第2章 感知器——神经元

授课人： 李仪
liyi1002@csu.edu.cn
http://faculty.csu.edu.cn/liyi
中南大学 自动化学院

主要内容

Contents:

01 感知器的定义

02 激活函数

03 感知器的训练

04 使用PyTorch框架

01

感知器的定义

感知器的概念：由若干输入的线性组合及其变换构成的计算单元：

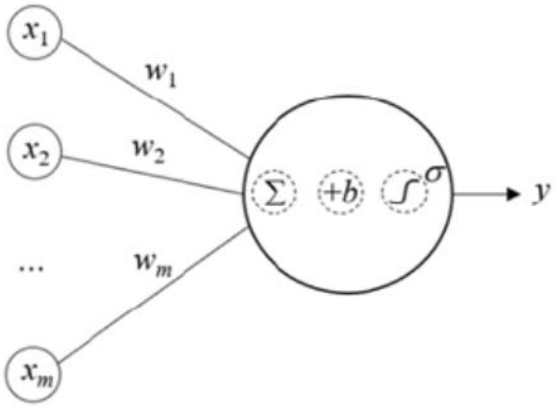


图 2-1 感知器（神经元）的基本结构

其中，$x_1, x_2, \ldots, x_m$ 为m 个输入，表示一个样本的m 个特征，Σ表示加权求和符号，b 为偏置项，σ表示一种函数变换，通常称为激活函数。每个输入用一个小圆圈节点表示，它们与大圆圈节点之间都有一条边连接，边上标注的$w1, w2, \ldots, wm$ 分别是对应输入的权重参数。

感知器的**数学模型**:

$$y = \sigma(w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_m \cdot x_m + b)$$
$$= \sigma(\sum_{j=1}^{m} w_j \cdot x_j + b)$$

说明: 一个感知器在工作时先对输入进行加权求和，然后再加上偏置项，最后用激活函数对其进行变换。

$x_1, x_2, \ldots, x_m$表示一个样本的 m 个特征值，该样本可记为 $x = (x_1, x_2, \ldots, x_m)$，表示输入的特征向量。令 $w = (w_1, w_2, \ldots, w_m)$，表示由权重参数 $w_1, w_2, \ldots, w_m$ 构成的权重向量。

这样，上述表达式可表示为向量相乘的形式:

$$y = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$$

02

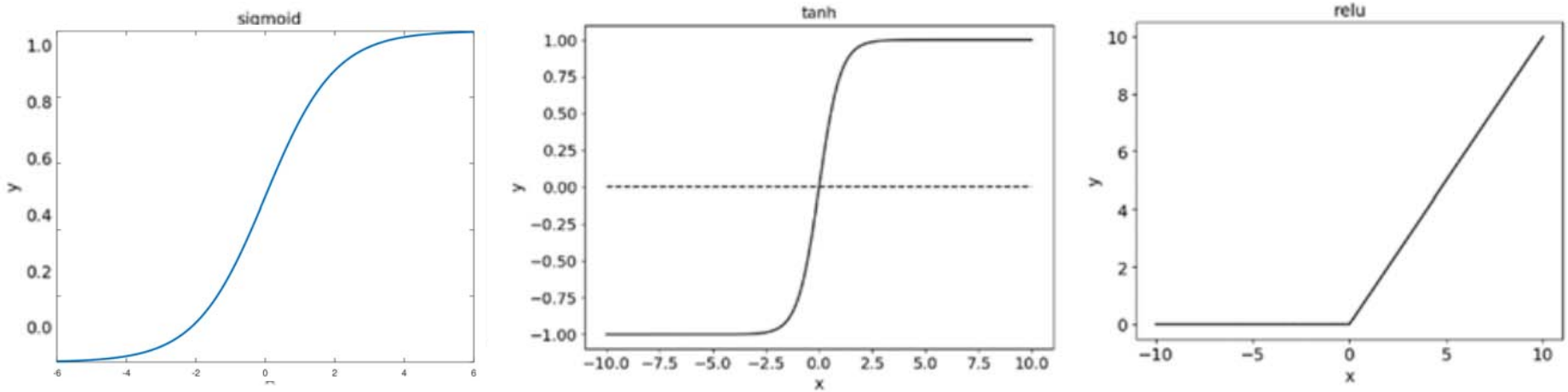
激活函数

常用的激活函数： sigmoid、tanh 和 relu（pronounce：瑞奥油） 函数，它们的数学公式如下：

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

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

$$\boldsymbol{ReLU(x) = \begin{cases} x, \text{当} x \geq 0 \\ 0, \text{其他} \end{cases}}$$



(a) sigmoid (b) tanh (c) relu

函数 sigmoid(x)是将实数区间(-∞, +∞)内的数映射到区间(0, 1)内，tanh(x)则将之映射到(-1, 1)内，而 relu(x)将小于 0的实数映射为 0，非负实数保持不变。sigmoid(x)和 tanh(x)是对实数 x 进行非线性变换，多用于全连接网络，为实现网络的非线性拟合功能奠定基础。relu(x)则多用于卷积神经网络，其取值范围为[0, 1]。

激活函数sigmoid(x)的导数：

令 y = sigmoid(x)，则不难推导出 sigmoid(x)关于 x 的导数函数可以表示为：

$$y(1-y)$$

特点：sigmoid(x)的导数可以其自身来表示。

例子：假设欲求函数 sigmoid(x)在 x = 0 上的导数，我们可以先求 y = sigmoid(x) = sigmoid(0) = 0.5，进而得知该导数值为 y(1-y) = 0.5。

激活函数tanh(x)的导数:

令 y = tanh(x)，则 tanh(x)的导数函数可以表示为:

$$(1+y)(1-y)$$

特点：sigmoid函数和tanh函数的特点之一就是它们的导数可用它们自身来表示。这使得它们在函数求导、运算速度等方面表现出优越的性能，是它们在全连接网络中受到广泛应用的一个重要原因。

其他激活函数：阶跃函数、恒等映射等函数在感知器、神经网络中也经常得到应用。

它们的数学公式如下：

$$f(x)=\begin{cases}0, & if\ x>\alpha\\ 1, & else\end{cases}$$ （阶跃函数，其中α为一实数值）

$$f(x)=x$$ （恒等映射）

03

感知器的训练

2.3.1 监督学习和无监督学习

机器学习方法大致可以分为三大类：监督学习、无监督学习和半监督学习。

在监督学习中，我们需要提供多个带标记（label）的训练样本，其中每个样本除了提供若干个特征值以外，还应提供样本所属的类别信息。
令：

$$D = \{\boldsymbol{s} = (\boldsymbol{x}, y) \mid \boldsymbol{x} = (x_1, x_2, \ldots, x_m) \in \Re, y \in \psi\}$$

表示一个样本数据集，其中 $x = (x_1, x_2, \ldots, x_m)$为特征向量，表示样本 s 的 m 个特征值；y 表示该样本的标记，一般为数值标量或离散符号；$\Re$和ψ分别表示 m 维特征空间和标记集。

2.3.1 监督学习和无监督学习

- **监督学习:** 利用样本的特征向量及其标记来学习数据分布规律的一种机器学习方法，即它既要求提供 特征向量x，也要提供标记 y。数据分类就是一种典型的监督学习方法，以神经网络为核心内容的深度学习也属于监督学习的范畴。

- **无监督学习:** 仅仅基于样本特征向量而无需样本标记的一种机器学习方法，即它仅要求提供x，而不需要提供 y。聚类方法就是一种典型的无监督学习方法。

- **半监督学习 :** 监督学习和无监督学习结合起来就产生了**半监督学习**。例如，利用少量的带标记的样本，通过聚类算法，为没有标记的样本构造标记，从而产生更多带标记的样本，进而满足监督学习方法对海量标记样本的需要。

2.3.2 面向回归问题的训练方法

1. 分类、线性回归与逻辑回归

回归（Regression）问题：模型的输出是连续值，是定量输出，即回归分析主要用于预测。

逻辑回归（Logistic Regression）：是在回归的基础上要求输出范围在(0, 1)之间，以用于解决分类问题，即逻辑回归主要用于分类，尤其是二分类。

分类（Classification）问题：模型的输出是有限个数的离散值，是定性输出。

2.3.2 面向回归问题的训练方法

2. 目标函数与随机梯度下降算法

目标函数：一种非负的函数，用来表示感知器的实际输出$\hat{y}$与期望输出y的接近程度。二者越接近，函数值越小；当二者完全吻合时，函数值达到最小值。记为

$$\mathcal{L}(\hat{y}, y)$$

其中$\hat{y}$表示感知器的实际输出，即$\hat{y}=\sigma(w\cdot x+b)$，y 为给定的样本标记，也是模型的期望输出。确定 w 和 b 的方法正是通过最小化目标函数来实现的。

2.3.2 面向回归问题的训练方法

2. 目标函数与随机梯度下降算法

损失函数（Loss Function）：也称为代价函数（Cost Function），通常是表示预测值与真实值之间的差异的函数。针对回归问题，每输入一个特征向量 x，模型的输出$\hat{y}$是一个连续型的数值。回归问题常用的目标函数设计如下：

$$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

对感知器而言，$\hat{y} = \sigma(\mathrm{w}\cdot\mathrm{x} + \mathrm{b})$，式中的"$\frac{1}{2}$"是为了导数计算方便。针对回归问题，在感知器中激活函数　通常设置为恒等映射，即相当于不需要激活函数；如果设置为其他激活函数，在梯度计算过程中需要对其进行求导。

2.3.2 面向回归问题的训练方法

2. 目标函数与随机梯度下降算法

目标函数$\mathcal{L}(\hat{y},y)$：关于$\hat{y}$的函数，而$\hat{y}$是关于 w 和 b 的函数，x 和 y 均为事先给定的样本数据，视为常数。因此，优化的目标是：找到适当的 w 和 b，使得$\mathcal{L}(\hat{y},y)$达到最小值。在数学上，该优化问题可表示为

$$\underset{w,b}{\operatorname{argmin}}\ \mathcal{L}(\hat{y},y)$$

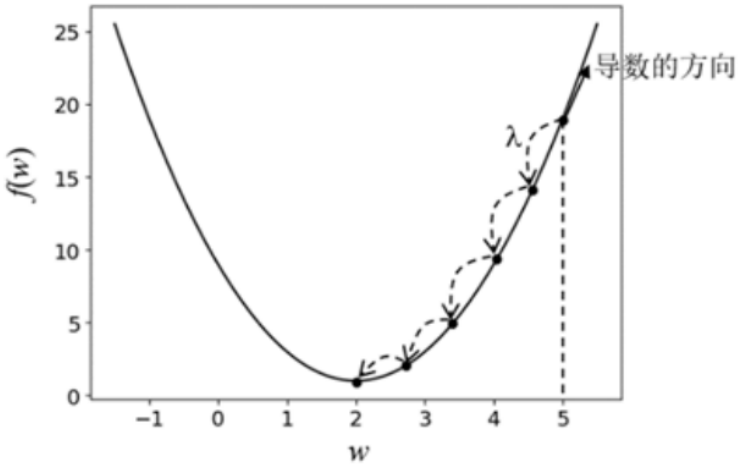
由于 $w = (w_1, w_2, \ldots, w_m)$，故$\mathcal{L}(\hat{y},y)$ 是 m+1 元函数（含一个偏置项 b），这说明一共有 m+1 个参数需要优化。

