

DL4J: 人工智能基础 (Java 版)

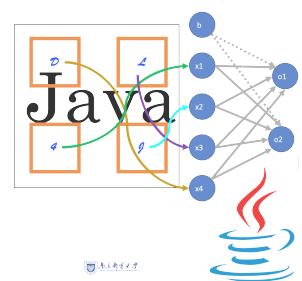
jdeepr 经典之作

作者: @YangJ9966, @imbbty, @simbaba

组织: jdeepr

时间: September 1, 2019

版本: 0.05



Victory won't come to us unless we go to it. — M. Moore

目 录

第一部分 Java 基础	1
1 开发环境	2
1.1 安装 JDK	2
1.2 安装 IDE	3
2 Java 基础	4
3 本章总结	5
 第二部分 数学基础与算法	 6
4 微积分	8
4.1 偏导数与梯度	9
4.2 线性拟合	10
4.3 优化问题	11
4.4 损失函数	14
5 概率与分布	17
5.1 全概率公式	18
5.2 最大似然	18
5.3 期望与方差	18
5.4 常见概率分布	19
5.5 矩估计	19
6 线性代数	21
6.1 矩阵和向量	21
6.2 雅克比矩阵	25
6.3 Hessian 矩阵	25
6.4 泰勒展开和牛顿法	26
6.5 向量与矩阵求导	26
7 数学基础-DEMO	27
7.1 向量运算	27
7.2 矩阵运算	30

7.3 梯度下降	32
7.4 概率分布	34
7.5 损失函数	34
7.6 拟合算法	35
8 机器学习	36
8.1 分类问题	36
8.2 线性回归	37
8.3 逻辑回归	40
8.4 正向传播	41
8.5 反向传播	41
8.6 全连接网络	43
9 有监督学习	44
9.1 训练过程	44
9.2 回归问题	44
9.3 SVM	46
9.4 卷积神经网络	48
9.5 手写识别示例	48
10 无监督学习	49
10.1 聚类算法	49
10.2 DL4J 示例	51
11 神经网络	52
11.1 评价标准	52
12 深度学习	53
12.1 多层感知机	53
12.2 PlayGround	54
第三部分 人工智能应用	56
13 ND4J	58
13.1 ND4J 介绍	58
13.2 使用 ND4J	58
13.3 线性回归示例	60
14 DL4J	61
14.1 DL4J 介绍	61
14.2 构造网络	61



14.3 准备数据	63
14.4 配置 DL4J	66
14.5 线性回归	66
14.6 逻辑回归应用	68
14.7 多分类问题	68
14.8 聚类问题	68
14.9 卷积神经网络	68
14.10 循环卷积神经网络	68
第四部分 综合运用	69
15 大数据和分布式	71
15.1 云计算	71
15.2 边缘计算	71
15.3 开源架构	71
15.4 示例：人脸识别	73
16 车牌识别	74
16.1 图像切割	74
16.2 算式识别	76
16.3 结果评估	76
17 写诗机器人	77
17.1 word2vec 介绍	77
17.2 如何写诗	77
第五部分 常见问题和附录	78
18 常见问题集	79
A 基本数学工具	81
A.1 求和算子与描述统计量	81



第一部分

Java 基础

第1章 开发环境

内容提要

- 开发环境
- 安装 IDE
- 安装 JDK

本书使用 Java 语言介绍机器学习算法，所有的演示代码可在 JDK8 以上正常运行。在开始学习之前，请先安装 JDK（Java Development Kit）和 IDE（Integrated Development Environment）。

表 1.1: 开发环境

序号	开发环境	
1.	JDK8+	建议 Oracle JDK
2.	IntelliJ IDEA	界面友好，操作简单
3.	VSCodium	强大开源，插件众多
4.	ND4J	Java 实现的矩阵/向量支持库
5.	DL4J	开源机器学习 Java 实现

1.1 安装 JDK

安装 JDK（Java Development Kit），有 2 个选择：Oracle JDK 和 Open JDK。在本书中，你可以无视它们两者之间的差异。使用 Linux 系统的同学，保持 Open JDK 即可。值得注意的是，近年来关于 Oracle JDK 的纠纷越来越多，实际上，在 Sun 被 Oracle 收购之前，JDK 也是要授权费的，但很少有公司愿意支付这笔费用。



