)，float 型
        retain_derived：是否保留中间变量，以便反向传播时再次使用，bool 型
        """
        if not self.is_initialized:  # 如果参数未初始化，先初始化参数
            self.n_in = X.shape[1]
            self._init_params()
        W = self.params["W"]
        b = self.params["b"]
        z = X @ W + b
        a = self.acti_fn.forward(z)
        if retain_derived:
            self.X.append(X)
        return a

    def backward(self, dLda, retain_grads=True, regular=None):
        """
        全连接网络的反向传播，原理见上文 反向传播算法 部分。

        参数说明：
        dLda：关于损失的梯度，为 (n_samples, n_out)，float 型
        retain_grads：是否计算中间变量的参数梯度，bool 型
        regular：正则化项
        """
        if not isinstance(dLda, list):
            dLda = [dLda]
        dX = []
        X = self.X
        for da, x in zip(dLda, X):
            dx, dw, db = self._bwd(da, x, regular)
            dX.append(dx)
            if retain_grads:
                self.gradients["W"] += dw
                self.gradients["b"] += db
        return dX[0] if len(X) == 1 else dX

    def _bwd(self, dLda, X, regular):
        W = self.params["W"]
        b = self.params["b"]
        Z = X @ W + b
        dZ = dLda * self.acti_fn.grad(Z)
        dX = dZ @ W.T
        dW = X.T @ dZ
        db = dZ.sum(axis=0, keepdims=True)
        if regular is not None:
            n = X.shape[0]
            dW_norm = regular.grad(self.params) / n
            dW += dW_norm
        return dX, dW, db

    @property
    def hyperparams(self):
        return {
            "layer": "FullyConnected",
            "init_w": self.init_w,
            "n_in": self.n_in,
            "n_out": self.n_out,
```

```
            "acti_fn": str(self.acti_fn),
            "optimizer": {
                "hyperparams": self.optimizer.hyperparams,
            },
            "components": {
                k: v for k, v in self.params.items()
            }
        }
```

[5]:
```
def minibatch(X, batchsize=256, shuffle=True):
    """
    函数作用：将数据集分割成 batch， 基于 mini batch 训练，具体可见第 8 章。
    """
    N = X.shape[0]
    idx = np.arange(N)
    n_batches = int(np.ceil(N / batchsize))
    if shuffle:
        np.random.shuffle(idx)
    def mb_generator():
        for i in range(n_batches):
            yield idx[i * batchsize : (i + 1) * batchsize]

    return mb_generator(), n_batches

class DFN(object):

    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=0.01)",
        init_w="std_normal",
        regular_act=None,
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.regular_act = regular_act
        self.regular = None
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用：模型初始化
        FC1 -> Sigmoid -> FC2 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["FC1"] = FullyConnected(
            n_out=self.hidden_dims_1,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC2"] = FullyConnected(
            n_out=self.hidden_dims_2,
            acti_fn="affine(slope=1, intercept=0)",
```

```
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        if self.regular_act is not None:
            self.regular = RegularizerInitializer(self.regular_act)()
        self.is_initialized = True

    def forward(self, X_train):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out, regular=self.regular)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        verbose：是否每个 batch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()

        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
```

```
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
            out, _ = self.forward(X_batch)
            y_pred_batch = softmax(out)
            batch_loss = self.loss(y_batch, y_pred_batch)
            # 正则化损失
            if self.regular is not None:
                for _, layerparams in self.hyperparams['components'].items():
                    assert type(layerparams) is dict
                    batch_loss += self.regular.loss(layerparams)
            grad = self.loss.grad(y_batch, y_pred_batch)
            _, _ = self.backward(grad)
            self.update()
            loss += batch_loss
            if self.verbose:
                fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))

        loss /= n_batch
        fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
        print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
        prev_loss = loss

    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "regular": str(self.regular_act),
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

[6]:
```
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1]*X_train.shape[2]).astype('float32')
X_test = X_test.reshape(-1, X_test.shape[1]*X_test.shape[2]).astype('float32')
```

```
print(X_train.shape, y_train.shape)
N = 20000   # 取 20000 条数据用以训练
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 784) (60000, 10)
(20000, 784) (20000, 10)
```

[7]:
```
"""
不引入正则化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10)
model.fit(X_train, y_train, n_epochs=20, batch_size=64)
```

```
[Epoch 1] Avg. loss: 2.286  Delta: inf (0.01m/epoch)
[Epoch 2] Avg. loss: 2.209  Delta: 0.078 (0.01m/epoch)
[Epoch 3] Avg. loss: 1.993  Delta: 0.215 (0.01m/epoch)
[Epoch 4] Avg. loss: 1.640  Delta: 0.353 (0.01m/epoch)
[Epoch 5] Avg. loss: 1.305  Delta: 0.335 (0.01m/epoch)
[Epoch 6] Avg. loss: 1.063  Delta: 0.242 (0.01m/epoch)
[Epoch 7] Avg. loss: 0.898  Delta: 0.166 (0.01m/epoch)
[Epoch 8] Avg. loss: 0.781  Delta: 0.117 (0.01m/epoch)
[Epoch 9] Avg. loss: 0.696  Delta: 0.085 (0.01m/epoch)
[Epoch 10] Avg. loss: 0.634  Delta: 0.062 (0.01m/epoch)
[Epoch 11] Avg. loss: 0.586  Delta: 0.048 (0.01m/epoch)
[Epoch 12] Avg. loss: 0.549  Delta: 0.037 (0.01m/epoch)
[Epoch 13] Avg. loss: 0.518  Delta: 0.031 (0.02m/epoch)
[Epoch 14] Avg. loss: 0.493  Delta: 0.025 (0.02m/epoch)
[Epoch 15] Avg. loss: 0.473  Delta: 0.021 (0.01m/epoch)
[Epoch 16] Avg. loss: 0.454  Delta: 0.018 (0.01m/epoch)
[Epoch 17] Avg. loss: 0.439  Delta: 0.015 (0.01m/epoch)
[Epoch 18] Avg. loss: 0.425  Delta: 0.014 (0.01m/epoch)
[Epoch 19] Avg. loss: 0.414  Delta: 0.012 (0.01m/epoch)
[Epoch 20] Avg. loss: 0.404  Delta: 0.010 (0.01m/epoch)
```

[8]:
```
print("without regularization -- accuracy:{}".format(model.evaluate(X_test, y_test)))

###### if show params ######
# print("regular", model.hyperparams["regular"], "\nparams:", model.hyperparams["components"])
```

```
without regularization -- accuracy:0.8961
```

[9]:
```
"""
引入 l2 正则化
"""
model_re = DFN(hidden_dims_1=200, hidden_dims_2=10, regular_act="l2(lambd=0.01)")
model_re.fit(X_train, y_train, n_epochs=20)
```

```
[Epoch 1] Avg. loss: 2.363  Delta: inf (0.02m/epoch)
[Epoch 2] Avg. loss: 2.284  Delta: 0.079 (0.02m/epoch)
[Epoch 3] Avg. loss: 2.068  Delta: 0.216 (0.02m/epoch)
[Epoch 4] Avg. loss: 1.729  Delta: 0.339 (0.02m/epoch)
[Epoch 5] Avg. loss: 1.428  Delta: 0.301 (0.02m/epoch)
[Epoch 6] Avg. loss: 1.226  Delta: 0.202 (0.02m/epoch)
[Epoch 7] Avg. loss: 1.096  Delta: 0.130 (0.02m/epoch)
```

```
[Epoch 8] Avg. loss: 1.013  Delta: 0.083 (0.02m/epoch)
[Epoch 9] Avg. loss: 0.958  Delta: 0.055 (0.02m/epoch)
[Epoch 10] Avg. loss: 0.923  Delta: 0.035 (0.01m/epoch)
[Epoch 11] Avg. loss: 0.899  Delta: 0.024 (0.01m/epoch)
[Epoch 12] Avg. loss: 0.883  Delta: 0.016 (0.01m/epoch)
[Epoch 13] Avg. loss: 0.872  Delta: 0.011 (0.01m/epoch)
[Epoch 14] Avg. loss: 0.865  Delta: 0.007 (0.01m/epoch)
[Epoch 15] Avg. loss: 0.860  Delta: 0.004 (0.01m/epoch)
[Epoch 16] Avg. loss: 0.858  Delta: 0.002 (0.01m/epoch)
[Epoch 17] Avg. loss: 0.858  Delta: 0.001 (0.01m/epoch)
[Epoch 18] Avg. loss: 0.858  Delta: -0.000 (0.02m/epoch)
[Epoch 19] Avg. loss: 0.859  Delta: -0.001 (0.01m/epoch)
[Epoch 20] Avg. loss: 0.860  Delta: -0.001 (0.01m/epoch)
```

[10]:
```
print("with L2 regularization -- accuracy:{}".format(model_re.evaluate(X_test, y_test)))


###### if show params ######
# print("regular", model_re.hyperparams["regular"], "\nparams:", model_re.hyperparams["components"])
```

```
with L2 regularization -- accuracy:0.8958
```

6.1.3 总结

相比 L^2 正则化，L^1 正则化会产生更稀疏的解。

假设 $\boldsymbol{w}^*$ 为未正则化的目标函数取得最优时的权重向量，并假设原目标函数有二阶导，将 $J(\boldsymbol{w})$ 在 $\boldsymbol{w}^*$ 处二阶泰勒展开 (最优值点一阶导数为 0)：

$$J(\boldsymbol{w}) \approx J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \mathrm{H}(\boldsymbol{w} - \boldsymbol{w}^*) \tag{6.13}$$

其中 H 是 $J(\boldsymbol{w})$ 在 $\boldsymbol{w}^*$ 处的海森矩阵。$J(\boldsymbol{w})$ 最小时满足上式导数为 0，于是：

$$\nabla J(\boldsymbol{w}) = \mathrm{H}(\boldsymbol{w} - \boldsymbol{w}^*) = 0 \tag{6.14}$$

我们再考虑 L^2 正则化条件下，$\Omega(\theta) = \alpha\frac{1}{2}\|\boldsymbol{w}\|_2^2$，则可以得到：

$$\nabla J(\boldsymbol{w}) = \mathrm{H}(\boldsymbol{w} - \boldsymbol{w}^*) + \alpha\boldsymbol{w} = 0 \tag{6.15}$$

于是，我们可以得到新的最优解 $\tilde{\boldsymbol{w}}$ 满足：

$$\tilde{\boldsymbol{w}} = (\mathrm{H} + \alpha\boldsymbol{I})^{-1}\mathrm{H}\boldsymbol{w}^* \tag{6.16}$$

如果考 Hessian 矩阵是对角正定矩阵，我们得到 L^2 正则化的最优解是 $\tilde{w}_i = \frac{\mathrm{H}_{i,i}}{\mathrm{H}_{i,i}+\alpha}w_i^*$。如果 $w_i^* \neq 0$，则 $\tilde{w}_i \neq 0$，这说明 L^2 正则化不会使参数变得稀疏。

我们再看 L^1 正则化的最优解，同样，我们得到考虑 L^1 正则化条件下的最优解，此时需要满足：

$$\nabla J(\tilde{\boldsymbol{w}}) = \mathrm{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) + \alpha sgn(\tilde{\boldsymbol{w}}) = 0 \tag{6.17}$$

为了简化讨论，我们假设 H 为对角阵，即 $\mathrm{H} = diag[\mathrm{H}_{1,1}, \mathrm{H}_{2,2}, ...\mathrm{H}_{n,n}], \mathrm{H}_{i,i} > 0$ (可以用 PCA 预处理输入特征得到)，此时

$$\tilde{w}_i = w_i^* - \frac{\alpha}{\mathrm{H}_{i,i}}sgn(\tilde{w}_i) \tag{6.18}$$

从这个式子也可以明显看出 $\tilde{\boldsymbol{w}}$ 和 $\boldsymbol{w}^*$ 是同号的。所以有：

$$\tilde{w}_i = w_i^* - \frac{\alpha}{\mathrm{H}_{i,i}}sgn(w_i^*) = sgn(w_i^*)\left(|w_i^*| - \frac{\alpha}{\mathrm{H}_{i,i}}\right) \tag{6.19}$$

同样，既然 $\tilde{\boldsymbol{w}}$ 和 $\boldsymbol{w}^*$ 是同号的，两边同乘 $sgn(\tilde{\boldsymbol{w}})$，得到：

$$|w_i^*| - \frac{\alpha}{\mathrm{H}_{i,i}} = |\tilde{w}_i| \geq 0 \tag{6.20}$$

于是刚才的式子可以进一步写为：

$$\tilde{w}_i = sgn(w_i^*)\max\left\{|w_i^*| - \frac{\alpha}{\mathrm{H}_{i,i}}, 0\right\} \tag{6.21}$$

可以看出，L^1 正则化有可能通过足够大的 α 实现稀疏。

- 正则化策略可以被解释为最大后验 (MAP) 贝叶斯推断。(详细内容见第五章)
  - L^2 正则化相当于权重是高斯先验的 MAP 贝叶斯推断；
  - L^1 正则化相当于权重是 `Laplace` 先验的 MAP 贝叶斯推断。

6.1.4 作为约束的范数惩罚

考虑参数范数正则化的代价函数：

$$\tilde{J}(\theta; \boldsymbol{X}, \boldsymbol{y}) = J(\theta; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\theta) \tag{6.22}$$

如果想约束 $\Omega(\theta) < k$，k 是某个常数，可以构造广义 Lagrange 函数：

$$\mathcal{L}(\theta, \alpha; \boldsymbol{X}, \boldsymbol{y}) = J(\theta; \boldsymbol{X}, \boldsymbol{y}) + \alpha(\Omega(\theta) - k) \tag{6.23}$$

该约束问题的解是：

$$\theta^* = \arg\min_{\theta} \max_{\alpha, \alpha \geq 0} \mathcal{L}(\theta, \alpha) \tag{6.24}$$

对于该问题，可以通过调节 α 与 k 的值来扩大或缩小权重的约束区域。较大的 α 将得到一个较小的约束区域；而较小的 α 将得到一个较大的约束区域。

另一方面，重新考虑 `1.1` 和 `1.2` 中的正则化，正则化式等价于带约束的目标函数中的约束项。例如以平方损失函数和 L^2 正则化为例，优化模型如下：

$$\begin{aligned} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) &= \sum_{i=1}^{n} (y_i - \theta^\top \mathrm{x_i})^2 \\ s.t. \|\boldsymbol{\theta}\|_2^2 &<= \mathrm{C} \end{aligned} \tag{6.25}$$

采用拉格朗日乘积算子法可以转化为无约束优化问题，即：

$$J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = \sum_{i=1}^{n} (y_i - \boldsymbol{\theta}^\top \boldsymbol{x}_i)^2 + \lambda(\|\boldsymbol{\theta}\|_2^2 - \mathrm{C}) \tag{6.26}$$

6.1.5 欠约束问题

机器学习中许多线性模型，如线性回归和 PCA，都依赖于矩阵 $\boldsymbol{X}^\top\boldsymbol{X}$ 求逆。如果 $\boldsymbol{X}^\top\boldsymbol{X}$ 不可逆，这些方法就会失效。这种情况下，**正则化的许多形式对应求逆 $\boldsymbol{X}^\top\boldsymbol{X} + \alpha\boldsymbol{I}$，且这个正则化矩阵是可逆的**。大多数正则化方法能够保证应用于欠定问题的迭代方法收敛。

例如线性回归的损失函数是平方误差之和：$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})$。

我们添加 L^2 正则项后，目标函数变为 $(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \frac{1}{2}\alpha\boldsymbol{w}^\top\boldsymbol{w}$。

这将普通方程的解从 $\boldsymbol{w} = (\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{y}$ 变为 $\boldsymbol{w} = (\boldsymbol{X}^\top\boldsymbol{X} + \alpha\boldsymbol{I})^{-1}\boldsymbol{X}^\top\boldsymbol{y}$。

6.2 数据增强

6.2.1 数据集增强

数据集增强是解决数据量有限的问题，**让机器学习模型泛化得更好的最好办法是使用更多的数据进行训练** (这在对象识别 (`object detection`) 问题很有效)。

方法:

- 类别不改变，平移不变性。比如对图像的平移，旋转，缩放。
- 注入噪声，可以使模型对噪声更健壮 (如去噪自编码器)。

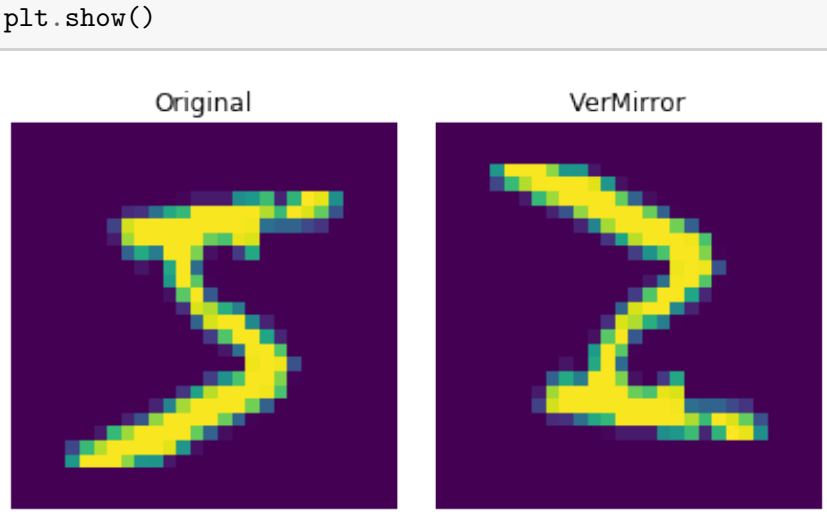
[11]:
```python
class Image(object):

    def __init__(self, image):
        self._set_params(image)

    def _set_params(self, image):
        self.img = image
        self.row = image.shape[0] # 图像高度
        self.col = image.shape[1] # 图像宽度
        self.transform = None

    def Translation(self, delta_x, delta_y):
        """
        平移。
```

```
        参数说明:
        delta_x: 控制左右平移，若大于 0 左移，小于 0 右移
        delta_y: 控制上下平移，若大于 0 上移，小于 0 下移
        """
        self.transform = np.array([[1, 0, delta_x],
                                   [0, 1, delta_y],
                                   [0,  0,  1]])

    def Resize(self, alpha):
        """
        缩放。

        参数说明:
        alpha: 缩放因子，不进行缩放设置为 1
        """
        self.transform = np.array([[alpha, 0, 0],
                                   [0, alpha, 0],
                                   [0,  0,  1]])

    def HorMirror(self):
        """
        水平镜像。
        """
        self.transform = np.array([[1,  0,  0],
                                   [0, -1, self.col-1],
                                   [0,  0,  1]])

    def VerMirror(self):
        """
        垂直镜像。
        """
        self.transform = np.array([[-1, 0, self.row-1],
                                   [0,  1,  0],
                                   [0,  0,  1]])

    def Rotate(self, angle):
        """
        旋转。

        参数说明:
        angle: 旋转角度
        """
        self.transform = np.array([[math.cos(angle),-math.sin(angle),0],
                                   [math.sin(angle), math.cos(angle),0],
                                   [    0,              0,         1]])

    def operate(self):
        temp = np.zeros(self.img.shape, dtype=self.img.dtype)
        for i in range(self.row):
            for j in range(self.col):
                temp_pos = np.array([i, j, 1])
                [x,y,z] = np.dot(self.transform, temp_pos)
                x = int(x)
                y = int(y)
                if x>=self.row or y>=self.col or x<0 or y<0:
                    temp[i,j,:] = 0
                else:
                    temp[i,j,:] = self.img[x,y]
        return temp
```

```
    def __call__(self, act):
        r = r"([a-zA-Z]*)=([^,)]*)"
        act_str = act.lower()
        kwargs = dict([(i, eval(j)) for (i, j) in re.findall(r, act_str)])
        if "translation" in act_str:
            self.Translation(**kwargs)
        elif "resize" in act_str:
            self.Resize(**kwargs)
        elif "hormirror" in act_str:
            self.HorMirror(**kwargs)
        elif "vermirror" in act_str:
            self.VerMirror(**kwargs)
        elif "rotate" in act_str:
            self.Rotate(**kwargs)
        return self.operate()
```

[12]:
```
"""
导入数据， 手写体数字
"""
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
```

[13]:
```
img = X_train[0].reshape(28, 28, 1)
Img = Image(img)
ax = plt.subplot(121)
plt.tight_layout()
plt.imshow(img.reshape(28,28))
plt.title('Original')
plt.axis('off')
ax = plt.subplot(122)
plt.imshow(Img('vermirror').reshape(28,28))
plt.title('VerMirror')
plt.axis('off')
plt.show()
```



6.2.2 噪声鲁棒性

- 将噪声**加入到输入**，等同于**数据集增强**。
- 将噪声**加入到权重**，能够**表现权重的不确定性**，这项技术主要用于循环神经网络。这可以被解释为关于权重的贝叶斯推断的随机实现，贝叶斯学习过程将权重视为不确定的，并且可以通过概率分布表示这种不确定性，向权重添加噪声是反映这种不确定性的一种实用的随机方法。

- 将噪声**加入到输出**，对噪声建模 (滤波)，**标签平滑**。由于大多数数据集的 $\boldsymbol{y}$ 标签都有一定错误，错误的 $\boldsymbol{y}$ 不利于最大化 $\log p(\boldsymbol{y} \mid \boldsymbol{x})$，避免这种情况的一种方法是显式地对标签上的噪声进行建模。

6.3 训练方案

6.3.1 半监督学习

监督学习指训练样本都是带标记的。然而在现实中，获取数据是容易的，但是收集到带标记的数据却是非常昂贵的。**半监督学习指的是既包含部分带标记的样本也有不带标记的样本**，通过这些数据来进行学习。在半监督学习的框架下，$P(\boldsymbol{x})$ 产生的未标记样本和 $P(\boldsymbol{x}, \boldsymbol{y})$ 中的标记样本都用于估计 $P(\boldsymbol{y} \mid \boldsymbol{x})$。在深度学习的背景下，半监督学习通常指的是学习一个表示 $h = f(\boldsymbol{x})$，**学习表示的目的是使相同类中的样本有类似的表示**。

我们可以构建这样一个模型，其中生成模型 $P(\boldsymbol{x})$ 或 $P(\boldsymbol{x}, \boldsymbol{y})$ 与判别模型 $P(\boldsymbol{y} \mid \boldsymbol{x})$ 共享参数，而不用分离无监督和监督部分。例如，我们可以这么做深度学习下的半监督学习，在损失函数中同时考虑两部分损失，一部分是有监督损失，另一部分是无监督损失 (无监督标签为 Pseudo Label，即直接取网络对无标签数据的预测的最大值为标签)。如果我们还鼓励网络学习数据内在的不变性，则可以构造无监督代价是对同一个输入在不同的正则和数据增强条件下的一致性。即要求在不同的条件下，模型的估计要一致。

6.3.2 多任务学习

多任务学习是基于共享表示，**把多个相关的任务放在一起学习**的一种机器学习方法。当模型的一部分被多个额外的任务共享时，这部分将被约束为良好的值，通常会带来**更好的泛化能力**。

从深度学习的观点看，**底层的先验知识**为：**能解释数据变化的因素中，某些因素是跨多个任务共享的**。

可以考虑对复杂的问题，分解为简单且相互独立的子问题来单独解决。做单任务学习时，各个任务之间的模型空间是相互独立的，但这忽略了问题之间所富含的丰富的关联信息；而多任务学习便把多个相关的任务放在一起学习，学习过程中通过一个在浅层的共享表示来互相分享、互相补充学习到的领域相关的信息，互相促进学习，提升泛化的效果。

多任务学习是正则化的一种方法，对于**与主任务相关的任务**，可以看作是添加额外信息，**数据增强**；与**主任务不相关的任务**，可以看作是**引入噪音**，从而提高泛化。

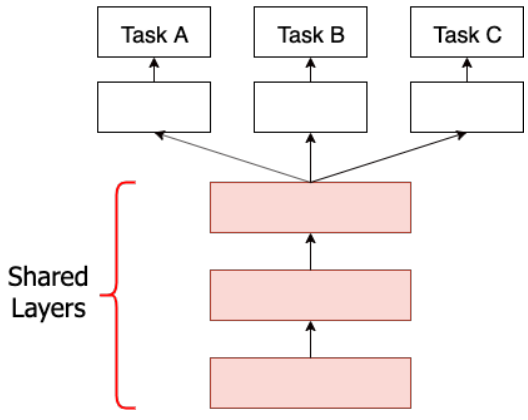


图 6.1. 多任务学习框架示意图，A, B, C 三个任务共享底层。

6.3.3 提前终止

当训练次数过多时会经常遭遇过拟合，此时训练误差会随时间推移减少，而验证集误差会再次上升。提前终止 (`Early Stopping`) 是一种交叉验证策略，我们将一部分训练集作为验证集 (`Validation Set`)。当我们看到验证集的性能越来越差时，我们立即停止对该模型的训练。

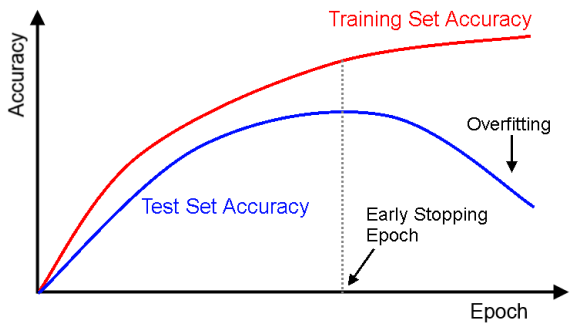


图 6.2. 学习曲线与提前终止。当测试集准确率下降时，可以提前终止。

提前终止的步骤如下：

- 将训练数据划分为训练集和测试集。

- 在训练集上训练，并在一段时间间隔内在测试集上预测。
- 当验证集上的误差高于上次时立即停止训练。
- 使用上一时刻中所得的权重来作为最终权重。

具体终止实现的策略可以多种：

- 策略一: 当泛化损失大于某个阈值时立即停止。
- 策略二: 假定过拟合仅仅发生在训练误差变化缓慢时。
- 策略三: 当 s 个连续时刻的泛化误差增大时停止。

一般而言，除非网络性能的小改进比训练时间更重要，否则选择第一个策略。

将测试集重新融入训练集 为了更好的利用所有数据，我们需要在完成提前终止的首次训练之后进行第二轮的训练，在第二轮中，所有的数据都被包括在内。对此我们有两个基本策略：

- 再次初始化模型，然后使用所有数据再次训练，在第二轮训练中，我们采用第一轮提前终止训练确定的最佳步数。
- 保持从第一轮训练获得的参数，然后使用验证集的数据继续训练，直到验证集的平均损失函数低于提前终止过程终止时的目标值。

提前终止具有正则化效果，其真正机制可理解**为将优化过程的参数空间限制在初始参数值 θ_0 的小邻域内**。考虑平方误差的简单线性模型，采用梯度下降法，可以证明假如学习率为 ϵ ，进行 τ 次训练迭代，则 $\frac{1}{\epsilon\tau}$ 等价于权重衰减系数 α 。

[14]:
```
"""
策略: 连续 4 个时刻验证集正确率没有增加
"""
def early_stopping(valid):
    """
    参数说明:
    valid: 验证集正确率列表
    """
    if len(valid) > 5:
        if valid[-1] < valid[-5] and valid[-2] < valid[-5] and valid[-3] < valid[-5] and valid[-4] < valid[-5]:
            return True
    return False
```

6.4 模型表示

6.4.1 参数绑定与共享

参数范数惩罚或约束是相对于固定区域或点，如 L^2 正则化是对参数偏离 0 进行惩罚。有时我们需要对模型参数之间的相关性进行惩罚，使模型参数尽量接近或者相等：

参数共享：强迫模型某些参数相等：

- 主要应用：卷积神经网络 (CNN)
- 优点：显著降低了 CNN 模型的参数数量 (CNN 模型参数数量经常是千万量级以上)，减少模型所占用的内存，并且显著提高了网络大小而不需要相应的增加训练数据。

6.4.2 稀疏表示

稀疏表示也是卷积神经网络经常用到的正则化方法。L^1 正则化会诱导稀疏的参数，使得许多参数为 0；而稀疏表示是惩罚神经网络的激活单元，稀疏化激活单元。换言之，稀疏表示是使得每个神经元的输入单元变得稀疏，很多输入是 0。如下图所示，相比于全连接层，隐藏层的 h 接受到稀疏的输入 x。

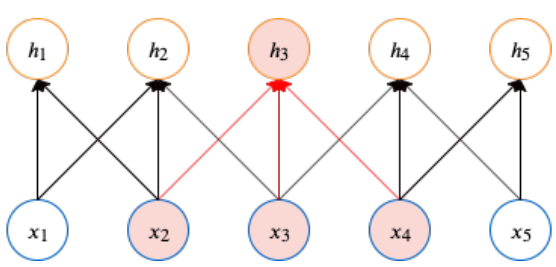


图 6.3. 稀疏表示示意图，每个隐藏层神经元最多连接三个输入单元。

6.4.3 Bagging 及其他集成方法

整合多个弱分类器，成为一个强大的分类器，这是集合学习的思想。集成学习主要有 Boosting 和 Bagging 两种思路。

Bagging 方法

在深度学习中，可以考虑用 Bagging 的思路来正则化。Bagging (`Bootstrap Aggregating`) 是通过重复采样生成新数据集 (Bootstrap)，再在新数据集上分别训练弱分类器，将多个弱分类器汇总成强分类器 (Aggregating)，从而可以降低泛化误差的技术，具体来说 Bagging 步骤：

- **构造 k 个不同的数据集**，每个数据集是从原始数据集中**重复采样**构成，和原始数据集具有相同数量的样本；
- 分别用这 k 个数据集去训练网络，**得到 k 个网络模型**；
- **最终输出的结果**可以用对 k 个网络模型的输出用**加权平均法**或者**投票法**来决定。

其中，我们通常选取的基础弱分类器是 CART 分类器 (见第五章)。而每个数据集 D_i 的构造：假设一共 m 个样本，则在单个数据集中 (需重复采样样本 m 次)，样本不会被采样到的概率约为 $(1-\frac{1}{m})^m \approx \frac{1}{e} = 36.8\%$。

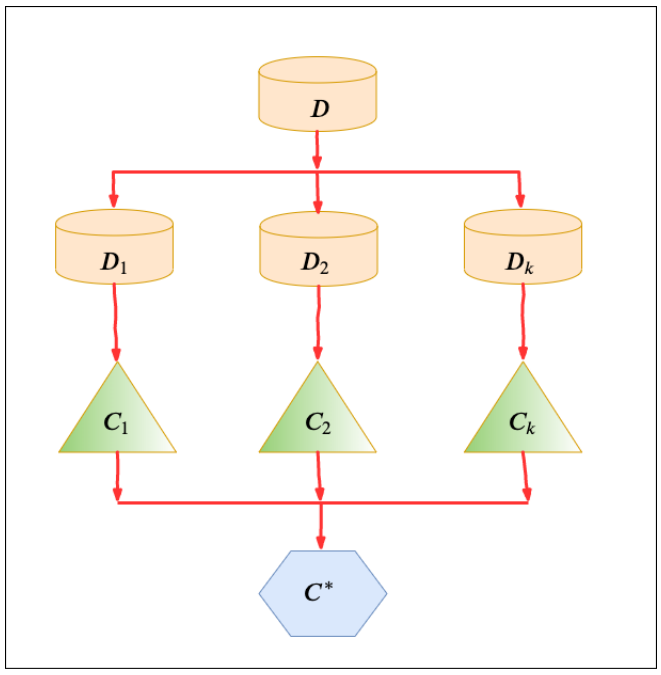


图 6.4. Bagging 方法示意图。

这种策略在机器学习中被称为模型平均 (`Model Averaging`)。模型平均是一个减少泛化误差的非常强大可靠的方法，例如我们假设有 k 个回归模型, 每个模型误差是 ϵ_i，误差服从零均值、方差为 v 、协方差为 c 的多维正态分布，则模型平均预测的误差为 $\frac{1}{k}\sum_i \epsilon_i$，均方误差的期望为

$$\mathbb{E}\left[(\frac{1}{k}\sum_i \epsilon_i)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i(\epsilon_i^2 + \sum_{i\neq j}\epsilon_i\epsilon_j)\right] = \frac{1}{k}v + \frac{k-1}{k}c \tag{6.27}$$

可见，在误差完全相关即 $c = v$ 的情况下，均方误差为 v，模型平均没有帮助。在误差完全不相关即 $c = 0$ 时，模型平均的均方误差的期望仅为 $\frac{1}{k}v$ ，这说明集成平方误差的期望随集成规模的增大而线性减少。

Bagging 方法的优缺点：

- 优点：Bagging 利用集成学习的优势，其中多个弱学习器的表现优于单个强学习器。它有助于减少方差，从而帮助我们避免过拟合。
- 缺点：该模型缺乏可解释性。如果建模不当，则可能存在高偏差的问题 (弱分类器很差，集成后预测性能仍然很差)。另一个重要的缺点是，尽管 Bagging 可以提高准确性，但计算昂贵。尤其在网络模型中使用，我们的网络模型本来就比较复杂，参数很多，现在参数又增加了 k 倍，从而导致训练这样的网络要花更加多的时间和空间。因此一般 k 的个数不能太多，比如 5 - 10 个就可以了。

随机森林

随机森林是对 Bagging 决策树的改进。像 CART 这样的决策树的问题在于它们的贪婪性，他们使用最小化误差的贪婪算法选择的要分割的特征变量。这样，即使使用 Bagging，各个决策树也可以具有很多结构相似性，进而它们的预测具有高度相关性。

如果来自子模型的预测不相关或充其量是弱相关的，则将多个模型中的预测组合在一起会更好。这是随机森林改动的动机，它更改了学习子模型的方式。这是一个简单的调整。在 CART 中，选择分割点时，允许学习算法浏览所有特征变量和所有变量值，以选择最佳的分割点。随机森林算法更改了此过程，学习算法仅限于要搜索特征的随机样本。

每个子模型可以搜索的特征的数目 p 是模型的超参数，可以用交叉验证优化，默认参数为：

- 分类问题：$p = \sqrt{n}$
- 回归问题：$p = n/3$

其中 p 是可以在分割点搜索的随机选择特征的数目，n 是输入特征变量的数目。

接下来，描述一下随机森林的训练过程：

1. 从训练数据中创建数据子集，从全部 n 个特征中随机选择 p 个特征，其中 $p \ll n$。
2. 在 p 个特征中，执行 CART 的步骤，生成一棵决策树。
3. 通过重复步骤 1 至 2 执行 k 次来构建森林，以创建数量为 k 的树。

随机森林的预测过程：

1. 采取测试特征并使用每个随机创建的决策树的规则来预测结果并存储预测结果 (目标)。
2. 计算每个预测目标的票数。
3. 将最高得票的预测目标视为随机森林算法的最终预测。

与其他分类技术相比，随机森林的优点：

- 对于分类问题中的应用，随机森林将避免过拟合问题。
- 对于分类和回归任务，可以使用相同的随机森林算法。
- 随机森林可用于从训练数据集中识别最重要的特征，换句话说，就是特征工程。

方法解决过拟合

在 “偏差和方差” 中我们介绍到 (第五章)，一个分类器的总期望误差是由偏差和方差这两部分之和构成的。Bagging 方法能够通过减少方差分量来降低期望误差值，包含的分类器越多，方差减少量就越大。

从泛化稳定性的角度来看，当学习方法不稳定时，即输入数据的微小变化能导致生成差别相当大的分类器。正则化技术是通过调整输入权重分配和校准输出的思路来缓解该问题的。而集成学习提供了另一种解决问题的思路，实际上，尽可能地使学习方法不稳定，增加集成分类器中的多样性，有助于提高分类器性能。当对决策树使用 Bagging 技术时，决策树已经是不稳定的，如果不对树进行剪枝，经常可以获得更好的性能，而这会使决策树变得不稳定。

更多关于集成方法及其衍生会在本章最后补充介绍（ Boosting，GBDT，XGBoost ）。

[15]:
```
from chapter5 import ClassificationTree
```

[16]:
```
# 进度条
bar_widgets = [
    'Training: ', progressbar.Percentage(), ' ', progressbar.Bar(marker="-", left="[", right="]"),
    ' ', progressbar.ETA()
]


def get_random_subsets(X, y, n_subsets, replacements=True):
    """ 从训练数据中抽取数据子集 (默认可重复抽样)"""
    n_samples = np.shape(X)[0]
    # 将 X 和 y 拼接，并将元素随机排序
    Xy = np.concatenate((X, y.reshape((1, len(y))).T), axis=1)
    np.random.shuffle(Xy)
    subsets = []
    # 如果抽样时不重复抽样，可以只使用 50% 的训练数据；如果抽样时可重复抽样，使用全部的训练数据，默认可重复抽样
    subsample_size = int(n_samples // 2)
    if replacements:
        subsample_size = n_samples
    for _ in range(n_subsets):
        idx = np.random.choice(
            range(n_samples),
            size=np.shape(range(subsample_size)),
            replace=replacements)
        X = Xy[idx][:, :-1]
        y = Xy[idx][:, -1]
        subsets.append([X, y])
    return subsets


class Bagging():
    """
```

```
    Bagging 分类器。使用一组分类树，这些分类树使用特征训练数据的随机子集。
    """
    def __init__(self, n_estimators=100, max_features=None, min_samples_split=2,
                 min_gain=0, max_depth=float("inf")):
        self.n_estimators = n_estimators    # 树的数目
        self.min_samples_split = min_samples_split    # 分割所需的最小样本数
        self.min_gain = min_gain            # 分割所需的最小纯度 (最小信息增益)
        self.max_depth = max_depth          # 树的最大深度
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)
        # 初始化决策树
        self.trees = []
        for _ in range(n_estimators):
            self.trees.append(
                ClassificationTree(
                    min_samples_split=self.min_samples_split,
                    min_impurity=min_gain,
                    max_depth=self.max_depth))

    def fit(self, X, y):
        # 对每棵树选择数据集的随机子集
        subsets = get_random_subsets(X, y, self.n_estimators)
        for i in self.progressbar(range(self.n_estimators)):
            X_subset, y_subset = subsets[i]
            # 用特征子集和真实值训练一棵子模型 (这里的数据也是训练数据集的随机子集)
            self.trees[i].fit(X_subset, y_subset)

    def predict(self, X):
        y_preds = np.empty((X.shape[0], len(self.trees)))
        # 每棵决策树都在数据上预测
        for i, tree in enumerate(self.trees):
            # 基于特征做出预测
            prediction = tree.predict(X)
            y_preds[:, i] = prediction
        y_pred = []
        # 对每个样本，选择最常见的类别作为预测
        for sample_predictions in y_preds:
            y_pred.append(np.bincount(sample_predictions.astype('int')).argmax())
        return y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

[17]:
```
class RandomForest():
    """
    随机森林分类器。使用一组分类树，这些分类树使用特征的随机子集训练数据的随机子集。
    """
    def __init__(self, n_estimators=100, max_features=None, min_samples_split=2,
                 min_gain=0, max_depth=float("inf")):
        self.n_estimators = n_estimators    # 树的数目
        self.max_features = max_features    # 每棵树的最大使用特征数
        self.min_samples_split = min_samples_split    # 分割所需的最小样本数
        self.min_gain = min_gain            # 分割所需的最小纯度 (最小信息增益)
        self.max_depth = max_depth          # 树的最大深度
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)
        # 初始化决策树
        self.trees = []
        for _ in range(n_estimators):
```

```
            self.trees.append(
                ClassificationTree(
                    min_samples_split=self.min_samples_split,
                    min_impurity=min_gain,
                    max_depth=self.max_depth))

    def fit(self, X, y):
        n_features = np.shape(X)[1]
        # 如果 max_features 没有定义，取默认值 sqrt(n_features)
        if not self.max_features:
            self.max_features = int(math.sqrt(n_features))
        # 对每棵树选择数据集的随机子集
        subsets = get_random_subsets(X, y, self.n_estimators)
        for i in self.progressbar(range(self.n_estimators)):
            X_subset, y_subset = subsets[i]
            # 选择特征的随机子集
            idx = np.random.choice(range(n_features), size=self.max_features, replace=True)
            # 保存特征的索引用于预测
            self.trees[i].feature_indices = idx
            # 选择索引对应的特征
            X_subset = X_subset[:, idx]
            # 用特征子集和真实值训练一棵子模型 (这里的数据也是训练数据集的随机子集)
            self.trees[i].fit(X_subset, y_subset)

    def predict(self, X):
        y_preds = np.empty((X.shape[0], len(self.trees)))
        # 每棵决策树都在数据上预测
        for i, tree in enumerate(self.trees):
            # 使用该决策树训练使用的特征
            idx = tree.feature_indices
            # 基于特征做出预测
            prediction = tree.predict(X[:, idx])
            y_preds[:, i] = prediction
        y_pred = []
        # 对每个样本，选择最常见的类别作为预测
        for sample_predictions in y_preds:
            y_pred.append(np.bincount(sample_predictions.astype('int')).argmax())
        return y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy
```

用自定义的 Bagging，乳腺癌数据集测试

[18]:
```
column_names = ['Sample code number','Clump Thickness',
                'Uniformity of Cell Size','Uniformity of Cell Shape',
                'Marginal Adhesion','Single Epithelial Cell Size',
                'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
data = pd.read_csv('../data/cancer/breast-cancer-wisconsin.data',names=column_names)
data = data.replace(to_replace='?',value=np.nan)  # 非法字符替代
data = data.dropna(how='any')  # 去掉空值，any；出现空值行则删除
print(data.shape)
# 随机采样 25% 的数据用于测试，剩下 75% 用于构建训练集
X_train,X_test,y_train,y_test = train_test_split(data[column_names[1:10]], data[column_names[10]],
                                                 test_size=0.25, random_state=1111)
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
```

```
# 修改标签为 0 和 1
print(y_train.shape)
y_train[y_train==2] = 0
y_train[y_train==4] = 1
y_test[y_test==2] = 0
y_test[y_test==4] = 1
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 数据标准化预处理，保证每个维度特征均值为 0，方差为 1
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
y_train = y_train.as_matrix()
y_test = y_test.as_matrix()
```

```
(683, 11)
2    328
4    184
Name: Class, dtype: int64
(512,)
0    328
1    184
Name: Class, dtype: int64
```

[19]:
```
model = Bagging(n_estimators=20)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
Training: 100% [----------------------------------------------] Time: 0:00:04

0.9473684210526315
```

用 sklearn 的 Bagging，乳腺癌数据集测试

[20]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn import tree
model = BaggingClassifier(tree.DecisionTreeClassifier(random_state=1))
model.fit(X_train, y_train)
model.score(X_test,y_test)
```

[20]: 0.9473684210526315

用自定义的随机森林，乳腺癌数据集测试

[21]:
```
model = RandomForest(n_estimators=20)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
Training: 100% [----------------------------------------------] Time: 0:00:01

0.9649122807017544
```

用 sklearn 的随机森林，乳腺癌数据集测试

[22]:
```
from sklearn.ensemble import RandomForestClassifier
model= RandomForestClassifier(random_state=1)
model.fit(X_train, y_train)
model.score(X_test,y_test)
```

[22]: 0.9415204678362573

6.4.4 Dropout

Dropout 的原理为：在每个迭代过程中，**随机选择某些神经元**，并且删除它们在网络中的前向和后向连接，相当于是**“去掉”这些神经元**。如图 6.5 所示，在每批样本训练时，将原始网络中部分隐藏层单元 “去掉”。当然，Dropout 并不意味着这些神经元永远的消失了，在下一批数据迭代前，我们会把网络恢复成最初的全连接网络，然后再用随机的方法去掉部分隐藏层的神经元，接着去迭代更新 $\boldsymbol{W}$, $\boldsymbol{b}$。Dropout 思想可以理解为每次训练时放弃部分神经元对剩下的神经元加重训练，使剩下的神经元具有更强的能力。

每个迭代过程都会有不同的神经元节点的组合，从而导致不同的输出。这可以看成机器学习中的集成方法 (`Ensemble Technique`)。集成模型一般优于单一模型，因为它们可以捕获更多的随机性；相似地，Dropout 使得神经网络模型优于正常的模型。

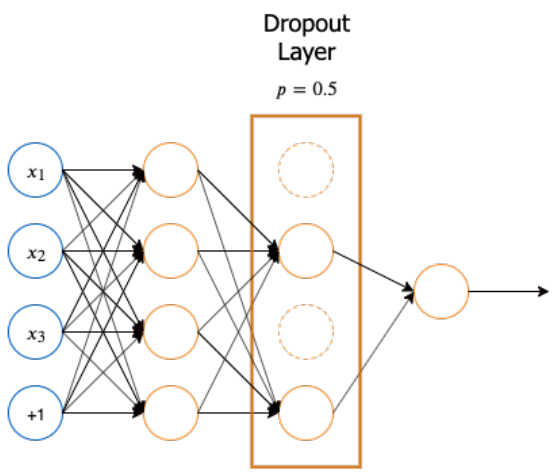


图 6.5. Dropout 示意图，此处 p 为 0.5，训练阶段随机“去掉”一半神经元。

Dropout 的实现步骤：

- 每次加载小批量样本，然后随机采样二值掩码，对于每个单元，掩码是独立采样的。**(将一些单元的输出乘零就能有效的删除一个单元)**；
- 传统的 Dropout，在训练阶段，用了 Dropout 的层，每个神经元以 p 的概率保留（或以 $1-p$ 的概率关闭），然后在测试阶段，不执行 Dropout，也就是所有神经元都不关闭，但是对训练阶段应用了 Dropout 的层上的神经元，为保证强度一致其输出激活值要乘以 p。**原理：为保证训练阶段与测试阶段强度一致，预测时对于每个隐层的输出，都乘以概率** p。可以从数学期望的角度去理解，我们考虑一个神经元的输出为 x (没有 Dropout 的情况下)，则它输出的数学期望为 $px + (1-p)0$，于是在测试阶段，我们直接把每个输出 x 都变换为 px，是可以保持一样的数学期望。而现在我们在训练阶段应用 Dropout 时并没有让神经元 a 的输出激活值除以 p，因此其期望值为 pa，在测试阶段如果不用 Dropout，所有神经元都保留，则输出期望值为 x，为了让测试阶段神经元的输出期望值和训练阶段保持一致 (这样才能正确评估训练出的模型)，就要给测试阶段的输出激活值乘上 p，使其输出期望值保持为 px；
- 现在主流的方法是 Inverted Dropout，和传统的 Dropout 方法有两点不同：在训练阶段，对执行了 Dropout 操作的层，其输出激活值要除以 p；测试阶段则不执行任何操作，既不执行 Dropout，也不用对神经元的输出乘 p。
- 其余部分，与之前一样，运行前向传播、反向传播和学习更新。

Dropout 不仅可以应用在隐含层，也可以应用在输入层。选择保留多少单元的概率值 p 是一个超参数。通常输入单元被保留的概率为 0.8，隐藏单元被保留的概率为 0.5。

Dropout 优点：

- 计算方便，训练过程中使用 Dropout 产生 n 个 (神经单元数目) 随机二进制数与状态相乘即可。
- 适用广 (几乎在所有使用分布式表示且可以用随机梯度下降训练的模型上都表现很好，如前馈神经网络、概率模型、受限波尔兹曼机、循环神经网络等)。
- 相比其他正则化方法 (如权重衰减、过滤器约束和稀疏激活) 更有效，也可与其他形式的正则化合并，得到进一步提升。

Dropout 缺点：

- 不适合宽度太窄的网络，否则大部分网络没有输入到输出的路径。
- 不适合训练数据太小（如小于 5000）的网络，训练数据太小时，Dropout 没有其他方法表现好。
- 不适合非常大的数据集，数据集大的时候正则化效果有限 (大数据集本身的泛化误差就很小)，使用 Dropout 的代价可能超过正则化的好处。

Dropout 与 Bagging 的比较

Dropout 模型中的参数 $\boldsymbol{W}$，$\boldsymbol{b}$ 是共享的，Dropout 下网络迭代时，更新的是同一组 $\boldsymbol{W}$，$\boldsymbol{b}$；而 Bagging 正则化中每个模型有自己的一套参数，相互之间是独立的。当然两种策略都是每次使用基于原始数据集得到的分批的数据集来训练模型。

[23]:
```
class Dropout(ABC):


    def __init__(self, wrapped_layer, p):
        """
        参数说明:
        wrapped_layer: 被 dropout 的层
        p: 神经元保留率
```

```
        """
        super().__init__()
        self._base_layer = wrapped_layer
        self.p = p
        self._init_wrapper_params()

    def _init_wrapper_params(self):
        self._wrapper_derived_variables = {"dropout_mask": None}
        self._wrapper_hyperparams = {"wrapper": "Dropout", "p": self.p}

    def flush_gradients(self):
        """
        函数作用：调用 base layer 重置更新参数列表
        """
        self._base_layer.flush_gradients()

    def update(self):
        """
        函数作用：调用 base layer 更新参数
        """
        self._base_layer.update()

    def forward(self, X, is_train=True):
        """
        参数说明：
        X：输入数组；
        is_train：是否为训练阶段，bool 型；
        """
        mask = np.ones(X.shape).astype(bool)
        if is_train:
            mask = (np.random.rand(*X.shape) < self.p) / self.p
            X = mask * X
        self._wrapper_derived_variables["dropout_mask"] = mask
        return self._base_layer.forward(X)

    def backward(self, dLda):
        return self._base_layer.backward(dLda)

    @property
    def hyperparams(self):
        hp = self._base_layer.hyperparams
        hpw = self._wrapper_hyperparams
        if "wrappers" in hp:
            hp["wrappers"].append(hpw)
        else:
            hp["wrappers"] = [hpw]
        return hp
```