2.3.2 面向回归问题的训练方法

2. 目标函数与随机梯度下降算法

一个例子：如何优化m+1个参数？考虑如下的一元函数：

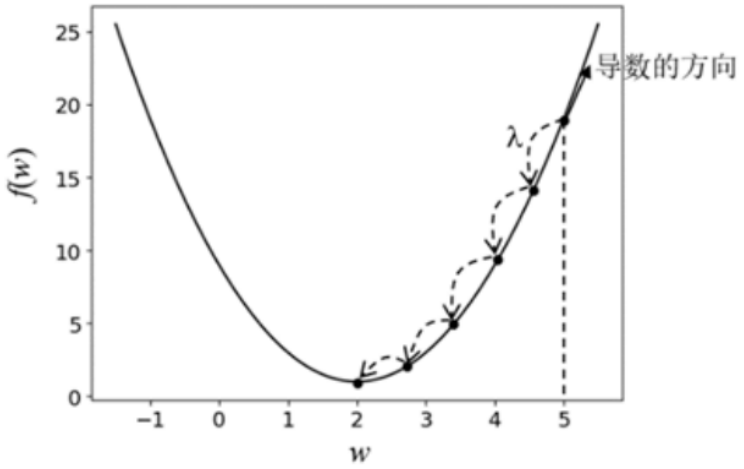
$$f(w) = 2(w\text{-}2)^2+1$$



函数 f(w)的最小值点位于 w = 2 处，最小值为 f(w) = 1。现任意选择一点，假设为点(5, 19)。易知，f(w)的导数函数为 f′(w) = 4(w-2)，其在该点上的导数为 f′(5) = 4(5-2) = 12，该导数的方向如图 2-3 中的箭头所示。该方向是函数 f(w)随 w 上升最快的方向。但我们要找的是最小值，而不是最大值，因此要沿着与此相反的方向才能以最快速度找到最小值。为此，我们需沿着与该导数方向相反的方向“走”，也就是沿着梯度下降的方向“走”（图中的虚线箭头）。

2.3.2 面向回归问题的训练方法

将 f(w)在 w = 5 上的导数值 12 乘以-1 以后得到-12（导数值的相反数），再用于修正 w，结果得到 w = 5+(-12) = -7。显然，这已经向左严重“跨过”了 w = 2 处，这说明“跨步”太大了。于是，学者们用一个小系数乘以导数值的相反数，用于控制“迈出”的步长。假如该小系数为 0.1，则“迈出”的步长为-1.2，以之修正 w 后得到 w = 5+(-1.2) = 3.8。然后，从 w = 3.8 开始，用同样方法就可以不断逼近 w = 2 的位置，总体上大致沿着图 2-3 所示的虚线箭头方向逼近。而这个小系数就是所谓的学习率（Learning Rate）它是一个超参数，需要手工设置，其取值通常为 0.1, 0.01等之类的非负小实数。



2.3.2 面向回归问题的训练方法

总结：可以得到参数 w 的如下更新方式：

$$w \leftarrow w - \lambda \frac{\partial f}{\partial w}$$

这种方法就是所谓的**梯度下降算法**。

2.3.2 面向回归问题的训练方法

【例2.1】给定函数 $f(w) = 2(w-2)^2+1$，请用梯度下降算法求该函数的最小值点。

用 PyTorch 来求出该函数的最小值点。为此，先定义实现 f(w)的 PyTorch函数，代码如下：

```
def f(w):
    t = 2 * (w - 2) ** 2 + 1
    return t
```

然后定义函数 f(w)的导数函数：

```
def df(w):
    t = 4 * (w - 2)
    return t
```

2.3.2 面向回归问题的训练方法

接着设置学习率和寻找的起点：

```
lr = 0.1 #学习率
w = torch.Tensor([5.0])       #设置寻找的起点
```

最后编写迭代循环代码：

```
for epoch in range(20):       #迭代循环
      w = w - lr*df(w)        #更新 w
y = f(w)
w, y = round(w.item(), 2), round(y.item(),2)
print("该函数的最小值点是：（%0.2f,%0.2f)"%(w, y))        #输出最小值点
```

2.3.2 面向回归问题的训练方法

运行该程序，输出结果如下：

```
该函数的最小值点是：(2.00,1.00)
```

在这个例子中，我们掌握了面向一元函数的梯度下降迭代算法，即每次都是沿着梯度下降最快的方向（与梯度方向相反）去更新参数，从而快速找到所需要的参数。

2.3.2 面向回归问题的训练方法

2. 目标函数与随机梯度下降算法

多元函数的梯度下降算法：要如何对众多的参数进行优化呢?

原理：当对某一个参数进行优化的时候，把其他参数看成是常数，这样就可以用上面介绍的方法对当前的参数进行优化，进而优化所有的参数。据此，针对 m+1 元的目标函数 $\mathcal{L}(\hat{y}, y)$，参数优化的核心操作可表示如下：

$$w_j \leftarrow w_j - \lambda \frac{\partial \mathcal{L}}{\partial w_j}, j = 1, 2, \ldots, m$$

$$b \leftarrow b - \lambda \frac{\partial \mathcal{L}}{\partial b}$$

2.3.2 面向回归问题的训练方法

2. 目标函数与随机梯度下降算法

如果每次更新时，只利用一个样本进行梯度计算，并以此更新参数，那么这种梯度下降算法通常称为**随机梯度下降算法**。

输入：待学习的感知器 $\boldsymbol{w}\cdot\boldsymbol{x}+b$，以及数据集 $D=\{(\boldsymbol{x}^{(i)},y^{(i)})\mid i=1,2,\ldots,n\}$