JRE（Java Runtime Environment）是 Java 程序的运行环境，不要下载错了。安装 JDK 的过程中，全部默认点选“下一步”即可。结束后，需要手动配置以下 2 个环境变量：`JAVA_HOME`、`CLASSPATH`。配置环境变量的主要目的是告诉 IDE 或系统，在哪里能找到某个程序或文件。把 `JAVA_HOME` 设定为 JDK 的安装目录，进一步在此基础上添加 `bin` 目录到环境变量。如果没有配置，或者设置不正确，在控制台（cmd、bash）就会提示错误。

```
# Ubuntu
parallels@ubuntu-vm:~$ java
Command 'java' not found, but can be installed with:
```

```
# Windows  
C:\Users\simbaba>java  
'java' 不是内部或外部命令，也不是可运行的程序  
或批处理文件。
```

Windows 上的配置相对比较简单，参考下图打开环境变量配置窗口。选择系统环境变量或者帐号环境变量里面都可以，区别是系统环境变量会影响所有用户。通常建议修改帐号环境变量，有些帐号可能也没权限修改系统环境变量。在其中，新建 2 个条目：

1. JAVA_HOME, java 的安装目录
2. 在 path 后面追加, %JAVA_HOME%\bin



Linux 和 Mac 的配资方法大致相同，需要修改配置文件。¹

```
# Mac、Linux  
export JAVA_HOME= JAVA安装目录  
export path=$PATH:$JAVA_HOME/bin
```

1.2 安装 IDE

目前主流的 IDE 有：Eclipse、IDEA、VSCode。

¹.bash_profile、.bashrc 或.profile 等。

第 2 章 Java 基础

XXXXX

第3章 本章总结

XXXX

第二部分

数学基础与算法

AI 最近变得非常流行了，但它的历史已经有六十多年。在 1956 年的达特茅斯会议，首次提出了 AI 的概念，而在一年后，Rosenblatt 就提出了感知机（perceptron）模型。

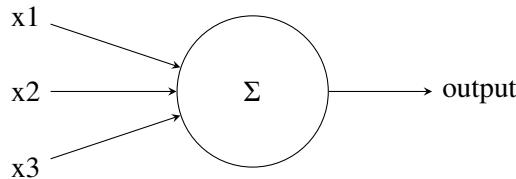


图 1：感知机

如上图 1，感知机就是二分类的线性分类模型，其输入为样本的特征向量，输出为样本的类别，取 +1 和 -1 二值。即通过某样本的特征，就可以准确判断该样本属于哪一类。顾名思义，感知机能够解决的问题首先要求特征空间是线性可分的，再者是二分类，即将样本分为 +1, -1 两类。

使用感知机可轻松解决异、或、非的问题，却存在一个致命缺陷：无法解决异或问题。导致 AI 进入第一个衰退期，直到 19 世纪八十年代，才逐渐复苏。1984 年，Hiton 教授提出 Boltzman 机模型。到 1986 年 Kumelhart 等人提出误差反向传播（Back Propagation）算法，简称 BP 网络。然而 90 年中期支持向量机（Support Vector Machines，SVM）强势登场，全方位碾压 NN。SVM 具有 1) 高效，可以快速训练；2) 无需调参，没有梯度消失问题；3) 高效泛化，全局最优解，不存在过拟合问题。几乎全方位的碾压 NN，NN 也再次被打入冷宫。

目前，BP 网络已成为广泛使用的网络。1987 年至今为发展期，神经网络受到各个国家的重视，纷纷展开研究，形成神经网络发展的另一个高潮。人工智能是一个将数学、算法和工程相结合的学科，包含微积分、概率论、统计学等数学知识。本章将为大家简要介绍人工智相关的数学基础理论。

第4章 微积分

微积分 (Calculus) 是数学的一个基础学科，包括极限、微分学、积分学及其应用。与机器学习相关的微积分问题有：极值问题、偏导数和梯度。在机器学习中，定义的所谓模型就是一个包含参数和特征的函数。定义如下：

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$$

其中， θ_i 称为参数，而 x_i 称为特征， h_{θ} 是预测模型。训练模型之前，都会采集很多组样本数据 (x_1, x_2, \dots, x_n) ， x_i 就是预设的特征。模型的准确性，取决于特征的完备性和数据的充分性。取什么样的特征（采集什么样的数据）有时候也很难确定，过少的或过多的特征，都影响模型的准确性。