[24]:
```
def minibatch(X, batchsize=256, shuffle=True):
    """
    函数作用：将数据集分割成 batch， 基于 mini batch 训练，具体可见第 8 章。
    """
    N = X.shape[0]
    idx = np.arange(N)
    n_batches = int(np.ceil(N / batchsize))
    if shuffle:
        np.random.shuffle(idx)
    def mb_generator():
        for i in range(n_batches):
```

```
            yield idx[i * batchsize : (i + 1) * batchsize]
    return mb_generator(), n_batches

class DFN(object):

    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=0.01)",
        init_w="std_normal",
        p=1.0,
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.p = p
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用：模型初始化
        FC1 -> Sigmoid -> FC2 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["FC1"] = Dropout(  # 这里引入 dropout
            FullyConnected(
                n_out=self.hidden_dims_1,
                acti_fn="sigmoid",
                init_w=self.init_w,
                optimizer=self.optimizer
            ), self.p
        )
        self.layers["FC2"] = FullyConnected(
                n_out=self.hidden_dims_2,
                acti_fn="affine(slope=1, intercept=0)",
                init_w=self.init_w,
                optimizer=self.optimizer
        )
        self.is_initialized = True

    def forward(self, X_train, is_train=True):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            try:  # 考虑 dropout
                out = v.forward(out, is_train=is_train)
            except:
                out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
```

```
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()
```

```
    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        verbose：是否每个 batch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch, is_train=True)
                y_pred_batch = softmax(out)
                batch_loss = self.loss(y_batch, y_pred_batch)
                grad = self.loss.grad(y_batch, y_pred_batch)
                _, _ = self.backward(grad)
                self.update()
                loss += batch_loss
                if self.verbose:
                    fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                    print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))
            loss /= n_batch
            fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
            print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
            prev_loss = loss

    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
```

```
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch, is_train=False)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "dropout keep ratio": self.p,
            "components": {k: v.hyperparams for k, v in self.layers.items()}
        }
```

[25]:
```
"""
引入 dropout
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, p=0.5)
model.fit(X_train, y_train, n_epochs=20, batch_size=64)
```

```
[Epoch 1] Avg. loss: 2.286  Delta: inf (0.02m/epoch)
[Epoch 2] Avg. loss: 2.215  Delta: 0.071 (0.02m/epoch)
[Epoch 3] Avg. loss: 2.017  Delta: 0.198 (0.02m/epoch)
[Epoch 4] Avg. loss: 1.681  Delta: 0.335 (0.02m/epoch)
[Epoch 5] Avg. loss: 1.356  Delta: 0.326 (0.02m/epoch)
[Epoch 6] Avg. loss: 1.124  Delta: 0.231 (0.02m/epoch)
[Epoch 7] Avg. loss: 0.965  Delta: 0.160 (0.02m/epoch)
[Epoch 8] Avg. loss: 0.851  Delta: 0.114 (0.02m/epoch)
[Epoch 9] Avg. loss: 0.779  Delta: 0.072 (0.02m/epoch)
[Epoch 10] Avg. loss: 0.718  Delta: 0.061 (0.02m/epoch)
[Epoch 11] Avg. loss: 0.677  Delta: 0.041 (0.02m/epoch)
[Epoch 12] Avg. loss: 0.643  Delta: 0.034 (0.02m/epoch)
[Epoch 13] Avg. loss: 0.615  Delta: 0.027 (0.02m/epoch)
[Epoch 14] Avg. loss: 0.591  Delta: 0.024 (0.02m/epoch)
[Epoch 15] Avg. loss: 0.573  Delta: 0.018 (0.02m/epoch)
[Epoch 16] Avg. loss: 0.554  Delta: 0.020 (0.02m/epoch)
[Epoch 17] Avg. loss: 0.541  Delta: 0.013 (0.02m/epoch)
[Epoch 18] Avg. loss: 0.530  Delta: 0.011 (0.02m/epoch)
[Epoch 19] Avg. loss: 0.525  Delta: 0.005 (0.02m/epoch)
[Epoch 20] Avg. loss: 0.516  Delta: 0.009 (0.02m/epoch)
```

[26]:
```
print("accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
accuracy:0.8889
```

6.5 样本测试

在正则化背景下，通过对抗训练可以减少原有独立同分布的测试集的错误率——在对抗扰动的训练集样本上训练网络。

主要原因之一是高度线性，神经网络主要是基于线性模块构建的。输入改变 ϵ ，则权重为 $\boldsymbol{w}$ 的线性函数将改变 $\epsilon\|\boldsymbol{w}\|_1$ ，对于高维的 $\boldsymbol{w}$ 这是一个非常大的数。对抗训练通过鼓励网络在训练数据附近的局部区域恒定来限制这一个高度敏感的局部线性行为。

6.6 补充材料

6.6.1 Boosting

在前面介绍的 Bagging 方法中，主要通过**对训练数据集进行随机采样**，以重新组合成不同的数据集，利用弱学习算法对不同的新数据集进行学习，得到一系列的预测结果，对这些预测结果做平均或者投票得到最终的预测。

而 Boosting 方法的主要目标是**将弱分类器“提升”为强分类器**，根据前一个弱分类器的训练效果对样本分布进行调整，再根据新的样本分布训练下一个弱分类器，如此迭代，最后将一系列弱分类器组合成一个强分类器。

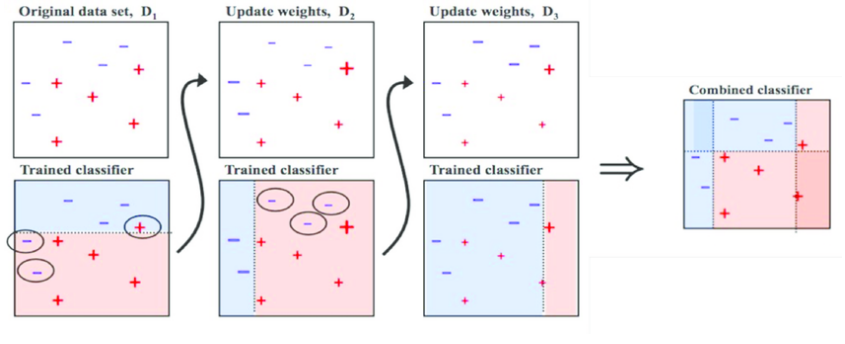


图 6.6. Boosting 方法示意图。

这里便存在几个问题：弱分类器选什么？如何调整样本分布？如何进行组合？针对这三个问题的不同回答就可以得到不同的 Boosting 方法。不过在具体介绍各种 Boosting 方法，我们需要先介绍前向分步加法模型。

前向分步加法模型

首先是加法模型 (`Addtive Model`)：

$$f(x) = \sum_{k=1}^{K} \beta_k \cdot b(\boldsymbol{x}; \gamma_k) \tag{6.28}$$

其中 $b(\boldsymbol{x}; \gamma_k)$ 是基函数，γ_k 是基函数的参数，β_k 是基函数的系数。

前向分步算法 (`Forward Stagewise Algorithm`)

在给定训练数据和损失函数 $\mathcal{L}(y, f(\boldsymbol{x}))$ 的情况下，学习加法模型 $f(\boldsymbol{x})$ 成为经验风险最小化即损失函数最小化的问题：

$$\min_{\beta_k, \gamma_k} \quad \sum_{i=1}^{m} \mathcal{L}\left(y^{(i)}, \sum_{k=1}^{K} \beta_k \cdot b(\boldsymbol{x}^{(i)}; \gamma_k)\right) \tag{6.29}$$

通常这是一个复杂的优化问题。前向分步算法求解这一优化问题的思路是：因为学习的是加法模型，如果能够从前向后，每一步只学习一个基函数及其系数，逐步逼近优化目标函数式，那么就可以简化优化的复杂度。具体地，每步只需优化如下损失函数：

$$\min_{\beta, \gamma} \quad \sum_{i=1}^{M} \mathcal{L}(y^{(i)}, \beta \cdot b(\boldsymbol{x}^{(i)}; \gamma)) \tag{6.30}$$

给定训练数据 $D = \{(\boldsymbol{x}^{(1)}, y^{(1)}), \cdots, (\boldsymbol{x}^{(m)}, y^{(m)})\}, \boldsymbol{x}^{(i)} \in \mathbb{R}^n, y^{(i)} \in \{-1, +1\}$，损失函数 $\mathcal{L}(y, f(\boldsymbol{x}))$ 和基函数集 $\{b(\boldsymbol{x}; \gamma)\}$。前向分步算法的步骤如下：

1. 初始化 $f_0(\boldsymbol{x}) = 0$。
2. 极小化损失函数 $(\beta_k, \gamma_k) = \arg\min_{\beta, \gamma} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, f_{k-1}(\boldsymbol{x}^{(i)}) + \beta \cdot b(\boldsymbol{x}; \gamma))$ 得到参数 β_k 和 γ_k。
3. 更新 $f_k(\boldsymbol{x}) = f_{k-1}(\boldsymbol{x}) + \beta_k \cdot b(\boldsymbol{x}; \gamma_k)$。
4. 重复步骤 2-3，$k = 1, 2, \cdots, K$。
5. 得到加法模型 $f(x) = f_K(x) = \sum_{k=1}^{K} \beta_k b(x; \gamma_k)$。

这样，前向分步算法将同时求解从 $k = 1$ 到 K 的所有参数 β_k, γ_k 的优化问题简化为逐次求解各个 β_k, γ_k 的优化问题。

AdaBoost 算法

回到三个问题，AdaBoost 的解决方案是什么？

- 弱分类器 (基分类器) 选什么？一般选法是决策树桩 (就是只有一层的决策树)。
- 如何调整样本分布？可以先赋予每个训练样本相同的权重；然后用弱分类器进行训练，每次训练后，对分类错误的样本加大权重 (重采样，具体做法后面再述)，使得在下一次的迭代中更加关注这些样本，如图 6.6 所示。
- 如何进行组合？组合方式即为加法模型 $f(\boldsymbol{x}) = sgn\left(\sum_{k=1}^{K} \alpha_k h_k(\boldsymbol{x})\right)$。其中 AdaBoost 的损失函数为指数损失 $\mathcal{L}(y, f(\boldsymbol{x})) = \exp(-yf(\boldsymbol{x}))$。

下面先分析为什么选择指数损失作为损失函数

假设 $f(\boldsymbol{x})$ 能使损失达到最小，对其求偏导：

$$\frac{\partial \exp(-yf(\boldsymbol{x}))}{\partial f(\boldsymbol{x})} = -\exp(-f(\boldsymbol{x}))P(y=1|\boldsymbol{x}) + \exp(f(\boldsymbol{x}))P(y=-1|\boldsymbol{x}) \tag{6.31}$$

然后我们令导数为 0，可以求得：

$$f(\boldsymbol{x}) = \frac{1}{2}\ln\frac{P(y=1 \mid \boldsymbol{x})}{P(y=-1 \mid \boldsymbol{x})} \tag{6.32}$$

接下来我们可以得到输出：

$$\begin{aligned} sgn(f(\boldsymbol{x})) &= sgn\left(\frac{1}{2}\ln\frac{P(y=1 \mid \boldsymbol{x})}{P(y=-1 \mid \boldsymbol{x})}\right) \\ &= \begin{cases} 1, P(y=1 \mid x) > P(y=-1 \mid x) \\ -1, P(y=1 \mid x) < P(y=-1 \mid x) \end{cases} \end{aligned} \tag{6.33}$$

这意味着 $sgn(f(\boldsymbol{x}))$ 达到了贝叶斯最优错误率。换言之，如果指数损失函数最小化，则分类错误率也将最小化。这说明指数损失函数是分类任务 0-1 损失函数的一致性替代函数。由于这个替代函数是单调连续可微函数，因此用它代替 0-1 损失函数作为优化目标。

基分类器权重 α 的更新

回到迭代过程，第一个基分类器 h_1 是通过直接将基学习算法用于初始训练数据 (初始数据分布) D_1 而得；此后迭代地生成 h_k 和 α_k。当基分类器 h_k 基于分布 D_k 产生后，该基分类器的权重 α_k 应当使得 $\alpha_k C_k$ 最小化指数损失函数

$$\begin{aligned} \min_{\alpha_k} \mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + \alpha_k \cdot h_k(\boldsymbol{x})) &= \min_{\alpha_k} \mathbb{E}_{\boldsymbol{x}\sim D_k}[\exp\left(-y\left(f_{k-1}(\boldsymbol{x}) + \alpha_k \cdot h_k(\boldsymbol{x})\right)\right)] \\ &= \min_{\alpha_k} \mathbb{E}_{\boldsymbol{x}\sim D_k}[\exp\left(-y\left(\alpha_k \cdot h_k\left(\boldsymbol{x}\right)\right)\right)] \\ &= \exp(-\alpha_k)P_{\boldsymbol{x}\sim D_k}(y = h_k(\boldsymbol{x})) + \exp(\alpha_k)P_{\boldsymbol{x}\sim D_k}(y \neq h_k(\boldsymbol{x})) \\ &= \exp(-\alpha_k)(1-\varepsilon_k) + \exp(\alpha_k)\varepsilon_k \end{aligned} \tag{6.34}$$

其中，$\varepsilon_k = P_{\boldsymbol{x}\sim D_k}(h_k(\boldsymbol{x}) \neq y)$，表示错误率。我们接下来对该式求导，可以得到：

$$\frac{\partial \mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + \alpha_k \cdot h_k\left(\boldsymbol{x}\right))}{\partial \alpha_k} = -\exp(-\alpha_k)(1-\varepsilon_k) + \exp(\alpha_k)\varepsilon_k \tag{6.35}$$

令导数为 0，有：

$$\alpha_k = \frac{1}{2}\ln\left(\frac{1-\varepsilon_k}{\varepsilon_k}\right) \tag{6.36}$$

这样我们便得到了在每次迭代，训练完基分类器后，加法模型中基函数的系数。

样本分布的更新与迭代过程中训练样本的生成

现在回到第二个问题的解答，我们如何调整样本分布。

在获得 f_{k-1} 之后样本的分布将进行调整，使下一轮基分类器 h_k 能纠正 f_{k-1} 的一些错误。理想的 h_k 能纠正 f_{k-1} 的全部错误，即最小化：

$$\begin{aligned} \mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + h_k\left(\boldsymbol{x}\right)) &= \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x})\right)] \\ &= \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)\exp\left(-yh_k(\boldsymbol{x})\right)] \end{aligned} \tag{6.37}$$

注意 $y^2 = h_k^2(\boldsymbol{x}) = 1$ (因为输出为 1 或 -1)，上式可将 $\exp\left(-yh_k(\boldsymbol{x})\right)$ 泰勒展开：

$$\begin{aligned} \mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x})) &\simeq \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)(1 - yh_k(\boldsymbol{x}) + \frac{y^2h_k^2(\boldsymbol{x})}{2})] \\ &= \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)(1 - f(x)h_t(x) + \frac{1}{2})] \end{aligned} \tag{6.38}$$

于是，理想的基分类器应满足：

$$\begin{aligned} h_k\left(\boldsymbol{x}\right) &= \arg\min_h \mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x})) \\ &= \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)yh_k(\boldsymbol{x})] \\ &= \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim D}\left[\frac{(-yf_{k-1}(\boldsymbol{x}))}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)]}yh_k(\boldsymbol{x})\right] \end{aligned} \tag{6.39}$$

这里的 $\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)]$ 表示一个常数。令 D_k 表示一个分布：

$$D_k\left(\boldsymbol{x}\right) = \frac{\exp\left(-yf_{k-1}(\boldsymbol{x})\right)}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)]}D(\boldsymbol{x}) \tag{6.40}$$

因为根据数学期望的定义，这里等价于令：

$$\begin{aligned} h_k(\boldsymbol{x}) &= \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim D}\left[\frac{\exp\left(-yf_{k-1}(\boldsymbol{x})\right)}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)]}yh_k(\boldsymbol{x})\right] \\ &= \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim D_k}[yG_k(\boldsymbol{x})] \end{aligned} \tag{6.41}$$

由于 $y, h_k \in \{+1, -1\}$，所以有：$y\, h_k = 1 - 2I\left(y \neq h_k\left(\boldsymbol{x}\right)\right)$。

则理想的基分类器：

$$h_k\left(\boldsymbol{x}\right) = \arg\min_h \mathbb{E}_{\boldsymbol{x}\sim D_k}[I\left(y \neq h_k\left(\boldsymbol{x}\right)\right)] \tag{6.42}$$

可见，理想的 $h_k\left(\boldsymbol{x}\right)$ 在分布 D_k 下最小化分类误差。因此，基分类器将基于分布 D_k 来训练，且针对 D_k 的分类误差应当小于 0.5。考虑到 D_k 和 D_{k+1} 的关系有：

$$\begin{aligned}
D_{k+1}\left(\boldsymbol{x}\right) &= \frac{\exp(-yf_k(\boldsymbol{x}))}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp(-yf_k(\boldsymbol{x}))]} D\left(\boldsymbol{x}\right) \\
&= D(\boldsymbol{x}) \exp\left(-yf_{k-1}(\boldsymbol{x})\right) \frac{\exp\left(-y\alpha_k \cdot h_k(\boldsymbol{x})\right)}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_k\left(\boldsymbol{x}\right)\right)]} \\
&= D_k(\boldsymbol{x}) \frac{\exp\left(-y\alpha_k \cdot h_k(\boldsymbol{x})\right) \mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_{k-1}(\boldsymbol{x})\right)]}{\mathbb{E}_{\boldsymbol{x}\sim D}[\exp\left(-yf_k(\boldsymbol{x})\right)]} \\
&= \frac{D_k(\boldsymbol{x}) \exp\left(-y\alpha_k \cdot h_k(\boldsymbol{x})\right)}{\mathbb{E}_{\boldsymbol{x}\sim D}\left[\frac{-yf_{k-1} - y\alpha_k h_k}{\mathbb{E}_{\boldsymbol{x}\sim D}\left[\exp(-yf_{k-1}\left(\boldsymbol{x}\right))\right]}\right]} \\
&= \frac{D_k(\boldsymbol{x}) \exp\left(-y\alpha_k \cdot h_k(\boldsymbol{x})\right)}{\mathbb{E}_{\boldsymbol{x}\sim D}[D_k(\boldsymbol{x}) \exp\left(-y\alpha_k \cdot h_k(\boldsymbol{x})\right)]}
\end{aligned} \tag{6.43}$$

于是，我们便得到了样本分布的迭代更新公式。

自定义实现

[27]:
```
# 进度条
bar_widgets = [
    'Training: ', progressbar.Percentage(), ' ', progressbar.Bar(marker="-", left="[", right="]"),
    ' ', progressbar.ETA()
]
```

[28]:
```
# 决策树桩，作为 Adaboost 算法的弱分类器 (基分类器)
class DecisionStump():

    def __init__(self):
        self.polarity = 1            # 表示决策树桩默认输出的类别为 1 或是 -1
        self.feature_index = None    # 用于分类的特征索引
        self.threshold = None        # 特征的阈值
        self.alpha = None            # 表示分类器准确性的值

class Adaboost():
    """
    Adaboost 算法。
    """
    def __init__(self, n_estimators=5):
        self.n_estimators = n_estimators    # 将使用的弱分类器的数量
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)

    def fit(self, X, y):
        n_samples, n_features = np.shape(X)
        # 初始化权重 (上文中的 D)，均为 1/N
        w = np.full(n_samples, (1 / n_samples))
        self.trees = []
        # 迭代过程
        for _ in self.progressbar(range(self.n_estimators)):
            tree = DecisionStump()
            min_error = float('inf')    # 使用某一特征值的阈值预测样本的最小误差
            # 迭代遍历每个 (不重复的) 特征值，查找预测 y 的最佳阈值
            for feature_i in range(n_features):
                feature_values = np.expand_dims(X[:, feature_i], axis=1)
                unique_values = np.unique(feature_values)
                # 将该特征的每个特征值作为阈值
                for threshold in unique_values:
                    p = 1
                    # 将所有样本预测默认值可以设置为 1
                    prediction = np.ones(np.shape(y))
```

```
                # 低于特征值阈值的预测改为 -1
                prediction[X[:, feature_i] < threshold] = -1
                # 计算错误率
                error = sum(w[y != prediction])
                # 如果错误率超过 50%，我们反转决策树桩默认输出的类别
                # 比如 error = 0.8 => (1 - error) = 0.2，
                # 原来计算的是输出到类别 1 的概率，类别 1 作为默认类别。反转后类别 0 作为默认类别
                if error > 0.5:
                    error = 1 - error
                    p = -1
                # 如果这个阈值导致最小的错误率，则保存
                if error < min_error:
                    tree.polarity = p
                    tree.threshold = threshold
                    tree.feature_index = feature_i
                    min_error = error

        # 计算用于更新样本权值的 alpha 值，也是作为基分类器的系数。
        tree.alpha = 0.5 * math.log((1.0 - min_error) / (min_error + 1e-10))
        # 将所有样本预测默认值设置为 1
        predictions = np.ones(np.shape(y))
        # 如果特征值低于阈值，则修改预测结果，这里还需要考虑弱分类器的默认输出类别
        negative_idx = (tree.polarity * X[:, tree.feature_index] < tree.polarity * tree.threshold)
        predictions[negative_idx] = -1
        # 计算新权值，未正确分类样本的权值增大，正确分类样本的权值减小
        w *= np.exp(-tree.alpha * y * predictions)
        w /= np.sum(w)
        # 保存分类器
        self.trees.append(tree)

def predict(self, X):
    n_samples = np.shape(X)[0]
    y_pred = np.zeros((n_samples, 1))
    # 用每一个基分类器预测样本
    for tree in self.trees:
        # 将所有样本预测默认值设置为 1
        predictions = np.ones(np.shape(y_pred))
        negative_idx = (tree.polarity * X[:, tree.feature_index] < tree.polarity * tree.threshold)
        predictions[negative_idx] = -1
        # 对基分类器加权求和，权重 alpha
        y_pred += tree.alpha * predictions
    # 返回预测结果 1 或 -1
    y_pred = np.sign(y_pred).flatten()
    return y_pred

def score(self, X, y):
    y_pred = self.predict(X)
    accuracy = np.sum(y == y_pred, axis=0) / len(y)
    return accuracy
```

用自定义的 Adaboost，乳腺癌数据集测试

[29]:
```
column_names = ['Sample code number','Clump Thickness',
                'Uniformity of Cell Size','Uniformity of Cell Shape',
                'Marginal Adhesion','Single Epithelial Cell Size',
                'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
data = pd.read_csv('../data/cancer/breast-cancer-wisconsin.data',names=column_names)
data = data.replace(to_replace='?',value=np.nan)  # 非法字符替代
data = data.dropna(how='any')  # 去掉空值，any；出现空值行则删除
```

```
print(data.shape)
# 随机采样 25% 的数据用于测试，剩下 75% 用于构建训练集
X_train,X_test,y_train,y_test = train_test_split(data[column_names[1:10]], data[column_names[10]],
                                                 test_size=0.25, random_state=1111)
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 修改标签为 -1 和 1
print(y_train.shape)
y_train[y_train==2] = -1
y_train[y_train==4] = 1
y_test[y_test==2] = -1
y_test[y_test==4] = 1
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 数据标准化预处理，保证每个维度特征均值为 0，方差为 1
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
y_train = y_train.as_matrix()
y_test = y_test.as_matrix()
```

```
(683, 11)
2    328
4    184
Name: Class, dtype: int64
(512,)
-1    328
 1    184
Name: Class, dtype: int64
```

[30]:
```
model = Adaboost(n_estimators=20)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
Training: 100% [----------------------------------------------] Time: 0:00:00

0.935672514619883
```

用 sklearn 的 Adaboost，乳腺癌数据集测试

[31]:
```
from sklearn.ensemble import AdaBoostClassifier
skl_model = AdaBoostClassifier()
skl_model.fit(X_train, y_train)
print(skl_model.score(X_test, y_test))
```

```
0.9532163742690059
```

GBDT 算法

Boosting Tree 算法

回到三个问题，Boosting Tree 的解决方案是什么?

- 弱分类器 (基分类器) 选什么? 选法是回归树 (CART，见第五章)。
- 如何调整样本分布? 样本不做修改，但每一次迭代的样本标签为真实结果和当前模型预测结果的残差。
- 如何进行组合? 组合方式即为加法模型 $f(\boldsymbol{x}) = \sum_{k=1}^{K} h_k(\boldsymbol{x})$，其中基学习器的系数为 1。如果用于回归 GBDT 的损失函数为平方损失 (`Square loss`) $\mathcal{L}(y, f(\boldsymbol{x})) = (y - f(\boldsymbol{x}))^2$，如果用于分类 GBDT 的损失函数为交叉熵 (`Cross-entropy`)。

同样分析损失函数的选择

我们先从回归问题来看，假设在第 k 次迭代时我们已有的可加模型为 $f_{k-1}(\boldsymbol{x})$。那么在新一轮迭代中最好的学习目标什么？学习真实结果和当前模型预测结果的残差 (residuals) r：

$$r(\boldsymbol{x}) = y - f(\boldsymbol{x}) \tag{6.44}$$

我们构造新一轮迭代过程中的数据集 $D = \{(\boldsymbol{x}^{(1)}, y^{(1)} - f(\boldsymbol{x}^{(1)})), \cdots, (\boldsymbol{x}^{(m)}, y^{(m)} - f(\boldsymbol{x}^{(m)}))\}, \boldsymbol{x}^{(i)} \in \mathbb{R}^n, y^{(i)} \in \mathbb{R}$。经过新一轮迭代训练得到的基学习器 $C_k(\boldsymbol{x})$ 如果是完美学习的，则等于当前的残差项 $r_{k-1}(\boldsymbol{x})$。这样一来，我们的可加模型就变为：

$$\begin{aligned} f_k(\boldsymbol{x}) &= f_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x}) \\ &= f_{k-1}(\boldsymbol{x}) + r_{k-1}(\boldsymbol{x}) \\ &= y \end{aligned} \tag{6.45}$$

这样一来，我们就可以得到完美的可加模型。那为什么选择平方损失，但平方损失有个很大的问题，对离群点 (Outliers) 呢？首先，看一下在第 k 次迭代时的损失函数 (优化目标)：

$$\begin{aligned} \mathcal{L}(y, f_k(\boldsymbol{x})) &= (y - f_{k-1}(\boldsymbol{x}) - h_k(\boldsymbol{x}))^2 \\ &= (r_{k-1}(\boldsymbol{x}) - h_k(\boldsymbol{x}))^2 \end{aligned} \tag{6.46}$$

所以，对于回归问题来说，如果是在平方损失函数的前提下，每一步确实只需要拟合当前模型的残差就可以了。至于分类问题求解，后面会进行介绍。

GBDT 算法

GBDT 是 Boosting Tree 的改进方法。回到三个问题，GBDT 的解决方案是什么？

- 弱分类器 (基分类器) 选什么？选法是回归树 (CART)。
- 如何调整样本分布？样本不做修改，但每一次迭代的样本标签为残差的负梯度。
- 如何进行组合？组合方式即为加法模型 $f(\boldsymbol{x}) = \sum_{k=1}^{K} h_k(\boldsymbol{x})$，其中基学习器的系数为 1。如果用于回归 GBDT 的损失函数为平方损失 (Square loss) $\mathcal{L}(y, f(\boldsymbol{x})) = (y - f(\boldsymbol{x}))^2$，绝对损失 (Absolute loss)，Huber 损失 (Huber loss)。如果用于分类 GBDT 的损失函数为交叉熵 (Cross-entropy)。

学习目标：残差的负梯度

先考虑第二个问题，残差的负梯度是什么？我们回到平方损失，假设在第 k 次迭代时我们已有的可加模型为 $f_{k-1}(\boldsymbol{x})$，于是会有：

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})} &= \frac{\partial (y - f_{k-1}(\boldsymbol{x}))^2}{\partial f_{k-1}(\boldsymbol{x})} \\ &= -(y - f_{k-1}(\boldsymbol{x})) \\ &= -r_k(\boldsymbol{x}) \end{aligned} \tag{6.47}$$

所以，我们在 Boosting Tree 算法中要学习的目标残差就等于负的梯度项 $-\frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})}$。所以我们可以考虑将学习目标从残差到残差的梯度项。但为什么要这么做？

现在，我们从泰勒展开的角度来分析。首先，回顾一阶泰勒展开公式：

$$f(x) \approx f(x_0) + f'(x_0)\Delta x \tag{6.48}$$

我们可以将在第 k 次迭代中已有的可加模型 $f_{k-1}(\boldsymbol{x})$ 视作 x_0，将需要学习的基学习器 $h_k(\boldsymbol{x})$ 视作 Δx。那么我们的损失函数可以写作：

$$\mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x})) \approx \mathcal{L}(y, f_{k-1}(\boldsymbol{x})) + \frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})} h_k(\boldsymbol{x}) \tag{6.49}$$

而在第 $k-1$ 次迭代后的损失为 $\mathcal{L}(y, f_{k-1}(\boldsymbol{x}))$。所以，经过第 k 次迭代后，损失的变化为 $\frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})} h_k(\boldsymbol{x})$。我们是希望损失越来越小的，而如果令 $h_k(\boldsymbol{x}) = -\frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})}$，则损失变化量为 $-(\frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})})^2$，这样一定是使损失递减。也就是说损失会向着下降的方向移动，这也是梯度下降的思想。它和 Boosting Tree 中使用的残差的区别在于，通过残差是在寻找全局最优值 (每一步都试图让结果达到最好)，而使用残差的梯度是在搜寻局部最优值 (每一步都试图让结果更好一些)。既然如此，使用残差的梯度优势在哪？这就要考虑损失函数了。

损失函数的使用

在前面的描述中，我们对回归问题默认的损失函数是平方损失，但平方损失有个很大的问题，对异常点 (Outliers) 很是敏感。比如下面这个例子，可以看出最后一个样本 (异常点) 会在下一次迭代中占据主要影响，下一个基学习器会过多关注于最后一个的异常点。而 Boosting Tree 使用平方损失正是因为这个损失可以帮助获得残差，可如果换个对异常点鲁棒的损失函数呢？这个时候 Boosting Tree 就无计可施了。

y	0.5	1.2	2	5
$f(\boldsymbol{x})$	0.6	1.4	1.5	1.7
$\mathcal{L} = (y - f(\boldsymbol{x}))^2$	0.005	0.02	0.125	5.445

但 GBDT 却可以做到。如果我们将损失函数换做绝对损失：

$$\mathcal{L}(y, f(\boldsymbol{x})) = |y - f(\boldsymbol{x})| \tag{6.50}$$

其负梯度为：$-\frac{\partial \mathcal{L}}{\partial f(\boldsymbol{x})} = sgn(y - f(\boldsymbol{x}))$。

或者我们将其换做 Huber 损失：

$$\mathcal{L}(y, f(\boldsymbol{x})) = \begin{cases} y - f(\boldsymbol{x}), & |y - f(\boldsymbol{x})| \leq \delta \\ \delta sgn(y - f(\boldsymbol{x})), & |y - f(\boldsymbol{x})| > \delta \end{cases} \tag{6.51}$$

其负梯度为：$-\frac{\partial \mathcal{L}}{\partial f(\boldsymbol{x})} = \begin{cases} \frac{1}{2}(y - f(\boldsymbol{x}))^2, & |y - f(\boldsymbol{x})| \leq \delta \\ \delta(|y - f(\boldsymbol{x})| - \frac{\delta}{2}), & |y - f(\boldsymbol{x})| > \delta \end{cases}$。

我们可以看一下采用绝对损失和 Huber 损失时的示例：

y	0.5	1.2	2	5
$f(\boldsymbol{x})$	0.6	1.4	1.5	1.7
Square loss	0.005	0.02	0.125	5.445
Absolute loss	0.1	0.2	0.5	3.3
Huber loss($\delta = 0.5$)	0.005	0.02	0.125	1.525

可以看到后两种损失对异常点要鲁棒些，当然使用这两种损失后梯度就不等于残差，但梯度仍可以作为一种近似。

Shrinkage 收缩

Shrinkage 收缩指，每次走一小步逐渐逼近结果的效果，要比每次迈一大步很快逼近结果的方式更容易得到精确值。就是说它不完全信任每一棵残差树，认为每棵树只学到了真实的一部分，于是累加的时候只累加一小部分再多学几棵树来弥补不足。这个技巧类似于梯度下降里的学习率。

实现 Shrinkage 收缩的一种简略方法是固定每一个基学习器的步长 (学习率) η：$f_k(\boldsymbol{x}) = f_{k-1}(\boldsymbol{x}) + \eta h_k(\boldsymbol{x})$。这个形式就回到了最初前向分步加法模型的描述，只是这里的基学习器的参数预先给定。

另一种方法是 Line Search，去寻找最优的步长：

$$\eta = \arg\min_{\eta} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, f_{k-1}(\boldsymbol{x}^{(i)}) + \eta h_k(\boldsymbol{x}^{(i)})) \tag{6.52}$$

当损失函数为平方误差时，显然有：

$$\begin{aligned} \eta &= \arg\min_{\eta} \sum_{i=1}^{m} \left((y^{(i)} - f_{k-1}(\boldsymbol{x}^{(i)})) - \eta h_k(\boldsymbol{x}^{(i)}) \right) \\ &= \arg\min_{\eta} \sum_{i=1}^{m} \left(-2\eta(y^{(i)} - f_{k-1}(\boldsymbol{x}^{(i)})) + (\eta h_k(\boldsymbol{x}^{(i)}))^2 \right) \end{aligned} \tag{6.53}$$

对其求导并令导数为 0，可以得到此时的最优 η：

$$\eta^* = \frac{\sum_{i=1}^{m} 2(y^{(i)} - f_{k-1}(\boldsymbol{x}^{(i)}))h_k(\boldsymbol{x}^{(i)})}{\sum_{i=1}^{m} h_k^2(\boldsymbol{x}^{(i)})} \tag{6.54}$$

分类问题下的损失函数

以上我们讨论完了回归问题，现在我们讨论分类问题。首先是**二分类问题**，在第五章我们介绍过概率监督学习 (逻辑回归)，逻辑回归实质上是用线性模型去拟合对数几率 (`log odds`)，$\log\left(\frac{p}{1-p}\right)$。如果说回归问题是用线性模型 (线性可加模型) 直接学习目标结果，那分类问题就是用线性模型 (线性可加模型) 去学习对数几率。所以，分类模型可以表达为：

$$\hat{y} = P(y = 1 \mid \boldsymbol{x}) = \frac{1}{1 + e^{-\sum\limits_{k} h_k(\boldsymbol{x})}} = \frac{1}{1 + e^{-f(\boldsymbol{x})}} \tag{6.55}$$

其中 $\hat{y}$ 表示分类模型的预测。所以，损失函数就可以表达为交叉熵：

$$\mathcal{L}(y, f(\boldsymbol{x})) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \tag{6.56}$$

在第六章我们介绍过二分类交叉熵的求导，所以在这里我们有：

$$-\frac{\partial \mathcal{L}(y, f(\boldsymbol{x}))}{\partial f(\boldsymbol{x})} = y - \hat{y} \tag{6.57}$$

可以看到，与回归问题类似，下一棵决策树的训练样本为：$\{(\boldsymbol{x}, y - \hat{y})\}$，其所需要拟合的残差为真实标签与预测概率之差。

再看一下多分类问题，假设一个有 C 个类别，每一次迭代的训练实际上是训练了 C 棵树去拟合每一个类别。如果一共再进行 K 次迭代的话，那么训练完之后总共有 $C \times K$ 棵树。

$$\begin{aligned} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x})) &= -\boldsymbol{y} \log \hat{\boldsymbol{y}} \\ &= -\sum_{c=1}^{C} y_c \log \hat{y}_c \end{aligned} \tag{6.58}$$

其中 $\hat{y}_c = \frac{e^{-f^c(\boldsymbol{x})}}{\sum_{l=1}^{C} e^{-f^l(\boldsymbol{x})}}$。同样在第六章我们介绍过多分类交叉熵的求导：

$$-\frac{\partial \mathcal{L}(y, f^c(\boldsymbol{x}))}{\partial f^c(\boldsymbol{x})} = y_c - \hat{y}_c \tag{6.59}$$

下一批基学习器 (C 棵决策树) 的训练样本为：$\{(\boldsymbol{x}, \boldsymbol{y} - \hat{\boldsymbol{y}})\}$，其所需要拟合的残差为真实标签与预测概率之差。

自定义实现

[32]:
```
from chapter5 import RegressionTree
```

[33]:
```
class Loss(ABC):

    def __init__(self):
        super().__init__()

    @abstractmethod
    def loss(self, y_true, y_pred):
        return NotImplementedError()

    @abstractmethod
    def grad(self, y, y_pred):
        raise NotImplementedError()

class SquareLoss(Loss):

    def __init__(self):
        pass

    def loss(self, y, y_pred):
        pass

    def grad(self, y, y_pred):
        return -(y - y_pred)

    def hess(self, y, y_pred):
        return 1

class CrossEntropyLoss(Loss):

    def __init__(self):
        pass

    def loss(self, y, y_pred):
        pass

    def grad(self, y, y_pred):
        return - (y - y_pred)

    def hess(self, y, y_pred):
        return y_pred * (1-y_pred)
```

[34]:
```
def softmax(x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return e_x / e_x.sum(axis=-1, keepdims=True)


def line_search(self, y, y_pred, h_pred):
    Lp = 2 * np.sum((y - y_pred) * h_pred)
    Lpp = np.sum(h_pred * h_pred)
    return 1 if np.sum(Lpp) == 0 else Lp / Lpp
```

```
def to_categorical(x, n_classes=None):
    """
    One-hot 编码
    """
    if not n_classes:
        n_classes = np.amax(x) + 1
    one_hot = np.zeros((x.shape[0], n_classes))
    one_hot[np.arange(x.shape[0]), x] = 1
    return one_hot


class GradientBoostingDecisionTree(object):
    """
    GBDT 算法。用一组基学习器 (回归树) 学习损失函数的梯度。
    """
    def __init__(self, n_estimators, learning_rate=1, min_samples_split=2,
                 min_impurity=1e-7, max_depth=float("inf"), is_regression=False, line_search=False):
        self.n_estimators = n_estimators          # 迭代的次数
        self.learning_rate = learning_rate        # 训练过程中沿着负梯度走的步长，也就是学习率
        self.min_samples_split = min_samples_split    # 分割所需的最小样本数
        self.min_impurity = min_impurity          # 分割所需的最小纯度
        self.max_depth = max_depth                # 树的最大深度
        self.is_regression = is_regression        # 分类问题或回归问题
        self.line_search = line_search            # 是否使用 line search
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)
        # 回归问题采用基础的平方损失，分类问题采用交叉熵损失
        self.loss = SquareLoss()
        if not self.is_regression:
            self.loss = CrossEntropyLoss()

    def fit(self, X, Y):
        # 分类问题将 Y 转化为 one-hot 编码
        if not self.is_regression:
            Y = to_categorical(Y.flatten())
        else:
            Y = Y.reshape(-1, 1) if len(Y.shape) == 1 else Y
        self.out_dims = Y.shape[1]
        self.trees = np.empty((self.n_estimators, self.out_dims), dtype=object)
        Y_pred = np.full(np.shape(Y), np.mean(Y, axis=0))
        self.weights = np.ones((self.n_estimators, self.out_dims))
        self.weights[1:, :] *= self.learning_rate
        # 迭代过程
        for i in self.progressbar(range(self.n_estimators)):
            for c in range(self.out_dims):
                tree = RegressionTree(
                        min_samples_split=self.min_samples_split,
                        min_impurity=self.min_impurity,
                        max_depth=self.max_depth)
                # 计算损失的梯度，并用梯度进行训练
                if not self.is_regression:
                    Y_hat = softmax(Y_pred)
                    y, y_pred = Y[:, c], Y_hat[:, c]
                else:
                    y, y_pred = Y[:, c], Y_pred[:, c]
                neg_grad = -1 * self.loss.grad(y, y_pred)
                tree.fit(X, neg_grad)
                # 用新的基学习器进行预测
                h_pred = tree.predict(X)
```

```
            # line search
            if self.line_search == True:
                self.weights[i, c] *= line_search(y, y_pred, h_pred)
            # 加法模型中添加基学习器的预测，得到最新迭代下的加法模型预测
            Y_pred[:, c] += np.multiply(self.weights[i, c], h_pred)
            self.trees[i, c] = tree

    def predict(self, X):
        Y_pred = np.zeros((X.shape[0], self.out_dims))
        # 生成预测
        for c in range(self.out_dims):
            y_pred = np.array([])
            for i in range(self.n_estimators):
                update = np.multiply(self.weights[i, c], self.trees[i, c].predict(X))
                y_pred = update if not y_pred.any() else y_pred + update
            Y_pred[:, c] = y_pred
        if not self.is_regression:
            # 分类问题输出最可能类别
            Y_pred = Y_pred.argmax(axis=1)
        return Y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy


class GradientBoostingRegressor(GradientBoostingDecisionTree):

    def __init__(self, n_estimators=200, learning_rate=1, min_samples_split=2,
                 min_impurity=1e-7, max_depth=float("inf"), is_regression=True, line_search=False):
        super(GradientBoostingRegressor, self).__init__(n_estimators=n_estimators,
            learning_rate=learning_rate,
            min_samples_split=min_samples_split,
            min_impurity=min_impurity,
            max_depth=max_depth,
            is_regression=is_regression,
            line_search=line_search)


class GradientBoostingClassifier(GradientBoostingDecisionTree):

    def __init__(self, n_estimators=200, learning_rate=1, min_samples_split=2,
                 min_impurity=1e-7, max_depth=float("inf"), is_regression=False, line_search=False):
        super(GradientBoostingClassifier, self).__init__(n_estimators=n_estimators,
            learning_rate=learning_rate,
            min_samples_split=min_samples_split,
            min_impurity=min_impurity,
            max_depth=max_depth,
            is_regression=is_regression,
            line_search=line_search)
```

用自定义的 **GBDT**，乳腺癌数据集测试

[35]:
```
column_names = ['Sample code number','Clump Thickness',
                'Uniformity of Cell Size','Uniformity of Cell Shape',
                'Marginal Adhesion','Single Epithelial Cell Size',
                'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
data = pd.read_csv('../data/cancer/breast-cancer-wisconsin.data',names=column_names)
```

```
data = data.replace(to_replace='?',value=np.nan)  # 非法字符替代
data = data.dropna(how='any')  # 去掉空值，any；出现空值行则删除
print(data.shape)
# 随机采样 25% 的数据用于测试，剩下 75% 用于构建训练集
X_train,X_test,y_train,y_test = train_test_split(data[column_names[1:10]], data[column_names[10]],
                                                 test_size=0.25, random_state=1111)
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 修改标签为 0 和 1
print(y_train.shape)
y_train[y_train==2] = 0
y_train[y_train==4] = 1
y_test[y_test==2] = 0
y_test[y_test==4] = 1
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 数据标准化预处理，保证每个维度特征均值为 0，方差为 1
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
y_train = y_train.as_matrix()
y_test = y_test.as_matrix()
```

```
(683, 11)
2    328
4    184
Name: Class, dtype: int64
(512,)
0    328
1    184
Name: Class, dtype: int64
```

[36]:
```
model = GradientBoostingClassifier(n_estimators=20)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
Training: 100% [----------------------------------------------] Time: 0:00:12

0.9532163742690059
```

用 sklearn 的 GBDT，乳腺癌数据集测试

[37]:
```
from sklearn.ensemble import GradientBoostingClassifier
skl_model= GradientBoostingClassifier()
skl_model.fit(X_train, y_train)
print(skl_model.score(X_test, y_test))
```

```
0.9473684210526315
```

XGBoost 算法

XGBoost 是 GBDT 的改进。再次回到三个问题，XGBoost 的解决方案是什么?