输出：训练好的感知器 $\boldsymbol{w}\cdot\boldsymbol{x}+b$

Begin

(1) 读入数据集 D，并设置学习率λ;

(2) 随机初始化 $\boldsymbol{w}$ 和 b，其中 $\boldsymbol{w}=(w_1,w_2,\ldots,w_m)$;

(3) 设计目标函数$\mathcal{L}(\hat{y},y)$，并导出其导数函数;

(4) While(未达到停止条件):

(5) For $i=1$ to n do: #通常先随机打乱 D 中样本的顺序

(6) 利用$\mathcal{L}(\hat{y}^{(i)},y^{(i)})$计算$\frac{\partial\mathcal{L}}{\partial w_j}$和$\frac{\partial\mathcal{L}}{\partial b}$;

(7) 令 $w_j\leftarrow w_j-\lambda\frac{\partial\mathcal{L}}{\partial w_j},j=1,2,\ldots,m$;

(8) 令 $b\leftarrow b-\lambda\frac{\partial\mathcal{L}}{\partial b}$;

(9) 输出 $\boldsymbol{w}\cdot\boldsymbol{x}+b$. #这时 $\boldsymbol{w}$ 和 b 都已经被优化

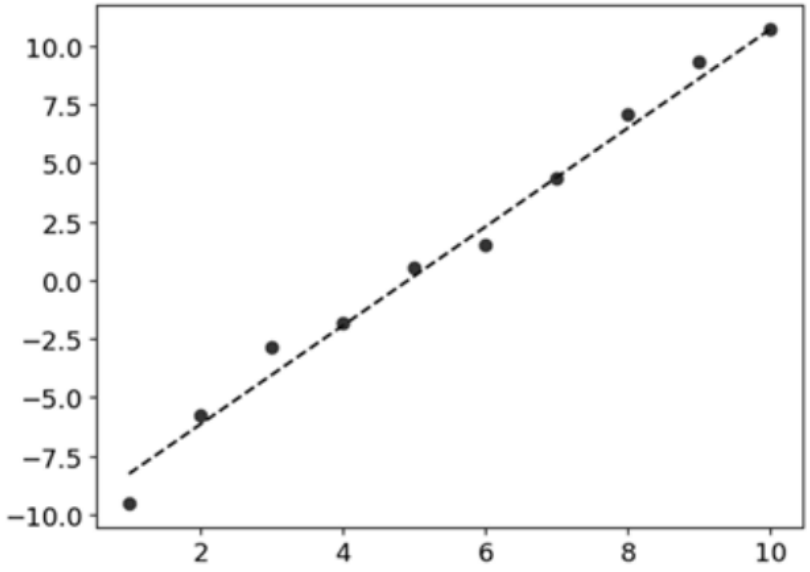
End

2.3.2 面向回归问题的训练方法

3. 随机梯度下降算法举例

【例2.2】构建一个感知器，使之可以对给定的若干离散点进行线性拟合（二维平面中的数据点）。

假设在一个二维平面中给定 10 个点，其坐标分别是：(1, -9.51), (2, -5.74), (3, -2.84), (4, -1.80), (5, 0.54), (6, 1.51), (7, 4.33), (8, 7.06), (9, 9.34), (10, 10.72)。如果将其标在一个坐标平面中，则结果如图 2-4 所示。下面构造一个线性感知器，以实现对这些离散点的拟合，即该感知器可以“绘出”图中的虚线。



2.3.2 面向回归问题的训练方法

根据输入数据的特点，每个样本数据只有一个特征，因此设置 m = 1，于是设计如下的感知器：

$$\hat{y} = w \cdot x + b$$

这是最简单的感知器，只需确定两个参数：w 和 b。主要步骤和代码如下：

首先，读取数据和定义感知器函数：

```
import torch
X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]     #读取数据
Y = [-9.51, -5.74, -2.84, -1.8, 0.54, 1.51, 4.33, 7.06, 9.34, 10.72]
X = torch.Tensor(X) #转换为张量
Y = torch.Tensor(Y)
def f(x):      #定义感知器函数
      t = w*x + b
return t
```

2.3.2 面向回归问题的训练方法

其中，“f(x)”相当于根据当前 w 和 b 的值来计算感知器的输出$\hat{y}$

然后，按照随机梯度下降算法的基本步骤编写 PyTorch 代码：

（1）随机初始化参数：

```
w, b = torch.rand(1), torch.rand(1) #随机初始化 w 和 b
```

（2）确定目标函数$\mathcal{L}(\hat{y},y)=\frac{1}{2}(\hat{y}-y)^2$，其关于 w 和 b 的导数函数分别是：

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial}{\partial w}\left(\frac{1}{2}(\hat{y}-y)^2\right)\\ &= (\hat{y}-y)\frac{\partial \hat{y}}{\partial w}\\ &= (\hat{y}-y)\,x\\ \frac{\partial \mathcal{L}}{\partial b} &= (\hat{y}-y)\end{aligned}$$

2.3.2 面向回归问题的训练方法

据此，编写关于 w 和 b 的导数函数的实现代码：

```
def dw(x,y):      #目标函数关于 w 的导数函数：
    t = (f(x) -y) * x
    return t
def db(x,y): #目标函数关于 b 的导数函数：
    t = (f(x) -y)
    return t
```

（3）设置学习率：

```
lr = torch.Tensor([0.01])      #设置学习率
```

2.3.2 面向回归问题的训练方法

（4）编写循环体代码，对一个样本数据，做一次梯度计算和参数更新，同时以循环代数作为算法停止条件：

```
for epoch in range(1000): #设置循环的代数
    for x,y in zip(X, Y): #注意，X 和 Y 中的元素一一对应
        dw_v, db_v = dw(x,y), db(x,y)
        w = w - lr * dw_v
        b = b - lr * db_v
```