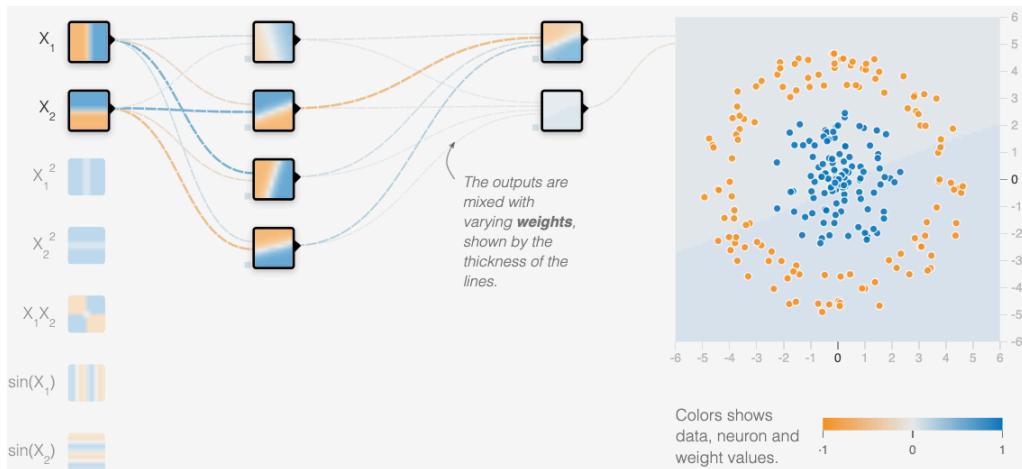


图 4.1: 线性特征分类

极端情况下，已有的样本数据少于特征的维度（相当于未知数多于方程个数），需要引入 λ 正则项。通过岭回归 (Ridge Regression) 得出最优的 λ 和 θ ，下一章线性代数部分会详细说明。

如果特征值 (x_1, x_2, \dots, x_n) 向量线性相关，就有必要减少特征的维度。很显然，特征空间的维度降低可以极大地简化模型。采集的样本数据通常都有一些噪点，要引入正则化降低它们对训练结果的影响，防止过拟合的出现。注意，欠拟合是指模型包含的特征维度低于现实情况，也无法训练出有效的结果。

由图 4.1 可知，两个线性特征 x_1 和 x_2 通过线性组合只能训练出多边形特征的模型。加入特征 x_3 并定义为 x_1^2 ，特征 x_4 定义为 x_2^2 ，改进后的模型分类预测更符合预期。实际上，需要哪些特征也并不是那么很明显。首先是我们能获取到哪些数据，主观判断是否相关。譬如，你可以拿到的数据：(年龄，性别，身高，体重，籍贯)，想用于预测某人的籍贯。

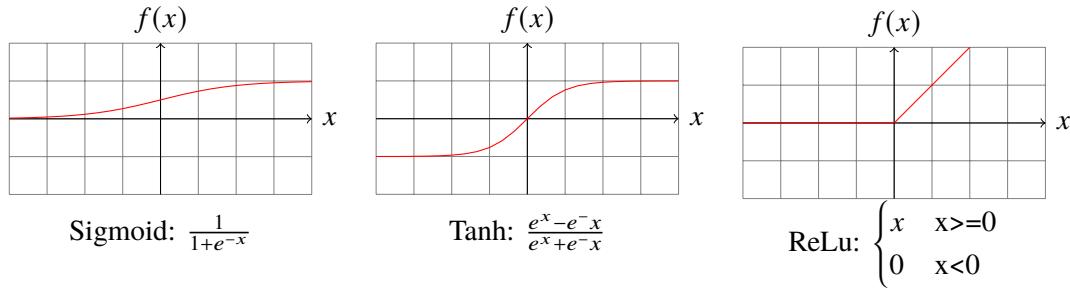
$$f(\text{年龄}, \text{性别}, \text{身高}, \text{体重}) \rightarrow \text{籍贯}$$

在定义模型之后，通常使用最小二乘法定义损失函数（代价函数，Cost Function）。然

后把准备好的大量样本喂给模型，进行迭代训练使其学习出最优的 θ ，使得代价函数的值尽量小。

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

在机器学习中，使用激活函数对结果进行逻辑分类：Sigmoid 函数，Tanh 函数，ReLU 函数。它们的主要区别是什么？请同学们仔细观察函数图形之间的共性和区别。