- 弱分类器 (基分类器) 选什么? 选法是 XGBoost 回归树 (见后文)。
- 如何调整样本分布? 不做更改。
- 如何进行组合? 组合方式即为加法模型 $f(\boldsymbol{x}) = \sum_{k=1}^{K} h_k(\boldsymbol{x})$，其中基学习器的系数为 1。损失函数同 GBDT。但是，在 XGBoost 中，显式地将树模型的复杂度作为正则项加在优化目标里。

XGBoost 回归树的学习策略

XGBoost 的损失函数中显示地添加了树模型的复杂度作为正则项 $\Omega(h_k)$：

$$\Omega(h_k) = \gamma T + \frac{\lambda}{2}\sum_{j=1}^{T} w_j^2 \tag{6.60}$$

其中 T 是基学习器叶子节点的数目，w_j 是每个叶子节点的权重。对叶子节点个数进行惩罚，相当于在训练过程中做了剪枝。对于样本 $\boldsymbol{x}$，我们用 $q(\boldsymbol{x})$ 表示将样本 $\boldsymbol{x}$ 分到了某个叶节点上，于是 $w_{q(\boldsymbol{x})}$ 表示回归树对样本的预测值。因此基学习器 $h_k = w_{q(\boldsymbol{x})}$。考虑正则项条件下，目标函数为：

$$J(y, f(\boldsymbol{x})) = \mathcal{L}(y, f(\boldsymbol{x})) + \sum_{k=1}^{K}\Omega(h_k) \tag{6.61}$$

然后我们再考虑学习目标，第 k 次迭代时：

$$\begin{aligned} J(y, f(\boldsymbol{x})) &= \mathcal{L}(y, f_{k-1}(\boldsymbol{x}) + h_k(\boldsymbol{x})) + \sum_{j=1}^{k-1}\Omega(h_j) + \Omega(h_k) \\ &\approx \mathcal{L}(y, f_{k-1}(\boldsymbol{x})) + \frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})}h_k(\boldsymbol{x}) + \frac{\partial^2 \mathcal{L}}{2\partial f_{k-1}^2(\boldsymbol{x})}h_k^2(\boldsymbol{x}) + \sum_{j=1}^{k-1}\Omega(h_j) + \Omega(h_k) \\ &= \mathcal{L}(y, f_{k-1}(\boldsymbol{x})) + \frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x})}h_k(\boldsymbol{x}) + \frac{\partial^2 \mathcal{L}}{2\partial f_{k-1}^2(\boldsymbol{x})}h_k^2(\boldsymbol{x}) + \Omega(h_k) + Const \end{aligned} \tag{6.62}$$

其中，从第一步到第二步用到了泰勒二阶展开：

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0) \tag{6.63}$$

简化一下书写，我们令第 i 个样本的：

$$\begin{aligned} g_i' &:= \frac{\partial \mathcal{L}}{\partial f_{k-1}(\boldsymbol{x}_i)} \\ h_i' &:= \frac{\partial^2 \mathcal{L}}{2\partial f_{k-1}^2(\boldsymbol{x}_i)} \end{aligned} \tag{6.64}$$

于是，损失函数为 (把样本编号信息也标注上)：

$$\begin{aligned} J(y, f(\boldsymbol{x})) &= \sum_{i=1}^{m}\left(\mathcal{L}(y, f_{k-1}(\boldsymbol{x})) + g_i' h_k(\boldsymbol{x}_i) + \frac{1}{2}h_i' h_k^2(\boldsymbol{x}_i)\right) + \Omega(h_k(\boldsymbol{x})) + Constant \\ &= \sum_{i=1}^{m}\left(\mathcal{L}(y, f_{k-1}(\boldsymbol{x})) + g_i' w_{q(\boldsymbol{x}_i)} + \frac{1}{2}h_i' w_{q(\boldsymbol{x}_i)}^2\right) + \gamma T + \frac{\lambda}{2}\sum_{j=1}^{T} w_j^2 + Constant \\ &= \sum_{i=1}^{m}\left(g_i' w_{q(\boldsymbol{x}_i)} + \frac{1}{2}h_i' w_{q(\boldsymbol{x}_i)}^2\right) + \gamma T + \frac{\lambda}{2}\sum_{j=1}^{T} w_j^2 + Constant' \end{aligned} \tag{6.65}$$

红色项是对样本的累加，蓝色项是对叶节点的累加。怎么统一起来？定义每个叶节点 j 上的样本集合为：$I_j = \{i \mid q(\boldsymbol{x}_i) = j\}$。则目标函数可以写成按叶节点累加的形式 (优化问题中常数项不影响结果，我们不再考虑)：

$$\begin{aligned} J(y, f(\boldsymbol{x})) &= \sum_{j=1}^{T}\left[(\sum_{i\in I_j} g_i')w_j + \frac{1}{2}(\sum_{i\in I_j} h_i' + \lambda)w_j^2\right] + \gamma T \\ &= \sum_{j=1}^{T}\left[(G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2\right] + \gamma T \end{aligned} \tag{6.66}$$

如果确定了树的结构 (即 $q(\boldsymbol{x})$ 确定)，为了使目标函数最小，可以令其导数为 0，解得每个叶节点的最优预测分数为：

$$w_j^* = -\frac{G_j}{H_j + \lambda}, \quad j = 1, 2, \cdots, T \tag{6.67}$$

代入目标函数，得到最小损失为：

$$J^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T \tag{6.68}$$

当回归树的结构确定时，我们前面已经推导出其最优的叶节点分数以及对应的最小损失值，问题是怎么确定树的结构？也就是说我们的基学习器是什么样的？一种是暴力枚举所有可能的树结构，选择损失值最小的，但这是 NP 难问题。另一种方法是贪心法，每次尝试分裂一个叶节点，计算分裂前后的增益，选择增益最大的 (信息增益见第五章)。

$$J^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T \tag{6.69}$$

标红部分衡量了每个叶子节点对总体损失的的贡献，我们希望损失越小越好，则标红部分的值越大越好。

因此，对一个叶子节点进行分裂，分裂前后的增益定义为：

$$\begin{aligned} Gain &= J^* - (J_L^* + J_R^*) \\ &= \left(-\frac{1}{2}\frac{(G_L + G_R)^2}{H_L + H_G + \lambda} + \gamma\right) - \left(-\frac{1}{2}\frac{G_L^2}{H_L + \lambda} + \gamma\right) - \left(-\frac{1}{2}\frac{G_R^2}{H_R + \lambda} + \gamma\right) \\ &= \frac{1}{2}\left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_G + \lambda}\right) - \gamma \end{aligned} \tag{6.70}$$

这个结果就可以用于在实践中评估候选分裂节点是不是应该分裂的划分依据，我们尽量找到使之最大的特征值分裂点。

在树学习中，另外一个关键问题是如何找到每一个特征上的分裂点。直觉的想法是枚举特征上所有可能的分裂点，然后计算上述的增益。这种算法称为 Exact Greedy Algorithm。当然为了有效率的找到最佳分裂节点，算法可以先将该特征的所有取值进行排序，之后按顺序取分裂值计算。但是当数据量很大时，数据不可能一次性的全部读入到内存中，或者在分布式计算中，也不可能事先对所有值进行排序，且无法使用所有数据来计算分裂节点之后的树结构的增益分数。

XGBoost 回归树的节点分裂算法

正如上面所说，尽管我们找到了寻找最佳分裂点的指标，但使用贪婪算法逐个特征值计算的计算量过高。为解决这个问题，有几种解决方案。

一种是**近似算法** (`Approximate Algo for Split Finding`)，近似算法首先按照特征取值中统计分布的百分位点确定一些候选分裂点，然后算法将连续的值映射到桶 (`buckets`) 中，接着汇总统计数据，并根据聚合统计数据在候选节点中找到最佳节点。XGBoost 采用的近似算法对于每个特征，只考察分位点，减少复杂度。例如，以三分位点举例：

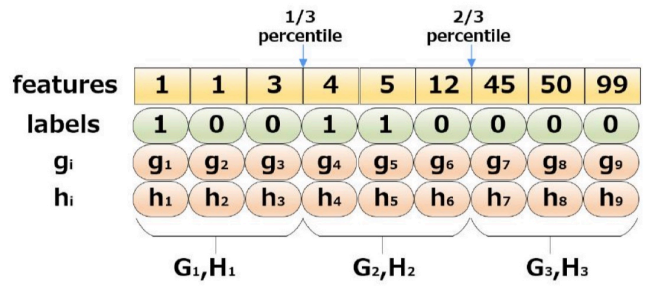


图 6.7. 只考察分位点的近似算法。

于是，我们找到其中最大的信息增益的划分方法：

$$Gain = \max\left(Gain, \frac{1}{2}\left(\frac{G_1^2}{H_1+\lambda}+\frac{G_{23}^2}{H_{23}+\lambda}-\frac{(G_{123})^2}{H_{123}+\lambda}\right)-\gamma, \frac{1}{2}\left(\frac{G_{12}^2}{H_{12}+\lambda}+\frac{G_3^2}{H_3+\lambda}-\frac{(G_{123})^2}{H_{123}+\lambda}\right)-\gamma\right) \tag{6.71}$$

然而，这种划分分位点的方法在实际中可能效果不是很好。

另一种是加权分位数 (`Weighted Quantile Sketch`)，我们需先构造一个集合：$\mathcal{D}_j = \left\{\left(x_{ij}, h'_{ij}\right)_{i=1,\ldots m}\right\}$，其中 x_{ij} 表示第 i 个样本的第 j 个特征值，h'_{ij} 是第 i 个样本的第 j 个特征的二阶梯度统计。我们现在定义一个排序函数 r_j：

$$r_j(z) = \frac{1}{\sum_{(x,h')\in\mathcal{D}_j} h'} \sum_{(x,h')\in\mathcal{D}_j, x<z} h' \tag{6.72}$$

表示特征 j 所有可取值中小于 z 的特征值的总权重占总的所有可取值的总权重和的比例，用 h' 加权。目标就是寻找候选分裂点集 $\{s_{j1}, s_{j2}, \ldots s_{jl}\}$，有

$$\left|r_j\left(s_{j,t}\right) - r_j\left(s_{j,t+1}\right)\right| < \epsilon, \quad s_{j1} = \min_i x_{ij}, s_{jl} = \max_i x_{ij} \tag{6.73}$$

ϵ 是近似因子或者说是扫描步幅，按照步幅 ϵ 挑选出特征 j 的取值候选点，组成候选点集，这意味着有大概 $\frac{1}{\epsilon}$ 个候选点。但是，我们为什么要用 h' 加权呢？我们把目标函数整理成以下形式：

$$\begin{aligned} J(y, f(\boldsymbol{x})) &\simeq \sum_{i=1}^{m}\left[g'_i h_k(\boldsymbol{x}_i) + \frac{1}{2}h'_i h_k^2(\boldsymbol{x}_i)\right] + \Omega\left(h_k\right) + Constant \\ &= \sum_{i=1}^{m}\left[g'_i h_k(\boldsymbol{x}_i) + \frac{1}{2}h'_i h_k^2(\boldsymbol{x}_i) + \underbrace{\frac{1}{2}\frac{g_i'^2}{h'_i}}_{\text{添加常数项}}\right] + \Omega\left(h_k\right) + Constant' \\ &= \sum_{i=1}^{m}\frac{1}{2}h'_i\left[h_k(\boldsymbol{x}_i) - \left(-\frac{g'_i}{h'_i}\right)\right]^2 + \Omega\left(h_k\right) + Constant' \end{aligned} \tag{6.74}$$

最后的代价函数就是一个加权平方误差，权值为 h'_i，标签为 $-\frac{g'_i}{h'_i}$，所以可以将特征 j 的取值权重看成对应的 h'_i。

XGBoost 的损失函数

最后，如何定义 $\mathcal{L}(y, f(\boldsymbol{x}))$？如果是回归模型，可以采用平方损失；如果是分类模型，可以采用交叉熵损失。具体原理同 GBDT。有了 $\mathcal{L}(y, f(\boldsymbol{x}))$ 的具体形式，我们便可以得到 H 和 G。(注：平方损失和交叉熵损失的 G 和 H 的推导可以参考第六章，在第六章我们推导了一阶导数的获得，而二阶导数只需要在一阶导数上再简单求导一次便可。)

XGBoost 的其它设置

除了上述描述的以外，XGBoost 在实现过程中，还使用了包括：

1. **缺失值处理**。当有缺失值时，系统将样本分到默认方向的叶子节点。每个分支都有两个默认方向，最佳的默认方向可以从训练数据中学习到。
2. **系统优化设计**。包括分块并行、缓存优化等。关于并行问题我们会在十二章介绍。
3. **列抽样**。这里借鉴了随机森林的做法，支持对特征采样，不仅能降低过拟合，还能减少计算。
4. Shrinkage。相当于学习率，这里同 GBDT 的做法，主要是为了削弱每棵树的影响，让后面有更大的学习空间。

[38]:
```
from chapter5 import DecisionTree
```

[39]:
```
class XGBoostRegressionTree(DecisionTree):
    """
    XGBoost 回归树。此处基于第五章介绍的决策树，故采用贪心算法找到特征上分裂点 (枚举特征上所有可能的分裂点)。
    """
    def __init__(self, min_samples_split=2, min_impurity=1e-7,
                 max_depth=float("inf"), loss=None, gamma=0., lambd=0.):
        super(XGBoostRegressionTree, self).__init__(min_impurity=min_impurity,
            min_samples_split=min_samples_split,
            max_depth=max_depth)
        self.gamma = gamma    # 叶子节点的数目的惩罚系数
        self.lambd = lambd    # 叶子节点的权重的惩罚系数
        self.loss = loss      # 损失函数

    def _split(self, y):
        # y 包含 y_true 在左半列，y_pred 在右半列
        col = int(np.shape(y)[1]/2)
        y, y_pred = y[:, :col], y[:, col:]
        return y, y_pred

    def _gain(self, y, y_pred):
        # 计算信息
        nominator = np.power((y * self.loss.grad(y, y_pred)).sum(), 2)
        denominator = self.loss.hess(y, y_pred).sum()
        return nominator / (denominator + self.lambd)

    def _gain_by_taylor(self, y, y1, y2):
        # 分割为左子树和右子树
        y, y_pred = self._split(y)
        y1, y1_pred = self._split(y1)
        y2, y2_pred = self._split(y2)
        true_gain = self._gain(y1, y1_pred)
        false_gain = self._gain(y2, y2_pred)
        gain = self._gain(y, y_pred)
        # 计算信息增益
        return 0.5 * (true_gain + false_gain - gain) - self.gamma

    def _approximate_update(self, y):
        y, y_pred = self._split(y)
        # 计算叶节点权重
        gradient = self.loss.grad(y, y_pred).sum()
        hessian = self.loss.hess(y, y_pred).sum()
        leaf_approximation = -gradient / (hessian + self.lambd)
        return leaf_approximation

    def fit(self, X, y):
        self._impurity_calculation = self._gain_by_taylor
        self._leaf_value_calculation = self._approximate_update
        super(XGBoostRegressionTree, self).fit(X, y)


class XGBoost(object):
    """
    XGBoost 学习器。
    """
    def __init__(self, n_estimators=200, learning_rate=0.001, min_samples_split=2,
                 min_impurity=1e-7, max_depth=2, is_regression=False, gamma=0., lambd=0.):
        self.n_estimators = n_estimators            # 树的数目
```

```
        self.learning_rate = learning_rate          # 训练过程中沿着负梯度走的步长，也就是学习率
        self.min_samples_split = min_samples_split  # 分割所需的最小样本数
        self.min_impurity = min_impurity            # 分割所需的最小纯度
        self.max_depth = max_depth                  # 树的最大深度
        self.gamma = gamma                          # 叶子节点的数目的惩罚系数
        self.lambd = lambd                          # 叶子节点的权重的惩罚系数
        self.is_regression = is_regression          # 分类或回归问题
        self.progressbar = progressbar.ProgressBar(widgets=bar_widgets)
        # 回归问题采用基础的平方损失，分类问题采用交叉熵损失
        self.loss = SquareLoss()
        if not self.is_regression:
            self.loss = CrossEntropyLoss()

    def fit(self, X, Y):
        # 分类问题将 Y 转化为 one-hot 编码
        if not self.is_regression:
            Y = to_categorical(Y.flatten())
        else:
            Y = Y.reshape(-1, 1) if len(Y.shape) == 1 else Y
        self.out_dims = Y.shape[1]
        self.trees = np.empty((self.n_estimators, self.out_dims), dtype=object)
        Y_pred = np.zeros(np.shape(Y))
        self.weights = np.ones((self.n_estimators, self.out_dims))
        self.weights[1:, :] *= self.learning_rate
        # 迭代过程
        for i in self.progressbar(range(self.n_estimators)):
            for c in range(self.out_dims):
                tree = XGBoostRegressionTree(
                        min_samples_split=self.min_samples_split,
                        min_impurity=self.min_impurity,
                        max_depth=self.max_depth,
                        loss=self.loss,
                        gamma=self.gamma,
                        lambd=self.lambd)
                # 计算损失的梯度，并用梯度进行训练
                if not self.is_regression:
                    Y_hat = softmax(Y_pred)
                    y, y_pred = Y[:, c], Y_hat[:, c]
                else:
                    y, y_pred = Y[:, c], Y_pred[:, c]
                    y, y_pred = y.reshape(-1, 1), y_pred.reshape(-1, 1)
                    y_and_ypred = np.concatenate((y, y_pred), axis=1)
                    tree.fit(X, y_and_ypred)
                    # 用新的基学习器进行预测
                    h_pred = tree.predict(X)
                    # 加法模型中添加基学习器的预测，得到最新迭代下的加法模型预测
                    Y_pred[:, c] += np.multiply(self.weights[i, c], h_pred)
                    self.trees[i, c] = tree

    def predict(self, X):
        Y_pred = np.zeros((X.shape[0], self.out_dims))
        # 生成预测
        for c in range(self.out_dims):
            y_pred = np.array([])
            for i in range(self.n_estimators):
                update = np.multiply(self.weights[i, c], self.trees[i, c].predict(X))
                y_pred = update if not y_pred.any() else y_pred + update
            Y_pred[:, c] = y_pred
```

```
        if not self.is_regression:
            # 分类问题输出最可能类别
            Y_pred = Y_pred.argmax(axis=1)
        return Y_pred

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = np.sum(y == y_pred, axis=0) / len(y)
        return accuracy


class XGBRegressor(XGBoost):

    def __init__(self, n_estimators=200, learning_rate=1, min_samples_split=2,
                 min_impurity=1e-7, max_depth=float("inf"), is_regression=True,
                 gamma=0., lambd=0.):
        super(XGBRegressor, self).__init__(n_estimators=n_estimators,
            learning_rate=learning_rate,
            min_samples_split=min_samples_split,
            min_impurity=min_impurity,
            max_depth=max_depth,
            is_regression=is_regression,
            gamma=gamma,
            lambd=lambd)


class XGBClassifier(XGBoost):

    def __init__(self, n_estimators=200, learning_rate=1, min_samples_split=2,
                 min_impurity=1e-7, max_depth=float("inf"), is_regression=False,
                 gamma=0., lambd=0.):
        super(XGBClassifier, self).__init__(n_estimators=n_estimators,
            learning_rate=learning_rate,
            min_samples_split=min_samples_split,
            min_impurity=min_impurity,
            max_depth=max_depth,
            is_regression=is_regression,
            gamma=gamma,
            lambd=lambd)
```

用自定义的 XGBoost，乳腺癌数据集测试

[40]:
```
column_names = ['Sample code number','Clump Thickness',
                'Uniformity of Cell Size','Uniformity of Cell Shape',
                'Marginal Adhesion','Single Epithelial Cell Size',
                'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
data = pd.read_csv('../data/cancer/breast-cancer-wisconsin.data',names=column_names)
data = data.replace(to_replace='?',value=np.nan)  # 非法字符替代
data = data.dropna(how='any')  # 去掉空值，any；出现空值行则删除
print(data.shape)
# 随机采样 25% 的数据用于测试，剩下 75% 用于构建训练集
X_train,X_test,y_train,y_test = train_test_split(data[column_names[1:10]], data[column_names[10]],
                                                 test_size=0.25, random_state=1111)
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 修改标签为 0 和 1
print(y_train.shape)
y_train[y_train==2] = 0
y_train[y_train==4] = 1
```

```
y_test[y_test==2] = 0
y_test[y_test==4] = 1
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 数据标准化预处理，保证每个维度特征均值为 0，方差为 1
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
y_train = y_train.as_matrix()
y_test = y_test.as_matrix()
```

```
(683, 11)
2    328
4    184
Name: Class, dtype: int64
(512,)
0    328
1    184
Name: Class, dtype: int64
```

[41]:
```
model = XGBClassifier(n_estimators=20)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
Training: 100% [----------------------------------------------] Time: 0:00:06

0.9590643274853801
```

[42]:
```
import numpy
import PIL
import matplotlib
import re
import pandas
import progressbar
import sklearn

print("numpy:", numpy.__version__)
print("PIL:", PIL.__version__)
print("matplotlib:", matplotlib.__version__)
print("re:", re.__version__)
print("pandas:", pandas.__version__)
print("progressbar:", progressbar.__version__)
print("sklearn:", sklearn.__version__)
```

```
numpy: 1.14.5
PIL: 6.2.1
matplotlib: 3.1.1
re: 2.2.1
pandas: 0.25.1
progressbar: 2.5
sklearn: 0.21.3
```

第七章　深度模型中的优化

7.1　基本优化算法

整体框架

7.1.1　梯度

梯度下降

- 微积分中使用**梯度**表示函数增长最快的方向，因此，神经网络中使用**负梯度**来指示目标函数下降最快的方向。
  - **梯度**实际上是损失函数对网络中每个参数的**偏导**所组成的向量；
  - **梯度**仅仅指示了对于每个参数各自增长最快的方向，因此，梯度无法保证**全局方向**就是函数为了达到最小值应该前进的方向；
  - 使用**梯度**的具体计算方法即**反向传播**。
- **梯度下降**是一种优化算法，通过**迭代**的方式寻找使模型**目标函数**达到**最小值**时的最优参数 (也称**最速下降法**)：
  - 当目标函数是**凸函数**时，梯度下降的解是全局最优解，但在一般情况下，**梯度下降无法保证全局最优**；
  - 梯度下降最常用的形式是批量梯度下降法 (`Batch Gradient Descent`，BGD)，其做法是在更新参数时使用所有的样本来进行更新。
- 反过来，如果求解**目标函数**达到**最大值**时的最优参数，就需要用**梯度上升法**进行迭代。
- **负梯度**中的每一项可以认为传达了**两个信息**：
  - 正负号在告诉输入向量应该调大还是调小 (正调大，负调小)；
  - 每一项的相对大小表明每个参数对函数值达到最值的**影响程度**。

随机梯度下降

首先，梯度下降法沿着梯度的反方向进行搜索，利用了函数的一阶导数信息。基本的梯度下降法每次使用**所有训练样本**的**平均损失**来更新参数。因此，经典的梯度下降在每次对模型参数进行更新时，需要遍历所有数据。当训练样本的数量很大时，这需要消耗相当大的计算资源，在实际应用中基本不可行。

而**随机梯度下降** (`Stochastic Gradient Descent`，SGD) 是随机抽取一批样本 (Batch)，以此为根据来更新参数。随机梯度下降每次使用 m 个样本的损失来近似平均损失，更新规则：

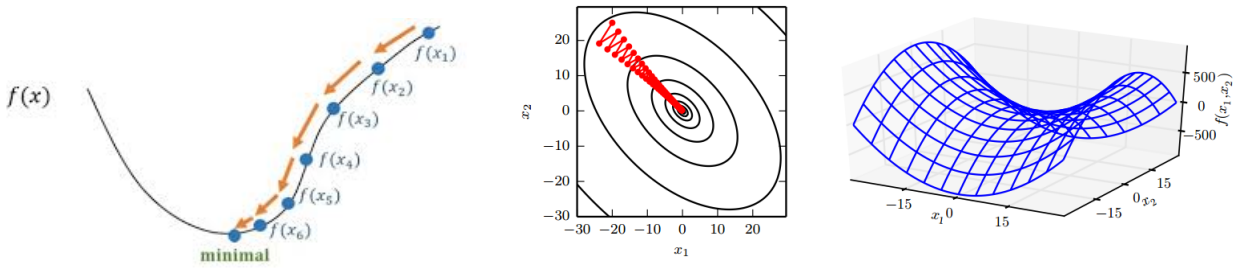
$$
\begin{aligned}
g &\leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\boldsymbol{x}^{(i)};\theta), y^{(i)}) \\
\theta &\leftarrow \theta - \epsilon g
\end{aligned}
\tag{7.1}
$$

随机梯度下降法存在的问题：

- 随机梯度下降 (SGD) 放弃了**梯度的准确性**，仅采用一部分样本来估计当前的梯度。因此 SGD 对梯度的估计常常出现偏差，造成目标函数收敛不稳定，甚至不收敛的情况。
- 无论是经典的梯度下降还是随机梯度下降，都可能陷入**局部极值点**。除此之外，SGD 还可能遇到 “**峡谷**” 和 “**鞍点**” 两种情况。
  - **峡谷**类似一个带有**坡度**的狭长小道，左右两侧是 “**峭壁**”；在**峡谷**中，准确的梯度方向应该沿着坡的方向向下，但粗糙的梯度估计使其稍有偏离就撞向两侧的峭壁，然后在两个峭壁间来回**震荡**。
  - **鞍点**的形状类似一个马鞍，一个方向两头翘，一个方向两头垂，而**中间区域近似平地**；一旦优化的过程中不慎落入鞍点，优化很可能就会停滞下来。

SGD 的改进遵循两个方向：**惯性保持**和**环境感知**。

- **惯性保持**指的是加入**动量**：**动量 (`Momentum`) 方法**。
- **环境感知**指的是根据不同参数的一些**经验性判断**，**自适应**地确定**每个参数的学习速率**：**自适应学习率的优化算法**。



(i) 梯度下降法和随机梯度下降法迭代过程 (ii) 随机梯度下降法存在的问题，峡谷与鞍点

图 7.1. 随机梯度下降法

[1]:
```
from abc import ABC, abstractmethod
import numpy as np
```

[2]:
```
class OptimizerBase(ABC):

    def __init__(self):
        pass

    def __call__(self):
        return self.update(params, params_grad, params_name)

    @abstractmethod
    def update(self, params, params_grad, params_name):
        """
        参数说明:
        params: 待更新参数， 如权重矩阵 W;
        params_grad: 待更新参数的梯度;
        params_name: 待更新参数名;
        """
        raise NotImplementedError
```

[3]:
```
class SGD(OptimizerBase):
    """
    sgd 优化方法
    """
    def __init__(self, lr=0.001):
        super().__init__()
        self.lr = lr
        self.cache = {}

    def __str__(self):
        return "SGD(lr={})".format(self.hyperparams["lr"])

    def update(self, params, params_grad, params_name):
        update_value = self.lr * params_grad
        return params - update_value

    @property
    def hyperparams(self):
        return {
            "op": "SGD",
            "lr": self.lr
        }
```

7.1.2 动量

Momentum 算法

引入**动量** (Momentum) 方法一方面是为了解决 “峡谷” 和 “鞍点” 问题，另一方面也可以用于 SGD 加速，特别是针对**高曲率、小幅但是方向一致**的梯度。

如果把原始的 SGD 想象成一个**纸团**在重力作用向下滚动，由于**质量小**受到山壁弹力的干扰大，导致来回震荡。或者在鞍点处因为**质量小**速度很快减为 0，导致无法离开这块平地。**动量**方法相当于把纸团换成了**铁球**。不容易受到外力的干扰，轨迹更加稳定，同时因为在鞍点处因为**惯性**的作用，更有可能离开平地。

参数更新公式

$$
\begin{aligned}
v &\leftarrow \alpha v - \epsilon \nabla_\theta (\frac{1}{m} \sum_{i=1}^{m} J(f(\boldsymbol{x^{(i)}}; \theta), y^{(i)})) \\
\theta &\leftarrow \theta + v
\end{aligned}
\tag{7.2}
$$

- 从形式上看，动量算法引入了变量 v 充当速度角色，以及相关的超参数 α，决定了之前的梯度贡献衰减得有多快；
- 原始 SGD 每次更新的步长只是梯度乘以学习率。现在，步长还取决于**历史梯度序列**的大小和排列。当许多连续的梯度指向**相同的方向**时，**步长会被不断增大**。

速度 v 累积了当前梯度元素 $\nabla_\theta(\frac{1}{m}\sum_{i=1}^{m} J(f(\boldsymbol{x^{(i)}}; \theta), y^{(i)}))$，相对于 ϵ，如果 α 越大，之前梯度对现在方向的影响也越大。实践中，α 一般取值为 0.5，0.9 和 0.99。

当前迭代点的下降方向不仅仅取决于**当前的梯度**，还受到**前面所有迭代点的影响**。动量方法以一种廉价的方式模拟了二阶梯度 (牛顿法)。

NAG 算法

Nesterov 提出了一个针对动量算法的改进措施。前面的动量算法是把历史的梯度和当前的梯度进行合并，来计算下降的方向。而 Nesterov 提出，让迭代点先按照历史梯度走一步，然后再合并。更新规则如下，改变主要在于梯度的计算上：

$$
\begin{aligned}
v &\leftarrow \alpha v - \epsilon \nabla_\theta (\frac{1}{m} \sum_{i=1}^{m} J(f(\boldsymbol{x^{(i)}}; \theta + \alpha v), y^{(i)})) \\
\theta &\leftarrow \theta + v
\end{aligned}
\tag{7.3}
$$

参数 α 和 ϵ 发挥了和标准动量方法中类似的作用，Nesterov 动量和标准动量之间的区别在于梯度的计算上。**NAG 把梯度计算放在对参数施加当前速度之后**，这个 “**提前量**” 的设计让算法有了对前方环境 “**预判**” 的能力，可以理解为 Nesterov 动量往标准动量方法中添加了一个校正因子。在凸优化问题使用批量梯度下降的情况下，Nesterov 动量将 k 步之后额外误差收敛率从 $O(\frac{1}{k})$ 提高到 $O(\frac{1}{k^2})$，对 SGD 没有改进收敛率。

如图 7.2 所示，左图显示了 Momentum 方法的轨迹，右图显示了 Nesterov 方法的轨迹。

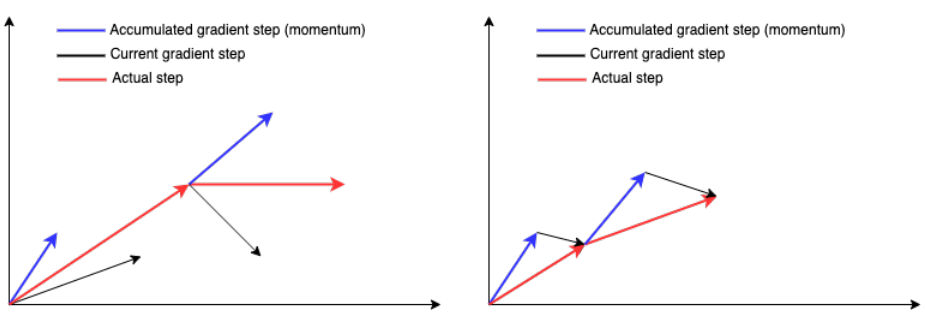


图 7.2. 动量方法的轨迹。蓝色线为历史的梯度，黑色线为当前的梯度，红色线为实际轨迹。

[4]:
```
"""
在 SGD 中考虑 momentum
"""
class Momentum(OptimizerBase):

    def __init__(
        self, lr=0.001, momentum=0.0, **kwargs
    ):
        """
        参数说明:
        lr: 学习率, float (default: 0.001)
        momentum: 考虑 Momentum 时的 alpha, 决定了之前的梯度贡献衰减得有多快, 取值范围 [0, 1], 默认 0
        """
        super().__init__()
```

```
        self.lr = lr
        self.momentum = momentum
        self.cache = {}

    def __str__(self):
        return "Momentum(lr={}, momentum={})".format(self.lr, self.momentum)

    def update(self, param, param_grad, param_name):
        C = self.cache
        lr, momentum = self.lr, self.momentum

        if param_name not in C:  # save v
            C[param_name] = np.zeros_like(param_grad)

        update = momentum * C[param_name] - lr * param_grad
        self.cache[param_name] = update
        return param + update

    @property
    def hyperparams(self):
        return {
            "op": "Momentum",
            "lr": self.lr,
            "momentum": self.momentum
        }
```

7.1.3 自适应学习率

由于 SGD 中随机采样 Batch 会引入噪声源，在极小点处梯度并不会消失。因此，随着梯度的降低，有必要逐步减小学习率。

AdaGrad 算法

算法的思想是独立地适应模型的每个参数：**一直较大偏导的参数相应有一个较小的学习率，初始学习率会下降的较快；而一直小偏导的参数则对应一个较大的学习率，初始学习率会下降的较慢**。具体来说，每个参数的学习率会缩放各参数反比于其**历史梯度平方值总和的平方根**，更新公式：

$$
\begin{aligned}
g &\leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\boldsymbol{x}^{(i)};\theta), y^{(i)}) \\
r &\leftarrow r + g \odot g \\
\theta &\leftarrow \theta - \frac{\epsilon}{\delta + \sqrt{r}} \odot g
\end{aligned}
\tag{7.4}
$$

r 为**累积平方梯度**。学习率相当于 $\frac{\epsilon}{\delta+\sqrt{r}}$，对于更新不频繁的参数，单次步长更大；对于更新频繁的参数，步长较小，使得学习到的参数更稳定，不至于被单个样本影响太多。

AdaGrad 算法存在的问题，历史梯度在分母上的**累积会越来越大**，所以**学习率会越来越小**，使得在中后期网络的学习能力越来越弱。

[5]:
```
class AdaGrad(OptimizerBase):

    def __init__(self, lr=0.001, eps=1e-7, **kwargs):
        """
        参数说明:
        lr: 学习率, float (default: 0.001)
        eps: delta 项, 防止分母为 0
        """
        super().__init__()
        self.lr = lr
        self.eps = eps
        self.cache = {}

    def __str__(self):
        return "AdaGrad(lr={}, eps={})".format(self.lr, self.eps)
```

```
    def update(self, param, param_grad, param_name):
        C = self.cache
        lr, eps = self.hyperparams["lr"], self.hyperparams["eps"]
        if param_name not in C:  # 保存 r
            C[param_name] = np.zeros_like(param_grad)
        C[param_name] += param_grad ** 2
        update = lr * param_grad / (np.sqrt(C[param_name]) + eps)
        self.cache = C
        return param - update

    @property
    def hyperparams(self):
        return {
            "op": "AdaGrad",
            "lr": self.lr,
            "eps": self.eps
        }
```

RMSProp 算法

RMSProp 主要是为了解决 AdaGrad 方法中**学习率过度衰减**的问题—— AdaGrad 根据平方梯度的整个历史来收缩学习率，可能使得学习率在达到局部最小值之前就变得太小而难以继续训练。RMSProp 使用**指数衰减平均** (递归定义) 以丢弃遥远的历史，使其能够在找到某个 “凸” 结构后快速收敛。此外，RMSProp 还加入了一个超参数 ρ 用于控制衰减速率：

$$
\begin{aligned}
g &\leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\boldsymbol{x}^{(i)};\theta), y^{(i)}) \\
r &\leftarrow \rho r + (1-\rho) g \odot g \\
\theta &\leftarrow \theta - \frac{\epsilon}{\sqrt{\delta + r}} \odot g
\end{aligned}
\tag{7.5}
$$

RMSProp 建议的**初始值**：全局学习率 $\epsilon = 1e-3$，衰减速率 $\rho = 0.9$。

[6]:
```
class RMSProp(OptimizerBase):

    def __init__(
        self, lr=0.001, decay=0.9, eps=1e-7, **kwargs
    ):
        """
        参数说明:
        lr: 学习率, float (default: 0.001)
        eps: delta 项, 防止分母为 0
        decay: 衰减速率
        """
        super().__init__()
        self.lr = lr
        self.eps = eps
        self.decay = decay
        self.cache = {}

    def __str__(self):
        return "RMSProp(lr={}, eps={}, decay={})".format(
            self.lr, self.eps, self.decay
        )

    def update(self, param, param_grad, param_name):
        C = self.cache
        lr, eps = self.hyperparams["lr"], self.hyperparams["eps"]
        decay = self.hyperparams["decay"]
        if param_name not in C:  # 保存 r
```

```
            C[param_name] = np.zeros_like(param_grad)
        C[param_name] = decay * C[param_name] + (1 - decay) * param_grad ** 2
        update = lr * param_grad / (np.sqrt(C[param_name]) + eps)
        self.cache = C
        return param - update

    @property
    def hyperparams(self):
        return {
            "op": "RMSProp",
            "lr": self.lr,
            "eps": self.eps,
            "decay": self.decay
        }
```

AdaDelta 算法

AdaDelta 和 RMSProp 一样使用**指数衰减平均** (递归定义) 以丢弃遥远的历史，但 AdaDelta 算法没有学习率这一超参数。AdaDelta 算法维护一个额外的变量 $\Delta\theta$，来计算自变量变化量按元素平方的指数加权移动平均：

$$
\begin{aligned}
g &\leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\boldsymbol{x}^{(i)};\theta), y^{(i)}) \\
r &\leftarrow \rho r + (1-\rho) g \odot g \\
g' &\leftarrow \frac{\sqrt{\delta + \Delta\theta}}{\sqrt{\delta + r}} \odot g \\
\theta &\leftarrow \theta - g' \\
\Delta\theta &\leftarrow \rho\Delta\theta + (1-\rho) g' \odot g'
\end{aligned}
\tag{7.6}
$$

可以看到，如不考虑 δ 的影响，AdaDelta 算法与 RMSProp 算法的不同之处在于使用 $\sqrt{\Delta\theta}$ 来替代超参数 ϵ。

[7]:
```
class AdaDelta(OptimizerBase):

    def __init__(
        self, lr=0.001, decay=0.95, eps=1e-7, **kwargs
    ):
        """
        参数说明:
        lr: 学习率, float (default: 0.001)
        eps: delta 项, 防止分母为 0
        decay: 衰减速率
        """
        super().__init__()
        self.lr = lr
        self.eps = eps
        self.decay = decay
        self.cache = {}

    def __str__(self):
        return "AdaDelta(eps={}, decay={})".format(self.eps, self.decay)

    def update(self, param, param_grad, param_name):
        C = self.cache
        eps = self.hyperparams["eps"]
        decay = self.hyperparams["decay"]
        if param_name not in C:  # 保存 r, delta_theta
            C[param_name] = {
                "r": np.zeros_like(param_grad),
                "d": np.zeros_like(param_grad)
            }
        C[param_name]["r"] = decay * C[param_name]["r"] + (1 - decay) * param_grad ** 2
```

```
        update = (np.sqrt(C[param_name]["d"] + eps)) * param_grad / (np.sqrt(C[param_name]["r"]) + eps)
        C[param_name]["d"] = decay * C[param_name]["d"] + (1 - decay) * update ** 2
        self.cache = C
        return param - update

    @property
    def hyperparams(self):
        return {
            "op": "AdaDelta",
            "eps": self.eps,
            "decay": self.decay
        }
```

Adam 算法

Adam 在 RMSProp 方法的基础上更进一步：除了加入**历史梯度平方的指数衰减平均** (s) 外，还保留了**历史梯度的指数衰减平均** (r)，相当于**动量**。Adam 行为就像一个带有摩擦力的小球，在误差面上倾向于平坦的极小值。

$$
\begin{aligned}
&g \leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\boldsymbol{x}^{(i)};\theta), y^{(i)}) \\
&t \leftarrow t+1 \\
&r \leftarrow \rho_1 r + (1-\rho_1) g \\
&s \leftarrow \rho_2 s + (1-\rho_2) g \odot g \\
&\hat{r} \leftarrow \frac{r}{1-\rho_1^t} \\
&\hat{s} \leftarrow \frac{s}{1-\rho_2^t} \\
&\Delta\theta \leftarrow -\epsilon \frac{\hat{r}}{\sqrt{\hat{s}+\delta}} \\
&\theta \leftarrow \theta + \Delta\theta
\end{aligned}
\tag{7.7}
$$

偏差修正

这里的 s 和 r 初始化为 0，且 ρ_1 和 ρ_2 推荐的初始值都很接近 1 (0.9 和 0.999)。注意到，在时间步 t 我们得到 $r_t = (1-\rho_1)\sum_{i=1}^{t}\rho_1^{t-i}g_i$，将过去各时间步小批量随机梯度的权值相加，得到 $(1-\rho_1)\sum_{i=1}^{t}\rho_1^{t-i} = 1-\rho_1^t$。可以看出，当 t 较小时，过去各时间步小批量随机梯度权值之和会较小。假设取 $\rho_1 = 0.9$，可以得到 $t=1$ 时 $r_1 = 0.1g_1$。因此我们需要偏差修正 (见式 7.7 第 5，6 行)，使过去各时间步小批量随机梯度权值之和为 1。

[8]:
```
class Adam(OptimizerBase):

    def __init__(
        self,
        lr=0.001,
        decay1=0.9,
        decay2=0.999,
        eps=1e-7,
        **kwargs
    ):
        """
        参数说明:
        lr: 学习率，float (default: 0.01)
        eps: delta 项，防止分母为 0
        decay1: 历史梯度的指数衰减速率，可以理解为考虑梯度均值 (default: 0.9)
        decay2: 历史梯度平方的指数衰减速率，可以理解为考虑梯度方差 (default: 0.999)
        """
        super().__init__()
        self.lr = lr
        self.decay1 = decay1
        self.decay2 = decay2
        self.eps = eps
        self.cache = {}
```

```
    def __str__(self):
        return "Adam(lr={}, decay1={}, decay2={}, eps={})".format(
            self.lr, self.decay1, self.decay2, self.eps
        )

    def update(self, param, param_grad, param_name):
        C = self.cache
        d1, d2 = self.hyperparams["decay1"], self.hyperparams["decay2"]
        lr, eps = self.hyperparams["lr"], self.hyperparams["eps"]
        if param_name not in C:
            C[param_name] = {
                "t": 0,
                "mean": np.zeros_like(param_grad),
                "var": np.zeros_like(param_grad),
            }
        t = C[param_name]["t"] + 1
        mean = C[param_name]["mean"]
        var = C[param_name]["var"]
        C[param_name]["t"] = t
        C[param_name]["mean"] = d1 * mean + (1 - d1) * param_grad
        C[param_name]["var"] = d2 * var + (1 - d2) * param_grad ** 2
        self.cache = C
        m_hat = C[param_name]["mean"] / (1 - d1 ** t)
        v_hat = C[param_name]["var"] / (1 - d2 ** t)
        update = lr * m_hat / (np.sqrt(v_hat) + eps)
        return param - update

    @property
    def hyperparams(self):
        return {
            "op": "Adam",
            "lr": self.lr,
            "eps": self.eps,
            "decay1": self.decay1,
            "decay2": self.decay2
        }
```

使用上述优化算法，MNIST 数据集测试

[9]:
```
from chapter6 import DFN
```

[10]:
```
"""
载入数据
"""
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1]*X_train.shape[2]).astype('float32')
X_test = X_test.reshape(-1, X_test.shape[1]*X_test.shape[2]).astype('float32')
print(X_train.shape, y_train.shape)
N = 20000   # 取 20000 条数据用以训练
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
```

```
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 784) (60000, 10)
(20000, 784) (20000, 10)
```

[11]:
```
"""
SGD 优化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="sgd(lr=0.01)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("sgd -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
sgd -- accuracy:0.8951
```

[12]:
```
"""
Momentum 优化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="momentum(lr=0.01, momentum=0.95)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("momentum -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
momentum -- accuracy:0.9604
```

[13]:
```
"""
AdaGrad 优化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="adagrad(lr=0.01, eps=1e-7)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("adagrad -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
adagrad -- accuracy:0.9503
```

[14]:
```
"""
RMSProp 优化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="rmsprop(lr=0.001, eps=1e-7, decay=0.95)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("rmsprop -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
rmsprop -- accuracy:0.9614
```

[15]:
```
"""
AdaDelta 优化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="adadelta(eps=1e-7, decay=0.95)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("adadelta -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
adadelta -- accuracy:0.9624
```

[16]:
```
"""
Adam 优化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="adam(lr=0.001, decay1=0.9, decay2=0.999, eps=1e-7)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("adam -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
adam -- accuracy:0.9667
```

7.1.4 二阶近似方法

牛顿法

梯度下降使用的梯度信息实际上是**一阶导数**，而牛顿法除了一阶导数外，还会使用**二阶导数**的信息。根据导数的定义，一阶导描述的是函数值的变化率，即**斜率**；二阶导描述的则是斜率的变化率，即曲线的弯曲程度——**曲率**。为简单起见，我们从泰勒展开说起：

$$f(x) \approx f(x_0) + f'(x)(x - x_0) + \frac{f''(x)}{2}(x - x_0)^2 \tag{7.8}$$

我们可以通过 x_0 去估计 x。牛顿法的思想也从中而来，我们可以通过在现有极小值点估计值的附近对 $f(x)$ 做二阶泰勒展开，进而找到极小值的下一个估计值。设 x_k 为当前的极小点估计值，则：

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k) + \frac{f''(x_k)}{2}(x - x_k)^2 \tag{7.9}$$

由极值的必要条件可知，新的极小值应该满足：$f'(x) = 0$。于是：

$$f(x_k) + \frac{f''(x)}{2}(x - x_k)^2 = 0 \tag{7.10}$$

进而得到新的极小值点：

$$x = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{7.11}$$

这样一来，我们就可以构造序列 $\{x_k\}$ 来逼近 $f(x)$ 的极小点。

我们推广到高维下的 x (维数为 n)，二阶泰勒展开可以写作：

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_k)^\top \nabla^2 f(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k) \tag{7.12}$$

其中：

$$\nabla^2 f(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial {x_2}^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ & & \ddots & \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{7.13}$$

我们通常称 ∇f 为 f 的梯度向量，同时记 $\nabla^2 f$ 为 $\mathrm{H}(f)$，称为海森矩阵，它是个对称矩阵。同样地，我们根据极值必要条件，可得 $\nabla f(\boldsymbol{x}) = 0$。

在泰勒展开等式两边同时作用一个梯度算子，可以得到：

$$\nabla f_k + \mathrm{H}_k(\boldsymbol{x} - \boldsymbol{x}_k) = 0 \tag{7.14}$$

如果矩阵 H_k 非奇异，则可以得到：

$$\boldsymbol{x} = \boldsymbol{x}_k - \mathrm{H}_k^{-1} \nabla f_k \tag{7.15}$$

于是可以得到迭代公式：

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \mathrm{H}_k^{-1} \nabla f_k, \quad k = 0, 1, ... \tag{7.16}$$

同样地，我们考虑损失函数下的问题，同样进行二阶泰勒展开：

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^\top \mathrm{H}(\theta - \theta_0) \tag{7.17}$$

于是，迭代公式为：

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathrm{H}_k^{-1} \nabla J_k, \quad k = 0, 1, ... \tag{7.18}$$

具体来说，更新公式为：

$$\begin{aligned} g &\leftarrow \frac{1}{m} \nabla_\theta \sum_i J(f(\boldsymbol{x^{(i)}}; \theta), y^{(i)}) \\ \mathrm{H} &\leftarrow \frac{1}{m} \nabla_\theta^2 \sum_i J(f(\boldsymbol{x^{(i)}}; \theta), y^{(i)}) \\ \Delta\theta &\leftarrow -\mathrm{H}^{-1} g \\ \theta &\leftarrow \theta + \Delta\theta \end{aligned} \tag{7.19}$$

当目标函数是二次函数时，则二次泰勒展开函数和原目标函数完全相同，对应此时的二次求导下的海森矩阵退化为常数矩阵。经过一次迭代后就可到达原目标函数的极小点，因此牛顿法是具有二次收敛性。而对非二次函数，如果函数的二次性态较强，或者迭代点进入极小点的邻域，此时牛顿法也很快的。

几何理解

牛顿法就是用一个**二次曲面**去拟合当前所处位置的局部曲面；而梯度下降法是用一个平面去拟合当前的局部曲面。

通常情况下，二次曲面的拟合会比平面更好，所以牛顿法选择的**下降路径**会更符合真实的最优下降路径。

通俗理解

找一条最短的路径走到一个盆地的最底部，梯度下降法每次只从当前所处位置选一个坡度最大的方向走一步；牛顿法在选择方向时，不仅会考虑坡度是否够大，还会考虑走了一步之后，**坡度是否会变得更大** (考虑梯度变化的趋势)。所以，牛顿法比梯度下降法看得更远，能**更快**地走到最底部。

- **优点**
  - 收敛速度快，能用更少的迭代次数找到最优解。
- **缺点**
  - 每一步都需要求解目标函数的 Hessian 矩阵的**逆矩阵**，计算复杂。
  - 牛顿法是局部收敛的，当初始点选择不当时，往往导致不收敛。
  - 二阶 Hessian 矩阵必须可逆，否则算法进行困难。

拟牛顿法

牛顿法虽然收敛快，但是需要计算海森矩阵，同时还要保证海森矩阵非奇异，这使得牛顿法失效。为了克服这两个问题，便提出来拟牛顿法。其基本思想是：**不用二阶偏导数，而是在“拟牛顿”的条件下优化目标函数，直接构造出可以近似海森矩阵 (或海森矩阵的逆) 的正定对称阵。**

拟牛顿条件

回到前面描述的二阶泰勒展开，在第 $k+1$ 次迭代进行展开：

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_{k+1}) + \nabla f(\boldsymbol{x}_{k+1})(\boldsymbol{x} - \boldsymbol{x}_{k+1}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_{k+1})^{\top} \mathrm{H}_{k+1}(\boldsymbol{x} - \boldsymbol{x}_{k+1}) \tag{7.20}$$

两边同时作用梯度算子 ∇，得到：

$$\nabla f(\boldsymbol{x}) \approx \nabla f(\boldsymbol{x}_{k+1}) + \mathrm{H}_{k+1}(\boldsymbol{x} - \boldsymbol{x}_{k+1}) \tag{7.21}$$

令 $\boldsymbol{x} = \boldsymbol{x}_k$：

$$\nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}) \approx \mathrm{H}_{k+1}(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) \tag{7.22}$$

现在，我们简化标记，令 $\boldsymbol{s}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$，$\boldsymbol{y}_k = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x})$。于是有：

$$\boldsymbol{y}_k \approx \mathrm{H}_{k+1}\boldsymbol{s}_k \quad \text{或者} \quad \boldsymbol{s}_k \approx \mathrm{H}_{k+1}^{-1}\boldsymbol{y}_k \tag{7.23}$$

这便是拟牛顿条件，它对迭代过程中的 H_{k+1} 做了约束。在迭代过程中，我们会用矩阵 $\boldsymbol{B}$ 去近似海森矩阵 H，用矩阵 $\boldsymbol{D}$ 去近似海森矩阵的逆 H^{-1}。那么我们可以进一步写成：

$$\boldsymbol{y}_k = \boldsymbol{B}_{k+1}\boldsymbol{s}_k \quad \text{或者} \quad \boldsymbol{s}_k = \boldsymbol{D}_{k+1}\boldsymbol{y}_k \tag{7.24}$$

DFP 算法

DFP 算法的核心是：通过迭代的方法，对 H^{-1} 近似。迭代过程为：

$$\boldsymbol{D}_{k+1} = \boldsymbol{D}_k + \Delta\boldsymbol{D}_k, \quad k = 0, 1... \tag{7.25}$$

其中 $\boldsymbol{D}_0$ 通常取单位矩阵，而 $\Delta\boldsymbol{D}_k$ 一般用“待定法”计算。具体来说，设校正矩阵 $\Delta\boldsymbol{D}_k = \alpha\boldsymbol{u}\boldsymbol{u}^{\top} + \beta\boldsymbol{v}\boldsymbol{v}^{\top}$，其中 α 和 β 是待定参数。这样的待定法可以保证 $\Delta\boldsymbol{D}_k$ 的对称性。

我们将“待定法”用到海森矩阵的近似中：

$$\begin{aligned} \boldsymbol{s}_k &= (\boldsymbol{D}_k + \alpha\boldsymbol{u}\boldsymbol{u}^{\top} + \beta\boldsymbol{v}\boldsymbol{v}^{\top})\boldsymbol{y}_k \\ &= \boldsymbol{D}_k\boldsymbol{y}_k + \boldsymbol{u}(\alpha\boldsymbol{u}^{\top}\boldsymbol{y}_k) + \boldsymbol{v}(\beta\boldsymbol{v}^{\top}\boldsymbol{y}_k) \\ &= \boldsymbol{D}_k\boldsymbol{y}_k + (\alpha\boldsymbol{u}^{\top}\boldsymbol{y}_k)\boldsymbol{u} + (\beta\boldsymbol{v}^{\top}\boldsymbol{y}_k)\boldsymbol{v} \end{aligned} \tag{7.26}$$

$\alpha\boldsymbol{u}^{\top}\boldsymbol{y}_k$ 和 $\beta\boldsymbol{v}^{\top}\boldsymbol{y}_k$ 既是常数，我们可以简单赋值为 1 和 -1。于是，我们可以得到：

$$\begin{aligned} \alpha &= \frac{1}{\boldsymbol{u}^{\top}\boldsymbol{y}_k} \\ \beta &= -\frac{1}{\boldsymbol{v}^{\top}\boldsymbol{y}_k} \end{aligned} \tag{7.27}$$