经过上述迭代循环以后，得到的 w 和 b 的值即为符合我们的需要。

2.3.2 面向回归问题的训练方法

以下是绘制离散点图的代码:

```
import matplotlib.pyplot as plt
plt.scatter(X,Y,c='r')
X2 = [X[0],X[len(X)-1]] #过两点绘制感知器函数直线图
Y2 = [f(X[0]),f(X[len(X)-1])]
plt.plot(X2,Y2,'--',c='b')
plt.tick_params(labelsize=13)
plt.show()
```

2.3.2 面向回归问题的训练方法

4. 批量梯度下降算法及举例

目标函数$\mathcal{L}(\hat{y},y)$实际上仅仅刻画单个样本在模型上造成的误差，故也称为单个样本的误差。然而，如果按照上面的思路，每处理一个样本，就按照上式做一次所有参数的更新，那么对大数据集而言，这种做法是低效的。

批量梯度下降算法（Batch Gradient Descent）：当输入多个样本后，利用多个样本的平均误差来做一次梯度计算和参数更新。

2.3.2 面向回归问题的训练方法

4. 批量梯度下降算法及举例

【例2.3】对例 2.2 所述的离散点线性拟合的问题，请改用批量梯度下降算法来解决。

与例 2.2 相比，这里需要做两点修改：（1）先对数据进行打包，即将数据集划分若干个包，形成多个批量；（2）以批量为单位，计算它们的平均误差，然后据此计算关于各个参数的梯度，进而用于更新相应的参数。由于例子比较简单，下面直接给出核心代码，其中批的大小设置为 4：

```
torch.manual_seed(123)
X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Y = [-9.51, -5.74, -2.84, -1.8, 0.54, 1.51, 4.33, 7.06, 9.34, 10.72]
X = torch.Tensor(X) #转换为张量
Y = torch.Tensor(Y)
```

2.3.2 面向回归问题的训练方法

4. 批量梯度下降算法及举例

```
#数据打包：
n = 4 #包的大小设置为 4
X1,Y1 = X[0:n],Y[0:n] #该包的大小为 4
X2,Y2 = X[n:2*n],Y[n:2*n] #该包的大小为 4
X3,Y3 = X[2*n:3*n],Y[2*n:3*n] #该包的大小为 2
X,Y = [X1,X2,X3],[Y1,Y2,Y3] #重新定义 X 和 Y
```

```
#定义感知器的函数
def f(x):
 t = w*x + b
      return t
def dw(x,y): #目标函数关于 w 的导数函数：
 t = (f(x) -y) * x
      return t
def db(x,y): #目标函数关于 b 的导数函数：
 t = (f(x) -y)
      return t
```

2.3.2 面向回归问题的训练方法

4. 批量梯度下降算法及举例

```
w, b = torch.rand(1), torch.rand(1)        #随机初始化 w 和 b
lr = torch.Tensor([0.01])                  #设置学习率
for epoch in range(1000):                  #设置循环的代数
    for bX, bY in zip(X, Y):               #此处与例 2.2 不同
        dw_v, db_v = dw(bX, bY), db(bX, bY)        #由于 bX 和 bY 均为张量,
        #所以函数 dw 和 db 不需要修改
        dw_v = dw_v.mean() #求平均值
        db_v = db_v.mean()
        w = w - lr * dw_v
        b = b - lr * db_v
    print("优化后，参数 w 和 b 的值分别为：%0.4f 和%0.4f"%(w,b))
```

2.3.3 面向分类问题的训练方法

分类问题可以分为二分类问题或多分类问题。一个感知器一般只能解决二分类问题。在二分类问题中，一般一个类别用 0 表示，另一个类别用 1 表示。考虑如下的感知器:

$$\hat{y} = \sigma(w \cdot x + b)$$

我们期望该感知器能够将属于 0 类的特征向量映射为接近 0 的数值，而将属于 1类的特征向量映射为接近 1 的数值，这些映射值都在(0, 1)范围内。为实现这种映射功能，感知器的激活函数　一般设置为 sigmoid 函数，其目标函数常定义为如下的交叉熵损失函数:

$$\mathcal{L}(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

其中，log 为自然对数，$\mathcal{L}(\hat{y}, y)$ 是关于 w 和 b 的函数，y 是标记，视为常数。

2.3.3 面向分类问题的训练方法

从该目标函数可以看出：如果 y = 0，则当模型输出$\hat{y}$越接近于 0 时，$\mathcal{L}(\hat{y}, y)$的值越小，而当$\hat{y}$越接近于 1 时，$\mathcal{L}(\hat{y}, y)$ 越大；如果 y = 1，则$\hat{y}$越接近于 1 时，$\mathcal{L}(\hat{y}, y)$ 越小，而当$\hat{y}$越接近于 0 时，$\mathcal{L}(\hat{y}, y)$ 越大。这说明，当通过不断修正 w 和 b 而使得$\mathcal{L}(\hat{y}, y)$ 被最小化时，该感知器能够将属于 0 类和 1 类的特征向量分别映射为接近 0 和 1 的数值。

二分类感知器的训练方法也是采用批量梯度下降算法或随机梯度下降算法，相应算法的描述跟针对回归问题的算法描述完全一样，不同的是，目标函数 $\mathcal{L}(\hat{y}, y)$ 的梯度函数发生了变化。