引入激活函数不仅用于逻辑分类，关键是还引入了非线性因素。Sigmoid 函数的输出在 (0,1) 之间，单调连续，并且容易求导。但它一旦输入落入饱和区，一阶导数就会接近于 0，容易产生梯度消失；Tanh 函数也是传统神经网络中常用的激活函数，与 Sigmoid 一样存在饱和问题。不过它的输出以 0 为中心，Tanh 函数看上去是放大的 Sigmoid 函数。

ReLU 是目前用的最多也是最受欢迎的激活函数，可加速 SGD(梯度下降算法) 的收敛。但 ReLU 随着训练的进行，部分输入也会落到硬饱和区。除了 ReLU，还有许多 ReLU 衍生的激活函数，比如：Leaky ReLU、ReLU6、SReLU、PReLU、RReLU、CReLU。

4.1 偏导数与梯度

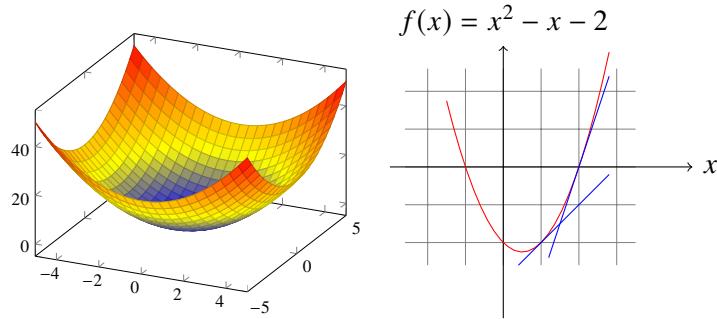
在微分中，导数表示函数的变化率，也是变化率最大的方向。梯度是全部偏导数构成的向量，一般将函数 f 的梯度记为 Δf 。根据方向导数的知识，可知梯度的方向即为函数值增长最快的方向。

$$\text{grad}f(x_0, x_1, \dots, x_n) = \left(\frac{\delta f}{\delta x_0}, \frac{\delta f}{\delta x_1}, \dots, \frac{\delta f}{\delta x_n} \right)$$

函数在某一点的梯度，是一个向量，与最大方向导数的方向一致，而它的模就是方向导数的最大值。既然，沿梯度方向具有最大的变化率，所以沿着负梯度方向去减小函数值，才能最快达到优化目标。

梯度下降就像是你低头只看脚下的下山过程，若想最快到达山底，最好每一步都找最陡峭的方向走，这就是梯度的负方向。由于视野的局限性，使用梯度下降也有可能落入局部最小值。

例子函数 $f(x) = x^2 - x - 2$, 梯度: $f'(x) = 2x - 1$



假设 $\alpha=0.2$ 可以这样推算：设起点为 $(4, 0)$

```

1次: 参数x=4, 4-0.2*(2*4-1)=2.6
2次: 参数x=2.6, 2.6-0.2*(2*2.6-1)= 1.76
3次: 参数x=1.76, 1.76-0.2*(2*1.76-1)=1.256
4次: 参数x=1.256, 1.256-0.2*(2*1.256-1)=0.9536
5次: 参数x=0.9536, 0.9536-0.2*(2*0.9536-1)=0.77216
.....
After 13 steps: x = 0.502743, f(x)= -2.249992.
After 14 steps: x = 0.501646, f(x)= -2.249997.
After 15 steps: x = 0.500987, f(x)= -2.249999.
After 16 steps: x = 0.500592, f(x)= -2.250000.
After 17 steps: x = 0.500355, f(x)= -2.250000.
After 18 steps: x = 0.500213, f(x)= -2.250000.