将 α 和 β 代入回去，可以得到：$\boldsymbol{u}-\boldsymbol{v}=\boldsymbol{s}_k-\boldsymbol{D}_k\boldsymbol{y}_k$。于是，我们不妨取 $\boldsymbol{u}=\boldsymbol{s}_k$，$\boldsymbol{v}=\boldsymbol{D}_k\boldsymbol{y}_k$。再代入回上述表达式中，得到：

$$\begin{aligned}\alpha&=\frac{1}{\boldsymbol{s}_k^\top\boldsymbol{y}_k}\\ \beta&=-\frac{1}{(\boldsymbol{D}_k\boldsymbol{y}_k)^\top\boldsymbol{y}_k}=-\frac{1}{\boldsymbol{y}_k^\top\boldsymbol{D}_k\boldsymbol{y}_k}\end{aligned}\tag{7.28}$$

最后，我们得到校正矩阵的形式：

$$\Delta\boldsymbol{D}_k=\frac{\boldsymbol{s}_k\boldsymbol{s}_k^\top}{\boldsymbol{s}_k^\top\boldsymbol{y}_k}-\frac{\boldsymbol{D}_k\boldsymbol{y}_k\boldsymbol{y}_k^\top\boldsymbol{D}_k}{\boldsymbol{y}_k^\top\boldsymbol{D}_k\boldsymbol{y}_k}\tag{7.29}$$

于是，我们得到 DFP 算法的步骤：

- 根据 $-\boldsymbol{D}_k\boldsymbol{g}_k$ 得到搜索方向 $\boldsymbol{d}_k$，再考虑步长 λ_k (可以用 Line Search 获得) 得到 $\boldsymbol{s}_k=\lambda_k\boldsymbol{d}_k$。这样便得到新一次迭代的 $\boldsymbol{x}_{k+1}$。
- 计算 $\nabla f(\boldsymbol{x}_k)$ (如果 $||\nabla f(\boldsymbol{x}_k)||<\epsilon$ 则结束)。进一步计算 $\boldsymbol{y}_k$ 和 $\Delta\boldsymbol{D}_k$，从而得到 $\boldsymbol{D}_{k+1}$。

BFGS 算法

BFGS 算法的思路同 DFP 算法，区别在于近似海森矩阵 H。设校正矩阵 $\Delta\boldsymbol{B}_k=\alpha\boldsymbol{u}\boldsymbol{u}^\top+\beta\boldsymbol{v}\boldsymbol{v}^\top$。可以化简得：

$$\boldsymbol{y}_k=\boldsymbol{B}_k\boldsymbol{s}_k+(\alpha\boldsymbol{u}^\top\boldsymbol{s}_k)\boldsymbol{u}+(\beta\boldsymbol{v}^\top\boldsymbol{s}_k)\boldsymbol{v}\tag{7.30}$$

令 $\alpha\boldsymbol{u}^\top\boldsymbol{s}_k=1$ 以及 $\beta\boldsymbol{v}^\top\boldsymbol{s}_k=-1$。同时，取 $\boldsymbol{u}=\boldsymbol{y}_k$，$\boldsymbol{v}=\boldsymbol{B}_k\boldsymbol{s}_k$。可得：

$$\begin{aligned}\alpha&=\frac{1}{\boldsymbol{y}_k^\top\boldsymbol{s}_k}\\ \beta&=-\frac{1}{(\boldsymbol{B}_k\boldsymbol{s}_k)^\top\boldsymbol{s}_k}=-\frac{1}{\boldsymbol{s}_k^\top\boldsymbol{B}_k\boldsymbol{s}_k}\end{aligned}\tag{7.31}$$

所以，校正矩阵的形式为：

$$\Delta\boldsymbol{B}_k=\frac{\boldsymbol{y}_k\boldsymbol{y}_k^\top}{\boldsymbol{y}_k^\top\boldsymbol{s}_k}-\frac{\boldsymbol{B}_k\boldsymbol{s}_k\boldsymbol{s}_k^\top\boldsymbol{B}_k}{\boldsymbol{s}_k^\top\boldsymbol{B}_k\boldsymbol{s}_k}\tag{7.32}$$

于是，我们得到 BFGS 算法的步骤：

- 根据 $-\boldsymbol{B}_k^{-1}\boldsymbol{g}_k$ 得到搜索方向 $\boldsymbol{d}_k$，再考虑步长 λ_k 得到 $\boldsymbol{s}_k=\lambda_k\boldsymbol{d}_k$。这样便得到新一次迭代的 $\boldsymbol{x}_{k+1}$。
- 计算 $\nabla f(\boldsymbol{x}_k)$ (如果 $||\nabla f(\boldsymbol{x}_k)||<\epsilon$ 则结束)。进一步计算 $\boldsymbol{y}_k$ 和 $\Delta\boldsymbol{B}_k$，从而得到 $\boldsymbol{B}_{k+1}$。

如何计算 $-\boldsymbol{B}_k^{-1}\boldsymbol{g}_k$

这里需要注意的是如何计算 $-\boldsymbol{B}_k^{-1}\boldsymbol{g}_k$。一种方法是解线性方程组 $\boldsymbol{B}_k\boldsymbol{d}_k=-\boldsymbol{g}_k$。另一种方法是借用 Sherman-Morrison 公式。

Sherman-Morrison 公式：

假设 $\boldsymbol{A}\in\mathbb{R}^{n\times n}$ 为非奇异矩阵，$\boldsymbol{x},\boldsymbol{y}\in\mathbb{R}^n$，如果 $1+\boldsymbol{y}^\top\boldsymbol{A}\boldsymbol{x}\neq0$，则有：

$$(\boldsymbol{A}+\boldsymbol{x}\boldsymbol{y}^\top)^{-1}=\boldsymbol{A}^{-1}-\frac{\boldsymbol{A}^{-1}\boldsymbol{x}\boldsymbol{y}^\top\boldsymbol{A}^{-1}}{1+\boldsymbol{y}^\top\boldsymbol{A}\boldsymbol{x}}\tag{7.33}$$

现在，我们用 Sherman-Morrison 公式，可以直接获得 $\boldsymbol{B}_{k+1}^{-1}$ 和 $\boldsymbol{B}_k^{-1}$ 间的递推关系。然后我们将 $\boldsymbol{B}_k^{-1}$ 表示为 $\boldsymbol{D}_k^{-1}$，这样就不用计算取逆操作。首先，我们看一下递推结果：

$$\boldsymbol{B}_{k+1}^{-1}=\left(I-\frac{\boldsymbol{s}_k\boldsymbol{y}_k^\top}{\boldsymbol{y}_k^\top\boldsymbol{s}_k}\right)\boldsymbol{B}_k^{-1}\left(I-\frac{\boldsymbol{y}_k\boldsymbol{s}_k^\top}{\boldsymbol{y}_k^\top\boldsymbol{s}_k}\right)+\frac{\boldsymbol{s}_k\boldsymbol{s}_k^\top}{\boldsymbol{y}_k^\top\boldsymbol{s}_k}\tag{7.34}$$

然后我们简单说一下证明思路，考虑原本的递推式为：

$$\boldsymbol{B}_{k+1}=\boldsymbol{B}_k+\frac{\boldsymbol{y}_k\boldsymbol{y}_k^\top}{\boldsymbol{y}_k^\top\boldsymbol{s}_k}-\frac{\boldsymbol{B}_k\boldsymbol{s}_k\boldsymbol{s}_k^\top\boldsymbol{B}_k}{\boldsymbol{s}_k^\top\boldsymbol{B}_k\boldsymbol{s}_k}\tag{7.35}$$

1. 首先，令 $\mathrm{H}_k=\boldsymbol{B}_k+\frac{\boldsymbol{y}_k\boldsymbol{y}_k^\top}{\boldsymbol{y}_k^\top\boldsymbol{s}_k}$。我们再用 Sherman-Morrison 公式，可以得到 $\boldsymbol{B}_{k+1}^{-1}$ 的表达式。
2. 再一次对 H_k 使用 Sherman-Morrison 公式得到 H_k^{-1} 的表达式。
3. 最后我们将 H_k^{-1} 的表达式代回 $\boldsymbol{B}_{k+1}^{-1}$ 的表达式中得到 $\boldsymbol{B}_{k+1}^{-1}$ 与 $\boldsymbol{B}_k^{-1}$ 的递推式。

我们令 $\boldsymbol{B}_k^{-1}$ 表示为 $\boldsymbol{D}_k^{-1}$，递推式如下：

$$\boldsymbol{D}_{k+1}=\left(I-\frac{\boldsymbol{s}_k\boldsymbol{y}_k^\top}{\boldsymbol{y}_k^\top\boldsymbol{s}_k}\right)\boldsymbol{D}_k\left(I-\frac{\boldsymbol{y}_k\boldsymbol{s}_k^\top}{\boldsymbol{y}_k^\top\boldsymbol{s}_k}\right)+\frac{\boldsymbol{s}_k\boldsymbol{s}_k^\top}{\boldsymbol{y}_k^\top\boldsymbol{s}_k}\tag{7.36}$$

最后，我们看一下此时 BFGS 算法的步骤：

- 根据 $-\boldsymbol{D}_k\boldsymbol{g}_k$ 得到搜索方向 $\boldsymbol{d}_k$，再考虑步长 λ_k 得到 $\boldsymbol{s}_k=\lambda_k\boldsymbol{d}_k$。这样便得到新一次迭代的 $\boldsymbol{x}_{k+1}$。
- 计算 $\nabla f(\boldsymbol{x}_k)$ (如果 $\|\nabla f(\boldsymbol{x}_k)\|<\epsilon$ 则结束)。进一步计算 $\boldsymbol{y}_k$，从而得到 $\boldsymbol{D}_{k+1}$。

L-BFGS 算法

前面的两个算法要构造 $n \times n$ 的矩阵 (n 为特征维数)，存储矩阵 (每次迭代都要存储一个矩阵) 仍需要耗费很多的资源。但考虑矩阵 $\boldsymbol{D}_k$ 的对称性，存储时只需存储一半矩阵数据。但是否可以还减少存储消耗?

这便有了 L-BFGS 算法 (`Limited-memory BFGS`)。其基本思想是：不再存储完整的矩阵 $\boldsymbol{D}_k$，而是存储计算过程中的 $\boldsymbol{s}_i$，$\boldsymbol{y}_i$。需要矩阵 $\boldsymbol{D}_k$ 时，利用向量计算得到。我们修改迭代公式：

$$\boldsymbol{D}_{k+1} = \left(I - \frac{\boldsymbol{s}_k \boldsymbol{y}_k^\top}{\boldsymbol{y}_k^\top \boldsymbol{s}_k}\right) \boldsymbol{D}_k \left(I - \frac{\boldsymbol{y}_k \boldsymbol{s}_k^\top}{\boldsymbol{y}_k^\top \boldsymbol{s}_k}\right) + \frac{\boldsymbol{s}_k \boldsymbol{s}_k^\top}{\boldsymbol{y}_k^\top \boldsymbol{s}_k} \tag{7.37}$$

如果令 $\rho_k = \frac{1}{\boldsymbol{y}_k^\top \boldsymbol{s}_k}$，$\boldsymbol{V}_k = I - \rho_k \boldsymbol{y}_k \boldsymbol{s}_k^\top$，则上式可以写作：

$$\boldsymbol{D}_{k+1} = \boldsymbol{V}_k^\top \boldsymbol{D}_k \boldsymbol{V}_k + \rho_k \boldsymbol{s}_k \boldsymbol{s}_k^\top \tag{7.38}$$

我们看一下 $\boldsymbol{D}_{k+1}$ 的生成，假设初始矩阵 $\boldsymbol{D}_0$ 为单位阵 I：

$$\begin{aligned} \boldsymbol{D}_1 &= \boldsymbol{V}_0^\top \boldsymbol{D}_0 \boldsymbol{V}_0 + \rho_0 \boldsymbol{s}_0 \boldsymbol{s}_0^\top \\ \boldsymbol{D}_2 &= \boldsymbol{V}_1^\top \boldsymbol{D}_1 \boldsymbol{V}_1 + \rho_1 \boldsymbol{s}_1 \boldsymbol{s}_1^\top = \boldsymbol{V}_1^\top \boldsymbol{V}_0^\top \boldsymbol{D}_0 \boldsymbol{V}_0 \boldsymbol{V}_1 + \boldsymbol{V}_1^\top \rho_0 \boldsymbol{s}_0 \boldsymbol{s}_0^\top \boldsymbol{V}_1 + \rho_1 \boldsymbol{s}_1 \boldsymbol{s}_1^\top \end{aligned} \tag{7.39}$$

更一般地，我们得到迭代公式：

$$\begin{aligned} \boldsymbol{D}_{k+1} = &(\boldsymbol{V}_k^\top \boldsymbol{V}_{k-1}^\top ... \boldsymbol{V}_0^\top) \boldsymbol{D}_0 (\boldsymbol{V}_0 ... \boldsymbol{V}_{k-1} \boldsymbol{V}_k) \\ &+ (\boldsymbol{V}_k^\top \boldsymbol{V}_{k-1}^\top ... \boldsymbol{V}_1^\top)(\rho_0 \boldsymbol{s}_0 \boldsymbol{s}_0^\top)(\boldsymbol{V}_1 ... \boldsymbol{V}_{k-1} \boldsymbol{V}_k) \\ &+ (\boldsymbol{V}_k^\top \boldsymbol{V}_{k-1}^\top ... \boldsymbol{V}_2^\top)(\rho_1 \boldsymbol{s}_1 \boldsymbol{s}_1^\top)(\boldsymbol{V}_2 ... \boldsymbol{V}_{k-1} \boldsymbol{V}_k) \\ &+ \cdots \\ &+ \boldsymbol{V}_k^\top (\rho_{k-1} \boldsymbol{s}_{k-1} \boldsymbol{s}_{k-1}^\top) \boldsymbol{V}_k \\ &+ \rho_{k-1} \boldsymbol{s}_{k-1} \boldsymbol{s}_{k-1}^\top \end{aligned} \tag{7.40}$$

7.2 优化策略

7.2.1 参数初始化

神经网络的训练过程中的参数学习是基于梯度下降法进行优化的。梯度下降法需要在开始训练时给每一个参数赋一个初始值。这个初始值的选取十分关键。一般希望**数据和参数的均值都为 0，输入和输出数据的方差一致**。在实际应用中，参数服从高斯分布或者均匀分布都是比较有效的初始化方式。

初始参数需要在不同单元间**破坏对称性**。如果具有相同激活函数的两个隐藏单元连接到相同的输入，那么这些单元必须**具有不同的初始参数**。**如果它们具有相同的初始参数**，然后应用到确定性损失和模型的确定性学习算法将**一直以相同的方式更新这两个单元**。

更大的初始权重具有**更强的破坏对称性**的作用，但也会在前向传播或反向传播中产生**爆炸的值**。如在循环网络中，很大的权重也可能导致混沌。因此，一般不要全零初始化或者具有较大数的随机初始化。

初始化策略有：

1、随机初始化方法

- 正态化的随机初始化 (Random Normal)：它是从以 0 为中心，标准差为 std 的正态分布中抽取样本。
- 标准化的随机初始化 (Random Uniform)：它是从 $[-1, 1]$ 中的均匀分布中抽取样本。

2、Glorot 初始化方法

Glorot 初始化，也称为 Xavier 初始化。首先，我们考虑隐藏层 $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x}$ 以及 $\boldsymbol{a} = \sigma(\boldsymbol{z})$，Glorot 初始化认为一个好的初始化应该在各个层的激活值 $\boldsymbol{a}$ 的方差保持一致，即对于第 i 层和第 j 层，有 $\mathrm{Var}(a^i) = \mathrm{Var}(a^j)$，同时各个层对状态 $\boldsymbol{z}$ 的梯度的方差要保持一致，即 $\mathrm{Var}(\frac{\partial J}{\partial z^i}) = \mathrm{Var}(\frac{\partial J}{\partial z^j})$。我们先分析一边，我们视每个神经元为一个特征。假设单层神经元上的特征的方差一样，即 $\mathrm{Var}(x_k) = \mathrm{Var}(\boldsymbol{x})$，每一层的输入 $\boldsymbol{a}$ 均值为 0，且初始时 $\boldsymbol{z}$ 落在激活函数的线性区域，即 $\sigma'(z_l) \approx 1$，$l = \{1, ..., m\}$，m 为该层神经元数目。于是：

$$\begin{aligned} \mathrm{Var}(a_l^i) &= \mathrm{Var}(\sigma(z_l^i)) \\ &= \mathrm{Var}(z_l^i) \\ &= \mathrm{Var}(\sum_{k=1}^{n} W_{lk} * a_k^{i-1}) \\ &= \sum_{k=1}^{n} \mathrm{Var}(W_{lk} * a_k^{i-1}) \\ &= n\,\mathrm{Var}(\mathrm{W}_{lk})\,\mathrm{Var}(a_k^{i-1}) \\ &= n\,\mathrm{Var}(\boldsymbol{W})\,\mathrm{Var}(a^{i-1}) \end{aligned} \tag{7.41}$$

由于 $\mathrm{Var}(\boldsymbol{a}_i) = \mathrm{Var}(\boldsymbol{a}_{i-1})$，故 $n\,\mathrm{Var}(\boldsymbol{W}) = 1$。于是考虑一边的情况下，$\mathrm{Var}(\boldsymbol{W}) = \frac{1}{n}$，$n$ 为输入神经元 (特征) 数目。同样考虑另一边，得到 $\mathrm{Var}(\boldsymbol{W}) = \frac{1}{m}$，$m$ 为输出神经元 (特征) 数目。我们综合考虑二者，得到 $\mathrm{Var}(\boldsymbol{W}) = \frac{2}{n+m}$。

- 正态分布的 Glorot 初始化 (Glorot Normal)：它是从以 0 为中心，标准差为 $std = \sqrt{\frac{2}{fan_{in}+fan_{out}}}$ 的截断正态分布中抽取样本，其中 fan_{in} 是权值张量中的输入单位的数量，fan_{out} 是权值张量中的输出单位的数量。
- 均匀分布的 Glorot 初始化 (Glorot Uniform)：它是从 $[-limit, limit]$ 中的均匀分布中抽取样本，其中 $limit = \sqrt{\frac{6}{fan_{in}+fan_{out}}}$ (将均匀分布的方差代入计算可得)，fan_{in} 是权值张量中的输入单位的数量，fan_{out} 是权值张量中的输出单位的数量。

3、Kaiming 初始化方法

Kaiming 初始化，也称为 He 初始化，也称之为 MSRA 初始化。前面的 Glorot 初始化要求了激活函数关于 0 对称，因此不适用于 ReLU。Kaiming 初始化的想法是一个好的初始化应该在各个层的状态值 $\boldsymbol{z}$ 的方差保持一致，即对于第 i 层和第 j 层，有 $\mathrm{Var}(z^i) = \mathrm{Var}(z^j)$。这边只假设初始化的均值为 0，即 $\mathbb{E}(\boldsymbol{W}) = 0$:

$$\begin{aligned}
\mathrm{Var}(z^i) &= n\,\mathrm{Var}(\boldsymbol{W}^\top a^{i-1}) \\
&= n\left[\mathbb{E}(\boldsymbol{W}^\top a^{i-1})^2 - (\mathbb{E}(\boldsymbol{W}^\top a^{i-1}))^2\right] \\
&= n\left[\mathbb{E}(\boldsymbol{W}^2)\mathbb{E}((a^{i-1})^2) - (\mathbb{E}(\boldsymbol{W}))^2(\mathbb{E}(a^{i-1}))^2\right] \\
&= n\,\mathbb{E}(\boldsymbol{W}^2)\mathbb{E}((a^{i-1})^2) \\
&= n\,\mathrm{Var}(\boldsymbol{W})\mathbb{E}((a^{i-1})^2)
\end{aligned} \tag{7.42}$$

而再计算 $\mathbb{E}((a^{i-1})^2)$，需要注意的是，由于 $\mathbb{E}(\boldsymbol{W}) = 0$ 且 $\boldsymbol{W}$ 与 a^{i-1} 独立，故 $\mathbb{E}(z^i) = 0$:

$$\begin{aligned}
\mathbb{E}(a^{i-1})^2 &= \mathbb{E}(g(z^{i-1}))^2 \\
&= \int_{-\infty}^{\infty} p(z^{i-1})(g(z^{i-1}))^2 dz^{i-1} \\
&= \int_{-\infty}^{0} p(z^{i-1})(g(z^{i-1}))^2 dz^{i-1} + \int_{0}^{\infty} p(z^{i-1})(g(z^{i-1}))^2 dz^{i-1} \\
&= 0 + \int_{0}^{\infty} p(z^{i-1})(z^{i-1})^2 dz^{i-1} \\
&= \frac{1}{2}\int_{-\infty}^{\infty} p(z^{i-1})(z^{i-1})^2 dz^{i-1} \\
&= \frac{1}{2}\mathbb{E}(z^{i-1})^2 \\
&= \frac{1}{2}\mathrm{Var}(z^{i-1})
\end{aligned} \tag{7.43}$$

于是我们得到 $\mathrm{Var}(z^i) = \frac{1}{2}n\,\mathrm{Var}(\boldsymbol{W})\,\mathrm{Var}(z^{i-1})$，由于 $\mathrm{Var}(z^i) = \mathrm{Var}(z^{i-1})$，可以得到 $\mathrm{Var}(\boldsymbol{W}) = \frac{2}{n}$。

- 正态分布的 Kaiming 初始化 (He Normal)：它是从以 0 为中心，标准差为 $std = \sqrt{\frac{2}{fan_{in}}}$ 的截断正态分布中抽取样本，其中 fan_{in} 是权值张量中的输入单位的数量。
- 均匀分布的 Kaiming 初始化 (He Uniform)：它是从 $[-limit, limit]$ 中的均匀分布中抽取样本，其中 $limit = \sqrt{\frac{6}{fan_{in}}}$，$fan_{in}$ 是权值张量中的输入单位的数量。

4、Batch Normalization 初始化方法

Batch Normalization 会在后文介绍，使用 BN 时，减少了网络对参数初始值尺度的依赖，此时使用较小的标准差 (如 0.01) 进行初始化即可。

5、Pre-Train 初始化方法

借助预训练模型中参数作为新任务参数初始化的方式是一种简便易行且十分有效的模型参数初始化方法，这也是目前的主流方法之一。

[17]:
```python
def calc_fan(weight_shape):
    """
    对权重矩阵计算 fan-in 和 fan-out

    参数说明:
    weight_shape: 权重形状
    """
    if len(weight_shape) == 2:  # 暂先考虑维数为 2
        fan_in, fan_out = weight_shape
    else:
        raise ValueError("Unrecognized weight dimension: {}".format(weight_shape))
    return fan_in, fan_out
```

[18]:
```python
def random_uniform(weight_shape, b=1.0):
    """
    初始化网络权重 W--- 基于 Uniform(-b, b)

    参数说明:
    weight_shape: 权重形状
```

```
    """
    return np.random.uniform(-b, b, size=weight_shape)
```

[19]:
```
def random_normal(weight_shape, std=1.0):
    """
    初始化网络权重 W--- 基于 TruncatedNormal(0, std)

    参数说明:
    weight_shape: 权重形状
    std: 权重标准差
    """
    return truncated_normal(0, std, weight_shape)
```

[20]:
```
def he_uniform(weight_shape):
    """
    初始化网络权重 W--- 基于 Uniform(-b, b)，其中 b=sqrt(6/fan_in)，常用于 ReLU 激活层

    参数说明:
    weight_shape: 权重形状
    """
    fan_in, fan_out = calc_fan(weight_shape)
    b = np.sqrt(6 / fan_in)
    return np.random.uniform(-b, b, size=weight_shape)
```

[21]:
```
def he_normal(weight_shape):
    """
    初始化网络权重 W--- 基于 TruncatedNormal(0, std)，其中 std=2/fan_in，常用于 ReLU 激活层

    参数说明:
    weight_shape: 权重形状
    """
    fan_in, fan_out = calc_fan(weight_shape)
    std = np.sqrt(2 / fan_in)
    return truncated_normal(0, std, weight_shape)
```

[22]:
```
def glorot_uniform(weight_shape, gain=1.0):
    """
    初始化网络权重 W--- 基于 Uniform(-b, b)，其中 b=gain*sqrt(6/(fan_in+fan_out))，
                        常用于 tanh 和 sigmoid 激活层

    参数说明:
    weight_shape: 权重形状
    """
    fan_in, fan_out = calc_fan(weight_shape)
    b = gain * np.sqrt(6 / (fan_in + fan_out))
    return np.random.uniform(-b, b, size=weight_shape)
```

[23]:
```
def glorot_normal(weight_shape, gain=1.0):
    """
    初始化网络权重 W--- 基于 TruncatedNormal(0, std)，其中 std=gain^2*2/(fan_in+fan_out)，
                        常用于 tanh 和 sigmoid 激活层

    参数说明:
    weight_shape: 权重形状
    """
    fan_in, fan_out = calc_fan(weight_shape)
    std = gain * np.sqrt(2 / (fan_in + fan_out))
    return truncated_normal(0, std, weight_shape)
```

[24]:
```python
def truncated_normal(mean, std, out_shape):
    """
    通过拒绝采样生成截断正态分布

    参数说明:
    mean: 正态分布均值
    std: 正态分布标准差
    out_shape: 矩阵形状
    """
    samples = np.random.normal(loc=mean, scale=std, size=out_shape)
    reject = np.logical_or(samples >= mean + 2 * std, samples <= mean - 2 * std)
    while any(reject.flatten()):
        resamples = np.random.normal(loc=mean, scale=std, size=reject.sum())
        samples[reject] = resamples
        reject = np.logical_or(samples >= mean + 2 * std, samples <= mean - 2 * std)
    return samples
```

7.3 批标准化

批标准化 (Batch Normalization) 并不是一个优化算法，而是一个**自适应的重参数化**的方法，试图解决训练**非常深的模型**的困难。

BatchNorm 是基于 SGD 的，因为**深层神经网络**在**非线性变换前**的激活输入值随着网络深度加深或者在训练过程中，其**分布逐渐发生偏移或者变动** (Covariate Shift)。之所以训练收敛慢，一般是**整体分布**逐渐往**非线性函数的取值区间的上下限两端靠近** (如对于 sigmoid 函数来说，激活输入值是大的负值或正值)，所以这导致反向传播时低层神经网络的**梯度消失**。

BatchNorm 是通过一定的规范化手段，把每层神经网络任意神经元这个输入值的分布强行拉回到**均值为 0、方差为 1 的标准正态分布**，把越来越偏的分布拉回标准分布，这样使得激活输入值落在**非线性函数对输入比较敏感的区域**，避免梯度消失问题产生，梯度变大使得学习收敛速度快。

BatchNorm 训练阶段的实现包括两个步骤：

- 对于某个神经元，如果 Batch 为 m 个样本，首先计算在当前神经元处的均值 μ 与方差 σ。再对数据进行规范化，使得输入每个特征的分布均值为 0，方差为 1。再将输入经过非线性函数。
- 步骤一让每一层网络的输入数据分布都变得稳定，但却导致了数据表达能力的缺失，因为通过变换操作改变了原有数据的信息表达。因此，引入参数 scale γ 和 offset β，再对规范化后的数据进行线性变换，**恢复数据本身的表达能力**。

具体如下：

$$
\begin{aligned}
\mu &\leftarrow \frac{1}{m}\sum_{i=1}^{m}\boldsymbol{x}^{(i)} \\
\sigma^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{x}^{(i)}-\mu)^2 \\
\hat{\boldsymbol{x}}^{(i)} &\leftarrow \frac{\boldsymbol{x}^{(i)}-\mu}{\sqrt{\sigma^2+\epsilon}} \\
\hat{\boldsymbol{z}}^{(i)} &\leftarrow \gamma\hat{\boldsymbol{x}}^{(i)}+\beta \\
\boldsymbol{a}^{(i)} &\leftarrow f(\hat{\boldsymbol{z}}^{(i)})
\end{aligned}
\tag{7.44}
$$

测试阶段使用训练阶段**整个样本的统计量**来对测试数据归一化。训练阶段保留每组 Batch 的 μ_{batch} 和 σ_{batch}，于是可以得到均值与方差的**无偏估计**。如下：

$$
\begin{aligned}
\mu &\leftarrow \mathbb{E}(\mu_{batch}) \\
\sigma^2 &\leftarrow \frac{m}{m-1}\mathbb{E}(\sigma^2_{batch})
\end{aligned}
\tag{7.45}
$$

反向传播过程：

- 首先，可以计算得出：

$$
\frac{\partial J}{\partial \gamma} = \frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}}\cdot\frac{\partial \hat{\boldsymbol{z}}^{(i)}}{\partial \gamma} = \boxed{\sum_{i=1}^{m}\frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}}\cdot\hat{\boldsymbol{x}}^{(i)}} \tag{7.46}
$$

$$
\frac{\partial J}{\partial \beta} = \frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}}\cdot\frac{\partial \hat{\boldsymbol{z}}^{(i)}}{\partial \beta} = \boxed{\sum_{i=1}^{m}\frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}}} \tag{7.47}
$$

$$
\frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} = \frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}}\cdot\frac{\partial \hat{\boldsymbol{z}}^{(i)}}{\partial \hat{\boldsymbol{x}}^{(i)}} = \boxed{\frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}}\cdot\gamma} \tag{7.48}
$$

然后，根据

$$\begin{cases} \dfrac{\partial \hat{\boldsymbol{x}}^{(i)}}{\partial \mu} = \dfrac{1}{\sqrt{\sigma^2+\epsilon}} \cdot (-1) \\ \dfrac{\partial \sigma^2}{\partial \mu} = \dfrac{1}{m}\displaystyle\sum_{i=1}^{m} 2 \cdot (\boldsymbol{x}^{(i)} - \mu) \cdot (-1) \\ \dfrac{\partial \hat{\boldsymbol{x}}}{\partial \sigma^2} = \displaystyle\sum_{i=1}^{m} (\boldsymbol{x}^{(i)} - \mu) \cdot (-0.5) \cdot (\sigma^2+\epsilon)^{-0.5-1} \end{cases} \tag{7.49}$$

可以得到：

$$\frac{\partial J}{\partial \sigma^2} = \boxed{\frac{\partial J}{\partial \hat{\boldsymbol{x}}} \cdot \frac{\partial \hat{\boldsymbol{x}}}{\partial \sigma^2}} \tag{7.50}$$

以及：

$$\begin{aligned} \frac{\partial J}{\partial \mu} =& \left(\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{-1}{\sqrt{\sigma^2+\epsilon}} \right) + \left(\frac{\partial J}{\partial \sigma^2} \cdot \frac{1}{m} \sum_{i=1}^{m} -2(\boldsymbol{x}^{(i)} - \mu) \right) \\ =& \left(\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{-1}{\sqrt{\sigma^2+\epsilon}} \right) + \left(\frac{\partial J}{\partial \sigma^2} \cdot (-2) \cdot \left(\frac{1}{m} \sum_{i=1}^{m} \boldsymbol{x}^{(i)} - \frac{1}{m} \sum_{i=1}^{m} \mu \right) \right) \\ =& \left(\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{-1}{\sqrt{\sigma^2+\epsilon}} \right) + \left(\frac{\partial J}{\partial \sigma^2} \cdot (-2) \cdot \left(\mu - \frac{m \cdot \mu}{m} \right) \right) \\ =& \boxed{\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{-1}{\sqrt{\sigma^2+\epsilon}}} \end{aligned} \tag{7.51}$$

再根据

$$\begin{cases} \dfrac{\partial \hat{\boldsymbol{x}}^{(i)}}{\partial \boldsymbol{x}^{(i)}} = \dfrac{1}{\sqrt{\sigma^2+\epsilon}} \\ \dfrac{\partial \mu}{\partial \boldsymbol{x}^{(i)}} = \dfrac{1}{m} \\ \dfrac{\partial \sigma^2}{\partial \boldsymbol{x}^{(i)}} = \dfrac{2(\boldsymbol{x}^{(i)} - \mu)}{m} \end{cases} \tag{7.52}$$

计算得出：

$$\begin{aligned} \frac{\partial J}{\partial \boldsymbol{x}^{(i)}} =& \left(\frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{1}{\sqrt{\sigma^2+\epsilon}} \right) + \left(\frac{\partial J}{\partial \mu} \cdot \frac{1}{m} \right) + \left(\frac{\partial J}{\partial \sigma^2} \cdot \frac{2(\boldsymbol{x}^{(i)} - \mu)}{m} \right) \\ =& \left(\frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{1}{\sqrt{\sigma^2+\epsilon}} \right) + \left(\frac{1}{m} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \cdot \frac{-1}{\sqrt{\sigma^2+\epsilon}} \right) - \left(0.5 \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \cdot (\boldsymbol{x}^{(j)} - \mu) \cdot (\sigma^2+\epsilon)^{-1.5} \cdot \frac{2(\boldsymbol{x}^{(i)} - \mu)}{m} \right) \\ =& \left(\frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot (\sigma^2+\epsilon)^{-0.5} \right) - \left(\frac{(\sigma^2+\epsilon)^{-0.5}}{m} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \right) - \left(\frac{(\sigma^2+\epsilon)^{-0.5}}{m} \cdot \frac{\boldsymbol{x}^{(i)} - \mu}{\sqrt{\sigma^2+\epsilon}} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \cdot \frac{(\boldsymbol{x}^{(j)} - \mu)}{\sqrt{\sigma^2+\epsilon}} \right) \\ =& \left(\frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot (\sigma^2+\epsilon)^{-0.5} \right) - \left(\frac{(\sigma^2+\epsilon)^{-0.5}}{m} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \right) - \left(\frac{(\sigma^2+\epsilon)^{-0.5}}{m} \cdot \hat{\boldsymbol{x}}^{(i)} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \cdot \hat{\boldsymbol{x}}^{(j)} \right) \\ =& \boxed{\frac{m \dfrac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} - \displaystyle\sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} - \hat{\boldsymbol{x}}^{(i)} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \cdot \hat{\boldsymbol{x}}^{(j)}}{m\sqrt{\sigma^2+\epsilon}}} \end{aligned} \tag{7.53}$$

[25]:
```
from chapter6 import LayerBase, CrossEntropy, OrderedDict, softmax, FullyConnected
```

[26]:
```
class BatchNorm1D(LayerBase):

    def __init__(self, momentum=0.9, epsilon=1e-5, optimizer=None):
        """
        参数说明:
        momentum: 动量项，越趋于 1 表示对当前 Batch 的依赖程度越小，running_mean 和 running_var 的计算越平滑
                  float 型 (default: 0.9)

        epsilon: 避免除数为 0, float 型 (default : 1e-5)
        optimizer: 优化器
        """
        super().__init__(optimizer)
        self.n_in = None
```

```
        self.n_out = None
        self.epsilon = epsilon
        self.momentum = momentum
        self.params = {
            "scaler": None,
            "intercept": None,
            "running_var": None,
            "running_mean": None,
        }
        self.is_initialized = False

    def _init_params(self):
        scaler = np.random.rand(self.n_in)
        intercept = np.zeros(self.n_in)
        running_mean = np.zeros(self.n_in)
        running_var = np.ones(self.n_in)
        self.params = {
            "scaler": scaler,
            "intercept": intercept,
            "running_mean": running_mean,
            "running_var": running_var,
        }
        self.gradients = {
            "scaler": np.zeros_like(scaler),
            "intercept": np.zeros_like(intercept),
        }
        self.is_initialized = True

    def reset_running_stats(self):
        self.params["running_mean"] = np.zeros(self.n_in)
        self.params["running_var"] = np.ones(self.n_in)

    def forward(self, X, is_train=True, retain_derived=True):
        """
        Batch 训练时 BN 的前向传播，原理见上文。

        [train]: Y = scaler * norm(X) + intercept, 其中 norm(X) = (X - mean(X)) / sqrt(var(X) + epsilon)

        [test]: Y = scaler * running_norm(X) + intercept,
                    其中 running_norm(X) = (X - running_mean) / sqrt(running_var + epsilon)

        参数说明:
        X: 输入数组，为 (n_samples, n_in)，float 型
        is_train: 是否为训练阶段，bool 型
        retain_derived: 是否保留中间变量，以便反向传播时再次使用，bool 型
        """
        if not self.is_initialized:
            self.n_in = self.n_out = X.shape[1]
            self._init_params()
        epsi, momentum = self.hyperparams["epsilon"], self.hyperparams["momentum"]
        rm, rv = self.params["running_mean"], self.params["running_var"]
        scaler, intercept = self.params["scaler"], self.params["intercept"]
        X_mean, X_var = self.params["running_mean"], self.params["running_var"]
        if is_train and retain_derived:
            X_mean, X_var = X.mean(axis=0), X.var(axis=0)
            self.params["running_mean"] = momentum * rm + (1.0 - momentum) * X_mean
            self.params["running_var"] = momentum * rv + (1.0 - momentum) * X_var
        if retain_derived:
            self.X.append(X)
```

```
        X_hat = (X - X_mean) / np.sqrt(X_var + epsi)
        y = scaler * X_hat + intercept
        return y

    def backward(self, dLda, retain_grads=True):
        """
        BN 的反向传播，原理见上文。

        参数说明：
        dLda：关于损失的梯度，为 (n_samples, n_out)，float 型
        retain_grads：是否计算中间变量的参数梯度，bool 型
        """
        if not isinstance(dLda, list):
            dLda = [dLda]
        dX = []
        X = self.X
        for da, x in zip(dLda, X):
            dx, dScaler, dIntercept = self._bwd(da, x)
            dX.append(dx)
            if retain_grads:
                self.gradients["scaler"] += dScaler
                self.gradients["intercept"] += dIntercept
        return dX[0] if len(X) == 1 else dX

    def _bwd(self, dLda, X):
        scaler = self.params["scaler"]
        epsi = self.hyperparams["epsilon"]
        n_ex, n_in = X.shape
        X_mean, X_var = X.mean(axis=0), X.var(axis=0)
        X_hat = (X - X_mean) / np.sqrt(X_var + epsi)
        dIntercept = dLda.sum(axis=0)
        dScaler = np.sum(dLda * X_hat, axis=0)
        dX_hat = dLda * scaler
        dX = (n_ex * dX_hat - dX_hat.sum(axis=0) - X_hat * (dX_hat * X_hat).sum(axis=0)) / (
            n_ex * np.sqrt(X_var + epsi)
        )
        return dX, dScaler, dIntercept

    @property
    def hyperparams(self):
        return {
            "layer": "BatchNorm1D",
            "acti_fn": None,
            "n_in": self.n_in,
            "n_out": self.n_out,
            "epsilon": self.epsilon,
            "momentum": self.momentum,
            "optimizer": {
                "cache": self.optimizer.cache,
                "hyperparams": self.optimizer.hyperparams,
            },
        }
```

引入 BatchNorm，MNIST 数据集测试

[27]:
```
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
```

```
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1]*X_train.shape[2]).astype('float32')
X_test = X_test.reshape(-1, X_test.shape[1]*X_test.shape[2]).astype('float32')
print(X_train.shape, y_train.shape)
N = 20000   # 取 20000 条数据用以训练
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 784) (60000, 10)
(20000, 784) (20000, 10)
```

[28]:
```
def minibatch(X, batchsize=256, shuffle=True):
    """
    函数作用：将数据集分割成 batch， 基于 mini batch 训练。
    """
    N = X.shape[0]
    idx = np.arange(N)
    n_batches = int(np.ceil(N / batchsize))
    if shuffle:
        np.random.shuffle(idx)
    def mb_generator():
        for i in range(n_batches):
            yield idx[i * batchsize : (i + 1) * batchsize]
    return mb_generator(), n_batches


class DFN(object):

    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=0.01)",
        init_w="std_normal",
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用：模型初始化
        FC1 -> Sigmoid -> BN -> FC2 -> Softmax
        """
        self.layers = OrderedDict()
```

```
        self.layers["FC1"] = FullyConnected(
            n_out=self.hidden_dims_1,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["BN"] = BatchNorm1D(optimizer=self.optimizer)
        self.layers["FC2"] = FullyConnected(
            n_out=self.hidden_dims_2,
            acti_fn="affine(slope=1, intercept=0)",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.is_initialized = True

    def forward(self, X_train, is_train=True):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            try:  # 考虑 BN
                out = v.forward(out, is_train=is_train)
            except:
                out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        verbose：是否每个 batch 输出损失
        """
        self.verbose = verbose
```

```
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch, is_train=True)
                y_pred_batch = softmax(out)
                batch_loss = self.loss(y_batch, y_pred_batch)
                grad = self.loss.grad(y_batch, y_pred_batch)
                _, _ = self.backward(grad)
                self.update()
                loss += batch_loss
                if self.verbose:
                    fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                    print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))
            loss /= n_batch
            fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
            print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
            prev_loss = loss

    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch, is_train=False)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

[29]:
```
model = DFN(hidden_dims_1=200, hidden_dims_2=10)
model.fit(X_train, y_train, n_epochs=20, batch_size=64)
```

```
[Epoch 1] Avg. loss: 0.979  Delta: inf (0.01m/epoch)
[Epoch 2] Avg. loss: 0.473  Delta: 0.506 (0.01m/epoch)
[Epoch 3] Avg. loss: 0.395  Delta: 0.078 (0.01m/epoch)
[Epoch 4] Avg. loss: 0.362  Delta: 0.033 (0.01m/epoch)
[Epoch 5] Avg. loss: 0.342  Delta: 0.019 (0.02m/epoch)
[Epoch 6] Avg. loss: 0.326  Delta: 0.016 (0.01m/epoch)
```

```
[Epoch 7] Avg. loss: 0.315  Delta: 0.011 (0.02m/epoch)
[Epoch 8] Avg. loss: 0.307  Delta: 0.008 (0.01m/epoch)
[Epoch 9] Avg. loss: 0.299  Delta: 0.008 (0.01m/epoch)
[Epoch 10] Avg. loss: 0.292  Delta: 0.008 (0.02m/epoch)
[Epoch 11] Avg. loss: 0.287  Delta: 0.005 (0.02m/epoch)
[Epoch 12] Avg. loss: 0.280  Delta: 0.007 (0.02m/epoch)
[Epoch 13] Avg. loss: 0.275  Delta: 0.005 (0.01m/epoch)
[Epoch 14] Avg. loss: 0.270  Delta: 0.005 (0.01m/epoch)
[Epoch 15] Avg. loss: 0.262  Delta: 0.008 (0.02m/epoch)
[Epoch 16] Avg. loss: 0.254  Delta: 0.008 (0.02m/epoch)
[Epoch 17] Avg. loss: 0.249  Delta: 0.005 (0.01m/epoch)
[Epoch 18] Avg. loss: 0.244  Delta: 0.005 (0.02m/epoch)
[Epoch 19] Avg. loss: 0.234  Delta: 0.010 (0.01m/epoch)
[Epoch 20] Avg. loss: 0.232  Delta: 0.002 (0.01m/epoch)
```

[30]:
```
print("accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
accuracy:0.9258
```

7.4 坐标下降

坐标下降 (Coordinate Descent) 是一种非梯度优化算法。算法在每次迭代中，在当前点处**沿一个坐标方向**进行**一维搜索**以求得一个函数的局部极小值。在整个过程中**循环使用不同的坐标方向**。如我们相对某个单一变量 x_i 最小化 $f(x)$，然后再相对另一个变量 x_j 计算，反复循环所有的变量，会保证到达 (局部) 极小值。这种做法被称为坐标下降，因为我们**一次优化一个坐标**。

对于不可拆分的函数而言，算法可能无法在较小的迭代步数中求得最优解。如果当**一个变量的值很大程度地影响另一个变量的最优值**时，坐标下降不是一个很好的方法。但在**稀疏矩阵上**的计算速度非常快，同时也是 Lasso 回归最快的解法。

7.5 Polyak 平均

Polyak 平均会平均优化算法在参数空间访问轨迹中的几个点。如果 t 次迭代梯度下降访问了点 $\theta^{(1)} \dots \theta^{(t)}$，那么 Polyak 平均算法输出的是：

$$\hat{\theta}^{(t)} = \frac{1}{t} \sum_i \theta^{(i)} \tag{7.54}$$

在梯度下降应用于某些问题，比如凸问题时，这种方法具有较强的收敛保证。当 Polyak 平均于**非凸问题时**，通常会用**指数衰减计算平均值**：

$$\hat{\theta}^{(t)} = \alpha \hat{\theta}^{(t-1)} + (1 - \alpha)\theta^{(t)} \tag{7.55}$$

滑动平均 (Exponential Moving Average)，或者叫做指数加权平均，可以用来估计变量的局部均值，使得**变量的更新与一段时间内的历史取值**有关。

对于 SGD，在**训练过程**仍然使用原来**不带滑动平均的权重**，在**测试过程**中使用**带滑动平均的权重**作为神经网络的权重，这样在测试数据上效果更好，带滑动平均的权重更新更加平滑。

7.6 监督预训练

如果模型太复杂难以优化，或是如果**任务非常困难**，直接训练模型来解决特定任务的挑战可能太大。**训练模型来求解一个简化的问题**，然后**转移**到最后的问题，有时也会更有效些。这些在直接训练目标模型求解目标问题之前，训练简单模型求解简化问题的方法统称为**预训练**。

贪心算法 (Greedy Algorithm) 将**问题分解成许多部分**，然后**独立地在每个部分求解最优值**。结合各个最佳的部分不能保证得到一个最佳的完整解，但贪心算法计算上比求解最优联合解的算法高效得多，并且贪心算法的解在不是最优的情况下，往往也是可以接受的。贪心算法也可以紧接一个**精调** (fine-tuning) 阶段，联合优化算法搜索全问题的最优解。使用贪心解初始化联合优化算法，可以极大地加速算法，并提高寻找到的解的质量。

一个与监督预训练有关的方法扩展了**迁移学习的想法**，另一条相关的工作线是 **FitNets 方法**。

7.7 设计有助于优化的模型

改进优化的最好方法并不总是改进优化算法，相反，在深度学习中的许多改进来自设计易于优化的模型。在实践中，**选择一族容易优化的模型**比使用一个强大的优化算法重要。

现代神经网络的设计选择体现在**层之间的线性变换**，**几乎处处可导的激活函数**，和**大部分定义域都有明显的梯度**，特别是**创新的模型**，比如 LSTM，整流线性单元和 maxout 单元都比先前的模型，比如 sigmoid 单元的深度网络，使用更多的线性函数，使得这些模型都有简化优化的作用。现代神经网络的设计方案旨在使其**局部梯度信息合理地对应着移动向一个遥远的解**。

[32]:
```
import numpy