2.3.3 面向分类问题的训练方法

下面给出此处 $\mathcal{L}(\hat{y}, y)$ 的梯度函数的推导过程：

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{\partial}{\partial w_j}(-y\log(\hat{y})-(1-y)\log(1-\hat{y}))$$

$$= \frac{-y}{\hat{y}}\frac{\partial \hat{y}}{\partial w_j} - \frac{1-y}{1-\hat{y}}\frac{\partial(-\hat{y})}{\partial w_j}$$

$$= \left(\frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}\right)\frac{\partial \hat{y}}{\partial w_j}$$

$$= \left(\frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}\right)\frac{\partial \sigma(\boldsymbol{w}\cdot\boldsymbol{x}+b)}{\partial w_j}$$

$$= \left(\frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}\right)\hat{y}(1-\hat{y})\frac{\partial(\boldsymbol{w}\cdot\boldsymbol{x}+b)}{\partial w_j}$$

$$= \left(\frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}\right)\hat{y}(1-\hat{y})x_j$$ （注：利用了函数 sigmoid 的导数性质）

$$= [(1-y)\hat{y} - (1-\hat{y})y]x_j$$

2.3.3 面向分类问题的训练方法

类似地，可以推导出：

$$\frac{\partial \mathcal{L}}{\partial b} = (1-y)\hat{y} - (1-\hat{y})y$$

针对二分类问题的感知器，其参数更新的核心操作是：

$$w_j \leftarrow w_j - \lambda[(1-y)\hat{y} - (1-\hat{y})y]x_j, j = 1, 2, \ldots, m$$

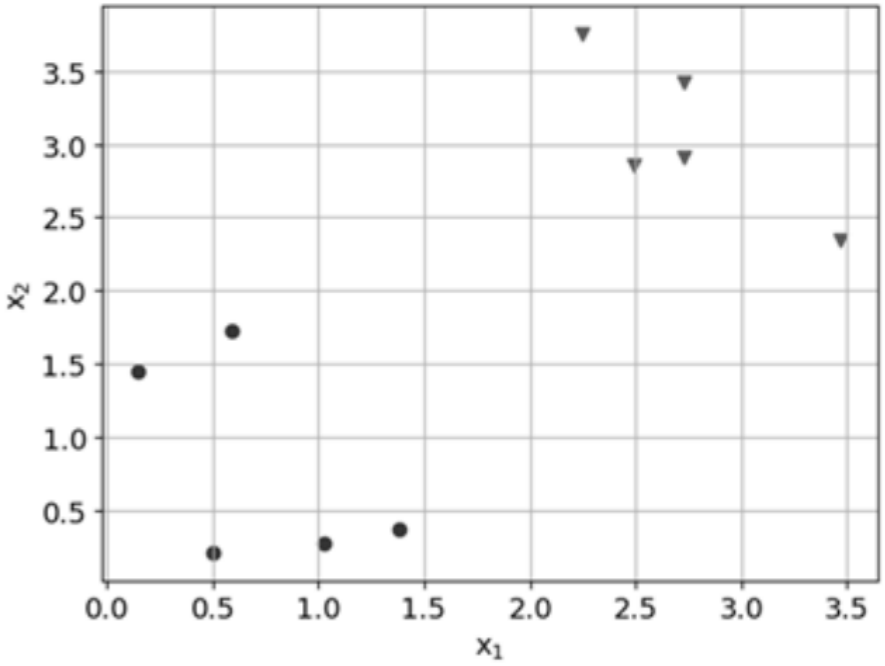
$$b \leftarrow b - \lambda[(1-y)\hat{y} - (1-\hat{y})y]$$

上述设计的感知器是在线性回归（只有一个实数值输出）的基础上，运用了激活函数sigmoid，使得感知器的输出范围在(0, 1)之间，因而这种感知器实际上是利用逻辑回归方法来解决二分类问题。

2.3.2 面向回归问题的训练方法

【例2.4】在一个二维平面中，对给定的若干个线性可分的离散点，其中部分属于 0 类，另一部分属于 1 类。请构造一个感知器，使之能够对这些点进行分类。

假设给定 10 个线性可分的离散点，其中 5 个点属于 0 类，另外 5 个点属于 1 类。这些点的坐标分别是(2.49, 2.86), (0.50, 0.21), (2.73, 2.91), (3.47, 2.34), (1.38, 0.37), (1.03, 0.27),(0.59, 1.73), (2.25, 3.75), (0.15, 1.45)和(2.73, 3.42)，它们的类别标记分别是 1, 0, 1, 1, 0, 0, 0, 1, 0 和 1。将这些点标记在一个坐标系中，结果如图 2-5 所示，其中圆黑点和下三角黑点分别表示 0 类和 1 类。



2.3.2 面向回归问题的训练方法

这属于典型的二分类问题，样本的特征值个数为 2，为此设计如下的感知器：

$$\hat{y} = \text{sigmoid}(\boldsymbol{w} \cdot \boldsymbol{x} + b)$$

其中，w = (w1, w2), x = (x1, x2), b 为偏置项，一共有 3 个参数待优化：w1, w2和 b。该感知器的目标函数构造如下：

$$\mathcal{L}(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

其中，y为样本的类标记，取值为 0 或 1。

2.3.2 面向回归问题的训练方法

按照随机梯度下降算法编写该感知器的训练代码，用面向对象编程方法来写这些代码：

```
import torch
import matplotlib.pyplot as plt
#读入数据
X1=[2.49, 0.50, 2.73, 3.47, 1.38, 1.03, 0.59, 2.25, 0.15, 2.73]
X2=[2.86, 0.21, 2.91, 2.34, 0.37, 0.27, 1.73, 3.75, 1.45, 3.42]
Y = [1, 0, 1, 1, 0, 0, 0, 1, 0, 1] #类标记
X1 = torch.Tensor(X1)
X2 = torch.Tensor(X2)
X = torch.stack((X1,X2),dim=1) #将所有特征数据“组装”为一个张量
#形状为 torch.Size([10, 2])
Y = torch.Tensor(Y) #形状为 torch.Size([10])
lr = torch.Tensor([0.1]) #设置学习率
```