```

4.2 线性拟合

样本的散点图有助于发现关键特征，建立正确的模型。如下图，可直观地识别出线性关系，尽管有干扰数据。

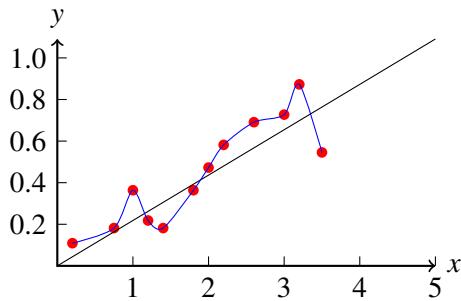


图 4.2: 线性拟合

拟合指的是你逼近目标函数的远近程度。欠拟合，是指模型复杂度过低，不能很好的拟合所有的数据，训练误差大，在训练和预测时表现都不好。欠拟合很容易被发现，在训练的时候就表现很差。过拟合，是指模型复杂度很高，但训练数据很少，导致训练误

差小，测试误差也大。过拟合，过度地学习训练数据中的细节和噪音，以至于模型在新的数据上表现很差，泛化性能变差。¹

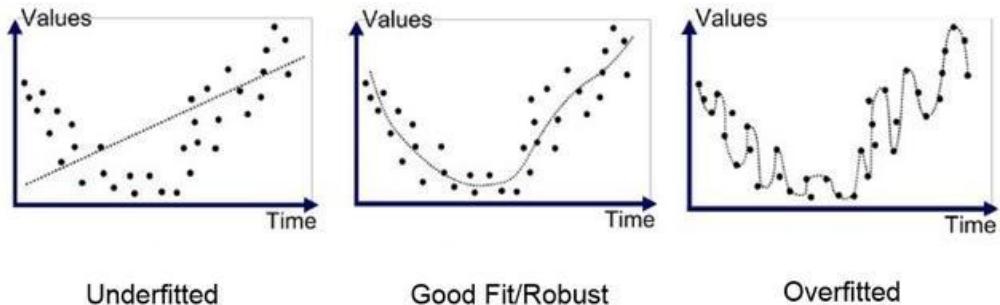


图 4.3: 拟合情况分类

4.2.1 最小二乘法

最小二乘法是勒让德 (A. M. Legendre) 于 1805 年在其著作《计算彗星轨道的新方法》中提出的。它的主要思想就是求解未知参数，使得理论值与观测值之差（即误差，或者说残差）的平方和达到最小。

$$E = \sum_{i=1}^n (y_i - \hat{y})^2$$

4.3 优化问题

拉格朗日乘子法 (Lagrange Multiplier)，用于求解带等式约束的极值问题，而 KKT (Karush Kuhn Tucker) 条件是拉格朗日乘子法的推广。在有等式约束时使用拉格朗日乘子法，在有不等约束时使用 KKT 条件。

4.3.1 拉格朗日乘子

拉格朗日乘子法是一种寻找多元函数在其变量受到一个或多个条件的约束时的极值的方法。设目标函数为 $f(x)$ ，约束条件为 $h_k(x)$ ，其中 s.t. 表示 subject to，“受限于”的意思， l 表示有 l 个约束条件。定义如下：

$$L(x, \lambda) = f(x) + \lambda h_k(x)$$

$$\min f(x) \quad | \quad \text{s.t. } h_k(x) = 0 \quad k = 1, 2, \dots, l$$

1. 构造拉格朗日函数
2. 解偏导方程

¹欠拟合 (underfitting)，或者叫作叫做高偏差 (bias)，过拟合 (overfitting)，也叫高方差 (variance)

² y_i 是样本数据， \hat{y} 是理论值

3. 代入上述函数

由拉格朗日乘子法的定义可知，它是一种寻找极值的方法，因此该方法并不能保证极值点是最低点或者最高点。

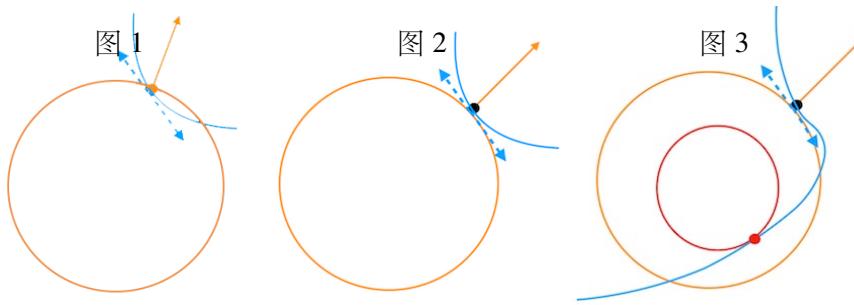


图 4.4: 拉格朗日算子法

这里不加任何推导，很明显相切的时候才能到最低点。橙线的梯度（上图 1 橙色箭头）和蓝线的切线（蓝色虚线）不是垂直关系。蓝线的两个切线方向，分布往函数高处走（与梯度的夹角小于 90 度），和往函数低处走（与梯度的夹角大于 90 度）。因此，两条曲线相交时，肯定不在最低点或最高点的位置。