print("numpy:", numpy.__version__)
```

```
numpy: 1.14.5
```

第八章　卷积网络

卷积网络是指那些至少在网络的一层中使用**卷积运算**来**替代一般的矩阵乘法运算**的神经网络。

8.1　卷积运算

卷积操作是指对**不同的数据窗口数据** (图像) 用**一组固定的权重** (滤波矩阵，可以看做一个恒定的滤波器 `Filter`) **逐个元素相乘再求和** (做内积)。

也可以用如下方式理解，如果想要低噪声估计，一种可行的方法是**对得到的测量结果进行平均**。可以认为**时间上越近**的测量**结果越相关**，所以采用一种加权平均的方法，对于**最近的测量结果赋予更高的权重**。

采用一个加权函数 $w(a)$ 来实现，其中 a 表示测量结果距当前时刻的时间间隔：$s(t)=\int x(a)w(t-a)da$。这就是卷积操作，可以用星号表示：$s(t)=(x*w)(t)$。其中 x 是输入 (`Input`)；w 是核 (`Kernel` 或 `Filter`)；输出的 s 是特征映射或特征图 (`Feature Map`) 或称输出 (`Output`)。

二维卷积运算

上面我们介绍了卷积运算的思想，但我们通常在卷积层中会使用更加直观的互相关 (`Cross-correlation`) 运算。例如，在二维卷积层中，一个二维输入数组 I 和一个二维核数组 K 通过互相关运算输出一个二维数组 O。如图所示，输入是一个高和宽均为 3 的二维数组，核数组 (也称卷积核或过滤器) 的高和宽分别为 2。因此，我们可以说输入数组形状为 $(3,3)$，卷积核窗口 (又称卷积窗口) 的形状为 $(2,2)$。

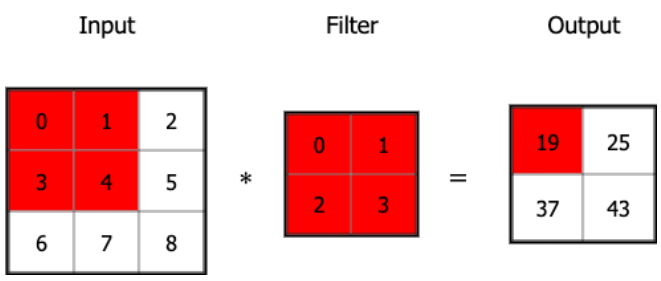


图 8.1.　二维卷积运算。输入的红色区域与核的红色区域通过运算得到输出的左上角红色区域。

红色区域显示了第一个输出元素的计算过程：$0\times0+1\times1+3\times2+4\times3=19$。

其他输出元素的计算过程同样可得：

$$\begin{aligned}1\times0+2\times1+4\times2+5\times3&=25,\\3\times0+4\times1+6\times2+7\times3&=37,\\4\times0+5\times1+7\times2+8\times3&=43.\end{aligned}\tag{8.1}$$

卷积背后的直觉

我们将卷积操作方式展开成前馈网络的形式，如图 8.2 所示，左图为原本的前馈网络，右图为卷积操作。在第六章介绍的深度前馈网络中，权重矩阵中的**每一个元素只会使用一次**，不会在不同的输入位置上共享参数。而在卷积操作中，卷积核的**每一个元素都作用在输入的每一位置**上。这个设计保证了我们只需要学习一个参数集合，而不用对每一位置去学习一个单独的参数。

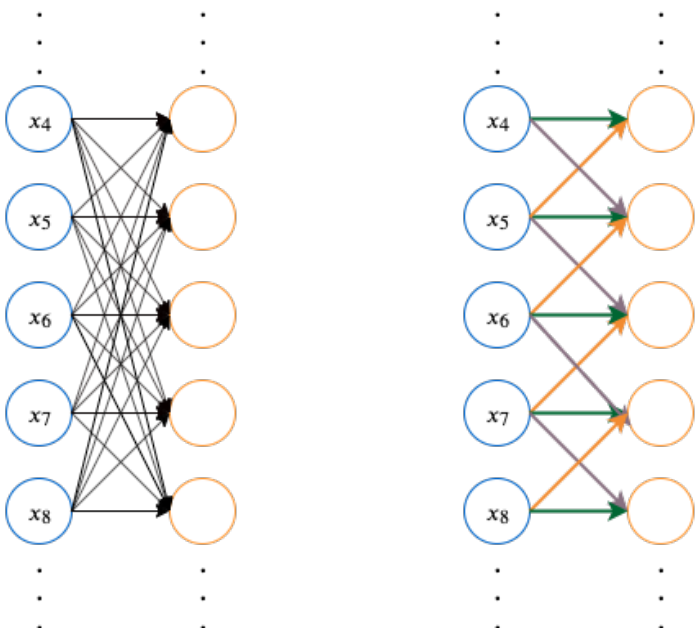


图 8.2. 卷积操作展开为深度前馈网络形式 (见第六章)，左图为前馈网络，右图为卷积网络展开。

这便是卷积的直觉，稀疏交互 (`Sparse Interactions`) 与参数共享 (`Parameter Sharing`)。前馈网络中每一个输出单元与每一个输入单元都产生交互，如果有 m 个输入和 n 个输出，那么矩阵乘法需要 $m \times n$ 个参数并且相应算法的时间复杂度为 $O(m \times n)$；卷积操作中如果我们限制每一个输出拥有的连接数为 k，那么稀疏的连接方法只需要 $k \times n$ 个参数以及 $O(k \times n)$ 的运行时间。

8.2 池化

在得到卷积特征后，下一步就是用来做分类。理论上，可以用提取到的所有特征训练分类器，但用所有特征的计算量开销会很大，而且也会使分类器倾向于过拟合。

为了解决这个问题，考虑到要描述一个大图像，一个自然的方法是在不同位置处对特征进行汇总统计。例如，可以计算在图像中某一区域的一个特定特征的平均值 (或最大值)，这样概括统计出来的数据，其规模 (相比使用提取到的所有特征) 就低得多，同时也可以改进分类结果 (使模型不易过拟合)。这样的聚集操作称为 “池化”。池化可以保留显著特征，降低特征规模。

池化运算

池化函数是使用某一位置的**相邻输出的总体统计特征**来代替网络在该位置的输出。

- **最大池化函数** (`Max Pooling`) 给出相邻矩形区域内的最大值；
- **平均池化** (`Average Pooling`) 计算相邻矩形区域内的平均值；
- 其他常用的池化函数包括 L^2 范数以及基于距中心像素距离的加权平均函数。

如图 8.3 所示，我们展示最大池化的运算过程：

$$\begin{aligned} \max(0,1,3,4) &= 4, \\ \max(2,0,5,0) &= 5, \\ \max(6,7,0,0) &= 7, \\ \max(8,0,0,0) &= 8. \end{aligned} \tag{8.2}$$

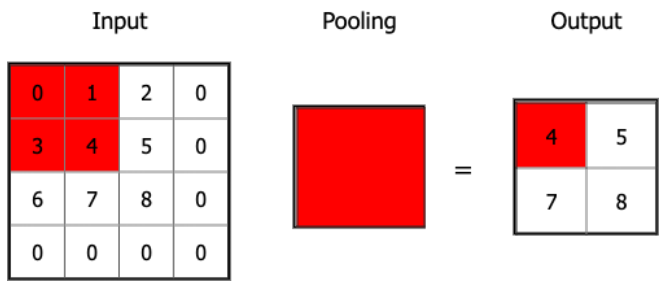


图 8.3. 最大池化运算。输入的红色区域经过池化得到输出的左上角红色区域。

不管采用什么样的池化函数，当输入作出少量平移时，池化能够帮助输入的表示近似不变。平移的不变性是指当我们对输入进行少量平移时，经过池化函数后的大多数输出并不会发生改变。**局部平移不变性**是一个很有用的性质，**尤其是当我们关心某个特征是否出现而不关心它出现的具体位置时**，后文我们会对这一性质进一步描述。

卷积与池化作为一种无限强的先验：

先验的强或者弱取决于先验中概率密度的集中程度：弱先验具有较高的熵值，例如方差很大的高斯分布，这样的先验允许数据对于参数的改变具有或多或少的自由性；强先验具有较低的熵值，例如方差很小的高斯分布，这样的先验在决定参数最终取值时起着更加积极的作用；无限强的先验需

要对一些参数的概率置零并且完全禁止对这些参数赋值，无论数据对于这些参数的值给出了多大的支持。因此，我们可以把卷积的使用当作是对网络中一层的参数引入了一个无限强的先验概率分布，要求该层学到的函数只能包含局部连接关系并且对平移具有等变性。

8.3 深度学习框架下的卷积：

8.3.1 多个并行卷积

通常指由多个并行卷积组成的运算，可以在每个位置提取多种类型的特征。

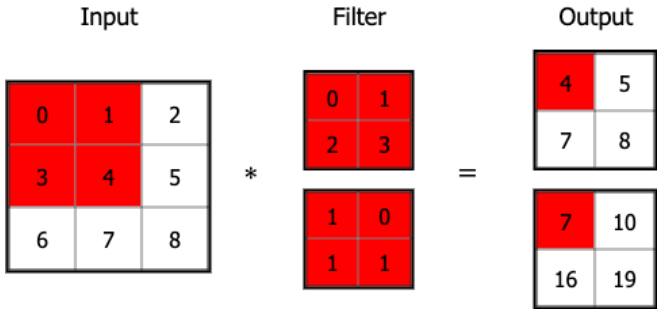


图 8.4. 多个并行卷积。输入的红色区域分别与两个核经过计算分别得到输出的左上角红色区域。

如图 8.4 所示，我们可以用两组卷积核去提取不同的特征。

8.3.2 输入值与核

输入通常也不仅仅是实值的网格，而是由一系列向量的网格，如一幅彩色图像在每一个像素点都会有红绿蓝三种颜色的亮度。当处理图像时，通常把卷积的输入输出都看作是 3 维的张量，其中一个索引用于标明不同的通道 (例如 RGB)，另外两个索引标明在每个通道上的空间坐标。我们可以将输入图像数组写作 V，它的每一个元素是 $\mathsf{V}_{i,j,k}$，表示处在通道 k 中第 i 行第 j 列的值。如下图所示，V 有 3 个通道，每个通道的形状均为 $(3,3)$。此时，我们还要定义卷积核数组 K，它的每一个元素是 $\mathsf{K}_{i,j,k,l}$，表示处在第 l 组卷积核通道 k 中第 i 行第 j 列的值。如图 8.5 所示，我们假设卷积核窗口的形状为 $(2,2)$，因为输入通道为 3，于是我们要对每个通道都定义一个卷积核，同时我们构造两组卷积核实现并行卷积，所以 K 的形状为 $(2,2,3,2)$。

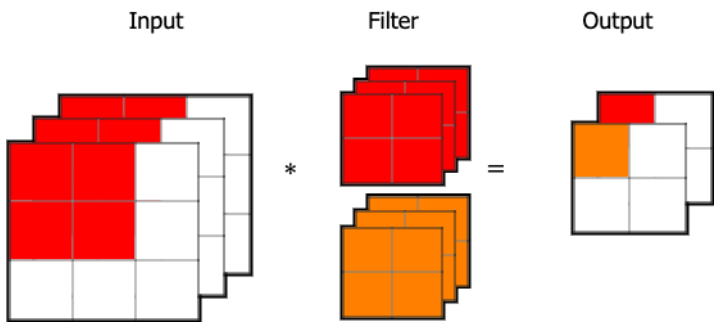


图 8.5. 输入数组的通道数为 3 (如图像数据)。用两组卷积核 (红色、橙色) 学习特征，而每组卷积核与输入运算再得到输出。

我们再看一下是如何获得输出 Z 值的，我们用 1 组卷积核，并且以输入为 2 通道为例，如下图 8.6，$\mathsf{Z}_{i,j,l} = \sum_{m,n,k} \mathsf{V}_{i+m,j+n,k}\mathsf{K}_{m,n,k,l}$。

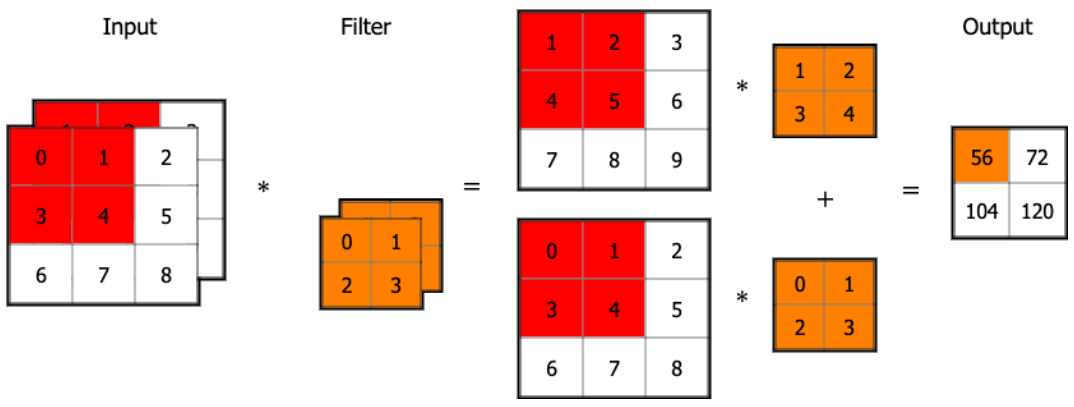


图 8.6. 输入数组的通道数为 2。展示用一组卷积核学习特征，卷积核的每个通道与输入的对应通道进行运算。再将各个通道的结果相加得到输出。

8.3.3 填充

假设输入图片的大小为 (m,n)，而卷积核的大小为 (f,f)，则卷积后的输出图片大小为 $(m-f+1, n-f+1)$，由此带来两个问题：

- 每次卷积运算后，输出图片的尺寸缩小。
- 原始图片的角落、边缘区像素点在输出中采用较少，输出图片丢失很多边缘位置的信息。

因此可以在进行卷积操作前，对原始图片在边界上进行填充 (Padding)，以增加矩阵的大小，通常将 0 作为填充值。

设每个方向扩展像素点数量为 p，则填充后原始图片的大小为 $(m+2p, n+2p)$，卷积核大小保持 (f, f) 不变，则输出图片大小 $(m+2p-f+1, n+2p-f+1)$。常用填充的方法有：

- 有效 (valid) 卷积，不填充，直接卷积，结果大小为 $(m-f+1, n-f+1)$。
- 相同 (same) 卷积，用 0 填充，并使得卷积后结果大小与输入一致，这样 $p=(f-1)/2$。
- 全 (full) 卷积，通过填充，使得输出尺寸为 $(m+f-1, n+f-1)$。

我们将之前的例子用 $p=1$ 填充，如图 8.7 所示。

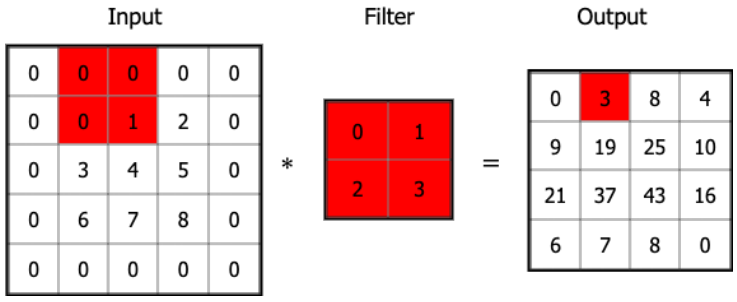


图 8.7. $p=1$ 填充，将 $(3,3)$ 的输入填充为 $(5,5)$。再与核运算得到输出。

8.3.4 卷积步幅

除了需要通过填充来避免信息损失，有时也需要通过设置步幅 (Stride) 来压缩一部分信息。步幅表示核在原始图片的水平方向和垂直方向上每次移动的距离。使用卷积步幅，跳过核中的一些位置 (看作对输出的下采样) 来降低计算的开销。如图 8.8 所示，在高上步幅为 3、在宽上步幅为 2 的卷积操作。当输出第一列第二个元素时，卷积窗口向下滑动了 3 行，而在输出第一行第二个元素时卷积窗口向右滑动了 2 列。通常我们设置在水平方向和垂直方向的步幅一样，如果步幅设为 s，则输出尺寸为 $(\lfloor\frac{m+2p-f}{s}+1\rfloor, \lfloor\frac{n+2p-f}{s}+1\rfloor)$。

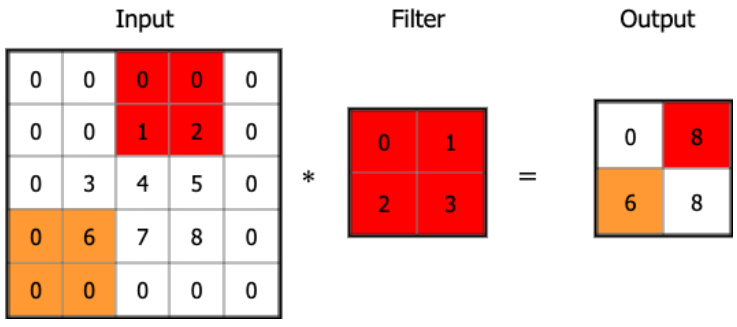


图 8.8. 考虑步幅下的卷积运算，在高上步幅为 3、在宽上步幅为 2。核第二步与输入的红色区域运算得到输出的右上角红色区域，第三步与输入的橙色区域运算得到输出的左下角橙色区域。

8.4 更多的卷积策略

8.4.1 深度可分离卷积

如图所示，对输入 3×3×2 的数组，经过 4 组 2×2×2 的卷积核，得到 2×2×4 的输出。这里我们一共需要 2×2×2×4=32 个参数。我们使用深度可分离卷积 (Depthwise Separable Convolution)，它将卷积过程分成两个步骤。第一步，在 Depthwise Convolution，输入有几个通道就设几个卷积核，如图 8.9 所示，输入一共 2 个通道，对每个通道分配一个卷积核，这里的每个卷积核只处理一个通道 (对比原始卷积过程每组卷积核处理所有通道)；第二步，在 Pointwise Convolution，由于在上一步不同通道间没有联系，因此这一步用 1×1 的卷积核组来获得不同通道间的联系。我们可以看出，在深度可分离卷积中，我们一共需要 2×2×2+1×1×2×4=16 个参数。

更形式化的，我们假设：

- 输入尺寸：$(\mathrm{H}_{in}, \mathrm{W}_{in}, c_1)$
- 卷积核尺寸：$(\mathrm{K}, \mathrm{K}, c_1)$
- 输出尺寸：$(\mathrm{H}_{out}, \mathrm{W}_{out}, c_2)$

我们需要的标准卷积核参数量为 $\mathrm{K}\times\mathrm{K}\times c_1\times c_2$。在深度可分离卷积中，第一步需要的参数量为 $\mathrm{K}\times\mathrm{K}\times 1\times c_1$，第二步需要的参数为 $1\times 1\times c_1\times c_2$，一共 $\mathrm{K}\times\mathrm{K}\times 1\times c_1+1\times 1\times c_1\times c_2$。所以深度可分离卷积参数量是标准卷积的 $\frac{1}{c_2}+\frac{1}{\mathrm{K}^2}$。

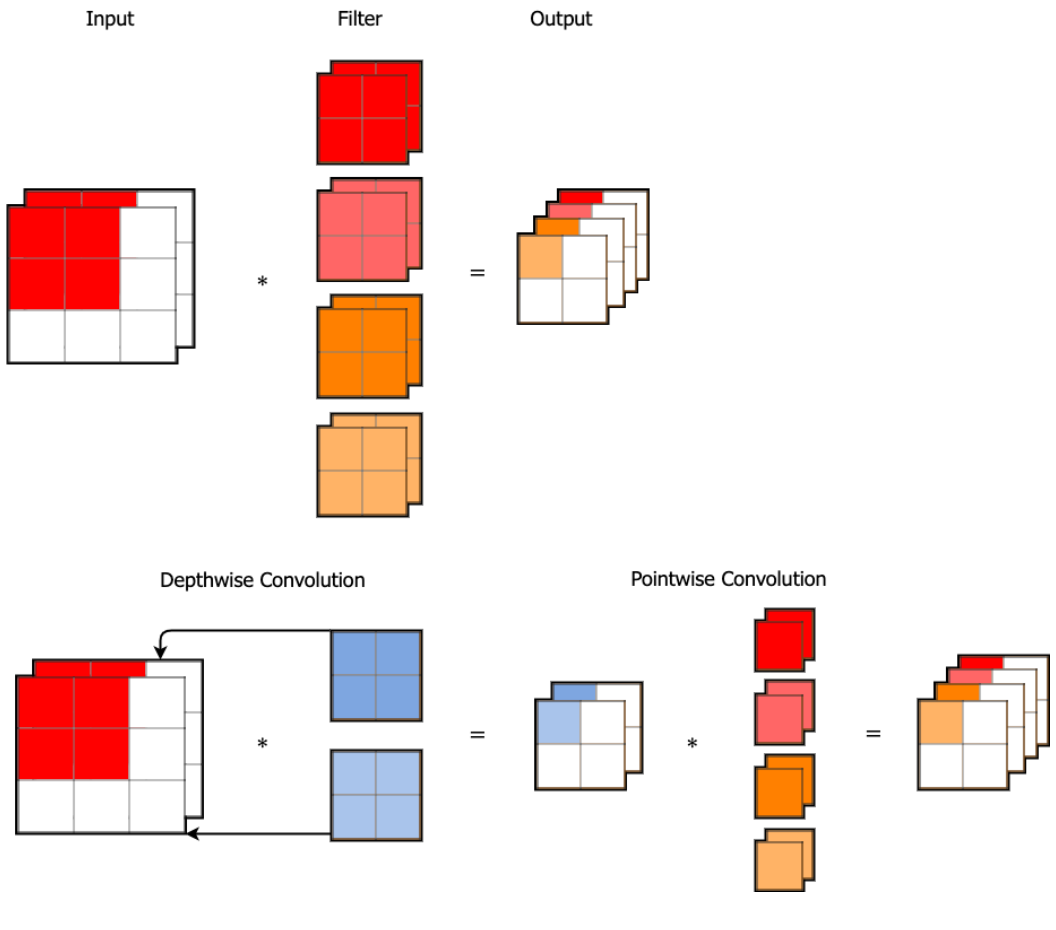


图 8.9. 上方显示了原始的卷积过程，构造四组通道数为 2 的卷积核组，经过运算得到输出。下方显示了深度可分离卷积，先经过一组卷积核，再经过四组通道数为 2 但更小的卷积核组。

8.4.2 分组卷积

如图 8.10 所示，我们先考虑标准卷积，对输入为 3×3×4 的数组，经过 2 组 2×2×4 的卷积核，得到 2×2×2 的输出。这里我们一共需要 2×2×4×2=32 个参数。再考虑分组卷积 (`Group Convolution`)，例如我们将输入数组依据通道分为 2 组，每组需要 2×2×2 的卷积核就可以得到 2×2×1 的输出，拼接在一起同样得到 2×2×2 的输出。在这个分组卷积中，我们一共需要 2×2×2×2=16 个参数。

更形式化的，我们假设：

- 输入尺寸：$(\mathrm{H}_{in}, \mathrm{W}_{in}, c_1)$
- 卷积核尺寸：$(\mathrm{K}, \mathrm{K}, c_1)$
- 输出尺寸：$(\mathrm{H}_{out}, \mathrm{W}_{out}, c_2)$

我们需要的标准卷积核参数量为 $\mathrm{K} \times \mathrm{K} \times c_1 \times c_2$。在分组卷积中，假设被分为 g 组，则每一组输入的尺寸为 $(\mathrm{H}_{in}, \mathrm{W}_{in}, c_1/g)$，对应该组需要的卷积核组的尺寸为 $(\mathrm{K}, \mathrm{K}, c_1/g, c_2/g)$，输出尺寸为 $(\mathrm{H}_{out}, \mathrm{W}_{out}, c_2/g)$。最后，将 g 组的结果拼接在一起，最终得到 $(\mathrm{H}_{out}, \mathrm{W}_{out}, c_2)$ 大小的输出。在这个过程中，分组卷积需要的卷积核参数量为 $\mathrm{K} \times \mathrm{K} \times (c_1/g) \times (c_2/g) \times g$，是标准卷积的 $\frac{1}{g}$。

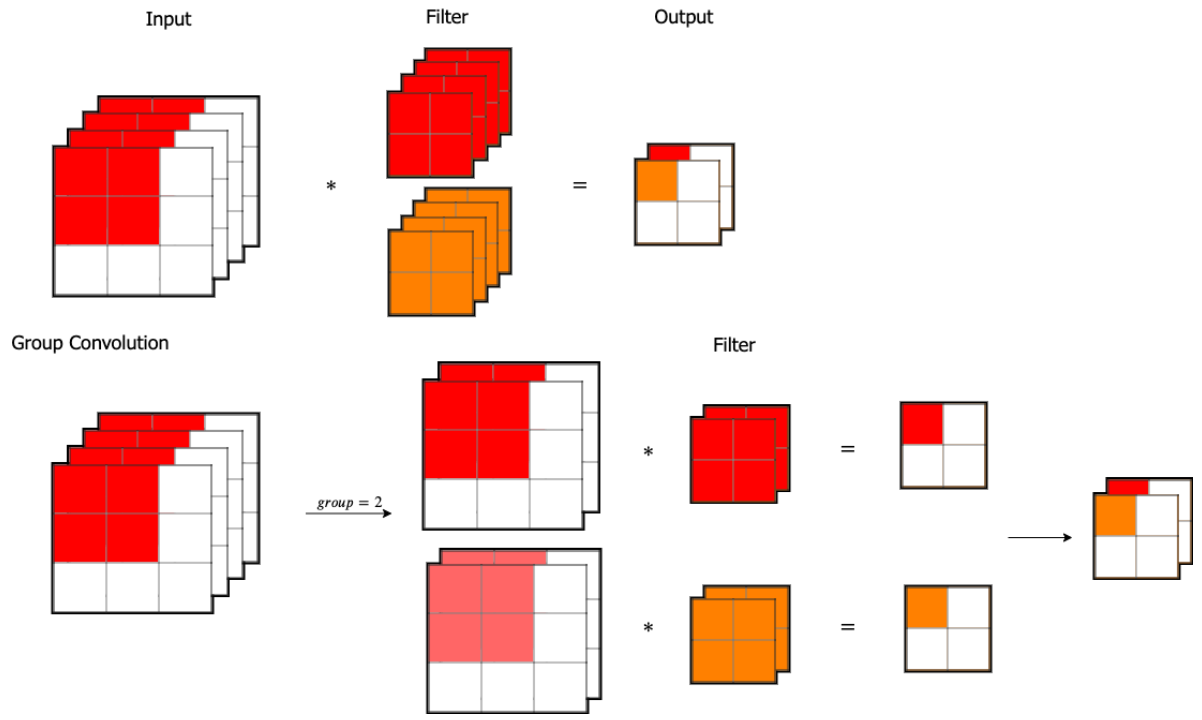


图 8.10. 上方显示了原始的卷积过程，构造两组通道数为 4 的卷积核组，经过运算得到输出。下方显示了分组卷积，将原始输入分为两组，每组通道数为 2。再分别经过一组通道数为 2 的卷积核。将分别得到的结果拼接得到最终输出。

8.4.3 扩张卷积

扩张卷积 (Dilated Convolution)，也称空洞卷积，它引入的参数被称为扩张率 (Dilation rate)，其定义了核内值之间的间距。如图 8.11 所示，图中扩张速率为 2 的 3×3 内核将具有与 5×5 内核相同的视野，但只使用 9 个参数。这种方法能以相同的计算成本，提供更大的感受野。在需要更大的观察范围，且无法承受多个卷积或更大的内核，可以用它。

如果输入图片的大小为 (m,n),而卷积核的大小为 (f,f),每个方向扩展像素点数量为 p,步幅设为 s,则标准卷积输出尺寸为 $(\lfloor\frac{m+2p-f}{s}+1\rfloor,\lfloor\frac{n+2p-f}{s}+1\rfloor)$。如果扩张率为 r，则扩张卷积输出尺寸为 $(\lfloor\frac{m+2p-[f+(f-1)(r-1)]}{s}+1\rfloor,\lfloor\frac{n+2p-[f+(f-1)(r-1)]}{s}+1\rfloor)$。

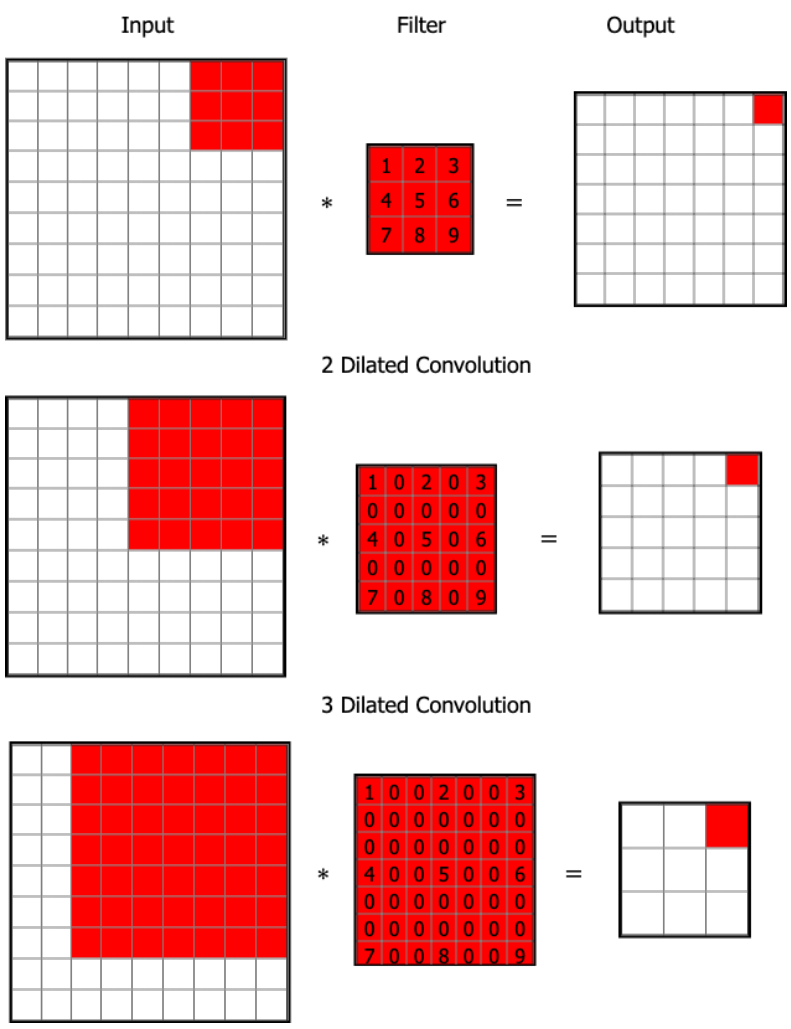


图 8.11. 分别显示了不考虑扩张率 (默认为 1)，以及考虑扩张率 (为 2 和 3) 的效果。

[1]:
```
import numpy as np
import time
import sys
sys.path.append('../')
from method.optimizer import OptimizerInitializer
from method.weight import WeightInitializer
from method.activation import ActivationInitializer
from chapter6 import LayerBase, CrossEntropy, FullyConnected, minibatch, softmax
from collections import OrderedDict
```

[2]:
```
"""
Padding 操作
"""
def calc_pad_dims_sameconv_2D(X_shape, out_dim, kernel_shape, stride, dilation=1):
    """
    当填充方式为相同卷积时，计算 padding 的数目，保证输入输出的大小相同。这里在卷积过程中考虑填充 (Padding)，
    卷积步幅 (Stride)，扩张率 (Dilation rate)。根据扩张卷积的输出公式可以得到 padding 的数目。

    参数说明:
    X_shape: 输入数组，为 (n_samples, in_rows, in_cols, in_ch)
    out_dim: 输出数组维数，为 (out_rows, out_cols)
    kernel_shape: 卷积核形状，为 (fr, fc)
    stride: 卷积步幅，int 型
    dilation: 扩张率，int 型，default=1
    """
    d = dilation
    fr, fc = kernel_shape
```

```
    out_rows, out_cols = out_dim
    n_ex, in_rows, in_cols, in_ch = X_shape
    # 考虑扩张率
    _fr, _fc = fr + (fr-1) * (d-1), fc + (fc-1) * (d-1)
    # 计算 padding 维数
    pr = int((stride * (out_rows-1) + _fr - in_rows) / 2)
    pc = int((stride * (out_cols-1) + _fc - in_cols) / 2)
    # 校验，如不等 (right/bottom 处) 添加不对称 0 填充
    out_rows1 = int(1 + (in_rows + 2 * pr - _fr) / stride)
    out_cols1 = int(1 + (in_cols + 2 * pc - _fc) / stride)
    pr1, pr2 = pr, pr
    if out_rows1 == out_rows - 1:
        pr1, pr2 = pr, pr + 1
    elif out_rows1 != out_rows:
        raise AssertionError
    pc1, pc2 = pc, pc
    if out_cols1 == out_cols - 1:
        pc1, pc2 = pc, pc + 1
    elif out_cols1 != out_cols:
        raise AssertionError
    # 返回对 X 的 Padding 维数 (left, right, up, down)
    return (pr1, pr2, pc1, pc2)


def pad2D(X, pad, kernel_shape=None, stride=None, dilation=1):
    """
    二维填充

    参数说明:
    X: 输入数组，为 (n_samples, in_rows, in_cols, in_ch)，
        其中 padding 操作是应用到 in_rows 和 in_cols
    pad: padding 数目，4-tuple, int, 或 'same', 'valid'
        在图片的左、右、上、下 (left, right, up, down) 0 填充
        若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
        若为 same，表示填充后为相同 (same) 卷积，
        若为 valid，表示填充后为有效 (valid) 卷积
    kernel_shape: 卷积核形状，为 (fr, fc)
    stride: 卷积步幅，int 型
    dilation: 扩张率，int 型，default=1
    """
    p = pad
    if isinstance(p, int):
        p = (p, p, p, p)
    if isinstance(p, tuple):
        X_pad = np.pad(
            X,
            pad_width=((0, 0), (p[0], p[1]), (p[2], p[3]), (0, 0)),
            mode="constant",
            constant_values=0,
        )
    # 'same' 卷积，首先计算 padding 维数
    if p == "same" and kernel_shape and stride is not None:
        p = calc_pad_dims_sameconv_2D(
            X.shape, X.shape[1:3], kernel_shape, stride, dilation=dilation
        )
        X_pad, p = pad2D(X, p)
    if p == "valid":
        p = (0, 0, 0, 0)
        X_pad, p = pad2D(X, p)
```

```
    return X_pad, p
```

[3]:
```
def conv2D(X, W, stride, pad, dilation=1):
    """
    二维卷积实现过程。

    参数说明:
    X: 输入数组，为 (n_samples, in_rows, in_cols, in_ch)
    W: 卷积层的卷积核参数，为 (kernel_rows, kernel_cols, in_ch, out_ch)
    stride: 卷积核的卷积步幅，int 型
    pad: padding 数目，4-tuple, int, 或 'same', 'valid' 型
        在图片的左、右、上、下 (left, right, up, down) 0 填充
        若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
        若为 same，表示填充后为相同 (same) 卷积，
        若为 valid，表示填充后为有效 (valid) 卷积
    dilation: 扩张率，int 型，default=1

    输出说明:
    Z: 卷积结果，为 (n_samples, out_rows, out_cols, out_ch)
    """
    s, d = stride, dilation
    X_pad, p = pad2D(X, pad, W.shape[:2], stride=s, dilation=d)
    pr1, pr2, pc1, pc2 = p
    fr, fc, in_ch, out_ch = W.shape
    n_samp, in_rows, in_cols, in_ch = X.shape
    # 考虑扩张率
    _fr, _fc = fr + (fr-1) * (d-1), fc + (fc-1) * (d-1)
    out_rows = int((in_rows + pr1 + pr2 - _fr) / s + 1)
    out_cols = int((in_cols + pc1 + pc2 - _fc) / s + 1)
    Z = np.zeros((n_samp, out_rows, out_cols, out_ch))
    for m in range(n_samp):
        for c in range(out_ch):
            for i in range(out_rows):
                for j in range(out_cols):
                    i0, i1 = i * s, (i * s) + fr + (fr-1) * (d-1)
                    j0, j1 = j * s, (j * s) + fc + (fc-1) * (d-1)
                    window = X_pad[m, i0 : i1 : d, j0 : j1 : d, :]
                    Z[m, i, j, c] = np.sum(window * W[:, :, :, c])
    return Z
```

8.5 GEMM 转换

在前面介绍的二维卷积运算代码中我们会有 4 个 for 循环，这需要很大的时间开销，我们能否有更快的计算方法?

矩阵乘 (General Matrix Multiplication，GEMM) 是深度学习的核心。前面的全连接层是通过矩阵乘实现的，卷积层是否也可以这样实现?

卷积核是在输入图像上按步长滑动，每次滑动操作在输入图像上的对应窗口的区域，将卷积核中的各个权值 w_* 与输入图像上对应窗口中的各个值 x_* 相乘，然后相加得到输出特征图上的一个值。卷积核对输入图像的每一次运算，可以看作是两个向量的内积，这意味着卷积操作完全可以转化为矩阵的乘法来实现。

我们将卷积层的卷积操作转化为卷积核的权值矩阵与输入数组转化的输入矩阵进行相乘。我们现在假设：

- 卷积核组：$(c_2, c_1, \mathrm{K}, \mathrm{K})$，单组卷积核 Kernel 的大小为 $(c_1, \mathrm{K}, \mathrm{K})$；
- 输入数组：$(c_1, \mathrm{H}_{in}, \mathrm{W}_{in})$；

转化如图 8.12 所示：图中单组卷积核 Kernel 在输入图像上滑动得到各个 Patch_*，Patch_* 的大小同卷积核。我们将 Kernel 和 Patch_* 均展开成一维向量。假设有 $n_k (n_k = c_2)$ 组卷积核，并且每组卷积核在输入上所有的滑动可以得到 n_p 个 Patch，于是可以得到输入矩阵 (l_{col}, n_p) 以及权值矩阵 (n_k, l_{col})。

- 权值矩阵：$(c_2, c_1 \times \mathrm{K} \times \mathrm{K})$；

- 输入矩阵：$(c_1 \times \mathrm{K} \times \mathrm{K}, \mathrm{H}_{out} \times \mathrm{W}_{out})$

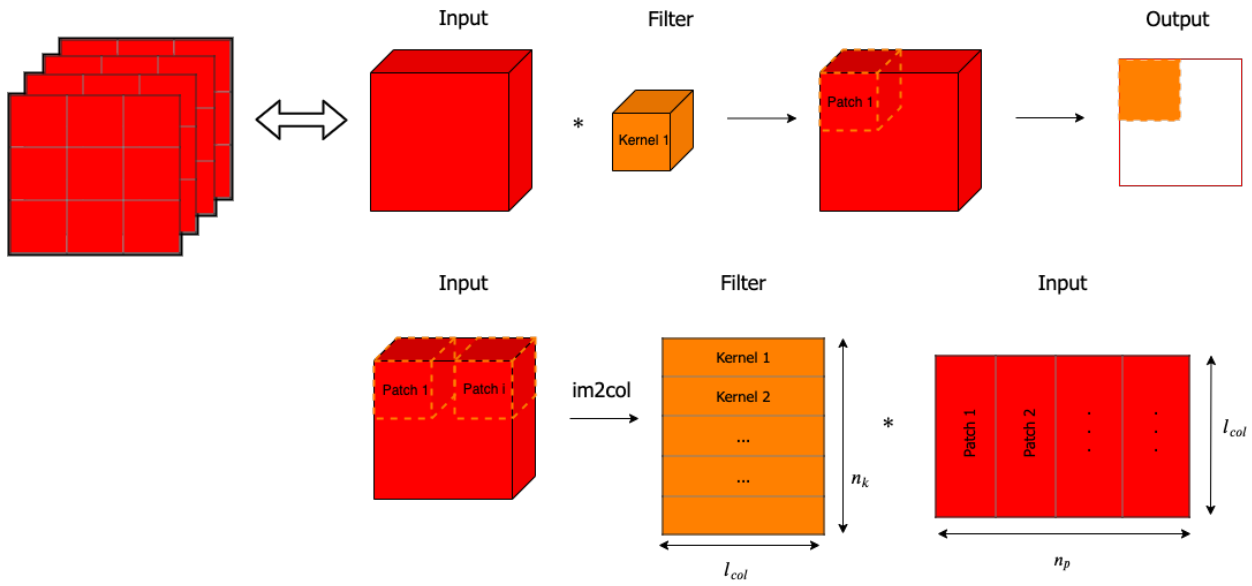


图 8.12. im2col 操作过程。一组卷积核在三维图像上滑动的过程，可以视作卷积核与输入图像上不同的块分别做运算。基于块可以将输入图像数组转化为 (l_{col}, n_p)。如果有 n_k 组卷积核，也将卷积核组转化为 (n_k, l_{col})。

再具体看一下 Kernel 和 Patch 的展开过程。对于 Kernel，我们可以依图 8.13 所示展开，图中所示为 1 组卷积核有 3 个通道，将卷积核先依通道再依行依次放入。

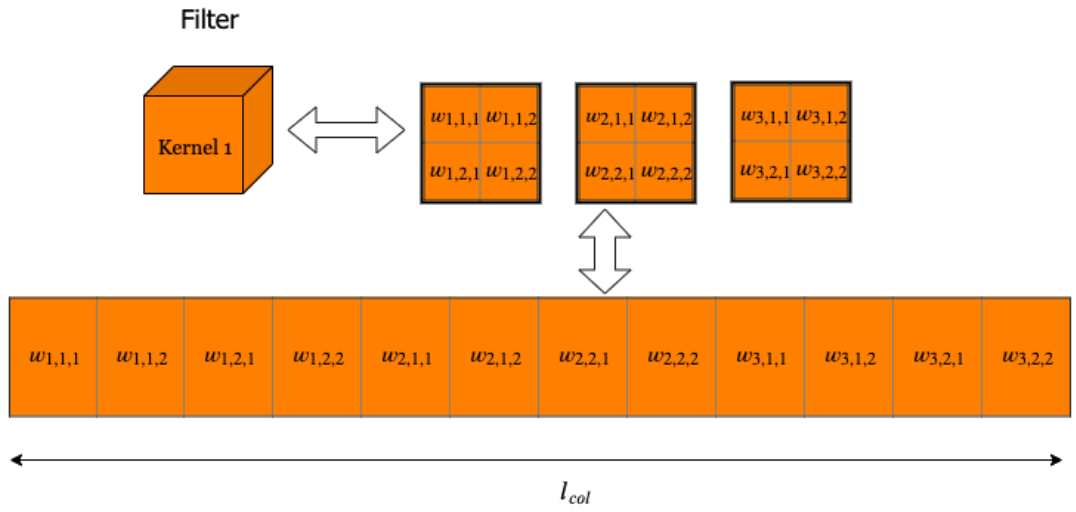


图 8.13. im2col 操作中过程，将单个卷积核组 (三维) 展开成一列向量的实现过程。

Patch 的展开过程同样。而每个 Patch 是 Kernel 在输入上滑动得到的，如果我们记 Patch 的右上角在图像中坐标为 $(:, i, j)$，则 Patch 中的元素为：$\{x_{c,i:i+h,j:j+h}\}, c = 1, ..., c_1, h = \mathrm{K}$，而 (i, j) 的取值 $i = 1, ..., \mathrm{H}_{out}, j = 1, ..., \mathrm{W}_{out}$。如果考虑扩张卷积，则元素可以写作：$\{x_{c,i:r:i+rh,j:r:j+rh}\}$。

[4]:
```
"""
下面展示 conv2D 的 GEMM 实现过程，将 X 和 W 转化为 2D 矩阵，
这里我们将 X 转化为 (kernel_rows*kernel_cols*in_ch, n_samples*out_rows*out_cols)
W 转化为 (out_ch, kernel_rows*kernel_cols*in_ch)
"""
def _im2col_indices(X_shape, fr, fc, p, s, d=1):
    """
    生成输入矩阵的 (c,h_in,w_in) 三个维度的索引

    输出说明:
    i: 输入矩阵的 i 值，(kernel_rows*kernel_cols*in_ch, out_rows*out_cols)，图示中第二维坐标
    j: 输入矩阵的 j 值，(kernel_rows*kernel_cols*in_ch, out_rows*out_cols)，图示中第三维坐标
    k: 输入矩阵的 c 值，(kernel_rows*kernel_cols*in_ch, 1)，图示中第一维坐标
    """
    pr1, pr2, pc1, pc2 = p
    n_ex, n_in, in_rows, in_cols = X_shape
    # 考虑扩张率
    _fr, _fc = fr + (fr-1) * (d-1), fc + (fc-1) * (d-1)
    out_rows = int((in_rows + pr1 + pr2 - _fr) / s + 1)
    out_cols = int((in_cols + pc1 + pc2 - _fc) / s + 1)
    # i0/i1/j0/j1: 用于得到 i, j, k。i0/j0 过程见图示，i1/j1 由滑动过程得出
    i0 = np.repeat(np.arange(fr), fc)
```

```
    i0 = np.tile(i0, n_in) * d
    i1 = s * np.repeat(np.arange(out_rows), out_cols)
    j0 = np.tile(np.arange(fc), fr * n_in) * d
    j1 = s * np.tile(np.arange(out_cols), out_rows)
    i = i0.reshape(-1, 1) + i1.reshape(1, -1)
    j = j0.reshape(-1, 1) + j1.reshape(1, -1)
    k = np.repeat(np.arange(n_in), fr * fc).reshape(-1, 1)
    return k, i, j


def im2col(X, W_shape, pad, stride, dilation=1):
    """
    im2col 实现

    参数说明:
    X: 输入数组，为 (n_samples, in_rows, in_cols, in_ch)，此时还未 0 填充 (padding)
    W_shape: 卷积层的卷积核的形状，为 (kernel_rows, kernel_cols, in_ch, out_ch)
    pad: padding 数目，4-tuple, int, 或 'same', 'valid' 型
        在图片的左、右、上、下 (left, right, up, down) 0 填充
        若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
        若为 same，表示填充后为相同 (same) 卷积，
        若为 valid，表示填充后为有效 (valid) 卷积
    stride: 卷积核的卷积步幅，int 型
    dilation: 扩张率，int 型，default=1

    输出说明:
    X_col: 输出结果，形状为 (kernel_rows*kernel_cols*n_in, n_samples*out_rows*out_cols)
    p: 填充数，4-tuple
    """
    fr, fc, n_in, n_out = W_shape
    s, p, d = stride, pad, dilation
    n_samp, in_rows, in_cols, n_in = X.shape
    X_pad, p = pad2D(X, p, W_shape[:2], stride=s, dilation=d)
    pr1, pr2, pc1, pc2 = p
    # 将输入的通道维数移至第二位
    X_pad = X_pad.transpose(0, 3, 1, 2)
    k, i, j = _im2col_indices((n_samp, n_in, in_rows, in_cols), fr, fc, p, s, d)
    # X_col.shape = (n_samples, kernel_rows*kernel_cols*n_in, out_rows*out_cols)
    X_col = X_pad[:, k, i, j]
    X_col = X_col.transpose(1, 2, 0).reshape(fr * fc * n_in, -1)
    return X_col, p


def conv2D_gemm(X, W, stride=0, pad='same', dilation=1):
    """
    二维卷积实现过程，依靠“im2col”函数将卷积作为单个矩阵乘法执行。

    参数说明:
    X: 输入数组，为 (n_samples, in_rows, in_cols, in_ch)
    W: 卷积层的卷积核参数，为 (kernel_rows, kernel_cols, in_ch, out_ch)
    stride: 卷积核的卷积步幅，int 型
    pad: padding 数目，4-tuple, int, 或 'same', 'valid' 型
        在图片的左、右、上、下 (left, right, up, down) 0 填充
        若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
        若为 same，表示填充后为相同 (same) 卷积，
        若为 valid，表示填充后为有效 (valid) 卷积
    dilation: 扩张率，int 型，default=1