2.3.2 面向回归问题的训练方法

```
class Perceptron2():
    def __init__(self):
        self.w1 = torch.Tensor([0.0])          #定义三个待优化参数（属性）
        self.w2 = torch.Tensor([0.0])
        self.b = torch.Tensor([0.0])
    def f(self, x):            #感知器函数的实现代码
        x1, x2 = x[0], x[1]
        t = self.w1 * x1 + self.w2 * x2 + self.b
        z = 1.0 / (1 + torch.exp(t)) #运用 sigmoid 函数
        return z
    def forward_compute(self, x): #前向计算
        pre_y = self.f(x)
        return pre_y #只有一个实数值输
```

2.3.2 面向回归问题的训练方法

```
perceptron2 = Perceptron2()          #创建实例 perceptron2
for ep in range(100): #迭代代数为 100
    for (x, y) in zip(X, Y):
        pre_y = perceptron2.forward_compute(x)          #执行前向计算
        x1, x2 = x[0], x[1]
        dw1 = ((1 - y) * pre_y - (1 - pre_y) * y) * x1          #目标函数关于 w1 的偏导数
        dw2 = ((1 - y) * pre_y - (1 - pre_y) * y) * x2          #目标函数关于 w2 的偏导数
        db = ((1 - y) * pre_y - (1 - pre_y) * y) * 1            #目标函数关于 b 的偏导数
        perceptron2.w1 = perceptron2.w1 + lr * dw1      #更新 w1
        perceptron2.w2 = perceptron2.w2 + lr * dw2      #更新 w2
        perceptron2.b = perceptron2.b + lr * db #更新 b
s = '学习到的感知器：pre_y = sigmoid(%0.2f*x1 + %0.2f*x2 + %0.2f)'\
%(perceptron2.w1,perceptron2.w2,perceptron2.b)
print(s)
```

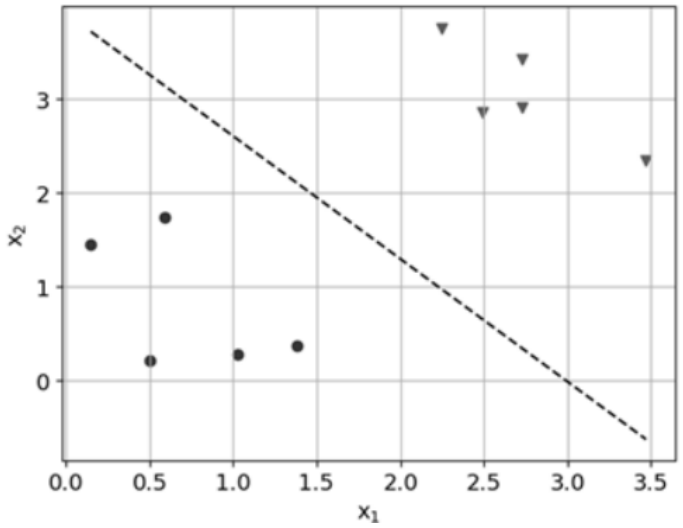
2.3.2 面向回归问题的训练方法

```
for (x, y) in zip(X, Y): #使用感知器做预测测试
    t = 1 if perceptron2.f(x) > 0.5 else 0 #阶跃变换
    s = ''
    if t == y.item():
        s = '点(%0.2f, %0.2f)被<正确>分类！'%(x[0],x[1])
    else:
        s = '点(%0.2f, %0.2f)被<错误>分类！' % (x[0], x[1])
    print(s)
```

2.3.2 面向回归问题的训练方法

执行上述代码，输出结果如下：

```
学习到的感知器：pre_y = sigmoid(-1.82*x1 + -1.39*x2 + 5.43)
点(2.49, 2.86)被<正确>分类！
点(0.50, 0.21)被<正确>分类！
点(2.73, 2.91)被<正确>分类！
点(3.47, 2.34)被<正确>分类！
点(1.38, 0.37)被<正确>分类！
点(1.03, 0.27)被<正确>分类！
点(0.59, 1.73)被<正确>分类！
点(2.25, 3.75)被<正确>分类！
点(0.15, 1.45)被<正确>分类！
点(2.73, 3.42)被<正确>分类！
```



04

使用PyTorch框架

2.4.1 PyTorch 框架的作用

编写程序的工作量主要体现在两方面：

（1）设计模型（包括感知器）的结构，以及让程序能够执行从输入到输出的前向计算；

（2）设计目标函数并让程序能够反向计算各参数的梯度及更新这些参数。

如果有一个工具，它能够自动帮助我们完成梯度计算和参数更新，那么我们就可以将更多的精力放在编写业务逻辑上面，不断提升程序的功能。深度学习框架正是出于这种需要的而推出的。实际上，它不但可以自动完成梯度计算和参数更新，而且提供了大量的辅助功能——通过调用其提供的函数可以轻而易举地实现复杂的业务逻辑，使得深度学习应用开发事半功倍。

2.4.1 PyTorch 框架的作用

常见的深度学习框架: Caffe/Caffe2、 Torch、 TensorFlow、 Keras、 MXNet等。

- 这些框架不但封装了常用的深度学习函数，如卷积、sigmoid、softmax 等，而且支持自动梯度计算和参数更新。利用这些框架，我们只需编写前向计算过程，它们会自动进行梯度计算和反向传播，完成参数的自动更新，大大减少编程人员的工作量。
- 在学术界，最常用的框架是 **PyTorch**，其次是 TensorFlow。当然，在进行深度学习应用开发时，具体选择哪一种框架，应根据个人的知识结构、易用程度、运行性能、是否真正开源等方面去斟酌。