如果两条曲线相切（上图 2），那么蓝线的切线和橙线的梯度是在切点这个位置垂直。这时，蓝线的切线方向指向橙线的等高线方向。图 3 中相切的点有两个，而红点的明显比黑点小。要判断找到的点是极低点还是极高点，需要将切点代入原函数再进行判断。在实际求解时，令各偏导为 0 就能满足相切的条件。

4.3.2 KKT 条件

在优化理论中，KKT 条件是非线性规划（Nonlinear Programming）最优解的必要条件。KKT 条件将拉格朗日算子法中的等式约束优化问题推广至不等式约束。将上一节中的约束等式 $h_k(x) = 0$ 推广为 $g_i(x) \leq 0$ ，优化问题如下：

$$\min f(x) \quad | \quad s.t. g(x) \leq 0$$

约束不等式 $g(x) \leq 0$ 称为 Primal Feasibility，定义可行域为（Feasible Region） $K = \{x \in \mathbb{R}^n : g(x) \leq 0\}$ 。设 x^* 为满足约束条件的最优解，可分为内部解、边界解两种情况。内部解的时候， $g(x)$ 不起作用，退化为无约束问题。边界解的时候，取不等式的边界，约束转换为 $g(x) = 0$ ，转化为 Lagrange 算子法。

1. $g(x^*) \leq 0$ ，最佳解位于 K 的内部，此时约束条件是无效的；
2. $g(x^*) = 0$ ，最佳解落在 K 的边界，此时约束条件是有效的。

KKT-图 1 中，把 Lagrange 条件变成一个区域，该图的切点处仍旧是最优解。KKT-图 2 中，在边界处 $g(x) = 0$ 等价于 Lagrange 算子法。KKT-图 3 中，最佳解位于 K 的内部，约束条件无效。KKT 是 SVM（support vector machine）支持向量机的重要理论基础。

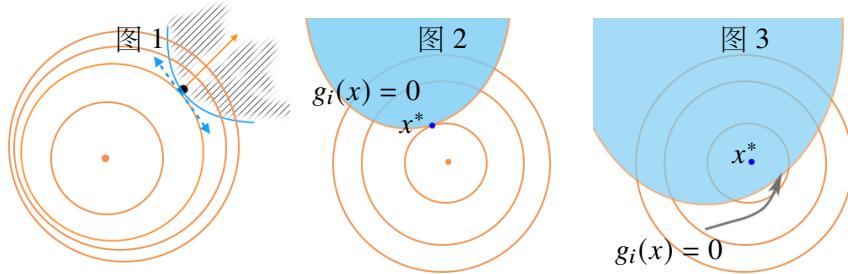


图 4.5: KKT

4.3.3 凸优化

凸优化在数学规划领域具有非常重要的地位。若能把一个实际问题表述为凸优化问题，基本上意味着该问题已经得到解决，这是非凸的优化问题所不具有的性质。机器学习中有很多优化问题都要通过凸优化来求解，即便是在非凸优化中，凸优化同样起着重要的作用。实际上，很多非凸优化问题，可以转化为凸优化问题来解决。

$$f_i(\alpha x + \beta y) \leq \alpha f_i(x) + \beta f_i(y), \quad \text{其中 } \alpha + \beta = 1, \alpha \geq 0, \beta \geq 0$$

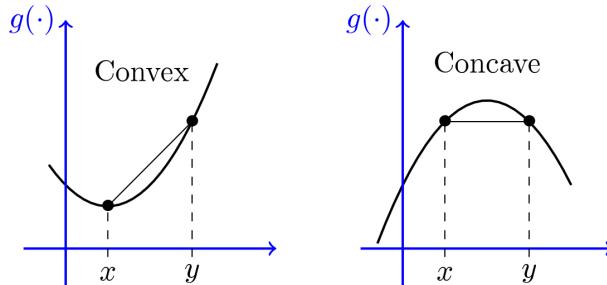


图 4.6: 凸函数和凹函数

如图 4.6 所示，凸函数的几何意义为任意两点连线上的取值大于该点在函数上的取值，而凹函数正好相反。很显然，凸函数总是在其任意一点的切线的上方。通常使用函数的二阶导来判断一个函数是否为一个凸函数。