    输出说明:
```

```
    Z: 卷积结果，为 (n_samples, out_rows, out_cols, out_ch)
    """
    s, d = stride, dilation
    _, p = pad2D(X, pad, W.shape[:2], s, dilation=dilation)
    pr1, pr2, pc1, pc2 = p
    fr, fc, in_ch, out_ch = W.shape
    n_samp, in_rows, in_cols, in_ch = X.shape
    # 考虑扩张率
    _fr, _fc = fr + (fr-1) * (d-1), fc + (fc-1) * (d-1)
    # 输出维数，根据上面公式可得
    out_rows = int((in_rows + pr1 + pr2 - _fr) / s + 1)
    out_cols = int((in_cols + pc1 + pc2 - _fc) / s + 1)
    # 将 X 和 W 转化为 2D 矩阵并乘积
    X_col, _ = im2col(X, W.shape, p, s, d)
    W_col = W.transpose(3, 2, 0, 1).reshape(out_ch, -1)
    Z = (W_col @ X_col).reshape(out_ch, out_rows, out_cols, n_samp).transpose(3, 1, 2, 0)
    return Z
```

8.6 卷积网络的训练

卷积的内部实现实际上是**矩阵相乘**，因此，卷积的**反向传播**过程实际上跟普通的全连接是类似的。

8.6.1 卷积网络示意图

CNN 的基本层包括卷积层和池化层，二者通常一起使用 (一个池化层紧跟一个卷积层之后)。这两层包括三个级联的函数：卷积，激励函数 (如 sigmoid 函数)，池化。其前向传播和后向传播的示意图如下：

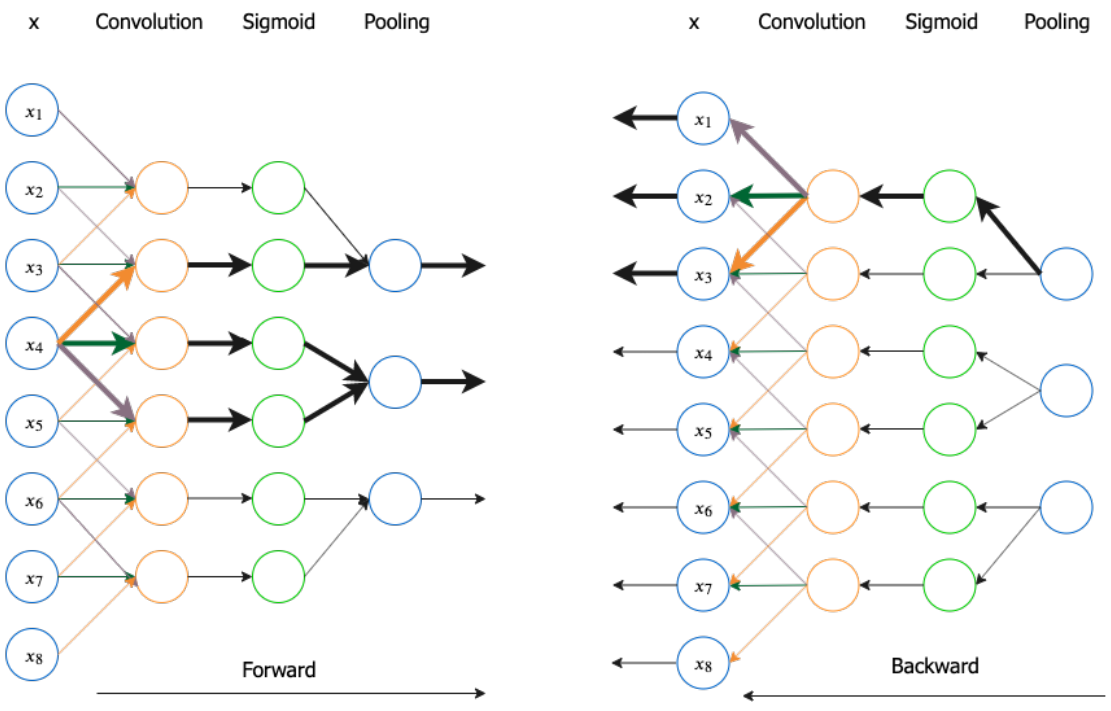


图 8.14. 卷积网络的前向传播与反向传播示意图 (这里输入为一维向量，即一维卷积)。

8.6.2 单层卷积层/池化层

卷积函数的导数及反向传播

从一维卷积入手，假设一个卷积过程的输入向量是 $\boldsymbol{x}$，输出向量是 $\boldsymbol{y}$，参数向量 (卷积算子) 是 $\boldsymbol{w}$。从输入到输出的过程为：

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \tag{8.3}$$

其中 $\boldsymbol{y}$ 的长度为 $|\boldsymbol{y}|$，$\boldsymbol{y}$ 中每一个元素的计算方法为：

$$y_n = (\boldsymbol{x} * \boldsymbol{w})[n] = \sum_{k=1}^{|\boldsymbol{w}|} x_{n+k-1} w_k = w^\top x_{n:n+|w|-1} \tag{8.4}$$

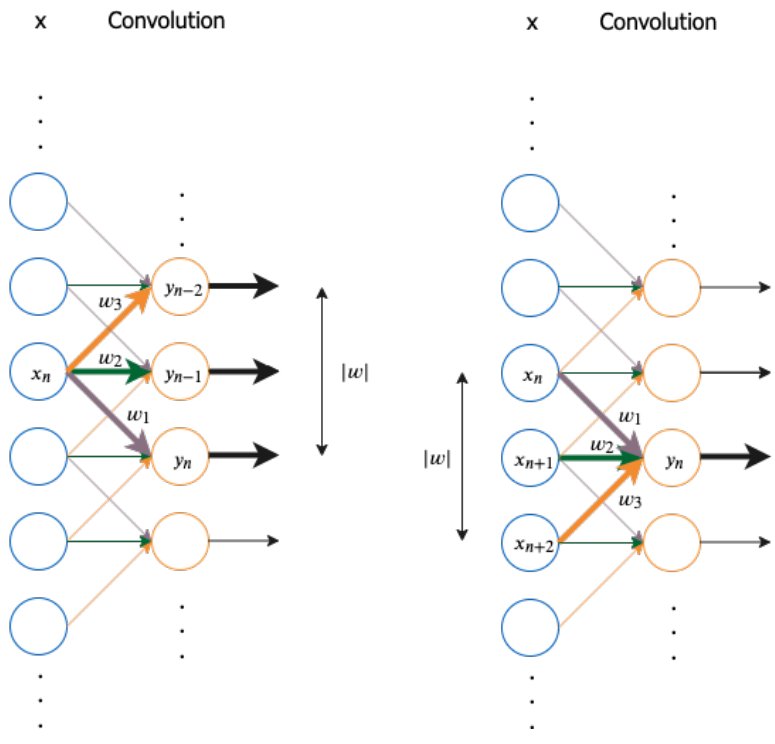


图 8.15. 卷积函数的前向传播与反向传播示意图 (一维卷积)。

$\boldsymbol{y}$ 中的元素与 $\boldsymbol{x}$ 中的元素有如下导数关系：

$$\frac{\partial y_{n-k+1}}{\partial x_n} = w_k, \quad \frac{\partial y_n}{\partial w_k} = x_{n+k-1}, \quad \text{for } 1 \le k \le |w| \tag{8.5}$$

进一步可以得到 J 关于 $\boldsymbol{w}$ 和 $\boldsymbol{x}$ 的导数：

$$\begin{aligned}
\delta_n^{(\boldsymbol{x})} = \frac{\partial J}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial x_n} = \sum_{k=1}^{|\boldsymbol{w}|}\frac{\partial J}{\partial y_{n-k+1}}\frac{\partial y_{n-k+1}}{\partial x_n} = \sum_{k=1}^{|\boldsymbol{w}|}\delta_{n-k+1}^{(\boldsymbol{y})} w_k = (\delta^{(\boldsymbol{y})} * \text{flip}(\boldsymbol{w}))[n] \\
\delta^{(\boldsymbol{x})} = \delta^{(\boldsymbol{y})} * \text{flip}(\boldsymbol{w}) \\
\frac{\partial}{\partial w_k}J = \frac{\partial J}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial w_k} = \sum_{n=1}^{|\boldsymbol{y}|}\frac{\partial J}{\partial y_n}\frac{\partial y_n}{\partial w_k} = \sum_{n=1}^{|\boldsymbol{y}|}\delta_n^{(\boldsymbol{y})} x_{n+k-1} = (\delta^{(\boldsymbol{y})} * \boldsymbol{x})[k] \\
\frac{\partial}{\partial \boldsymbol{w}}J = \delta^{(\boldsymbol{y})} * \boldsymbol{x}
\end{aligned} \tag{8.6}$$

因此，通过 $\delta^{(\boldsymbol{y})}$ 与 $\text{flip}(\boldsymbol{w})$ 可得到 J 关于 $\boldsymbol{x}$ 的导数 $\delta^{(\boldsymbol{x})}$，通过 $\delta^{(\boldsymbol{y})}$ 与 $\boldsymbol{x}$ 可计算出 $\boldsymbol{w}$ 的梯度 $\frac{\partial}{\partial \boldsymbol{w}}J$。

如果考虑偏置和激活函数，则前向传播时，$\boldsymbol{y}$ 经过激活函数 g 得到 $\boldsymbol{a} = g(\boldsymbol{y}) = g(\boldsymbol{x} * \boldsymbol{w} + \boldsymbol{b})$；反向传播时，$\delta^{(\boldsymbol{y})} = \delta^{(\boldsymbol{a})} \odot g'(\boldsymbol{y})$。于是 $\delta^{(\boldsymbol{x})} = \delta^{(\boldsymbol{y})} * \text{flip}(\boldsymbol{w}) \odot g'(\boldsymbol{y})$。

至此，我们拆解了卷积核作用在输入上的导数与反向传播过程。映射到二维卷积，这也是得到了一个二维卷积核作用在二维输入数组上的结论。推导二维卷积的导数与反向传播的过程类似。

在实际的卷积核中，我们还需要考虑通道。上文介绍的多通道输入，后一层的每个通道都是由前一层的各个通道经过卷积再求和得到的。我们将这个过程可以理解为全连接，节点为一个通道的输入数组。

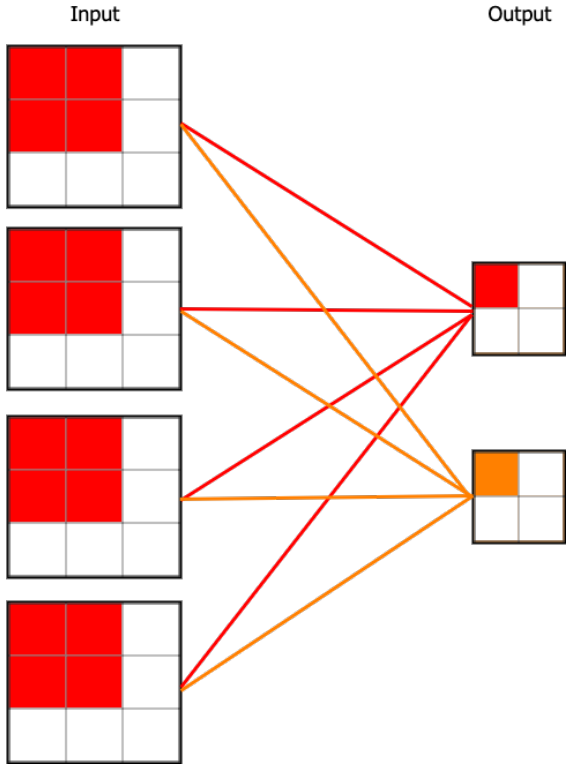


图 8.16. 考虑多通道下的卷积运算。输入的每个通道先与不同卷积核组相应的卷积核运算，再累加得到输出的对应位置。这个过程可以视作全连接，将单个卷积核视作权重。

[5]:
```
class Conv2D(LayerBase):

    def __init__(
        self,
        out_ch,
        kernel_shape,
        pad=0,
        stride=1,
        dilation=1,
        acti_fn=None,
        optimizer=None,
        init_w="glorot_uniform",
    ):
        """
        二维卷积

        参数说明:
        out_ch: 卷积核组的数目，int 型
        kernel_shape: 单个卷积核形状，2-tuple
        acti_fn: 激活函数，str 型
        pad: padding 数目，4-tuple, int, 或 'same', 'valid' 型
            在图片的左、右、上、下 (left, right, up, down) 0 填充
            若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
            若为 same，表示填充后为相同 (same) 卷积，
            若为 valid，表示填充后为有效 (valid) 卷积
        stride: 卷积核的卷积步幅，int 型
        dilation: 扩张率，int 型，default=1
        init_w: 权重初始化方法，str 型
        optimizer: 优化方法，str 型
        """
        super().__init__(optimizer)

        self.pad = pad
        self.in_ch = None
        self.out_ch = out_ch
        self.stride = stride
        self.dilation = dilation
        self.kernel_shape = kernel_shape
        self.init_w = init_w
        self.init_weights = WeightInitializer(mode=init_w)
        self.acti_fn = ActivationInitializer(acti_fn)()
        self.parameters = {"W": None, "b": None}
        self.is_initialized = False

    def _init_params(self):
        fr, fc = self.kernel_shape
        W = self.init_weights((fr, fc, self.in_ch, self.out_ch))
        b = np.zeros((1, 1, 1, self.out_ch))

        self.params = {"W": W, "b": b}
        self.gradients = {"W": np.zeros_like(W), "b": np.zeros_like(b)}
        self.derived_variables = {"Y": []}
        self.is_initialized = True

    def forward(self, X, retain_derived=True):
        """
        卷积层的前向传播，原理见上文。

        参数说明:
```

```
        X: 输入数组，形状为 (n_samples, in_rows, in_cols, in_ch)
        retain_derived: 是否保留中间变量，以便反向传播时再次使用，bool 型

        输出说明:
        a: 卷积层输出，形状为 (n_samples, out_rows, out_cols, out_ch)
        """
        if not self.is_initialized:
            self.in_ch = X.shape[3]
            self._init_params()
        W = self.params["W"]
        b = self.params["b"]
        n_samp, in_rows, in_cols, in_ch = X.shape
        s, p, d = self.stride, self.pad, self.dilation
        # 卷积操作
        Y = conv2D(X, W, s, p, d) + b
        a = self.acti_fn(Y)
        if retain_derived:
            self.X.append(X)
            self.derived_variables["Y"].append(Y)
        return a

    def backward(self, dLda, retain_grads=True):
        """
        卷积层的反向传播，原理见上文。

        参数说明:
        dLda: 关于损失的梯度，为 (n_samples, out_rows, out_cols, out_ch)
        retain_grads: 是否计算中间变量的参数梯度，bool 型

        输出说明:
        dXs: 即 dX，当前卷积层对输入关于损失的梯度，为 (n_samples, in_rows, in_cols, in_ch)
        """
        if not isinstance(dLda, list):
            dLda = [dLda]
        W = self.params["W"]
        b = self.params["b"]
        Ys = self.derived_variables["Y"]
        Xs, d = self.X, self.dilation
        (fr, fc), s, p = self.kernel_shape, self.stride, self.pad
        dXs = []
        for X, Y, da in zip(Xs, Ys, dLda):
            n_samp, out_rows, out_cols, out_ch = da.shape
            X_pad, (pr1, pr2, pc1, pc2) = pad2D(X, p, self.kernel_shape, s, d)
            dY = da * self.acti_fn.grad(Y)
            dX = np.zeros_like(X_pad)
            dW, db = np.zeros_like(W), np.zeros_like(b)
            for m in range(n_samp):
                for i in range(out_rows):
                    for j in range(out_cols):
                        for c in range(out_ch):
                            i0, i1 = i * s, (i * s) + fr + (fr-1) * (d-1)
                            j0, j1 = j * s, (j * s) + fc + (fc-1) * (d-1)
                            wc = W[:, :, :, c]
                            kernel = dY[m, i, j, c]
                            window = X_pad[m, i0:i1:d, j0:j1:d, :]
                            db[:, :, :, c] += kernel
                            dW[:, :, :, c] += window * kernel
                            dX[m, i0:i1:d, j0:j1:d, :] += (
                                wc * kernel
```

```
                        )
            if retain_grads:
                self.gradients["W"] += dW
                self.gradients["b"] += db
            pr2 = None if pr2 == 0 else -pr2
            pc2 = None if pc2 == 0 else -pc2
            dXs.append(dX[:, pr1:pr2, pc1:pc2, :])
        return dXs[0] if len(Xs) == 1 else dXs

    @property
    def hyperparams(self):
        return {
            "layer": "Conv2D",
            "pad": self.pad,
            "init_w": self.init_w,
            "in_ch": self.in_ch,
            "out_ch": self.out_ch,
            "stride": self.stride,
            "dilation": self.dilation,
            "acti_fn": str(self.acti_fn),
            "kernel_shape": self.kernel_shape,
            "optimizer": {
                "cache": self.optimizer.cache,
                "hyperparams": self.optimizer.hyperparams,
            },
        }
```

如果我们使用 GEMM 转换实现前向传播和反向传播，权值矩阵 $\boldsymbol{W}$ 形状为 $(c_2, c_1 \times \mathrm{K} \times \mathrm{K})$，输入矩阵 $\boldsymbol{X}$ 形状为 $(c_1 \times \mathrm{K} \times \mathrm{K}, \mathrm{H}_{out} \times \mathrm{W}_{out})$。

于是 $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b}$，$\boldsymbol{b}$ 的形状为 c_2。

我们回顾第六章介绍的 DFN 的反向传播算法，在第 l 个全连接层有：

$$\begin{aligned} \boldsymbol{z}_{l+1} &= \boldsymbol{W}_l \boldsymbol{a}_l + \boldsymbol{b}_l \\ \boldsymbol{a}_{l+1} &= g(\boldsymbol{z}_{l+1}) \end{aligned} \tag{8.7}$$

我们可以得到：

$$\begin{cases} \delta_l^{(z)} = (\boldsymbol{W}_l)^\top \delta_{l+1}^{(z)} \odot g'(\boldsymbol{z}_l) \\ \nabla_{\boldsymbol{W}_l} J = \delta_{l+1}^{(z)} (\boldsymbol{a}_l)^\top \\ \nabla_{\boldsymbol{b}_l} J = \delta_{l+1}^{(z)} \end{cases} \tag{8.8}$$

同样的，我们这里有：

$$\begin{cases} \delta^{(x)} = \boldsymbol{W}^\top \delta^{(a)} \odot g'(\boldsymbol{y}) \\ \nabla_{\boldsymbol{W}} J = \delta^{(y)} \boldsymbol{X}^\top \\ \nabla_{\boldsymbol{b}} J = \delta^{(y)} \end{cases} \tag{8.9}$$

[6]:
```
def col2im(X_col, X_shape, W_shape, pad, stride, dilation=0):
    """
    col2im 实现，“col2im” 函数将 2D 矩阵变为 4D 图像

    参数说明:
    X_col: X 经过 im2col 后 (列) 的矩阵，形状为 (Q, Z)，具体形状见上文
    X_shape: 原始的输入数组形状，为 (n_samples, in_rows, in_cols, in_ch)，
             此时还未 0 填充 (padding)
    W_shape: 卷积核组形状，4-tuple 为 (kernel_rows, kernel_cols, in_ch, out_ch)
    pad: padding 数目，4-tuple
            在图片的左、右、上、下 (left, right, up, down) 0 填充
    stride: 卷积核的卷积步幅，int 型
    dilation: 扩张率，int 型，default=1

    输出说明:
```

```
    img: 输出结果，形状为 (n_samples, in_rows, in_cols, in_ch)
    """
    s, d = stride, dilation
    pr1, pr2, pc1, pc2 = pad
    fr, fc, n_in, n_out = W_shape
    n_samp, in_rows, in_cols, n_in = X_shape
    X_pad = np.zeros((n_samp, n_in, in_rows + pr1 + pr2, in_cols + pc1 + pc2))
    k, i, j = _im2col_indices((n_samp, n_in, in_rows, in_cols), fr, fc, pad, s, d)
    X_col_reshaped = X_col.reshape(n_in * fr * fc, -1, n_samp)
    X_col_reshaped = X_col_reshaped.transpose(2, 0, 1)
    np.add.at(X_pad, (slice(None), k, i, j), X_col_reshaped)
    pr2 = None if pr2 == 0 else -pr2
    pc2 = None if pc2 == 0 else -pc2
    return X_pad[:, :, pr1:pr2, pc1:pc2]
```

[7]:
```
class Conv2D_gemm(LayerBase):

    def __init__(
        self,
        out_ch,
        kernel_shape,
        pad=0,
        stride=1,
        dilation=1,
        acti_fn=None,
        optimizer=None,
        init_w="glorot_uniform",
    ):
        """
        二维卷积

        参数说明:
        out_ch: 卷积核组的数目，int 型
        kernel_shape: 单个卷积核形状，2-tuple
        acti_fn: 激活函数，str 型
        pad: padding 数目，4-tuple, int, 或 'same', 'valid' 型
            在图片的左、右、上、下 (left, right, up, down) 0 填充
            若为 int，表示在左、右、上、下均填充数目为 pad 的 0，
            若为 same，表示填充后为相同 (same) 卷积，
            若为 valid，表示填充后为有效 (valid) 卷积
        stride: 卷积核的卷积步幅，int 型
        dilation: 扩张率，int 型，default=1
        init_w: 权重初始化方法，str 型
        optimizer: 优化方法，str 型
        """
        super().__init__(optimizer)
        self.pad = pad
        self.in_ch = None
        self.out_ch = out_ch
        self.stride = stride
        self.dilation = dilation
        self.kernel_shape = kernel_shape
        self.init_w = init_w
        self.init_weights = WeightInitializer(mode=init_w)
        self.acti_fn = ActivationInitializer(acti_fn)()
        self.parameters = {"W": None, "b": None}
        self.is_initialized = False

    def _init_params(self):
```

```
        fr, fc = self.kernel_shape
        W = self.init_weights((fr, fc, self.in_ch, self.out_ch))
        b = np.zeros((1, 1, 1, self.out_ch))
        self.params = {"W": W, "b": b}
        self.gradients = {"W": np.zeros_like(W), "b": np.zeros_like(b)}
        self.derived_variables = {"Y": []}
        self.is_initialized = True

    def forward(self, X, retain_derived=True):
        """
        卷积层的前向传播，原理见上文。

        参数说明：
        X：输入数组，形状为 (n_samples, in_rows, in_cols, in_ch)
        retain_derived：是否保留中间变量，以便反向传播时再次使用，bool 型

        输出说明：
        a：卷积层输出，形状为 (n_samples, out_rows, out_cols, out_ch)
        """
        if not self.is_initialized:
            self.in_ch = X.shape[3]
            self._init_params()
        W = self.params["W"]
        b = self.params["b"]
        n_samp, in_rows, in_cols, in_ch = X.shape
        s, p, d = self.stride, self.pad, self.dilation
        # 卷积操作
        Y = conv2D_gemm(X, W, s, p, d) + b
        a = self.acti_fn(Y)
        if retain_derived:
            self.X.append(X)
            self.derived_variables["Y"].append(Y)
        return a

    def backward(self, dLda, retain_grads=True):
        """
        卷积层的反向传播，原理见上文。

        参数说明：
        dLda：关于损失的梯度，为 (n_samples, out_rows, out_cols, out_ch)
        retain_grads：是否计算中间变量的参数梯度，bool 型

        输出说明：
        dX：当前卷积层对输入关于损失的梯度，为 (n_samples, in_rows, in_cols, in_ch)
        """
        if not isinstance(dLda, list):
            dLda = [dLda]
        dX = []
        X = self.X
        Y = self.derived_variables["Y"]
        for da, x, y in zip(dLda, X, Y):
            dx, dw, db = self._bwd(da, x, y)
            dX.append(dx)
            if retain_grads:
                self.gradients["W"] += dw
                self.gradients["b"] += db
        return dX[0] if len(X) == 1 else dX

    def _bwd(self, dLda, X, Y):
```

```
        W = self.params["W"]
        d = self.dilation
        fr, fc, in_ch, out_ch = W.shape
        n_samp, out_rows, out_cols, out_ch = dLda.shape
        (fr, fc), s, p = self.kernel_shape, self.stride, self.pad
        dLdy = dLda * self.acti_fn.grad(Y)
        dLdy_col = dLdy.transpose(3, 1, 2, 0).reshape(out_ch, -1)
        W_col = W.transpose(3, 2, 0, 1).reshape(out_ch, -1).T
        X_col, p = im2col(X, W.shape, p, s, d)
        dW = (dLdy_col @ X_col.T).reshape(out_ch, in_ch, fr, fc).transpose(2, 3, 1, 0)
        db = dLdy_col.sum(axis=1).reshape(1, 1, 1, -1)
        dX_col = W_col @ dLdy_col
        dX = col2im(dX_col, X.shape, W.shape, p, s, d).transpose(0, 2, 3, 1)
        return dX, dW, db

    @property
    def hyperparams(self):
        return {
            "layer": "Conv2D",
            "pad": self.pad,
            "init_w": self.init_w,
            "in_ch": self.in_ch,
            "out_ch": self.out_ch,
            "stride": self.stride,
            "dilation": self.dilation,
            "acti_fn": str(self.acti_fn),
            "kernel_shape": self.kernel_shape,
            "optimizer": {
                "cache": self.optimizer.cache,
                "hyperparams": self.optimizer.hyperparams,
            },
        }
```

池化函数的导数及后向传播

池化函数是一个下采样函数，对于大小为 m 的池化区域，池化函数及其导数可以定义为：

- 最大池化：$g(x) = \max(x)$，导数为 $\frac{\partial g}{\partial x_i} = \begin{cases} 1 & \text{if } x_i = \max(x) \\ 0 & \text{otherwise} \end{cases}$
- 均值池化：$g(x) = \frac{\sum_{k=1}^{m} x_k}{m}$，导数为 $\frac{\partial g}{\partial x_i} = \frac{1}{m}$
- p 范数池化：$g(x) = \|x\|_p = (\sum_{k=1}^{m} |x_k|^p)^{1/p}$ ，导数为 $\frac{\partial g}{\partial x_i} = (\sum_{k=1}^{m} |x_k|^p)^{1/p-1} |x_i|^{p-1}$

下采样的后向传播过程为上采样，其示意图为：

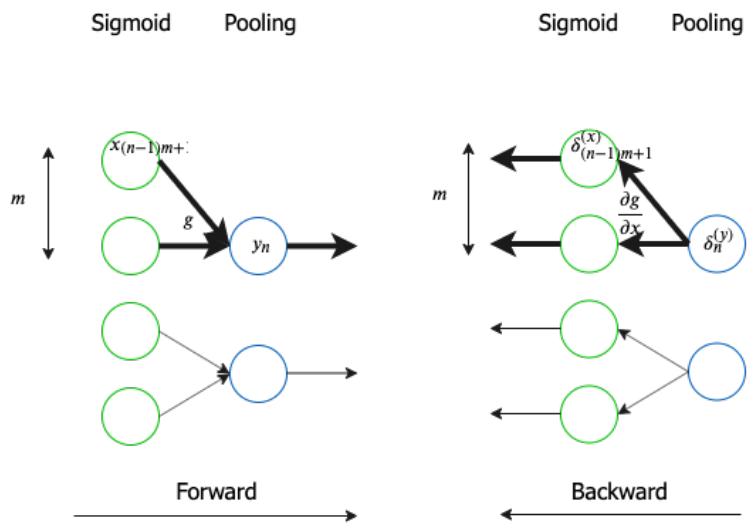


图 8.17. 池化函数的前向传播与反向传播示意图 (一维卷积)。

该后向传播过程就是利用 g 的导数将误差信号传递到 g 的输入。

$$\begin{aligned}\delta^{(\boldsymbol{x})}_{(n-1)m+1:nm} = \frac{\partial}{\partial x_{(n-1)m+1:nm}} J = \frac{\partial J}{\partial y_n}\frac{\partial y_n}{\partial x_{(n-1)m+1:nm}} = \delta^{(\boldsymbol{y})}_n g'_n \\ \delta^{(\boldsymbol{x})} = upsample(\delta^{(\boldsymbol{y})}, g')\end{aligned} \tag{8.10}$$

上采样 (`upsampling`) 具体展示如下，假设池化大小为 $2,2$，步幅为 2，且 $\delta^{(\boldsymbol{y})}$ 第 k 个子矩阵为：

$$\delta^{(y)}_k = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix} \tag{8.11}$$

先将 $\delta^{(\boldsymbol{y})}_k$ 还原成：

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 6 & 8 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{8.12}$$

如果是最大池化，假设之前在前向传播时记录的最大值位置分别是左上，右上，左下，右下，则转换后的矩阵为：

$$\begin{pmatrix} 2 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 8 \end{pmatrix} \tag{8.13}$$

如果是均值池化，根据公式可以得到：

$$\begin{pmatrix} 0.5 & 0.5 & 1.0 & 1.0 \\ 0.5 & 0.5 & 1.0 & 1.0 \\ 1.5 & 1.5 & 2.0 & 2.0 \\ 1.5. & 1.5 & 2.0 & 2.0 \end{pmatrix} \tag{8.14}$$

[8]:
```python
class Pool2D(LayerBase):

    def __init__(self, kernel_shape, stride=1, pad=0, mode="max", optimizer=None):
        """
        二维池化

        参数说明:
        kernel_shape: 池化窗口的大小，2-tuple
        stride: 和卷积类似，窗口在每一个维度上滑动的步长，int 型
        pad: padding 数目，4-tuple, int, 或 str('same','valid') 型 (default: 0)
            和卷积类似
        mode: 池化函数，str 型 (default: 'max')，可选 {"max","average"}
        optimizer: 优化方法，str 型
        """
        super().__init__(optimizer)
        self.pad = pad
        self.mode = mode
        self.in_ch = None
        self.out_ch = None
        self.stride = stride
        self.kernel_shape = kernel_shape
        self.is_initialized = False

    def _init_params(self):
        self.derived_variables = {"out_rows": [], "out_cols": []}
        self.is_initialized = True

    def forward(self, X, retain_derived=True):
        """
        池化层前向传播

        参数说明:
```

```
        X: 输入数组，形状为 (n_samp, in_rows, in_cols, in_ch)
        retain_derived: 是否保留中间变量，以便反向传播时再次使用，bool 型

        输出说明:
        Y: 输出结果，形状为 (n_samp, out_rows, out_cols, out_ch)
        """
        if not self.is_initialized:
            self.in_ch = self.out_ch = X.shape[3]
            self._init_params()
        n_samp, in_rows, in_cols, nc_in = X.shape
        (fr, fc), s, p = self.kernel_shape, self.stride, self.pad
        X_pad, (pr1, pr2, pc1, pc2) = pad2D(X, p, self.kernel_shape, s)
        out_rows = int((in_rows + pr1 + pr2 - fr) / s + 1)
        out_cols = int((in_cols + pc1 + pc2 - fc) / s + 1)
        if self.mode == "max":
            pool_fn = np.max
        elif self.mode == "average":
            pool_fn = np.mean
        Y = np.zeros((n_samp, out_rows, out_cols, self.out_ch))
        for m in range(n_samp):
            for i in range(out_rows):
                for j in range(out_cols):
                    for c in range(self.out_ch):
                        i0, i1 = i * s, (i * s) + fr
                        j0, j1 = j * s, (j * s) + fc
                        xi = X_pad[m, i0:i1, j0:j1, c]
                        Y[m, i, j, c] = pool_fn(xi)
        if retain_derived:
            self.X.append(X)
            self.derived_variables["out_rows"].append(out_rows)
            self.derived_variables["out_cols"].append(out_cols)
        return Y

    def backward(self, dLdy, retain_grads=True):
        """
        池化层的反向传播，原理见上文。

        参数说明:
        dLdy: 关于损失的梯度，为 (n_samples, out_rows, out_cols, out_ch)
        retain_grads: 是否计算中间变量的参数梯度，bool 型

        输出说明:
        dXs: 即 dX，当前卷积层对输入关于损失的梯度，为 (n_samples, in_rows, in_cols, in_ch)
        """
        if not isinstance(dLdy, list):
            dLdy = [dLdy]
        Xs = self.X
        out_rows = self.derived_variables["out_rows"]
        out_cols = self.derived_variables["out_cols"]
        (fr, fc), s, p = self.kernel_shape, self.stride, self.pad
        dXs = []
        for X, dy, out_row, out_col in zip(Xs, dLdy, out_rows, out_cols):
            n_samp, in_rows, in_cols, nc_in = X.shape
            X_pad, (pr1, pr2, pc1, pc2) = pad2D(X, p, self.kernel_shape, s)
            dX = np.zeros_like(X_pad)
            for m in range(n_samp):
                for i in range(out_row):
                    for j in range(out_col):
                        for c in range(self.out_ch):
```

```
                        i0, i1 = i * s, (i * s) + fr
                        j0, j1 = j * s, (j * s) + fc
                        if self.mode == "max":
                            xi = X[m, i0:i1, j0:j1, c]
                            mask = np.zeros_like(xi).astype(bool)
                            x, y = np.argwhere(xi == np.max(xi))[0]
                            mask[x, y] = True
                            dX[m, i0:i1, j0:j1, c] += mask * dy[m, i, j, c]
                        elif self.mode == "average":
                            frame = np.ones((fr, fc)) * dy[m, i, j, c]
                            dX[m, i0:i1, j0:j1, c] += frame / np.prod((fr, fc))
            pr2 = None if pr2 == 0 else -pr2
            pc2 = None if pc2 == 0 else -pc2
            dXs.append(dX[:, pr1:pr2, pc1:pc2, :])
        return dXs[0] if len(Xs) == 1 else dXs

    @property
    def hyperparams(self):
        return {
            "layer": "Pool2D",
            "acti_fn": None,
            "pad": self.pad,
            "mode": self.mode,
            "in_ch": self.in_ch,
            "out_ch": self.out_ch,
            "stride": self.stride,
            "kernel_shape": self.kernel_shape,
            "optimizer": {
                "cache": self.optimizer.cache,
                "hyperparams": self.optimizer.hyperparams,
            },
        }
```

8.6.3 多层卷积层/池化层

在第六章介绍了多层神经网络反向传播的推导过程。这里同前面所述，假设损失函数 $J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{y})$，第 $l+1$ 层的输入和输出分别是 $\boldsymbol{a}_l$ 和 $\boldsymbol{a}_{l+1}$，参数为 $\boldsymbol{W}_l$ 和 $\boldsymbol{b}_l$，仿射结果为中间变量 $\boldsymbol{z}_{l+1} = \boldsymbol{W}_l \boldsymbol{a}_l + \boldsymbol{b}_l$。其中 $\boldsymbol{a}_{l+1} = g(\boldsymbol{z}_{l+1})$，$g$ 为激励函数，第一层的输出 $\boldsymbol{a}_1 = \boldsymbol{x}$，为整个网络的输入，最后一层的输出 $\boldsymbol{a}_L$ 是代价函数的输入。计算得到第 l 层 J 对 $\boldsymbol{z}_l$ 的偏导数和 J 对参数 $\boldsymbol{W}_l$ 和 $\boldsymbol{b}_l$ 的梯度。

$$\begin{cases} \delta_l^{(z)} = (\boldsymbol{W}_l)^\top \delta_{l+1}^{(z)} \odot g'(\boldsymbol{z}_l) \\ \nabla_{\boldsymbol{W}_l} J = \delta_{l+1}^{(z)} (\boldsymbol{a}_l)^\top \\ \nabla_{\boldsymbol{b}_l} J = \delta_{l+1}^{(z)} \end{cases} \tag{8.15}$$

同样的，可以得到在卷积网络反向传播中，假设在第 $l+1$ 层经过卷积操作 (包括激励函数) 或池化 (激励函数为池化函数)。

1. 对第 l 层计算得到 J 对 z_l 的偏导数；

   - 如果是卷积层：

   $$\delta_l^{(z)} = \delta_{l+1}^{(z)} * \text{flip}(\boldsymbol{W}_l) \odot g'(\boldsymbol{z}_l) \tag{8.16}$$

   - 如果是池化层：

   $$\delta_l^{(z)} = upsample(\delta_{l+1}^{(z)}, g') \tag{8.17}$$

2. 对第 l 层计算得到 J 对参数 $\boldsymbol{W}_l$ 和 $\boldsymbol{b}_l$ 的梯度；

$$\begin{aligned} \nabla_{\boldsymbol{W}_l} J &= \delta_{l+1}^{(z)} * \boldsymbol{a}_l \\ \nabla_{\boldsymbol{b}_l} J &= \delta_{l+1}^{(z)} \end{aligned} \tag{8.18}$$

8.6.4 Flatten 层 & 全连接层

通过多层卷积层/池化层后我们可以获得很多个特征图 (`Feature Map`)，这些特征图从不同角度得到模型的特征，但是我们无法直接用这些特征图拿来连接到分类。对于一个分类模型，我们既要综合考虑所有的特征图，又要能够连接到分类，可以考虑的方案就是将最后得到的特征图 “拍平” (丢

到 Flatten 层)，然后把 Flatten 层的输出放到全连接层里 (一般至少 2 层以保证全连接层能够学到拟合函数)。最后采用 softmax 对其进行分类。

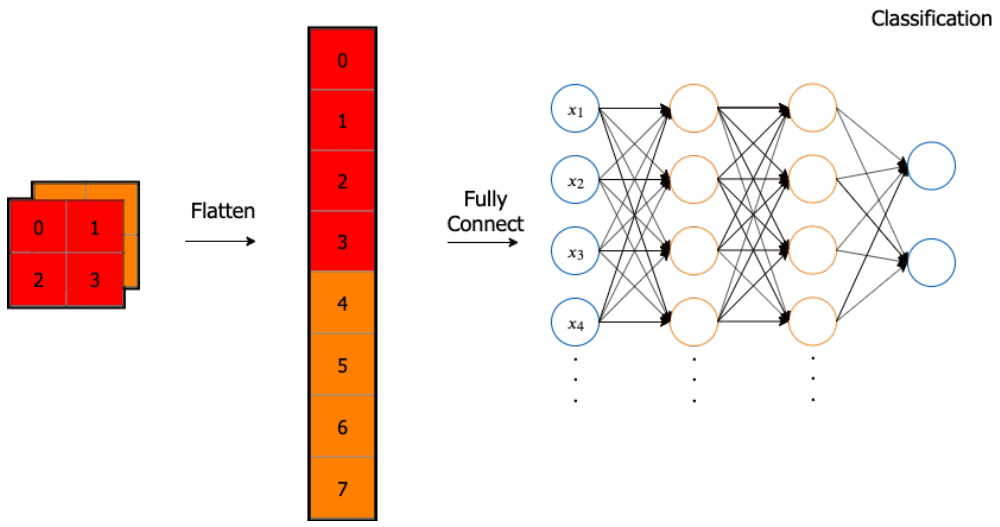


图 8.18. Flatten 层的实现过程：对最后得到的卷积核组依次展开，每个值作为一个神经元节点。

如果是将最后一层的特征图 Flatten 再放入全连接层，会存在的问题是输入的参数量过大，这会降低训练的速度，同时很容易过拟合。既然是针对这些特征图进行分类，那么另外一种可以考虑的方案是全局平均池化 (`Global Average Pooling`)。如图所示，GAP 的操作是将最后一层的特征图取均值。我们得到的 GAP 层中的每个节点对应不同的特征图，连接 GAP 层和最后的密集层的权重编码了每个特征图对预测目标分类的贡献。

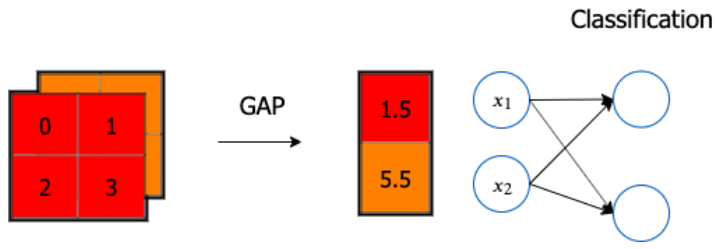


图 8.19. GAP 层的实现过程：对最后得到的卷积核组，每个卷积核取其均值作为一个神经元节点。

[9]:
```python
class Flatten(LayerBase):

    def __init__(self, keep_dim="first", optimizer=None):
        """
        将多维输入展开

        参数说明:
        keep_dim: 展开形状，str (default : 'first')
                对于输入 X，keep_dim 可选 'first'-> 将 X 重构为 (X.shape[0], -1)，
                'last'-> 将 X 重构为 (-1, X.shape[0])，'none'-> 将 X 重构为 (1,-1)
        optimizer: 优化方法
        """
        super().__init__(optimizer)
        self.keep_dim = keep_dim
        self._init_params()

    def _init_params(self):
        self.X = []
        self.gradients = {}
        self.params = {}
        self.derived_variables = {"in_dims": []}

    def forward(self, X, retain_derived=True):
        """
        前向传播

        参数说明:
        X: 输入数组
        retain_derived: 是否保留中间变量，以便反向传播时再次使用，bool 型
        """
        if retain_derived:
            self.derived_variables["in_dims"].append(X.shape)
```

```
        if self.keep_dim == "none":
            return X.flatten().reshape(1, -1)
        rs = (X.shape[0], -1) if self.keep_dim == "first" else (-1, X.shape[-1])
        return X.reshape(*rs)

    def backward(self, dLdy, retain_grads=True):
        """
        反向传播

        参数说明：
        dLdy：关于损失的梯度
        retain_grads：是否计算中间变量的参数梯度，bool 型

        输出说明：
        dX：将对输入的梯度进行重构为原始输入的形状
        """
        if not isinstance(dLdy, list):
            dLdy = [dLdy]
        in_dims = self.derived_variables["in_dims"]
        dX = [dy.reshape(*dims) for dy, dims in zip(dLdy, in_dims)]
        return dX[0] if len(dLdy) == 1 else dX

    @property
    def hyperparams(self):
        return {
            "layer": "Flatten",
            "keep_dim": self.keep_dim,
            "optimizer": {
                "cache": self.optimizer.cache,
                "hyperparams": self.optimizer.hyperparams,
            },
        }
```

8.7 平移等变

卷积神经网络具有平移不变性 (Translation Invariance) 或称平移等变。平移不变性意味着系统产生完全相同的响应 (输出)，不管它的输入是如何平移的。

- 卷积：图像经过平移，相应的特征图上的表达也是平移的。如图 8.20，输入图像的左下角有一个人脸，经过卷积后人脸的特征 (眼睛，鼻子) 也位于特征图的左下角。但假如人脸特征在图像的左上角，那么卷积后对应的特征也在特征图的左上角。无论目标出现在图像中的哪个位置，它都会检测到同样的这些特征，输出同样的响应，所以卷积具有平移不变性。

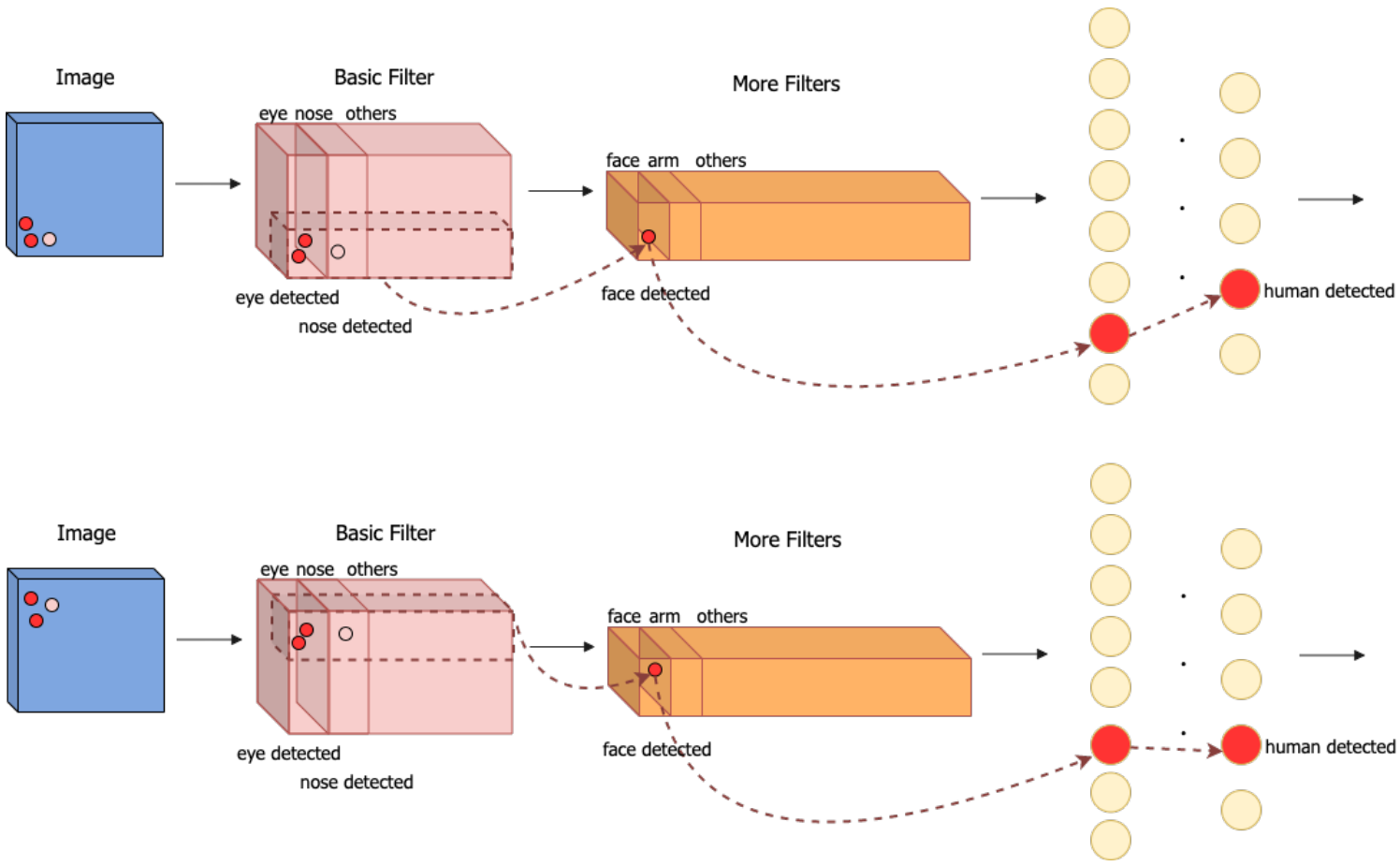


图 8.20. 卷积的平移不变性。

- 池化：如最大池化，可以返回感受野中的最大值，如果最大值被移动了，但仍然在这个感受野中，那么池化层也仍然会输出相同的最大值，所以卷积也可能具有平移不变性。

8.8 代表性的卷积神经网络

8.8.1 卷积神经网络 (LeNet)

LeNet [1] 分为卷积层块和全连接层块两个部分。卷积层块里的基本单位是卷积层后接最大池化层，卷积层块由两个这样的基本单位重复堆叠构成。

在卷积层块中，每个卷积层都使用 5×5 的窗口，并在输出上使用 sigmoid 激活函数。第一个卷积层输出通道数为 6，第二个卷积层输出通道数则增加到 16。这是因为卷积层未使用 0 填充，第二个卷积层比第一个卷积层的输入的高和宽要小，所以增加输出通道使两个卷积层的参数尺寸类似。卷积层块的两个最大池化层的窗口形状均为 2×2，且步幅为 2。由于池化窗口与步幅形状相同，池化窗口在输入上每次滑动所覆盖的区域互不重叠，其池化效果为池化函数的导数及后向传播部分中示例所示。

卷积层块的输出形状为 (批量大小, 通道, 高, 宽)。当卷积层块的输出传入全连接层块时，全连接层块会将小批量中每个样本变平 (Flatten)。即将全连接层的输入形状将变成二维，其中第一维是小批量中的样本，第二维是每个样本变平后的向量表示，且向量长度为通道、高和宽的乘积。全连接层块含 3 个全连接层，它们的输出个数分别是 120、84 和 10，其中 10 为输出的类别个数，如图所示。

我们接下来代码演示的是 LeNet 的实现，但与论文 [1] 中的实现稍有不同：论文 [1] 中的 LeNet 网络在池化层之后再进行非线性处理 (即 sigmoid 激活函数)，现在通用的操作是经过卷积之后就经过非线性处理 (sigmoid 激活函数)，然后再进行池化操作。

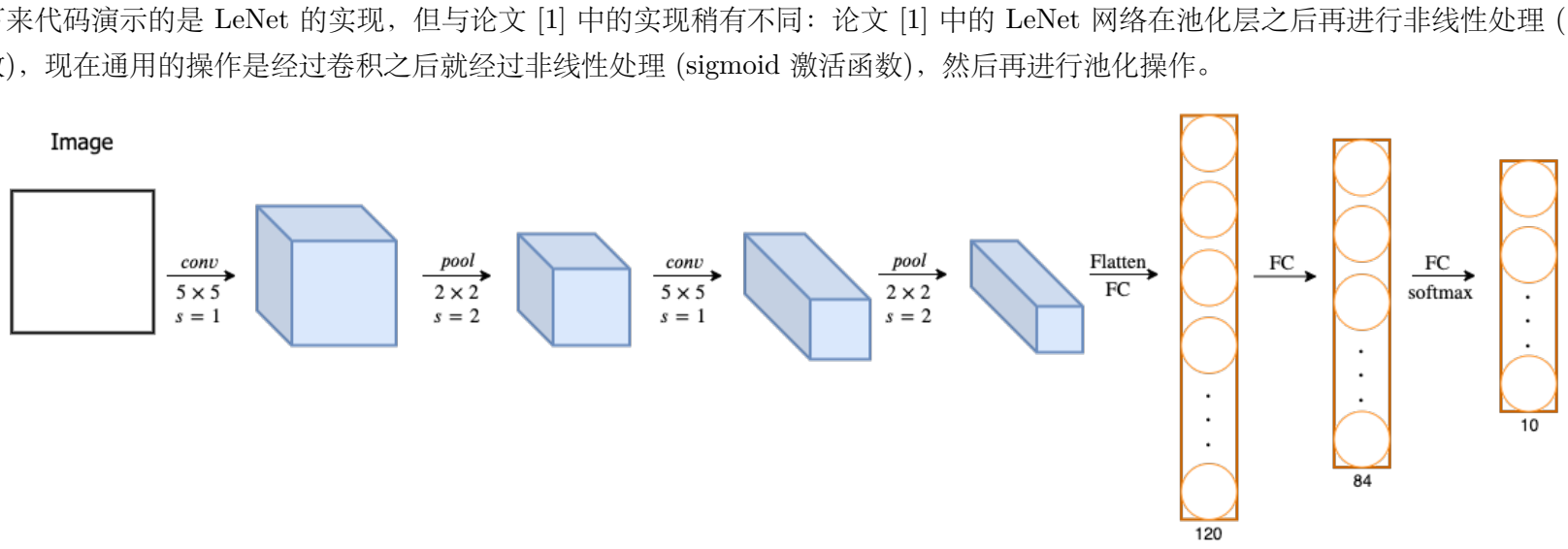


图 8.21. LeNet 网络实现框架。

[1]: Lecun, Y. , Bottou, L. , Bengio, Y. , & Haffner, P. . (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[10]:
```
class LeNet(object):
```

```
    def __init__(
        self,
        fc3_out=128,
        fc4_out=84,
        fc5_out=10,
        conv1_pad=0,
        conv2_pad=0,
        conv1_out_ch=6,
        conv2_out_ch=16,
        conv1_stride=1,
        pool1_stride=2,
        conv2_stride=1,
        pool2_stride=2,
        conv1_kernel_shape=(5, 5),
        pool1_kernel_shape=(2, 2),
        conv2_kernel_shape=(5, 5),
        pool2_kernel_shape=(2, 2),
        optimizer="adam",
        init_w="glorot_normal",
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.fc3_out = fc3_out
        self.fc4_out = fc4_out
        self.fc5_out = fc5_out
        self.conv1_pad = conv1_pad
        self.conv2_pad = conv2_pad
        self.conv1_stride = conv1_stride
        self.conv1_out_ch = conv1_out_ch
        self.pool1_stride = pool1_stride
        self.conv2_out_ch = conv2_out_ch
        self.conv2_stride = conv2_stride
        self.pool2_stride = pool2_stride
        self.conv2_kernel_shape = conv2_kernel_shape
        self.pool2_kernel_shape = pool2_kernel_shape
        self.conv1_kernel_shape = conv1_kernel_shape
        self.pool1_kernel_shape = pool1_kernel_shape
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用: 模型初始化
        Conv1 -> Pool1 -> Conv2 -> Pool2 -> Flatten -> FC3 -> FC4 -> FC5 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["Conv1"] = Conv2D(
            out_ch=self.conv1_out_ch,
            kernel_shape=self.conv1_kernel_shape,
            pad=self.conv1_pad,
            stride=self.conv1_stride,
            acti_fn="sigmoid",
            optimizer=self.optimizer,
            init_w=self.init_w,
        )
        self.layers["Pool1"] = Pool2D(
            mode="max",
            optimizer=self.optimizer,
```