2.4.2 使用 PyTorch 框架实现感知器

【例2.5】使用 PyTorch 框架实现例 2.4 的感知器功能。

为了运用 PyTorch 框架，一般都需要使用面向对象编程方法，即先定义类，然后创建类的对象，进而通过调用对象的方法来实现学习功能。这是因为我们需要通过创建新类才能继承类 nn.Module，而该类封装了 PyTorch 框架提供的诸多强大的功能，包括梯度计算和参数更新功能等。

在例 2.4 中，我们已经通过定义类 Perceptron2 来实现感知器功能。在此，为运用 PyTorch 框架，我们首先对类 Perceptron2 做两个地方的修改：一是让类 Perceptron2 继承类 nn.Module，同时要在代码开头引入 torch.nn 模块：

```
import torch.nn as nn
```

2.4.2 使用 PyTorch 框架实现感知器

二是要在类 Perceptron2 的__init__(self)方法中添加一种特殊的方法：super(Perceptron2,self).__init__()，它解决了子类调用父类方法的一些问题；或者去掉所有的参数，写成super().__init__()。

在此基础上，主要再修改三个地方： （1）告诉 PyTorch 框架哪些参数需要更新。为此，删__init__(self)方法中原来定义三个变量 w1, w2 和 b 的代码：

```
self.w1 = torch.Tensor([0.0])
self.w2 = torch.Tensor([0.0])
self.b = torch.Tensor([0.0])
```

修改为下列代码：

```
self.w1 = nn.Parameter(torch.Tensor([0.0]))
self.w2 = nn.Parameter(torch.Tensor([0.0]))
self.b = nn.Parameter(torch.Tensor([0.0]))
```

2.4.2 使用 PyTorch 框架实现感知器

（2）选择优化器和设置学习率。优化器有很多种，其中 Adam 已经被验证在多种场合下均有较好性能的优化器，一般选择它作为优化器。在本例中，我们设置如下：

```
optimizer = torch.optim.Adam(perceptron2.parameters(), lr=0.1)
```

其中，perceptron2.parameters()包含了实例 perceptron2 中所有需要优化的参数，Adam 将自动对这些参数进行求导和优化。我们也可以直接打印这些参数出来查看：

```
for e in perceptron2.parameters():
    print(e.data)
```

另外，lr=0.1 表示学习率设置为 0.1，也可根据需要进行更改。其默认值为 0.01。

2.4.2 使用 PyTorch 框架实现感知器

（3）让 PyTorch 框架执行反向传播和自动的梯度计算和参数更新。为此，可删除原来程序中用于计算梯度和更新参数的代码：

```
dw1 = ((1 - y) * pre_y - (1 - pre_y) * y) * x1 #目标函数关于 w1 的偏导数
dw2 = ((1 - y) * pre_y - (1 - pre_y) * y) * x2 #目标函数关于 w2 的偏导数
db = ((1 - y) * pre_y - (1 - pre_y) * y) * 1 #目标函数关于 b 的偏导数
perceptron2.w1 = perceptron2.w1 + lr * dw1 #更新 w1
perceptron2.w2 = perceptron2.w2 + lr * dw2 #更新 w2
perceptron2.b = perceptron2.b + lr * db #更新 b
```

改为如下代码：

```
optimizer.zero_grad() #对参数的梯度清零，去掉以前保存的梯度，否则会自动累加梯度
loss.backward() #反向转播并计算各参数的梯度
optimizer.step() #利用梯度更新参数
```

2.4.2 使用 PyTorch 框架实现感知器

另外，我们还利用 nn 模块提供的函数来构造目标函数（损失函数），代码如下：

```
loss = nn.BCELoss()(pre_y, y)
```

该语句实际上就是用 PyTorch 函数来表示我们定义的目标函数。该目标函数的数学公式如下：

$$\mathcal{L}(\hat{y}, y) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

我们可以自己编写目标函数的实现代码，然后调用它。例如：

```
def L(pre_y, y): #定义目标函数
    loss = -y * torch.log(pre_y) - (1.0 - y) * torch.log(1.0 - pre_y)
    return loss
loss = L(pre_y, y)
```

2.4.2 使用 PyTorch 框架实现感知器

一个感知器可以视为一个神经元，而神经网络是由多个神经元组成的计算网络。PyTorch 框架提供了 nn.Linear()函数来创建一个由多个神经元组成的神经网络层。当一个网络中只有一个网络层，而且该层中只有一个神经元，那么该网络实际上就是由一个神经元（感知器）组成了。带偏置项的单个神经元（感知器）可用下列代码创建：

```
nn.Linear(in_features=m, out_features=1, bias=True)
```

其中，in_features=m 表示输入样本的特征个数为 m，out_features=1 表示仅有一个神经元，因而也只有一个输出，bias=True 表示为每个神经元设置一个偏置项（如果有多个神经元的话）。但该函数没有设置任何的激活函数。 执行该函数时，其涉及的 m+1 参数都自动被设置为可训练参数，而不需要 nn.Parameter()再做设置。

2.4.2 使用 PyTorch 框架实现感知器

【例 2.6】利用 nn.Linear()函数来创建感知器，实现例 2.4 和例 2.5 的功能要求。

从上述对 nn.Linear()函数的介绍可以知道，由于一个样本有两个特征值，要实现本例的要求，主要是修改例 2.5 程序中的下面代码：

```
self.w1 = nn.Parameter(torch.Tensor([0.0]))
self.w2 = nn.Parameter(torch.Tensor([0.0]))
self.b = nn.Parameter(torch.Tensor([0.0]))
```

将其改为：

```
self.fc = nn.Linear(in_features=2, out_features=1, bias=True)
```

该语句创建了有两个特征值输入、带一个偏置项的感知器。其他与此相关的代码也做相应的变动。

本章内容：

- 感知器
- 监督学习和无监督学习
- 线性回归、逻辑归回、分类
- 感知器的训练
- pytorch框架