```
            stride=self.pool1_stride,
            kernel_shape=self.pool1_kernel_shape,
        )
        self.layers["Conv2"] = Conv2D(
            out_ch=self.conv1_out_ch,
            kernel_shape=self.conv1_kernel_shape,
            pad=self.conv1_pad,
            stride=self.conv1_stride,
            acti_fn="sigmoid",
            optimizer=self.optimizer,
            init_w=self.init_w,
        )
        self.layers["Pool2"] = Pool2D(
            mode="max",
            optimizer=self.optimizer,
            stride=self.pool2_stride,
            kernel_shape=self.pool2_kernel_shape,
        )
        self.layers["Flatten"] = Flatten(optimizer=self.optimizer)
        self.layers["FC3"] = FullyConnected(
            n_out=self.fc3_out,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC4"] = FullyConnected(
            n_out=self.fc4_out,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC5"] = FullyConnected(
            n_out=self.fc5_out,
            acti_fn="affine(slope=1, intercept=0)",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.is_initialized = True

    def forward(self, X_train):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
```

```
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False, epo_verbose=True):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        verbose：是否每个 batch 输出损失
        epo_verbose：是否每个 epoch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch)
                y_pred_batch = softmax(out)
                batch_loss = self.loss(y_batch, y_pred_batch)
                grad = self.loss.grad(y_batch, y_pred_batch)
                _, _ = self.backward(grad)
                self.update()
                loss += batch_loss
                if self.verbose:
                    fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                    print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))
            loss /= n_batch
            if epo_verbose:
                fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
                print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
            prev_loss = loss

    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
```

```
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "fc3_out": self.fc3_out,
            "fc4_out": self.fc4_out,
            "fc5_out": self.fc5_out,
            "conv1_pad": self.conv1_pad,
            "conv2_pad": self.conv2_pad,
            "conv1_stride": self.conv1_stride,
            "conv1_out_ch": self.conv1_out_ch,
            "pool1_stride": self.pool1_stride,
            "conv2_out_ch": self.conv2_out_ch,
            "conv2_stride": self.conv2_stride,
            "pool2_stride": self.pool2_stride,
            "conv2_kernel_shape": self.conv2_kernel_shape,
            "pool2_kernel_shape": self.pool2_kernel_shape,
            "conv1_kernel_shape": self.conv1_kernel_shape,
            "pool1_kernel_shape": self.pool1_kernel_shape,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

用 LeNet，MNIST 数据集测试

[11]:
```
"""
载入数据，取 10000 条数据用以训练，测试集不变
"""
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1], X_train.shape[2], 1).astype('float32')
X_test = X_test.reshape(-1, X_train.shape[1], X_train.shape[2], 1).astype('float32')
print(X_train.shape, y_train.shape)
N = 10000   # 取 10000 条数据用以训练，测试集不变
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 28, 28, 1) (60000, 10)
(10000, 28, 28, 1) (10000, 10)
```

[12]:
```
model = LeNet()
model.fit(X_train, y_train, n_epochs=15, batch_size=64, epo_verbose=True)
print("Test Accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
[Epoch 1] Avg. loss: 2.265  Delta: inf (17.67m/epoch)
[Epoch 2] Avg. loss: 1.369  Delta: 0.896 (17.33m/epoch)
[Epoch 3] Avg. loss: 0.703  Delta: 0.666 (17.33m/epoch)
[Epoch 4] Avg. loss: 0.489  Delta: 0.214 (17.35m/epoch)
[Epoch 5] Avg. loss: 0.389  Delta: 0.100 (17.41m/epoch)
[Epoch 6] Avg. loss: 0.330  Delta: 0.059 (17.31m/epoch)
[Epoch 7] Avg. loss: 0.287  Delta: 0.043 (17.39m/epoch)
[Epoch 8] Avg. loss: 0.255  Delta: 0.032 (17.31m/epoch)
[Epoch 9] Avg. loss: 0.230  Delta: 0.025 (17.30m/epoch)
[Epoch 10] Avg. loss: 0.209  Delta: 0.022 (17.38m/epoch)
[Epoch 11] Avg. loss: 0.192  Delta: 0.017 (17.31m/epoch)
[Epoch 12] Avg. loss: 0.178  Delta: 0.014 (17.31m/epoch)
[Epoch 13] Avg. loss: 0.163  Delta: 0.015 (17.32m/epoch)
[Epoch 14] Avg. loss: 0.155  Delta: 0.008 (17.32m/epoch)
[Epoch 15] Avg. loss: 0.143  Delta: 0.012 (17.32m/epoch)
Test Accuracy:0.959
```

以下我们再用 GEMM 转换的卷积计算实现 LeNet，比较两者的时间差值。

[13]:
```python
class LeNet_gemm(object):

    def __init__(
        self,
        fc3_out=128,
        fc4_out=84,
        fc5_out=10,
        conv1_pad=0,
        conv2_pad=0,
        conv1_out_ch=6,
        conv2_out_ch=16,
        conv1_stride=1,
        pool1_stride=2,
        conv2_stride=1,
        pool2_stride=2,
        conv1_kernel_shape=(5, 5),
        pool1_kernel_shape=(2, 2),
        conv2_kernel_shape=(5, 5),
        pool2_kernel_shape=(2, 2),
        optimizer="adam",
        init_w="glorot_normal",
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.fc3_out = fc3_out
        self.fc4_out = fc4_out
        self.fc5_out = fc5_out
        self.conv1_pad = conv1_pad
        self.conv2_pad = conv2_pad
        self.conv1_stride = conv1_stride
        self.conv1_out_ch = conv1_out_ch
        self.pool1_stride = pool1_stride
        self.conv2_out_ch = conv2_out_ch
        self.conv2_stride = conv2_stride
        self.pool2_stride = pool2_stride
        self.conv2_kernel_shape = conv2_kernel_shape
        self.pool2_kernel_shape = pool2_kernel_shape
        self.conv1_kernel_shape = conv1_kernel_shape
        self.pool1_kernel_shape = pool1_kernel_shape
```

```
        self.is_initialized = False

    def _set_params(self):
        """
        函数作用：模型初始化
        Conv1 -> Pool1 -> Conv2 -> Pool2 -> Flatten -> FC3 -> FC4 -> FC5 -> Softmax
        """
        self.layers = OrderedDict()
        self.layers["Conv1"] = Conv2D_gemm(
            out_ch=self.conv1_out_ch,
            kernel_shape=self.conv1_kernel_shape,
            pad=self.conv1_pad,
            stride=self.conv1_stride,
            acti_fn="sigmoid",
            optimizer=self.optimizer,
            init_w=self.init_w,
        )
        self.layers["Pool1"] = Pool2D(
            mode="max",
            optimizer=self.optimizer,
            stride=self.pool1_stride,
            kernel_shape=self.pool1_kernel_shape,
        )
        self.layers["Conv2"] = Conv2D_gemm(
            out_ch=self.conv1_out_ch,
            kernel_shape=self.conv1_kernel_shape,
            pad=self.conv1_pad,
            stride=self.conv1_stride,
            acti_fn="sigmoid",
            optimizer=self.optimizer,
            init_w=self.init_w,
        )
        self.layers["Pool2"] = Pool2D(
            mode="max",
            optimizer=self.optimizer,
            stride=self.pool2_stride,
            kernel_shape=self.pool2_kernel_shape,
        )
        self.layers["Flatten"] = Flatten(optimizer=self.optimizer)
        self.layers["FC3"] = FullyConnected(
            n_out=self.fc3_out,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC4"] = FullyConnected(
            n_out=self.fc4_out,
            acti_fn="sigmoid",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.layers["FC5"] = FullyConnected(
            n_out=self.fc5_out,
            acti_fn="affine(slope=1, intercept=0)",
            init_w=self.init_w,
            optimizer=self.optimizer
        )
        self.is_initialized = True
```

```
    def forward(self, X_train):
        Xs = {}
        out = X_train
        for k, v in self.layers.items():
            Xs[k] = out
            out = v.forward(out)
        return out, Xs

    def backward(self, grad):
        dXs = {}
        out = grad
        for k, v in reversed(list(self.layers.items())):
            dXs[k] = out
            out = v.backward(out)
        return out, dXs

    def update(self):
        """
        函数作用：梯度更新
        """
        for k, v in reversed(list(self.layers.items())):
            v.update()
        self.flush_gradients()

    def flush_gradients(self, curr_loss=None):
        """
        函数作用：更新后重置梯度
        """
        for k, v in self.layers.items():
            v.flush_gradients()

    def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False, epo_verbose=True):
        """
        参数说明：
        X_train：训练数据
        y_train：训练数据标签
        n_epochs：epoch 次数
        batch_size：每次 epoch 的 batch size
        verbose：是否每个 batch 输出损失
        epo_verbose：是否每个 epoch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch)
                y_pred_batch = softmax(out)
                batch_loss = self.loss(y_batch, y_pred_batch)
                grad = self.loss.grad(y_batch, y_pred_batch)
                _, _ = self.backward(grad)
```

```
                self.update()
                loss += batch_loss
                if self.verbose:
                    fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                    print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))
            loss /= n_batch
            if epo_verbose:
                fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
                print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
            prev_loss = loss

    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "fc3_out": self.fc3_out,
            "fc4_out": self.fc4_out,
            "fc5_out": self.fc5_out,
            "conv1_pad": self.conv1_pad,
            "conv2_pad": self.conv2_pad,
            "conv1_stride": self.conv1_stride,
            "conv1_out_ch": self.conv1_out_ch,
            "pool1_stride": self.pool1_stride,
            "conv2_out_ch": self.conv2_out_ch,
            "conv2_stride": self.conv2_stride,
            "pool2_stride": self.pool2_stride,
            "conv2_kernel_shape": self.conv2_kernel_shape,
            "pool2_kernel_shape": self.pool2_kernel_shape,
            "conv1_kernel_shape": self.conv1_kernel_shape,
            "pool1_kernel_shape": self.pool1_kernel_shape,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

[14]:
```
model = LeNet_gemm()
model.fit(X_train, y_train, n_epochs=15, batch_size=64, epo_verbose=True)
print("Test Accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
[Epoch 1] Avg. loss: 2.297  Delta: inf (5.20m/epoch)
[Epoch 2] Avg. loss: 1.569  Delta: 0.728 (5.24m/epoch)
[Epoch 3] Avg. loss: 0.751  Delta: 0.818 (5.62m/epoch)
[Epoch 4] Avg. loss: 0.484  Delta: 0.267 (5.52m/epoch)
[Epoch 5] Avg. loss: 0.370  Delta: 0.114 (5.61m/epoch)
[Epoch 6] Avg. loss: 0.310  Delta: 0.059 (5.82m/epoch)
[Epoch 7] Avg. loss: 0.271  Delta: 0.040 (5.62m/epoch)
[Epoch 8] Avg. loss: 0.241  Delta: 0.030 (5.28m/epoch)
```

```
[Epoch 9] Avg. loss: 0.217  Delta: 0.024 (5.28m/epoch)
[Epoch 10] Avg. loss: 0.201  Delta: 0.017 (5.28m/epoch)
[Epoch 11] Avg. loss: 0.183  Delta: 0.017 (5.30m/epoch)
[Epoch 12] Avg. loss: 0.170  Delta: 0.013 (5.27m/epoch)
[Epoch 13] Avg. loss: 0.157  Delta: 0.013 (5.25m/epoch)
[Epoch 14] Avg. loss: 0.147  Delta: 0.010 (5.26m/epoch)
[Epoch 15] Avg. loss: 0.138  Delta: 0.009 (5.24m/epoch)
Test Accuracy:0.9567
```

更多的卷积神经网络及其应用将在第十二章呈现

[15]:
```
import numpy


print("numpy:", numpy.__version__)
```

```
numpy: 1.14.5
```

第九章　实践方法论

9.1　实践方法论

到目前为止，我们一直在谈论很多内容，涉及深度学习的理论方面。但是，理论与实际可行之间还存在很大差距。人们可能需要做出各种选择，包括收集哪种类型的数据，在哪里找到该数据，是否应该收集更多数据，更改模型复杂性，更改 (添加/删除) 正则化，改进优化，调试软件实现等等。建议的实用设计过程如下：

1. 确定目标，决定使用什么样的度量指标 (即用一个单一的数字指标来评估你的模型)——这代表了最终目标。这个度量指标的选择取决于该系统旨在解决的问题。

2. 尽快建立**端到端的工作流程**，包括评估所需指标。也就是说，我们在设计一个系统时，要考虑 “数据”、“模型” 和 “分析” 三个模块的实现。要尽快保证系统可以正确接受输入并预处理 (“数据” 模块)，系统可以以正确的格式生成输出 (“模型” 模块)，系统可以计算度量指标并可视化结果 (“分析” 模块)。于是，我们最早在设计 “模型” 模块时，可以先用一个非常简单的模型。接下来，我们就可以只专注于改进 “模型” 模块中的模型，然后就可以立即获得最终结果，并检查该改进是否优化了度量指标。

3. 很好地对系统进行检测，以确定性能瓶颈，这需要**诊断哪些部分的性能比预期的差**，并了解导致性能不佳的原因——过拟合、欠拟合、建模问题、数据问题、软件实现错误等等。

4. 根据上述诊断，可以通过**添加更多数据、增加模型的容量、调整超参数或通过更好的标注来改善数据质量等等来不断地改进算法**。

为此，本章讨论的内容包括如下：

- 性能度量指标
- 默认的基准模型
- 确定是否收集更多数据
- 选择超参数
- 调试策略

9.2　性能度量指标

如上所述，决定使用哪种错误指标非常重要，因为这最终将指导你如何取得进展。它应该足以代表你要实现的最终目标。例如，我们考虑一个二分类问题–良/恶性乳腺癌肿瘤预测。现在，选择什么合理的指标来代表这里的最终目标呢?

9.2.1　错误率与准确性

首先考虑错误率 (`Error rate`) 和准确性 (`Accuracy`)。

- 错误率：分类错误的样本数占样本总数的比例 err。
- 准确性：分类正确的样本数占样本总数的比例 $acc = 1 - err$。

现在，我们考虑一种情况，如果我们有 1000 个样本，其中 995 个是良性的，只有 5 个人是恶性的。假设我们只关注预测的错误率与准确性，那么如何设计好的模型? 可以设计模型无论是什么样本输入，全部输出良性。此时的准确性是 99.5%，是不是很满意呢? 但这种模型根本没有用。所以说对于数据的类别存在不平衡的情况，不能只看错误率/准确性。

这便引出了查准率、查全率、ROC、AUC。

9.2.2　查准率、查全率与 $\mathbf{F}_1$ 值

混淆矩阵

首先定义**混淆矩阵** (`Confusion matrix`)：

	Positive Prediction	Negative Prediction
Positive Class	TP	FN

	Positive Prediction	Negative Prediction
Negative Class	FP	TN

其中：

- TP (`True Positive`)：表示将正样本预测为正例的数目。即真实结果为 1，预测结果也为 1。
- TN (`True Negative`)：表示将负样本预测为负例的数目。即真实结果为 0，预测结果也为 0。
- FP (`False Positive`)：表示将负样本预测为正例的数目。即真实结果为 0，预测结果为 1。
- FN (`False Negative`)：表示将正样本预测为负例的数目。即真实结果为 1，预测结果为 0。

混淆矩阵的示例如图 9.1 所示，考虑 8 个样本的真实结果和预测结果，根据定义我们可以得到其混淆矩阵。

Class	1	1	1	1	0	0	0	0
Prediction	1	0	1	1	0	1	1	0

	Positive Prediction (1)	Negative Prediction (0)
Positive Class (1)	3	2
Negative Class (0)	1	2

图 9.1. 混淆矩阵示例。上方显示真实类别与预测类别，下方显示混淆矩阵。

通过混淆矩阵，我们可以获得前面描述的错误率和准确性的数学定义：

$$
\begin{aligned}
acc &= \frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}} \\
err &= \frac{\mathrm{FP}+\mathrm{FN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}
\end{aligned}
\tag{9.1}
$$

查准率和查全率的定义与关联

现在我们定义查准率和查全率：

- 查准率 (`Precision`)：也叫精度，简记为 P 或 PPV，表示预测为正例的样本中（$\mathrm{TP}+\mathrm{FP}$）有多少是真正的正样本（TP）。

$$
\mathrm{P} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}} \tag{9.2}
$$

- 查全率 (`Recall`)：也叫召回率，简记为 R 或 TPR，表示在实际真正的正样本中（$\mathrm{TP}+\mathrm{FN}$），预测为正例的样本数（TP）所占的比例。

$$
\mathrm{R} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}} \tag{9.3}
$$

为什么要引入这两个指标?

查准率表示**宁愿漏掉，不可错杀**。在识别垃圾邮件中偏向这种思路，因为我们不希望正常邮件 (对应为负样本，通常将占多数的类别视为负类) 被误杀，这样会造成严重的困扰。

查全率表示**宁愿错杀，不可漏掉**。在金融风控领域偏向这种思路，我们希望系统能够筛选出所有有风险的行为或用户 (对应为正样本)，然后交给人工鉴别，漏掉一个可能造成灾难性后果。

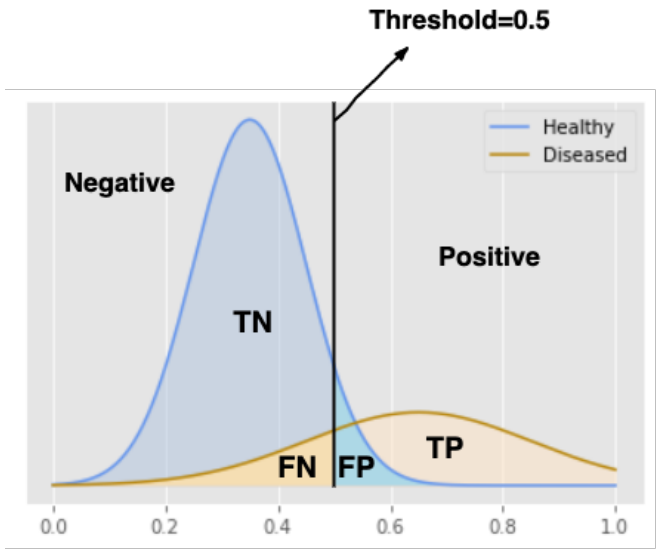


图 9.2. 肿瘤恶性/良性分类器示例。暗黄色高斯曲线表示真正的恶性肿瘤分布，淡蓝色高斯曲线表示真正的良性肿瘤分布。Positive 区域是大于阈值即肿瘤分类器预测为正例的样本。Negative 区域是肿瘤分类器预测为负例的样本。

查准率代表实际检测结果的百分比，而查全率则代表成功检测到的真实事件的比例。如图 9.2 所示，我们构造一个肿瘤恶性/良性分类器 (并非基于真实数据绘制)。暗黄色高斯曲线表示真正的恶性肿瘤分布，对应所有正例样本数 (暗黄色曲线下方面积) 为 TP + FN。淡蓝色高斯曲线表示真正的良性肿瘤分布，对应所有负例样本数 (淡蓝色曲线下方面积) 为 FP + TN。Positive 区域是大于阈值即肿瘤分类器预测为正例的样本 TP + FP。Negative 区域是肿瘤分类器预测为负例的样本 TN + FN。于是，查准率 P 表示在 Positive 中，肿瘤分类器预测对的比重，所以越大越好。查全率 R 表示在暗黄色曲线中，落在 Positive 区域的比重，所以越大越好。

现在，回到最初的问题，考虑如果模型指出所有的肿瘤预测都不是恶性的，那么召回率为 0 (此时一个真正的正样本都没预测到)。查准率和查全率通常是一对矛盾的度量。例如，卖瓜的时候，如果希望查准率高，那就选择那些最有把握的瓜，但这样有些好瓜就不会被选到，查全率并不高；但如果希望查全率高，那就将有把握的和模棱两可的瓜都选上，这样虽然查全率高，但查准率又下降了。

$\mathbf{F}_1$ 值

而很多时候，我们实际上只希望有一个单一的指标来判断，而不是在两个指标之间进行权衡。F_1 值是 “查准率和查全率” 的调和平均值，它是一个广为接受的指标：

$$\mathrm{F}_1 = \frac{2\mathrm{PR}}{\mathrm{P}+\mathrm{R}} \tag{9.4}$$

但是，F_1 值会在查准率和查全率上给予同等的权重。在某些情况下，你可能想偏重一个，因此我们获得了更一般的 F_β 值：

$$\mathrm{F}_\beta = (1+\beta^2)\frac{\mathrm{PR}}{\beta^2\mathrm{P}+\mathrm{R}} \tag{9.5}$$

其中 β 用于调整权重，当 $\beta = 1$ 时两者权重相同，即为 F_1 值。如果认为查准率更重要，则减小 β；若认为查全率更重要，则增大 β。

9.2.3 PR 曲线

但如果你确实想讨论查准率/查全率的关系时，PR 曲线 (`Precision-Recall Curve`) 可以提供帮助。PR 曲线是针对不同阈值的查准率 P (y 轴) 和查全率 R (x 轴) 的图。算法对样本进行分类时都会有置信度，即表示该样本是正例的概率，比如 99% 的概率认为样本 A 是正例，或者 15% 的概率认为样本 B 是正例。通过选择合适的阈值 (比如 50%)，就对样本进行划分，概率大于阈值 (50%) 的就认为是正例，小于阈值 (50%) 的就是负例。

因此，我们考虑选择不同的阈值，并计算不同阈值情况下的查准率和查全率又是如何。具体做法是通过置信度对所有样本进行排序，再逐个样本的选择阈值 (以该样本的置信度作为阈值)，在该样本之前的都视作正例，该样本之后的都视作负例。将每一个样本分别作为划分阈值，并计算对应的查准率和查全率，就可以绘制 PR 曲线。

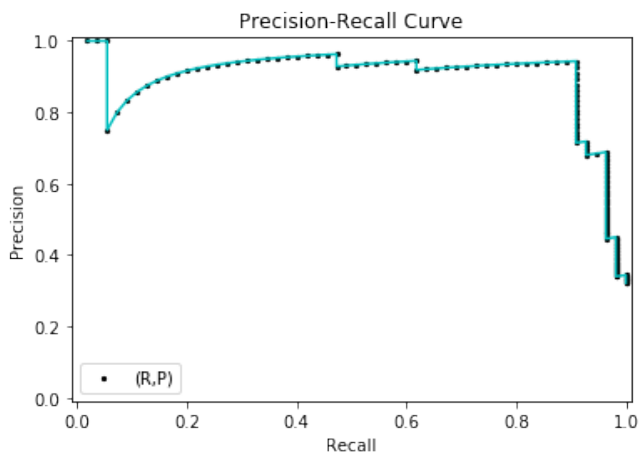


图 9.3. PR 曲线示意图。实现代码见后文代码实现部分。

PR 曲线示意图如图 9.3 所示 (实现代码见后文)。因为是经过排序后逐个样本作为阈值划分点，可以知道随着划分点的移动，被视作正例的样本逐渐增加，于是查全率是单调递增的 (分母考虑所有的，所以不会变化)。而查准率并非递减，越来越多的样本被视作正例，则 TP 和 FP 都可能增加，因此查准率可能会振荡，但整体趋势会下降。一个好的模型表现是给正样本高置信度，负样本低置信度。在此情况下，我们陆续选择置信度作为阈值，好的模型的表现是经过对置信度排序后排在前列的都是正样本，则对应于查全率由 0 到 1 的过程中，查准率一直等于 1 或接近 1。

PR 曲线的注意点：

1. 由于在最后所有的样本都会被视作正例，因此 FN=0，所以 R=TP/(TP+FN)=1。同时，FP= 所有的负样本数，因此 P=TP/(TP+FP)= 正样本占所有样本的比例，可以知道除非样本中负样本数很多，否则 P 不会为 0。如果看到 PR 曲线的终点接近 (1,0) 点，这可能是因为样本中负样本远远多于正样本。

2. 具有完美表现的模型会绘制坐标 (1,1) 的点，PR 曲线表现为一条 y 轴值为 1 的水平线段，外加在 x 轴值为 1 处的垂直线段 (线段终点由样本中正样本所占比例决定)，这表示经过置信度排序后正样本都排在了前列。一个模型的表现可以由一条向坐标 (1,1) 弯曲的曲线表示。无技能的分类器 (即该分类器没有学习) 将是图上的一条水平线，其查准率与数据集中正样本数量成比例，对于一个平衡的数据集而言将是 0.5。

PR 曲线集中在少数类上 (这里我们将少数类作为正样本)，使其成为不平衡二元分类模型的有效诊断。同时，当你需要考虑适合你业务需求的阈值时 (阈值可能非 0.5)，使用 PR 曲线也是不错的选择。

9.2.4 ROC 曲线与 AUC 值

对于二分类的度量，除了上面的查准率 P、查全率 R 以及引申的 PR 曲线，我们还可以通过以下度量方法。首先，需要定义真正例率 (`True Positive Rate`) 和假正例率 (`False Positive Rate`)：

- 真正例率：简记为 TPR，表示当前被预测为正例的样本中，真正的正样本（TP）占实际所有的正样本（TP+FN）的比例。其实也就是查全率或召回率 (表示召回的正样本比例)。

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{9.6}$$

- 假正例率：简记为 FPR，表示当前被错误预测为正例的样本中，真正的负样本（FP）占实际所有的负样本（FP+TN）的比例。

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} \tag{9.7}$$

回到图中，真正例率 TPR 意义等同于前面描述的查全率 R。假正例率 FPR 表示淡蓝色曲线中，落在预测正例 Positive 区域的比重，所以越小越好。

ROC 曲线

如果我们以假正例率 FPR 为 x 轴，真正例率 TPR 为 y 轴，并且随着与在 PR 曲线中相同思路的阈值改变，我们将得到 ROC 曲线 (`Receiver Operating Characteristic Curve`)。示意图如图 9.4 所示 (实现代码见后文)。

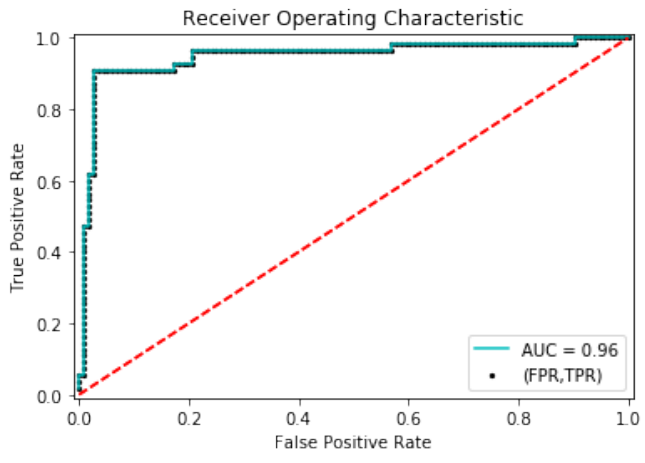


图 9.4. ROC 曲线示意图。实现代码见后文代码实现部分。

ROC 曲线的注意点：

1. 初始时所有样本均视为负例，此时 TP 和 FP 均为 0，故曲线必然经过 (0,0)。最后所有的样本都会被视作正例，此时 FN 和 TN 均为 0，故曲线必然经过 (1,1)。由于阈值点的移动，TP 和 FP 会不断增加，于是 FPR 和 FPR 都是单调递增的 (分母考虑所有的，所以不会变化)。

2. 可以想象，如果一个模型的性能较好，自然是 FPR 越小、TPR 越大，这样越好。所以，理想情况下的 ROC 曲线经过坐标 (0,1) 的点。如果模型的表现较好，我们自然希望在 FPR 很小的时候 TPR 就比较大 (可以参考图理解)，因此一个模型的表现可以由一条向坐标 (0,1) 弯曲的曲线表示。再考虑无技能的分类器 (即该分类器没有学习)，将是图上的一条 $y = x$ 的直线段，因为 TPR 和 FPR 都明显与被视作正例的数目成比例 (分母是除以各自对应的总数，因此没有学习的情况下都是从 0 到 1 沿相同斜率增加)。

AUC 值的计算方法

尽管 ROC 曲线是一种有用的诊断工具，但根据两个或多个分类器的曲线比较它们可能会具有挑战性。为了得到一个描述曲线的数字，我们可以计算 ROC 曲线下的面积或称 ROC AUC 值。显然，ROC 曲线的左上方程度越多，面积越大，ROC AUC 值越高。

那么如何 计算 AUC 值 呢?

梯形法

采用面积的积分公式，计算曲线下各个小矩形的面积之和。

概率法

现在，我们考虑数据集 D：$(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_m, y_m) \in \mathbb{R}^n \times \{0, 1\}$，其中 $\boldsymbol{x}_i$ 是第 i 个 n 维特征的样本。y_i 是第 i 个样本的真实类别 (0 或 1)。

对于一个新的观测样本 $\boldsymbol{x} \in \mathbb{R}^n$，我们通过训练后的分类模型为其得到预测概率 $\hat{p}(\boldsymbol{x})$，表示模型对 $\boldsymbol{x}$ 的标签 $y = 1$ 的置信度。接下来，我们可以选择不同的阈值，然后对每个样本根据置信度和阈值的比较将其分类到类别 0 或类别 1。更进一步，我们计算真正例率和假正例率，并绘制 ROC 曲线。

假设最终的类别 (0 或 1) 是由阈值 $t \in [0, 1]$ 决定，那么真正例率 TPR 可以写作条件概率：

$$T(t) := P[\hat{p}(\boldsymbol{x}) > t \mid \boldsymbol{x} \text{ 属于类别 } 1] \tag{9.8}$$

假正例率 FPR 也可以写作：

$$F(t) := P[\hat{p}(\boldsymbol{x}) > t \mid \boldsymbol{x} \text{不属于类别 } 1] \tag{9.9}$$

简化起见，我们用 $y(\boldsymbol{x}) = 1$ 表示 $\boldsymbol{x}$ 属于类别 1，$y(\boldsymbol{x}) = 0$ 表示 $\boldsymbol{x}$ 不属于类别 1。

ROC 曲线绘制 t 由 1 至 0 时 $T(t)$ 和 $F(t)$ 的关系，于是我们将 T 视作 F 的函数：

$$\begin{aligned}
\text{AUC} &= \int_0^1 T(F_0)\,\mathrm{d}F_0 \\
&= \int_0^1 P(\,\hat{p}(\boldsymbol{x}) > F^{-1}(F_0) \mid y(\boldsymbol{x}) = 1)\,\mathrm{d}F_0 \\
&= \int_1^0 P(\,\hat{p}(\boldsymbol{x}) > F^{-1}(F(t)) \mid y(\boldsymbol{x}) = 1) \cdot \frac{\partial F(t)}{\partial t}\,\mathrm{d}t \\
&= \int_0^1 P(\,\hat{p}(\boldsymbol{x}) > t \mid y(\boldsymbol{x}) = 1) \cdot P(\hat{p}(\boldsymbol{x'}) = t \mid y(\boldsymbol{x'}) = 0)\,\mathrm{d}t \\
&= \int_0^1 P(\,\hat{p}(\boldsymbol{x}) > \hat{p}(\boldsymbol{x'}), \hat{p}(\boldsymbol{x'}) = t \mid y(\boldsymbol{x}) = 1, y(\boldsymbol{x'}) = 0)\,\mathrm{d}t \\
&= P(\,\hat{p}(\boldsymbol{x}) > \hat{p}(\boldsymbol{x'}) \mid y(\boldsymbol{x}) = 1, y(\boldsymbol{x'}) = 0)
\end{aligned}$$

其中，从第三步到第四步是用到累积分布函数 $P[\hat{p}(\boldsymbol{x'}) \le t \,|\, y(\boldsymbol{x'}) = 0] = 1 - F(t)$ 关于 t 的导数为概率密度函数 $P[\hat{p}(\boldsymbol{x'}) = t \,|\, y(\boldsymbol{x'}) = 0]$。所以，我们可以看出，如果给定一个随机选择的观测样本 $\boldsymbol{x}$ 属于类别 1，以及另一个随机选择的观测样本 $\boldsymbol{x'}$ 属于类别 0，AUC 是分类模型给 $\boldsymbol{x}$ 的打分要高于给负例 $\boldsymbol{x'}$ 的打分的概率，即 $\hat{p}(\boldsymbol{x}) > \hat{p}(\boldsymbol{x'})$ 的条件概率。

这便得到了概率法的做法：随机抽出一对样本 (一个正样本，一个负样本)，然后用训练好的分类模型对这两个样本进行预测，预测得到正样本的概率大于负样本概率的概率。在实现时，我们可以先对预测概率升序排序，然后统计有多少正负样本对满足：正样本预测值 > 负样本预测值, 再除以总的正负样本对的数目。统计的过程为：构造初始的当前负样本数目 $Count_{Neg} = 0$，如果样本 $\boldsymbol{x'}$ 的标签 (真实类别) 为 0，则计入当前的负样本数目 $Count_{Neg}$；如果样本 $\boldsymbol{x}$ 的标签为 1，则满足条件的正负样本对数目加上 $Count_{Neg}$，这是因为我们是对概率升序排序，所以一定满足 $\hat{p}(\boldsymbol{x}) > \hat{p}(\boldsymbol{x'})$，于是正样本 $\boldsymbol{x}$ 和 $Count_{Neg}$ 计数的负样本都是满足条件的。这样做的时间复杂度为 $O(m \log m)$，m 为样本数。

9.2.5 覆盖

在一些应用中，机器学习系统可能会拒绝做出判断。如果机器学习算法能够估计所作判断的置信度，这就会非常有用，可以帮助机器决定是否要做出判断。如果错误判断会导致严重危害，而且操作人员又可以偶尔地人工接管，那么当机器学习系统认为某个样本或数据无法操作时，最好的办法是由人来做。但是，只有当机器学习系统确实能够大量降低需要人工操作处理的工作时，它才是有用的。这便是覆盖 (Coverage) 的度量指标：

- 覆盖 (Coverage)：表示机器学习系统能够产生响应的比例。

9.2.6 指标性能的瓶颈

在大多数应用中，即便拥有了无限量的数据，也可能无法实现绝对的零误差，这可能是由于特征的表示能力不足或者系统本质上的随机性。系统可能的最小误差量称为系统的贝叶斯误差。

指标性能的主要瓶颈通常是训练数据有限。现在，如果从 MNIST 之类的标准数据集转入更多实际问题，你就会意识到，获取准确数据要比最初看起来要困难得多，而且在大多数情况下，这并不是免费的。

因此，重要的事情是你需要事先确定好应用程序的实际期望误差率，并用来指导接下来的设计决策。

自定义实现度量指标

[1]:
```
from abc import ABC, abstractmethod
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from chapter5 import Sigmoid, LogisticRegression
import matplotlib.pyplot as plt
%matplotlib inline
import itertools
import time
import re
from scipy.stats import norm
```

[2]:
```
def cal_conf_matrix(labels, preds):
    """
    计算混淆矩阵。

```

```
    参数说明:
    labels: 样本标签 (真实结果)
    preds: 预测结果
    """
    n_sample = len(labels)
    result = pd.DataFrame(index=range(0,n_sample),columns=('probability','label'))
    result['label'] = np.array(labels)
    result['probability'] = np.array(preds)
    cm = np.arange(4).reshape(2,2)
    cm[0,0] = len(result[result['label']==1][result['probability']>=0.5]) # TP, 注意这里是以 0.5 为阈值
    cm[0,1] = len(result[result['label']==1][result['probability']<0.5])  # FN
    cm[1,0] = len(result[result['label']==0][result['probability']>=0.5]) # FP
    cm[1,1] = len(result[result['label']==0][result['probability']<0.5])  # TN
    return cm
```

[3]:
```
def cal_PRF1(labels, preds):
    """
    计算查准率 P, 查全率 R, F1 值。
    """
    cm = cal_conf_matrix(labels, preds)
    P = cm[0,0]/(cm[0,0]+cm[1,0])
    R = cm[0,0]/(cm[0,0]+cm[0,1])
    F1 = 2*P*R/(P+R)
    return P, R, F1
```

[4]:
```
def cal_PRcurve(labels, preds):
    """
    计算 PR 曲线上的值。
    """
    n_sample = len(labels)
    result = pd.DataFrame(index=range(0,n_sample),columns=('probability','label'))
    y_pred[y_pred>=0.5] = 1
    y_pred[y_pred<0.5] = 0
    result['label'] = np.array(labels)
    result['probability'] = np.array(preds)
    result.sort_values('probability',inplace=True,ascending=False)
    PandR = pd.DataFrame(index=range(len(labels)),columns=('P','R'))
    for j in range(len(result)):
        # 以每一个概率为分类的阈值, 统计此时正例和反例的数量
        result_j = result.head(n=j+1)
        P = len(result_j[result_j['label']==1])/float(len(result_j))  # 当前实际为正的数量/当前预测为正的数量
        R = len(result_j[result_j['label']==1])/float(len(result[result['label']==1]))  # 当前真正例的数量/实际为正的数量
        PandR.iloc[j] = [P,R]
    return PandR
```

[5]:
```
def cal_ROCcurve(labels, preds):
    """
    计算 ROC 曲线上的值。
    """
    n_sample = len(labels)
    result = pd.DataFrame(index=range(0,n_sample),columns=('probability','label'))
    y_pred[y_pred>=0.5] = 1
    y_pred[y_pred<0.5] = 0
    result['label'] = np.array(labels)
    result['probability'] = np.array(preds)
    # 计算 TPR,FPR
    result.sort_values('probability',inplace=True,ascending=False)
    TPRandFPR=pd.DataFrame(index=range(len(result)),columns=('TPR','FPR'))
    for j in range(len(result)):
```

```
        # 以每一个概率为分类的阈值，统计此时正例和反例的数量
        result_j=result.head(n=j+1)
        TPR=len(result_j[result_j['label']==1])/float(len(result[result['label']==1]))  # 当前真正例的数量/实际为正的数量
        FPR=len(result_j[result_j['label']==0])/float(len(result[result['label']==0]))  # 当前假正例的数量/实际为负的数量
        TPRandFPR.iloc[j]=[TPR,FPR]
    return TPRandFPR
```

[6]:
```
def timeit(func):
    """
    装饰器，计算函数执行时间
    """
    def wrapper(*args, **kwargs):
        time_start = time.time()
        result = func(*args, **kwargs)
        time_end = time.time()
        exec_time = time_end - time_start
        print("{function} exec time: {time}s".format(function=func.__name__,time=exec_time))
        return result
    return wrapper

@timeit
def area_auc(labels, preds):
    """
    AUC 值的梯度法计算
    """
    TPRandFPR = cal_ROCcurve(labels, preds)
    # 计算 AUC，计算小矩形的面积之和
    auc = 0.
    prev_x = 0
    for x, y in zip(TPRandFPR.FPR,TPRandFPR.TPR):
        if x != prev_x:
            auc += (x - prev_x) * y
            prev_x = x
    return auc

@timeit
def naive_auc(labels, preds):
    """
    AUC 值的概率法计算
    """
    n_pos = sum(labels)
    n_neg = len(labels) - n_pos
    total_pair = n_pos * n_neg  # 总的正负样本对的数目
    labels_preds = zip(labels, preds)
    labels_preds = sorted(labels_preds,key=lambda x:x[1])  # 对预测概率升序排序
    count_neg = 0  # 统计负样本出现的个数
    satisfied_pair = 0   # 统计满足条件的样本对的个数
    for i in range(len(labels_preds)):
        if labels_preds[i][0] == 1:
            satisfied_pair += count_neg  # 表明在这个正样本下，有哪些负样本满足条件
        else:
            count_neg += 1
    return satisfied_pair / float(total_pair)
```

用自定义的度量指标，乳腺癌数据集测试，第五章描述的逻辑回归用于模型训练

[7]:
```
column_names = ['Sample code number','Clump Thickness',
                'Uniformity of Cell Size','Uniformity of Cell Shape',
                'Marginal Adhesion','Single Epithelial Cell Size',
                'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
```

```
data = pd.read_csv('../data/cancer/breast-cancer-wisconsin.data',names=column_names)
data = data.replace(to_replace='?',value=np.nan)  # 非法字符替代
data = data.dropna(how='any')  # 去掉空值，any；出现空值行则删除
print(data.shape)
# 随机采样 25% 的数据用于测试，剩下 75% 用于构建训练集
X_train,X_test,y_train,y_test = train_test_split(data[column_names[1:10]], data[column_names[10]],
                                                 test_size=0.25, random_state=1111)
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 修改标签为 0 和 1
print(y_train.shape)
y_train[y_train==2] = 0
y_train[y_train==4] = 1
y_test[y_test==2] = 0
y_test[y_test==4] = 1
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 数据标准化预处理，保证每个维度特征均值为 0，方差为 1
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
```

```
(683, 11)
2    328
4    184
Name: Class, dtype: int64
(512,)
0    328
1    184
Name: Class, dtype: int64
```

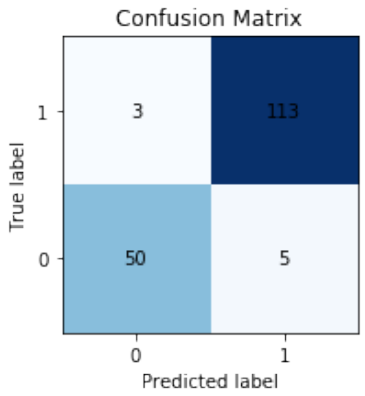
[8]:
```
model = LogisticRegression()
# 使用逻辑回归训练
model.fit(X_train,y_train)
# 预测测试集
y_pred = model.predict(X_test)
```

自定义混淆矩阵测试

[9]:
```
# 计算混淆矩阵
cm = cal_conf_matrix(y_test, y_pred)
print(cm)
# 绘制混淆矩阵
classes = [0,1]
plt.figure(figsize=(3,3))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
tick_marks = np.arange(len(classes))
plt.xlim(-0.5,1.5)
plt.ylim(-0.5,1.5)
plt.xticks(tick_marks, classes)
plt.yticks(tick_marks, classes)
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, cm[i, j], horizontalalignment="center", verticalalignment='center')
    plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')
```

```
[[ 50   5]
 [  3 113]]
```

[9]:
```
Text(0.5, 6.800000000000011, 'Predicted label')
```



用 sklearn 实现混淆矩阵测试

[10]:
```
from sklearn.metrics import confusion_matrix

y_pred_cla = y_pred.copy()
y_pred_cla[y_pred_cla>=0.5] = 1
y_pred_cla[y_pred_cla<0.5] = 0
cm = confusion_matrix(y_test, y_pred_cla)
print(cm)
```

```
[[113   3]
 [  5  50]]
```

自定义 $\mathbf{P}$、$\mathbf{R}$、$\mathbf{F}_1$ 值测试

[11]:
```
P, R, F1 = cal_PRF1(y_test, y_pred)
print(P, R, F1)
```

```
0.9433962264150944 0.9090909090909091 0.9259259259259259
```

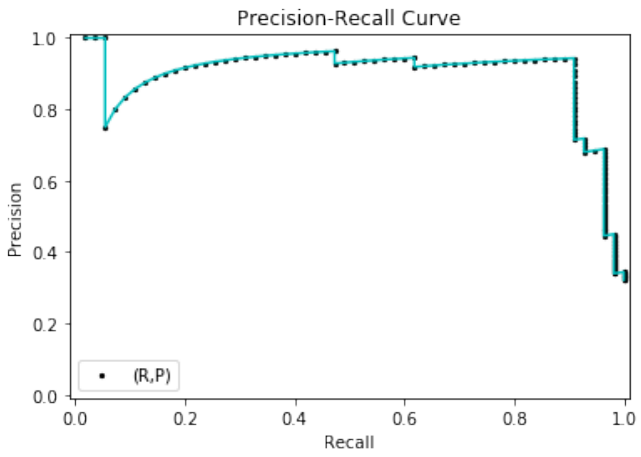
用 sklearn 实现 $\mathbf{P}$、$\mathbf{R}$、$\mathbf{F}_1$ 值测试

[12]:
```
from sklearn.metrics import precision_score, recall_score, f1_score

y_pred_cla = y_pred.copy()
y_pred_cla[y_pred_cla>=0.5] = 1
y_pred_cla[y_pred_cla<0.5] = 0
print(precision_score(y_test, y_pred_cla))
print(recall_score(y_test, y_pred_cla))
print(f1_score(y_test, y_pred_cla))
```

```
0.9433962264150944
0.9090909090909091
0.9259259259259259
```

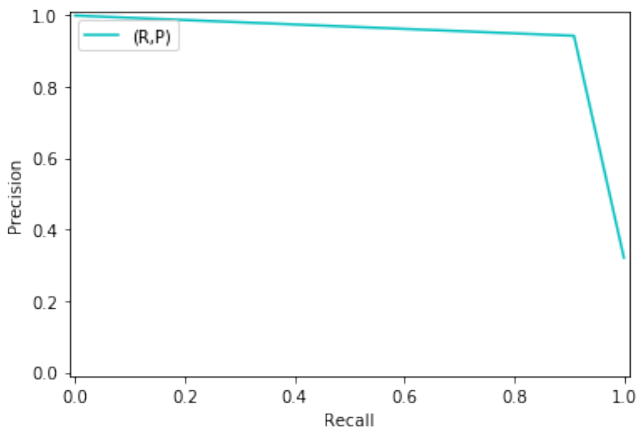
自定义 PR 曲线测试

[13]:
```
# 计算 PR 曲线并绘制
PandR = cal_PRcurve(y_test, y_pred)
plt.scatter(x=PandR['R'],y=PandR['P'],label='(R,P)',color='k',s=5)
plt.plot(PandR['R'], PandR['P'],color='c')
plt.title('Precision-Recall Curve')
plt.xlim([-0.01,1.01])
plt.ylim([-0.01,1.01])
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```



用 sklearn 实现 PR 曲线测试

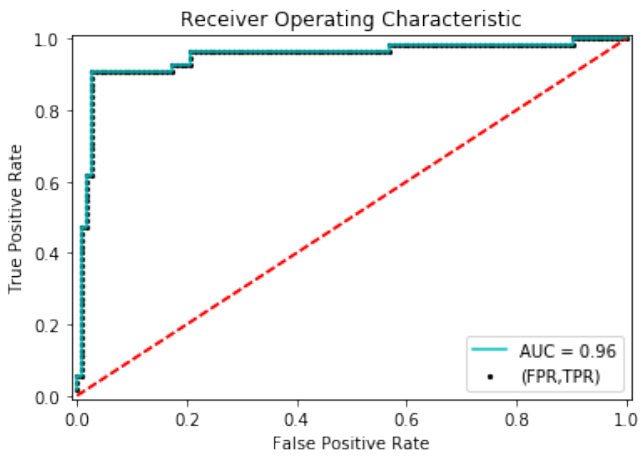
[14]:
```
from sklearn.metrics import precision_recall_curve
precision, recall, _ = precision_recall_curve(y_test, y_pred)
plt.plot(recall, precision, color='c', label='(R,P)')
plt.xlim([-0.01,1.01])
plt.ylim([-0.01,1.01])
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.legend()
plt.show()
```



自定义 ROC 曲线和 AUC 值测试

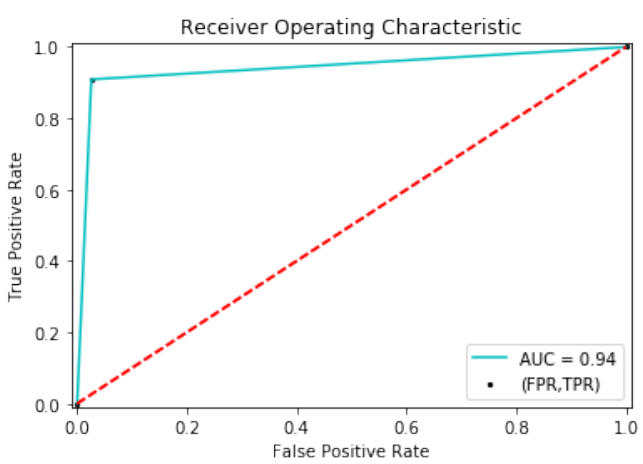
[15]:
```
# 绘制 ROC 曲线和 AUC 值
area_AUC= area_auc(y_test, y_pred)
naive_AUC = naive_auc(y_test, y_pred)
TPRandFPR = cal_ROCcurve(y_test, y_pred)
plt.scatter(x=TPRandFPR['FPR'],y=TPRandFPR['TPR'],label='(FPR,TPR)',color='k',s=5)
plt.plot(TPRandFPR['FPR'], TPRandFPR['TPR'], 'c',label='AUC = %0.2f'% naive_AUC)
plt.legend(loc='lower right')
plt.title('Receiver Operating Characteristic')
plt.plot([(0,0),(1,1)],'r--')
plt.xlim([-0.01,1.01])
plt.ylim([-0.01,01.01])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

```
area_auc exec time: 0.4615659713745117s
naive_auc exec time: 0.00011801719665527344s
```



用 sklearn 实现 ROC 曲线和 AUC 值测试

[16]:
```
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score

fpr, tpr, _ = roc_curve(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred)
plt.scatter(x=fpr,y=tpr,label='(FPR,TPR)',color='k',s=5)
plt.plot(fpr, tpr, 'c',label='AUC = %0.2f'% roc_auc)
plt.legend(loc='lower right')
plt.title('Receiver Operating Characteristic')
plt.plot([(0,0),(1,1)],'r--')
plt.xlim([-0.01,1.01])
plt.ylim([-0.01,01.01])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



9.3 默认基准模型

如开始时提到的，尽快建立一个有效的端到端系统非常重要。根据问题的复杂性，我们甚至可能选择从一个非常简单的模型开始，例如逻辑回归。但是，如果你要解决的问题属于 “完全 AI” 的，例如图像分类，语音识别等，那么从深度学习模型入手几乎总是更好。

首先，你要根据数据的结构选择适合的基准模型。如果你的数据由固定大小的向量组成，并且你打算执行监督学习任务，那么可以使用多层感知器。如果你的数据具有固定的拓扑结构，则使用卷积神经网络可能是最好的方法。同样，如果你的数据具有顺序模式，则循环神经网络将是理想的起点。然而，深度学习在不断发展，默认的算法可能会改变。例如，3-4 年前，AlexNet 将是基于图像任务的理想起点。但是，现在 ResNets 是广泛接受的默认选择。

为了训练模型，一个合理的起点是使用 Adam 优化器。除此之外，具有动量和学习率衰减的 SGD 也被广泛使用，其中，学习率呈指数衰减直至一个点，然后每次验证误差稳定时，线性减小 2-10 倍。通常，批标准化可以提供稳定性并允许使用更大的学习率帮助更快地达到收敛，从而提高性能。

随着模型复杂度的增加，由于训练数据有限，最终将变得易于过拟合。因此，建议也为模型添加一些正则化。常见的选择包括损失函数的 L^2 正则化，Dropout，提前终止和批标准化。使用批标准化可以不用 Dropout。(请详见第七章和第八章)

如果你的任务与已有的先前工作的一些其他任务相似，建议你仅从后者复制模型 (以及权重)，然后将其用作任务的初始值。这种训练方式称为迁移学习 (`Transfer Learning`)。例如，在 Kaggle 上的 Dogs Vs Cats 图像分类挑战中，使用 ImageNet 上预训练的包含相似图像的模型作为获得最佳性能的起点，而不是从头开始训练模型。

最后，某些领域 (如自然语言处理 (NLP)) 在初始化过程中使用无监督的学习方法会极大地受益。在当前应用于 NLP 的深度学习趋势中，通常将每个单词表示为词嵌入 (向量)，并且存在诸如 word2vec 和 GLoVe 之类的无监督学习方法来学习这些词嵌入 (Word Embedding)。

9.4 确定是否收集更多数据

很多人容易犯的错误是，他们不断尝试使用不同的算法来提高其模型的性能，但**简单地改善他们拥有的数据或收集更多数据则可能是最好的改进方法**。由于数据是使 AI 方案正常工作的最重要部分，因此我们现在将对此进行更详细的探讨。

那么，你如何决定何时获取更多数据？首先，如果模型在训练集上的性能很差，则说明它没有充分利用数据中存在的信息，在这种情况下，你需要通过增加层数或增加每层中隐藏单元的数量来增加模型的复杂性。同样，**调整超参数**是要执行的重要步骤。你可能会惊讶于选择正确的超参数对使模型正常工作会产生多么大的影响。例如，学习率是你最重要的也是最需要调整的超参数。为你的问题设置正确的学习率值可以节省大量时间。但是，如果你的模型相当复杂并且对优化进行了仔细的调整，但性能仍未达到所需的水平，则问题可能出在数据的质量上，你必须收集更干净的数据。

为了强调数据在现代深度网络中的重要性，对于那些可能不知道的人来说，深度学习之所以开始受到关注的原因是 ImageNet 竞赛，其中在 2012 年，深度学习模型以显着优势远远超过了以前的最佳模型。ImageNet 由数百万个带标签的图像组成，并且创建类似的大标签数据集是当今诸如对象检测之类的极为复杂的问题已成为解决问题的原因。

训练误差通常会随着数据集大小的增加而增加。这是因为该模型会发现现在很难准确地适合所有数据点。而且，通过增加数据集的大小，由于模型现在将变得更加通用，因此你的验证误差 (dev) 或者测试误差将减少。另外一种情况，考虑模型，训练误差低但测试误差高的特定情况称为过拟合，是训练深度模型中最常见的问题之一，在这种情况下，正则化可能会有所帮助。

9.5 选择超参数

大多数深度学习算法具有许多需要正确选择的超参数。不同的超参数控制模型的不同方面。有些影响内存成本，例如要使用的层数，而另一些影响性能，如 Dropout 的保留概率，学习率，动量等。广泛地，有两种选择这些超参数的方法。第一个是手动选择它们，这需要了解超参数的作用以及它们如何影响训练和泛化。另一种方法是自动选择超参数，这大大降低了复杂性，但以计算能力为代价。现在，我们将更详细地讨论这两种方法：

9.5.1 手动超参数调整

手动超参数调整需要大量领域知识，并对训练误差，泛化误差，学习理论等有基本的了解。手动超参数调整的主要目的是通过平衡内存和运行时间，实现与任务复杂度匹配的有效容量。影响有效容量的因素是模型的表示能力，表示能力指最小化训练模型的代价函数以及正则化程度的学习算法的能力。

许多超参数以不同的方式影响过拟合 (或欠拟合)。例如，增加某些超参数 (如隐藏单元的数量) 会增加过拟合的可能，而增加其他超参数 (如权重衰减) 会减少过拟合的可能。它们中的一些是离散的，例如隐藏单元的数量，而另一些则可能是二进制的，例如是否使用批标准化。一些超参数具有隐式限制它们的界限，例如权重衰减只能减少容量。因此，如果模型欠拟合，则无法通过改变权重衰减使其过拟合。

如前所述，如果你只能调整一个超参数，请调整学习率。在正确的学习率下，模型的有效容量是最高的，这里学习率既不要不太高也不要太低。将学习率设置得太低会降低训练速度，甚至可能导致算法陷入局部极小值；设置得太高可能会由于剧烈振荡而导致训练不稳定。

如果训练误差高，通常的方法是添加更多的层或更多的隐藏单元以增加容量。如果训练误差低但测试误差高，则需要减小训练误差和测试误差之间的差距，而又不要过多增加训练误差。通常，一个充分大的且经过良好正则化的模型 (例如，通过使用 Dropout，批处理归一化，权重衰减等) 效果最佳。实现低泛化的最终目标的两种主要方法是：向模型添加正则化和增加数据集大小。下表显示了每个超参数如何影响容量：

超参数	容量何时增加	原因	注意事项
隐藏单元数量	增加	增加隐藏单元数量会增加模型的表示能力。	几乎模型每个操作所需的时间和内存代价都会随隐藏单元数量的增加而增加。
学习率	调至最优	不正确的学习速率，不管是太高还是太低都会由于优化失败而导致低有效容量的模型。	
卷积核宽度	增加	增加卷积核宽度会增加模型的参数数量。	较宽的卷积核导致较窄的输出尺寸，除非使用隐式零填充减少此影响，否则会降低模型容量。较宽的卷积核需要更多的内存存储参数，并会增加运行时间，但较窄的输出会降低内存代价。
隐式零填充	增加	在卷积之前隐式添加零能保持较大尺寸的表示。	大多数操作的时间和内存代价会增加。
权重衰减系数	降低	降低权重衰减系数使得模型参数可以自由地变大。	

超参数	容量何时增加	原因	注意事项
Dropout 比率	降低	较少地丢弃单元可以更多地让单元彼此“协力”来适应训练集。	

9.5.2 自动超参数优化算法

超参数调整可以看作是优化过程本身，它可以优化目标函数 (例如验证)，有时还受到训练时间、内存限制等约束。因此，我们可以设计超参数优化 (`Hyperparameter Optimization`) 算法包装学习算法并选择其超参数。不幸的是，这些 HO 算法有它们自己的一组超参数，但是这些超参数通常更容易选择，我们现在将讨论这些 HO 算法：

网格搜索 (`Grid Search`)

对于网格搜索，首先选择一个你认为适合每个超参数的值范围。然后，为超参数值的每种可能组合训练模型。为简化起见，如果你有 2 个超参数并为每个参数选择一个 N 个值的范围，则需要针对所有可能的 N^2 的组合训练模型。通常，你会根据自己的理解 (或经验) 来设置范围的最大值和最小值，然后通常以对数刻度在两者之间选择一个值。例如，学习率的可能值：{0.1, 0.01, 0.001}，隐藏单位数：{50, 100, 200, 400} 等。

此外，网格搜索在重复执行时效果最佳。例如，如果初始设置的范围是 {0.1, 0, 1}，而性能最好的时候值是 1，则可能是该范围设置错误。应该再次检查较高的范围，例如 {1, 2, 3}。如果此时效果最好的值为 0，则应在 {-0.1, 0, 0.1} 之间进行更精细的搜索。

网格搜索的主要问题是计算成本。如果要调整 m 个超参数，并且每个参数都可以取 N 值，则训练和评估试验的次数将以 $O(N^m)$ 形式增长。

随机搜索 (`Random Search`)

一种更好、更快的方法是随机搜索。在这种情况下，你可以在值的选择上定义一些分布，例如，对于二进制的用二项分布，对于离散的用多项分布，在对数刻度上用均匀分布 (学习率)：

然后，对于每次运行，根据每个超参数的分布对其值进行随机抽样。事实证明，这比网格搜索更有效。下图对此进行了解释：

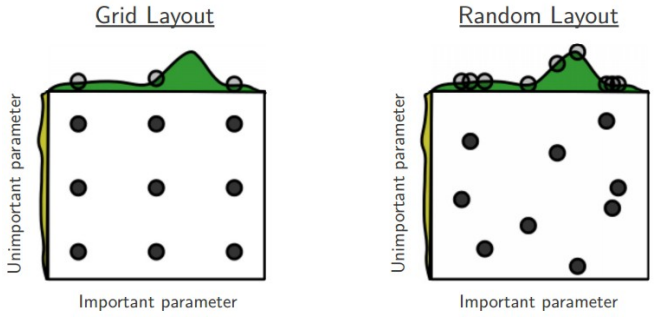


图 9.5. 左图为网格搜索，右图为随机搜索，横轴表示重要超参数的值，纵轴表示不重要超参数的值。

如图 9.5 所示，左边为网格搜索，横轴表示重要超参数的值，纵轴表示不重要超参数的值。当有超参数对性能度量没有显著影响时 (纵轴)，随机搜索比网格搜索高效。原因体现在**没有浪费的实验**。例如左图中给定横轴的一个超参数值，网格会对纵轴超参数的 3 个不同值给出相同结果，其实造成了浪费，这样的结果是虽然做了 9 次实验但本质上只有 3 次有用的。但是在右图的随机搜索中，每次超参数通常具有不同值，每次都是独立的探索和实验，这样就实质上做了 9 次实验。所以，更清楚地说，随机搜索比网格搜索更快的主要原因就是它不执行任何浪费的计算。

基于模型的超参数优化 (`Model-based Hyperparameter Optimization`)

如上所述，超参数调整可以看作是一个优化过程。在简化的设置中，可以针对超参数在验证集上采用一些可微分的误差度量的梯度，再使用梯度下降即可。但是，在大多数实际设置中，这种梯度是不可用的，可能是因为高额的计算代价和存储成本，也可能是压根儿就不可导。为了弥补这一点，你可以对验证误差建模并对该模型执行优化。一种通用方法是建立贝叶斯回归模型以估计验证误差的期望值以及该估计的不确定性。但贝叶斯超参数优化 (`Bayesian Hyperparameter Optimization`) 仍处于新生阶段，不够可靠。其基本想法是为了找到目标函数 ($f(\theta)$) 的最大值，这里的 θ 代表一组超参数，$f(\theta)$ 是超参数放入黑盒模型 (目标函数) 后的输出 (比如这组超参对应的神经网络的验证集的准确率)，即 $\theta_t = \arg\max f(\theta)$。第 t 次迭代我们生成一个 θ_t，将 $(\theta_t, f(\theta_t))$ 加到我们已有的观测到的数据集合里，然后进行下一次迭代，得到下一个样本 θ_{t+1}。现在问题的核心是在每次迭代里如何选择要观测哪个 θ_t。

在贝叶斯优化中 θ_t 是通过优化另一个函数来选择的：采集函数 (`Acquisition Function`) α_t。即 $\theta_t = \arg\max \alpha_t(\theta)$。在选下一个超参数点 θ_t 的时候，我们既想要去尝试那些我们之前没有探索过的区域的点 (探索，`Exploration`)，又想要去根据我们目前已经观测到的所有点的预测选择预测值可能比较大的点 (开发，`Exploitation`)。为了能很好地平衡两个需求，对于域中里面任意一个点 θ_t，我们既需要预测对应的 $f(\theta_t)$ 的值 (针对开发)，又需要知道对应的 $f(\theta_t)$ 的不确定性程度 (`Uncertainty`) (针对探索)。我们通常使用代理函数 (`Surrogate Function`) 同时得到这两个需求，首选方法是高斯过程回归。再通过采集函数基于这两个需求采样下一个点。如此迭代。

那么，这里我们就需要详细说明高斯过程回归，再描述采集函数的方法。为了阐述高斯过程回归，下面会依贝叶斯线性回归、核函数到高斯过程回归的顺序进行描述。

贝叶斯线性回归

在第七章介绍过，线性回归在当噪声服从高斯分布的时候，最小二乘损失最后导出的结果相当于对概率模型应用 MLE。而进一步引入参数的先验时，如果先验分布是高斯分布，那么 MAP 的结果相当于岭回归的正则化，如果先验是拉普拉斯分布，那么相当于 Lasso 的正则化。这两种方案都是点估计方法。我们希望利用贝叶斯方法来求解参数的后验分布。

这里线性回归的模型假设为：

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{w}^\top \boldsymbol{x} \\ y &= f(\boldsymbol{x}) + \epsilon \\ \epsilon &\sim N(0, \sigma^2) \end{aligned} \tag{9.10}$$

在贝叶斯方法中，需要解决推断和预测两个问题。

推断

引入高斯先验：

$$p(\boldsymbol{w}) = N(\mathbf{0}, \boldsymbol{\Sigma}_w) \tag{9.11}$$

对参数的后验分布进行推断：

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) = \frac{p(\boldsymbol{w}, \boldsymbol{y} \mid \boldsymbol{X})}{p(\boldsymbol{y} \mid \boldsymbol{X})} = \frac{p(\boldsymbol{y} \mid \boldsymbol{w}, \boldsymbol{X})\, p(\boldsymbol{w} \mid \boldsymbol{X})}{\int p(\boldsymbol{y} \mid \boldsymbol{w}, \boldsymbol{X})\, p(\boldsymbol{w} \mid \boldsymbol{X}) d\boldsymbol{w}} \tag{9.12}$$

分母和参数无关，而 $p(\boldsymbol{w} \mid \boldsymbol{X}) = p(\boldsymbol{w})$，代入先验得到：

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto \prod_{i=1}^{m} p(y^{(i)} \mid \boldsymbol{w}^\top \boldsymbol{x}^{(i)}) N(\mathbf{0}, \boldsymbol{\Sigma}_w) \tag{9.13}$$

高斯分布取高斯先验的共轭分布依然是高斯分布，于是最终得到的后验分布也会是高斯分布。当然，我们首先看一下这里的第一项：

$$\begin{aligned} \prod_{i=1}^{m} p(y^{(i)} \mid \boldsymbol{w}^\top \boldsymbol{x}^{(i)}) &= \frac{1}{(2\pi)^{m/2}\sigma^m} \exp\left(-\frac{1}{2\sigma^2} \sum_{i=1}^{m} (y^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}^{(i)}) \right) \\ &= \frac{1}{(2\pi)^{m/2}\sigma^m} \exp\left(-\frac{1}{2} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^\top (\sigma^{-2}\boldsymbol{I}) (\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{w}) \right) \\ &= N(\boldsymbol{X}\boldsymbol{w}, \sigma^{-2}\boldsymbol{I}) \end{aligned} \tag{9.14}$$

代回原式：

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto \exp\left(-\frac{1}{2} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^\top (\sigma^{-2}\boldsymbol{I}) (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) - \frac{1}{2} \boldsymbol{w}^\top \boldsymbol{\Sigma}_w^{-1} \boldsymbol{w} \right) \tag{9.15}$$

假定最后得到的高斯分布记为 $N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$。于是对于上面的分布，采用配方的方式来得到 $\boldsymbol{\mu}$ 和 $\boldsymbol{\Sigma}$。

- 首先分析指数上面的二次项：

$$-\frac{1}{2\sigma^2} \boldsymbol{w}^\top \boldsymbol{X}^\top \boldsymbol{X} \boldsymbol{w} - \frac{1}{2} \boldsymbol{w}^\top \boldsymbol{\Sigma}_w^{-1} \boldsymbol{w} \quad \Rightarrow \quad \boldsymbol{\Sigma}^{-1} = \sigma^{-2} \boldsymbol{X}^\top \boldsymbol{X} + \boldsymbol{\Sigma}_w^{-1} \triangleq \boldsymbol{A} \tag{9.16}$$

- 首先分析指数上面的一次项：

$$\begin{aligned} \frac{1}{2\sigma^2} 2 \boldsymbol{y}^\top \boldsymbol{X} \boldsymbol{w} = \sigma^{-2} \boldsymbol{y}^\top \boldsymbol{X} \boldsymbol{w} \quad &\Rightarrow \quad \boldsymbol{\mu} \boldsymbol{\Sigma}^{-1} = \sigma^{-2} \boldsymbol{y}^\top \boldsymbol{X} \\ &\Rightarrow \quad \boldsymbol{\mu} = \sigma^{-2} \boldsymbol{A}^{-1} \boldsymbol{y}^\top \boldsymbol{X} \end{aligned} \tag{9.17}$$

预测

给定一个 $\boldsymbol{x}^*$，求解 y^*，此时是有 $f(\boldsymbol{x}^*) = \boldsymbol{w}^\top \boldsymbol{x}^* = \boldsymbol{x}^{*\top} \boldsymbol{w}$。代入参数后验，有 $\boldsymbol{x}^{*\top} \boldsymbol{w} = N(\boldsymbol{x}^{*\top} \boldsymbol{\mu}, \boldsymbol{x}^{*\top} \boldsymbol{\Sigma} \boldsymbol{x}^*)$。再考虑噪声，得到 $y^* = f(\boldsymbol{x}^*) + \epsilon$：

$$\begin{aligned} p(y^* \mid \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{x}^*) &= \int p(y^* \mid \boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{x}^*) p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{x}^*) d\boldsymbol{w} \\ &= \int p(y^* \mid \boldsymbol{w}, \boldsymbol{x}^*) p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) d\boldsymbol{w} \\ &= N(\boldsymbol{x}^{*\top} \boldsymbol{\mu}, \boldsymbol{x}^{*\top} \boldsymbol{\Sigma} \boldsymbol{x}^* + \sigma^2) \end{aligned} \tag{9.18}$$

核函数

在第五章介绍支持向量机时我们简单提过核方法，在分类问题中，核方法是将低维空间中的严格不可分的数据转化为高维空间中线性可分的数据(核函数的应用之一)。我们用一个特征转换函数作用到低维空间的数据。而不可分数据在通过特征变换后，往往需要求得变换后的内积。但是将数据从低维变到高维后，我们其实是很难求得变换函数的内积。

现在，我们直接引入内积的变换函数：

$$\forall \boldsymbol{x}, \boldsymbol{x'} \in \mathcal{X}, \exists \phi \in \mathcal{H} : x \to z, \quad k(\boldsymbol{x}, \boldsymbol{x'}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{x'}) \tag{9.19}$$

称 $k(\boldsymbol{x}, \boldsymbol{x'})$ 为核函数，ϕ 为特征转换函数，其中 $\mathcal{H}$ 是 Hilbert 空间 (完备的线性内积空间)。我们直接求解变换后的内积复杂难算，在实际中，通常都是使用核函数求解内积。

例如，考虑 RBF 核 $k(\boldsymbol{x},\boldsymbol{x'})=\exp\left(-\frac{(\boldsymbol{x}-\boldsymbol{x'})^2}{2\sigma^2}\right)$，便有：

$$\begin{aligned}\exp\left(-\frac{(\boldsymbol{x}-\boldsymbol{x'})^2}{2\sigma^2}\right)&=\exp\left(-\frac{\boldsymbol{x}^2}{2\sigma^2}\right)\exp\left(-\frac{\boldsymbol{x}\boldsymbol{x'}}{\sigma^2}\right)\exp\left(-\frac{\boldsymbol{x'}^2}{2\sigma^2}\right)\\&=\exp\left(-\frac{\boldsymbol{x}^2}{2\sigma^2}\right)\sum_{n=0}^{\infty}\frac{\boldsymbol{x}^n\boldsymbol{x'}^n}{\sigma^{2n}n!}\exp\left(-\frac{\boldsymbol{x'}^2}{2\sigma^2}\right)\\&=\exp\left(-\frac{\boldsymbol{x}^2}{2\sigma^2}\right)\psi^\top(\boldsymbol{x})\psi(\boldsymbol{x'})\exp\left(-\frac{\boldsymbol{x'}^2}{2\sigma^2}\right)\\&=\phi^\top(\boldsymbol{x})\phi(\boldsymbol{x'})\end{aligned}\tag{9.20}$$

这里，我们的复杂的特征转换函数 $\phi(\boldsymbol{x})=\exp\left(-\frac{\boldsymbol{x}^2}{2\sigma^2}\right)\psi(\boldsymbol{x})$。

我们常说的核函数其实是正定核函数。正定核函数要求核函数满足两个条件：

- 对称性，即 $k(\boldsymbol{x},\boldsymbol{x'})=k(\boldsymbol{x'},\boldsymbol{x})$。
- 正定性，即 $\forall m,\boldsymbol{x}^{(1)},...,\boldsymbol{x}^{(m)}\in\mathcal{X}$，对应的 Gram 矩阵 $K=[k(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)})]$ 是半正定的。

高斯过程回归 (核贝叶斯线性回归)

贝叶斯线性回归可以通过加入核函数的方法来解决非线性函数的问题，将 $f(\boldsymbol{x})=\boldsymbol{x}^\top\boldsymbol{w}$ 这个函数变为 $f(\boldsymbol{x})=\phi(\boldsymbol{x})^\top\boldsymbol{w}$，变换到更高维的空间，有：

$$f(\boldsymbol{x}^*)=N(\phi(\boldsymbol{x}^*)^\top\sigma^{-2}\boldsymbol{A}^{-1}\boldsymbol{\Phi}^\top\boldsymbol{y},\phi(\boldsymbol{x}^*)^\top\boldsymbol{A}^{-1}\phi(\boldsymbol{x}^*))\tag{9.21}$$

其中 $\boldsymbol{A}=\sigma^{-2}\boldsymbol{\Phi}^\top\boldsymbol{\Phi}+\boldsymbol{\Sigma}_w^{-1}$，$\boldsymbol{\Phi}=(\phi(\boldsymbol{x}^1),\phi(\boldsymbol{x}^2)...\phi(\boldsymbol{x}^m))^\top$。为了求解 $\boldsymbol{A}^{-1}$，可以利用 Woodbury Formula，即：

$$(A+UCV)^{-1}=A^{-1}+A^{-1}U(C^{-1}+VA^{-1}U)^{-1}VA^{-1}\tag{9.22}$$

令 $A=\boldsymbol{\Sigma}_w^{-1},C=\sigma^{-2}\boldsymbol{I}$，可以得到协方差矩阵 $\boldsymbol{\Sigma}$：

$$\boldsymbol{A}^{-1}=\boldsymbol{\Sigma}_w-\boldsymbol{\Sigma}_w\boldsymbol{\Phi}(\sigma^2\boldsymbol{I}+\boldsymbol{\Phi}^\top\boldsymbol{\Sigma}_w\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^\top\boldsymbol{\Sigma}_w\tag{9.23}$$

另一方面，我们可以对 $\boldsymbol{A}=\sigma^{-2}\boldsymbol{\Phi}^\top\boldsymbol{\Phi}+\boldsymbol{\Sigma}_w^{-1}$ 两边变换 (同时左乘/右乘矩阵)，得到均值表达式 $\boldsymbol{\mu}$：

$$\begin{aligned}&\boldsymbol{A}=\sigma^{-2}\boldsymbol{\Phi}^\top\boldsymbol{\Phi}+\boldsymbol{\Sigma}_w^{-1}\\\Rightarrow\quad&\boldsymbol{A}\boldsymbol{\Sigma}_w\boldsymbol{\Phi}^\top=\sigma^{-2}\boldsymbol{\Phi}^\top\boldsymbol{\Phi}\boldsymbol{\Sigma}_w\boldsymbol{\Phi}^\top+\boldsymbol{\Phi}^\top=\sigma^{-2}\boldsymbol{\Phi}^\top(\boldsymbol{\Phi}\boldsymbol{\Sigma}_w\boldsymbol{\Phi}^\top+\sigma^2\boldsymbol{I})\\\Rightarrow\quad&\boldsymbol{\Sigma}_w\boldsymbol{\Phi}^\top=\sigma^{-2}\boldsymbol{A}^{-1}\boldsymbol{\Phi}^\top(\boldsymbol{\Phi}\boldsymbol{\Sigma}_w\boldsymbol{\Phi}^\top+\sigma^2\boldsymbol{I})\\\Rightarrow\quad&\sigma^{-2}\boldsymbol{A}^{-1}\boldsymbol{\Phi}^\top=\boldsymbol{\Sigma}_w\boldsymbol{\Phi}^\top(\boldsymbol{\Phi}\boldsymbol{\Sigma}_w\boldsymbol{\Phi}^\top+\sigma^2\boldsymbol{I})^{-1}\\\Rightarrow\quad&\phi(\boldsymbol{x}^*)^\top\sigma^{-2}\boldsymbol{A}^{-1}\boldsymbol{\Phi}^\top\boldsymbol{y}=\phi(\boldsymbol{x}^*)^\top\boldsymbol{\Sigma}_w^\top\boldsymbol{\Phi}(\boldsymbol{\Phi}\boldsymbol{\Sigma}_w\boldsymbol{\Phi}^\top+\sigma^2\boldsymbol{I})^{-1}\boldsymbol{y}\end{aligned}\tag{9.24}$$

我们同时将 $\boldsymbol{A}^{-1}$ 代回预测过程：

$$\phi(\boldsymbol{x}^*)^\top\boldsymbol{A}^{-1}\phi(\boldsymbol{x}^*)=\phi(\boldsymbol{x}^*)^\top\boldsymbol{\Sigma}_w\phi(\boldsymbol{x}^*)-\phi(\boldsymbol{x}^*)^\top\boldsymbol{\Sigma}_w\boldsymbol{\Phi}(\sigma^2\boldsymbol{I}+\boldsymbol{\Phi}^\top\boldsymbol{\Sigma}_w\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^\top\boldsymbol{\Sigma}_w\phi(\boldsymbol{x}^*)\tag{9.25}$$

我们可以看到，在均值向量和协方差中，由这四项构成：$\phi(\boldsymbol{x}^*)^\top\boldsymbol{\Sigma}_w^\top\boldsymbol{\Phi}$，$\phi(\boldsymbol{x}^*)^\top\boldsymbol{\Sigma}_w\phi(\boldsymbol{x}^*)$，$\phi(\boldsymbol{x}^*)^\top\boldsymbol{\Sigma}_w\boldsymbol{\Phi}$，$\boldsymbol{\Phi}^\top\boldsymbol{\Sigma}_w\phi(\boldsymbol{x}^*)$。将 $\boldsymbol{\Phi}$ 的表达式代入展开，那么它们是有着通式：$k(\boldsymbol{x},\boldsymbol{x'})=\phi(\boldsymbol{x})^\top\boldsymbol{\Sigma}_w\phi(\boldsymbol{x'})=\sqrt{\boldsymbol{\Sigma}_w}\phi(\boldsymbol{x})\cdot\sqrt{\boldsymbol{\Sigma}_w}\phi(\boldsymbol{x'})$。其中由于 $\boldsymbol{\Sigma}_w$ 是正定对称的方差矩阵，所以，这是一个核函数。

$$\begin{aligned}\phi(\boldsymbol{x}^*)^\top\boldsymbol{\mu}&=k(\boldsymbol{x}^*,\boldsymbol{X})[k(\boldsymbol{X},\boldsymbol{X})+\sigma^2\boldsymbol{I}]^{-1}\boldsymbol{y}\\\phi(\boldsymbol{x}^*)^\top\boldsymbol{\Sigma}\phi(\boldsymbol{x}^*)&=k(\boldsymbol{x}^*,\boldsymbol{x}^*)-k(\boldsymbol{x}^*,\boldsymbol{X})[k(\boldsymbol{X},\boldsymbol{X})+\sigma^2\boldsymbol{I}]^{-1}k(\boldsymbol{X},\boldsymbol{x}^*)+\sigma^2\boldsymbol{I}\end{aligned}\tag{9.26}$$

这样我们就可以和贝叶斯线性回归中进行一样的预测步骤。

函数空间的角度

现在，我们再换个角度看高斯过程回归。在刚刚的视角中，我们是基于概率分布的权重。那现在，我们上升一下，直接到函数级别。回顾一下最开始的函数定义 (式 9.10)，那么预测就可以写作：

$$p(y^*\mid\boldsymbol{X},\boldsymbol{y},\boldsymbol{x}^*)=\int_f p(y^*\mid f,\boldsymbol{X},\boldsymbol{y},\boldsymbol{x}^*)p(f\mid\boldsymbol{X},\boldsymbol{y},\boldsymbol{x}^*)df\tag{9.27}$$

这里可以对比一下 (式 9.18)。就数据集来说，取 $f(\boldsymbol{X})\sim N(\mu(\boldsymbol{X}),k(\boldsymbol{X},\boldsymbol{X}))$，$\boldsymbol{y}=f(\boldsymbol{X})+\epsilon\sim N(\mu(\boldsymbol{X}),k(\boldsymbol{X},\boldsymbol{X})+\sigma^2\boldsymbol{I})$。这里体现核函数另一个作用，平滑。以 RBF 核为例，当 $\boldsymbol{x}^*=\boldsymbol{x}$ 时，核函数值最大；当两个输入样本变得越来越远时，曲线逐渐呈平滑下降趋势。为了实现平滑度，我们会希望这两个样本的协方差就等于核函数。

现在预测任务的目的是给定一个新数据样本序列 $\boldsymbol{X}^*$，得到对应的预测 $\boldsymbol{y}^*=f(\boldsymbol{X}^*)+\epsilon$。然后，我们便可以写出：

$$\begin{pmatrix}\boldsymbol{y}\\f(\boldsymbol{X}^*)\end{pmatrix}\sim N\left(\begin{pmatrix}\mu(\boldsymbol{X})\\\mu(\boldsymbol{X}^*)\end{pmatrix},\begin{pmatrix}k(\boldsymbol{X},\boldsymbol{X})+\sigma^2\boldsymbol{I}&k(\boldsymbol{X},\boldsymbol{X}^*)\\k(\boldsymbol{X}^*,\boldsymbol{X})&k(\boldsymbol{X}^*,\boldsymbol{X}^*)\end{pmatrix}\right)\tag{9.28}$$

这里回顾一下高斯分布：

$$\begin{pmatrix} x_a \\ x_b \end{pmatrix} \sim N\left(\begin{pmatrix} \mu_a \\ \mu_b \end{pmatrix}, \begin{pmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{pmatrix}\right) \tag{9.29}$$

则会有：

$$\begin{aligned} & x_b \mid x_a \sim N(\mu_{b|a}, \Sigma_{b|a}) \\ & \mu_{b|a} = \Sigma_{ba}\Sigma_{aa}^{-1}(x_a - \mu_a) + \mu_b \\ & \Sigma_{b|a} = \Sigma_{bb} - \Sigma_{ba}\Sigma_{aa}^{-1}\Sigma_{ab} \end{aligned} \tag{9.30}$$

于是，我们可以直接写出：

$$\begin{aligned} p(f(\boldsymbol{X}^*) \mid \boldsymbol{X}, \boldsymbol{y}, \boldsymbol{X}^*) &= p(f(\boldsymbol{X}^*) \mid \boldsymbol{y}) \\ &= N\left(k(\boldsymbol{X}^*, \boldsymbol{X})[k(\boldsymbol{X}, \boldsymbol{X}) + \sigma^2\boldsymbol{I}]^{-1}(\boldsymbol{y} - \mu(\boldsymbol{X})) + \mu(\boldsymbol{X}^*), k(\boldsymbol{X}^*, \boldsymbol{X}^*) - k(\boldsymbol{X}^*, \boldsymbol{X})[k(\boldsymbol{X}, \boldsymbol{X}) + \sigma^2\boldsymbol{I}]^{-1}k(\boldsymbol{X}, \boldsymbol{X}^*)\right) \end{aligned} \tag{9.31}$$

我们通常会设均值为 $\mu(\boldsymbol{X}) = \mu(\boldsymbol{X}^*) = 0$：

$$p(f(\boldsymbol{X}^*) \mid \boldsymbol{y}) = N\left(k(\boldsymbol{X}^*, \boldsymbol{X})[k(\boldsymbol{X}, \boldsymbol{X}) + \sigma^2\boldsymbol{I}]^{-1}\boldsymbol{y}, k(\boldsymbol{X}^*, \boldsymbol{X}^*) - k(\boldsymbol{X}^*, \boldsymbol{X})[k(\boldsymbol{X}, \boldsymbol{X}) + \sigma^2\boldsymbol{I}]^{-1}k(\boldsymbol{X}, \boldsymbol{X}^*)\right) \tag{9.32}$$

如果再有 $\boldsymbol{y}^* = f(\boldsymbol{X}^*) + \epsilon$：

$$p(\boldsymbol{y}^* \mid \boldsymbol{y}) = N\left(k(\boldsymbol{X}^*, \boldsymbol{X})[k(\boldsymbol{X}, \boldsymbol{X}) + \sigma^2\boldsymbol{I}]^{-1}\boldsymbol{y}, k(\boldsymbol{X}^*, \boldsymbol{X}^*) - k(\boldsymbol{X}^*, \boldsymbol{X})[k(\boldsymbol{X}, \boldsymbol{X}) + \sigma^2\boldsymbol{I}]^{-1}k(\boldsymbol{X}, \boldsymbol{X}^*) + \sigma^2\boldsymbol{I}\right) \tag{9.33}$$

于是，我们同样可以做到预测。但可以看到，函数空间的角度更加简单易于求解。那么我们看一下高斯进程回归的基本步骤：

- 选择适当的均值函数 μ 和核函数 k，以及噪声 σ。其中核函数的选择尤其重要，根据不同的应用选择不同的核，一般选法是 RBF 高斯核。
- 对于已有的训练样本 D，计算核矩阵 $K = k(\boldsymbol{X}, \boldsymbol{X})$。再考虑待预测的样本 $\boldsymbol{x}^*$，计算 $K_* = k(\boldsymbol{x}^*, \boldsymbol{X})$ 和 $K_{**} = k(\boldsymbol{x}^*, \boldsymbol{x}^*)$。
- 计算出 $\boldsymbol{\mu}$ 和 $\boldsymbol{\Sigma}$，代入预测公式 (高斯函数) 中，得到预测值的均值、标准差、置信区间。通常，我们可以将均值就作为预测结果。

最后，我们再考虑一个问题，同样是已有一些样本点再对新样本做出预测，这和我们在第五章讨论的最小二乘回归有什么区别？最小二乘回归最后对新样本点 $\boldsymbol{x}^*$ 的预测 $f(\boldsymbol{x}^*)$ 是一个点估计，而在这里的 $f(\boldsymbol{x}^*)$ 其实是个后验概率分布。既然得到了分布，那么我们同样可以得到预测值的点估计 (均值)，但还可以知道这个估计有多少信心在里面。

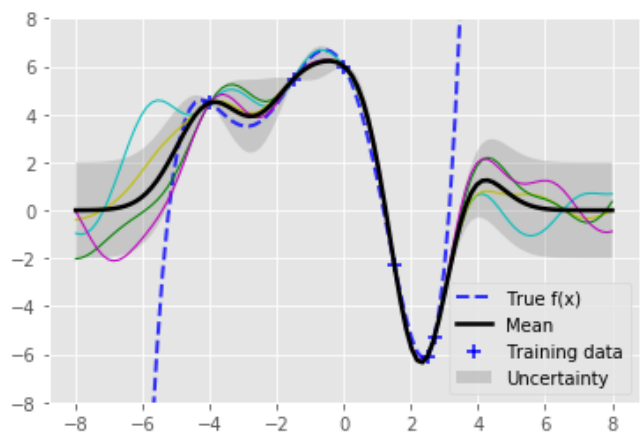
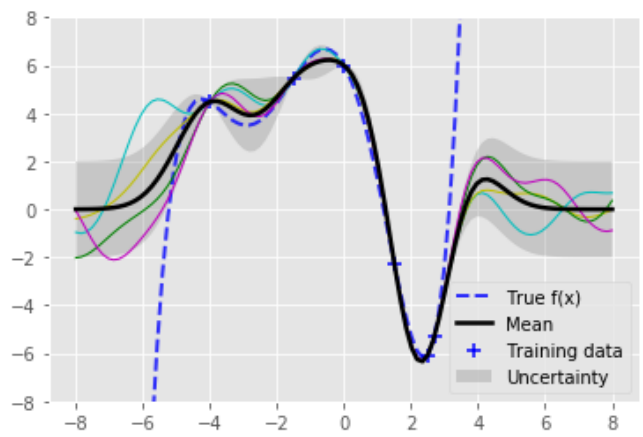


图 9.6. 高斯过程回归用于指导采样。已知点为图中蓝色点，真实曲线为图中蓝色虚线。对其采样 4 次，每次采样构造 100 个样本的测试集，基于“样本-预测” 可以绘制一条曲线。黑色曲线为进行多次采样平均后的曲线。

其另一个作用可以指导采样。如图 9.6 所示 (实现代码见下文)，训练集 D 中就 6 个点 (蓝色点)，真实曲线为图中蓝色虚线。我们可以构造一个 100 个样本的测试集。于是，我们便可以得到后验分布，我们从后验分布中随机采样得到每个 $\boldsymbol{x}^*$ 的预测结果 (视作标签)。再根据这些样本 (训练集 + 测试集) 以及它们的标签拟合一条曲线。在图中我们随机采样 4 次，得到 4 个拟合函数。如果是采样很多次，再取所有的拟合函数的平均值就得到均值曲线 (图中黑线)。含有已知数据 (训练集数据) 的地方，这些函数都很接近 (方差很低)；而没有数据的时候，变化范围就比较大 (灰色区域表示均值上下 2 个标准差)。这样，我们可以在预测值和方差之间折中，选择下一个采样点。

[18]:
```python
def k(xs, xt, sigma=1):
    """
    协方差函数。
    """
    dx = np.expand_dims(xs, 1) - np.expand_dims(xt, 0)
    return np.exp(-(dx**2) / (2*sigma**2))


def m(x):
    """
    均值函数。
    """
    return np.zeros_like(x)
```

```
def f(x):
    """
    目标函数。
    """
    coefs = [6, -2.5, -2.4, -0.1, 0.2, 0.03]
    y = 0
    for i, coef in enumerate(coefs):
            y += coef * (x ** i)
    return y

x = np.array([-4, -1.5, 0, 1.5, 2.5, 2.7])
y = f(x)
x_star = np.linspace(-8, 8, 100)
K = k(x, x)
K_star = k(x, x_star)
K_star_star = k(x_star, x_star)
mu = m(x_star) + K_star.T@np.linalg.pinv(K)@(y-m(y))
Sigma = K_star_star - K_star.T@np.linalg.pinv(K)@K_star
y_true = f(x_star)
plt.style.use('ggplot')
plt.plot(x_star, y_true, linewidth=2, color='b', alpha=0.8,
         linestyle='dashed', label='True f(x)')
plt.scatter(x, y, s=70, c='b', marker='+', label='Training data')
stds = np.sqrt(Sigma.diagonal())
up_lower = mu + 2*stds
plt.fill_between(x_star, mu+2*stds, mu-2*stds, facecolor='grey', alpha=0.3,
                 label='Uncertainty')
Colors = ['c', 'y', 'g', 'm']
for color in Colors:
    y_star = np.random.multivariate_normal(mu, Sigma)
    plt.plot(x_star, y_star, linewidth=1, color=color)
plt.ylim(-8, 8)
plt.plot(x_star, mu, linewidth=2.5, color='k', label='Mean')
plt.legend()
plt.show()
```



采集函数

前面我们通过高斯过程回归得到均值 $\boldsymbol{\mu}$ 和协方差 $\boldsymbol{\Sigma}$，这两个可以分别理解为用来做开发和探索的信息。然后基于这两个信息我们可以设计不同的采集函数。我们这里看一个简单的方法 Thompson Sampling。

Thompson Sampling (TS)

第一步先从当前的高斯过程后验里面采样得到一个函数 (例如在上图中从后验分布中采样了 4 个函数)，不妨记为 $\hat{f}_t$。

第二步我们要新生成的点就是：

$$\boldsymbol{x}^* = \arg\max \hat{f}_t(\boldsymbol{x}^*) \tag{9.34}$$

[19]:
```python
class KernelBase(ABC):

    def __init__(self):
        super().__init__()
        self.params = {}
        self.hyperparams = {}

    @abstractmethod
    def _kernel(self, X, Y):
        raise NotImplementedError

    def __call__(self, X, Y=None):
        return self._kernel(X, Y)

    def __str__(self):
        P, H = self.params, self.hyperparams
        p_str = ", ".join(["{}={}".format(k, v) for k, v in P.items()])
        return "{}({})".format(H["op"], p_str)

    def summary(self):
        return {
            "op": self.hyperparams["op"],
            "params": self.params,
            "hyperparams": self.hyperparams,
        }


class RBFKernel(KernelBase):

    def __init__(self, sigma=None):
        """
        RBF 核。
        """
        super().__init__()
        self.hyperparams = {"op": "RBFKernel"}
        self.params = {"sigma": sigma}  # 如果 sigma 未赋值则默认为 np.sqrt(n_features/2)，n_features 为特征数。

    def _kernel(self, X, Y=None):
        """
        对 X 和 Y 的行的每一对计算 RBF 核。如果 Y 为空，则 Y=X。

        参数说明：
        X：输入数组，为 (n_samples, n_features)
        Y：输入数组，为 (m_samples, n_features)
        """
        X = X.reshape(-1, 1) if X.ndim == 1 else X
        Y = X if Y is None else Y
        Y = Y.reshape(-1, 1) if Y.ndim == 1 else Y
        assert X.ndim == 2 and Y.ndim == 2, "X and Y must have 2 dimensions"
        sigma = np.sqrt(X.shape[1] / 2) if self.params["sigma"] is None else self.params["sigma"]
        X, Y = X / sigma, Y / sigma
        D = -2 * X @ Y.T + np.sum(Y**2, axis=1) + np.sum(X**2, axis=1)[:, np.newaxis]
        D[D < 0] = 0
        return np.exp(-0.5 * D)


class KernelInitializer(object):

    def __init__(self, param=None):
```

```
        self.param = param

    def __call__(self):
        r = r"([a-zA-Z0-9]*)=([^,)]*)"
        kr_str = self.param.lower()
        kwargs = dict([(i, eval(j)) for (i, j) in re.findall(r, self.param)])
        if "rbf" in kr_str:
            kernel = RBFKernel(**kwargs)
        else:
            raise NotImplementedError("{}".format(kr_str))
        return kernel
```

[20]:
```
class GPRegression:
    """
    高斯过程回归
    """
    def __init__(self, kernel="RBFKernel", sigma=1e-10):
        self.kernel = KernelInitializer(kernel)()
        self.params = {"GP_mean": None, "GP_cov": None, "X": None}
        self.hyperparams = {"kernel": str(self.kernel), "sigma": sigma}

    def fit(self, X, y):
        """
        用已有的样本集合得到 GP 先验。

        参数说明:
        X: 输入数组，为 (n_samples, n_features)
        y: 输入数组 X 的目标值，为 (n_samples)
        """
        mu = np.zeros(X.shape[0])
        Cov = self.kernel(X, X)
        self.params["X"] = X
        self.params["y"] = y
        self.params["GP_cov"] = Cov
        self.params["GP_mean"] = mu

    def predict(self, X_star, conf_interval=0.95):
        """
        对新的样本 X 进行预测。

        参数说明:
        X_star: 输入数组，为 (n_samples, n_features)
        conf_interval: 置信区间，浮点型 (0, 1)，default=0.95
        """
        X = self.params["X"]
        y = self.params["y"]
        K = self.params["GP_cov"]
        sigma = self.hyperparams["sigma"]
        K_star = self.kernel(X_star, X)
        K_star_star = self.kernel(X_star, X_star)
        sig = np.eye(K.shape[0]) * sigma
        K_y_inv = np.linalg.pinv(K + sig)
        mean = K_star @ K_y_inv @ y
        cov = K_star_star - K_star @ K_y_inv @ K_star.T
        percentile = norm.ppf(conf_interval)
        conf = percentile * np.sqrt(np.diag(cov))
        return mean, conf, cov
```

[21]:
```
class BayesianOptimization:

    def __init__(self):
        self.model = GPRegression()

    def acquisition_function(self, Xsamples):
        mu, _, cov = self.model.predict(Xsamples)
        mu = mu if mu.ndim==1 else (mu.T)[0]
        ysample = np.random.multivariate_normal(mu, cov)
        return ysample

    def opt_acquisition(self, X, n_samples=20):
        # 样本搜索策略，一般方法有随机搜索、基于网格的搜索，或局部搜索
        # 我们这里就用简单的随机搜索，这里也可以定义样本的范围
        Xsamples = np.random.randint(low=1,high=50,size=n_samples*X.shape[1])
        Xsamples = Xsamples.reshape(n_samples, X.shape[1])
        # 计算采集函数的值并取最大的值
        scores = self.acquisition_function(Xsamples)
        ix = np.argmax(scores)
        return Xsamples[ix, 0]

    def fit(self, f, X, y):
        # 拟合 GPR 模型
        self.model.fit(X, y)
        # 优化过程
        for i in range(15):
            x_star = self.opt_acquisition(X)  # 下一个采样点
            y_star = f(x_star)
            mean, conf, cov = self.model.predict(np.array([[x_star]]))
            # 添加当前数据到数据集合
            X = np.vstack((X, [[x_star]]))
            y = np.vstack((y, [[y_star]]))
            # 更新 GPR 模型
            self.model.fit(X, y)
        ix = np.argmax(y)
        print('Best Result: x=%.3f, y=%.3f' % (X[ix], y[ix]))
        return X[ix], y[ix]
```

用自定义的度量指标，乳腺癌数据集测试，第七章描述的随机森林用于模型训练

[22]:
```
from chapter7 import RandomForest
```

[23]:
```
column_names = ['Sample code number','Clump Thickness',
                'Uniformity of Cell Size','Uniformity of Cell Shape',
                'Marginal Adhesion','Single Epithelial Cell Size',
                'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
data = pd.read_csv('../data/cancer/breast-cancer-wisconsin.data',names=column_names)
data = data.replace(to_replace='?',value=np.nan)  # 非法字符替代
data = data.dropna(how='any')  # 去掉空值，any；出现空值行则删除
print(data.shape)
# 随机采样 25% 的数据用于测试，剩下 75% 用于构建训练集
X_train,X_test,y_train,y_test = train_test_split(data[column_names[1:10]], data[column_names[10]],
                                                 test_size=0.25, random_state=1111)
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 修改标签为 0 和 1
print(y_train.shape)
y_train[y_train==2] = 0
y_train[y_train==4] = 1
y_test[y_test==2] = 0
```

```
y_test[y_test==4] = 1
# 再查看训练样本的数量和类别分布
print(y_train.value_counts())
# 数据标准化预处理，保证每个维度特征均值为 0，方差为 1
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
y_train = y_train.as_matrix()
y_test = y_test.as_matrix()
```

```
(683, 11)
2    328
4    184
Name: Class, dtype: int64
(512,)
0    328
1    184
Name: Class, dtype: int64
```

[24]:
```
def func_black_box(k):
    model = RandomForest(n_estimators=k)
    model.fit(X_train, y_train)
    return model.score(X_test, y_test)

X0 = np.array([[4]])
y0 = np.array([func_black_box(4)])
print("initial n_estimators={}, score={}".format(4, y0[0]))
BO = BayesianOptimization()
Xl, yl = BO.fit(func_black_box, X0, y0)
print("Best n_estimators={}, score={}".format(Xl, yl))
```

```
initial n_estimators=4, score=0.9122807017543859
Best Result: x=19.000, y=0.965
Best n_estimators=[19], score=[0.96491228]
```

[25]:
```
import numpy
import matplotlib
import re
import pandas
import sklearn
import itertools
import scipy

print("numpy:", numpy.__version__)
print("matplotlib:", matplotlib.__version__)
print("pandas:", pandas.__version__)
print("re:", re.__version__)
print("sklearn:", sklearn.__version__)
print("scipy:", scipy.__version__)
```

```
numpy: 1.14.5
matplotlib: 3.1.1
pandas: 0.25.1
re: 2.2.1
sklearn: 0.21.3
scipy: 1.3.1
```