$$\nabla_x^2 f(x) \geq 0$$

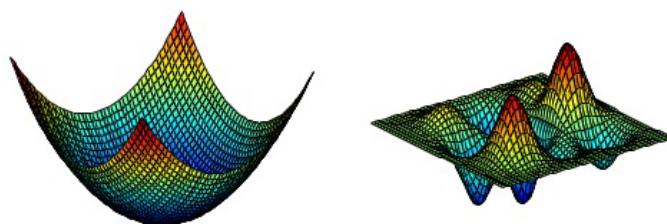


图 4.7: 凹函数转化为多个凸函数优化问题

凸优化相对简单，具有良好的几何性质，比如分离平面和支撑平面。通常凸问题的

局部最优解也是全局的最优解，Lagrange 对偶是凸优化理论中最重要的工具。往往只要把某一问题抽象为凸问题，就可近似认为这个问题已经解决了。

SVM 本身就是把一个分类问题抽象为凸优化问题，利用凸优化的各种工具（如 Lagrange 对偶）求解和解释。深度学习中关键的算法是反向传播（Back Propagation），本质就是凸优化算法中的梯度下降算法。总的来说，凸优化在工程领域的应用中有着无可撼动的地位。

实际上，生活中几乎所有问题的本质都是非凸的。如图 4.7 所示，很多非凸问题可以转化成多个凸优化问题，加速问题的求解。

4.4 损失函数

在机器学习中，机器的学习需要某个指标来表示现在的状态，然后，以这个指标为基准，寻找最优权重参数。机器学习利用已知样本，推演隐藏在背后的真实曲线。这里假设该某曲线为 $h_\theta = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$ ，通过样本求解 $\theta_i (i = 0 \dots 4)$ 。不过，这个假设（Hypothesis）很有可能不是最好的。实际上，我们获得的训练数据总是有误差，不能完全拟合这些数据点，否则就会导致过拟合（overfit）的问题。

评价训练过程是否有效，可使用方差衡量源数据和期望值相差的偏离程度。若是逐渐减小，就是一个有效的训练过程。这个方差也是一个函数，称为损失函数（loss function）³，用于计算预测值 $f(x)$ 与真实值 y 的不一致程度。它是一个非负的实值函数，通常使用 $L(Y, f(x))$ 来表示。损失函数值越小，拟合效果就越好。当样本个数为 n 时，此时的损失函数变为：

$$L(Y, f(x)) = \frac{1}{N} \sum_N (Y - f(x))^2$$

除 2 是为了方便求导

$Y - f(X)$ 表示的是残差，整个式子表示的是残差的平方和，而我们的目的就是最小化残差的平方和。在实际应用中，通常使用均方误差（mean squared error）作为衡量指标，如下：

$$E = \frac{1}{2} \sum_k (y_k - t_k)^2$$

在上式中 k 表示着数据处于哪一个维度， t_k 表示着各维度的监督数据。通常情况下，监督数据仅将正确解标签表示为 1，而其他非正确解则表示为 0。而机器学习输出的结果 y_k ，则是会在各个维度都显示该维度是正确解标签的可能性。均方误差通过计算机器学习的输出和正确解监督数据的各个元素之差的平方，再求总和。从而得到该权重参数的输出结果与正确结果之间的偏差。

我们现在再介绍另一种常用函数，交叉熵误差（cross entropy error）。其公式如下：

$$E = - \sum_k t_k \log y_k$$

³也称为代价函数 (Cost Function)



在上式中, y_k 代表着机器学习的输出, \log 表示以 e 为底数的自然对数 (\log_e)。 t_k 代表着正确解标签, 仅当解标签为正确时, t_k 的索引才为 1。其余情况都为 0。因此, E 所代表的实际为解标签为正确时所输出的自然对数。

线性回归是确定两个或两个以上变量间关系的一种常见统计分析方法, 被广泛用于回归分析。只有一个自变量的情况称为简单回归, 多于一个自变量的情况叫做多元回归。故可分为一元线性回归分析方程和多元线性回归分析方程。给定一个随机样本 $(Y_i, X_{i1}, \dots, X_{ip})$, 线性回归模型表示为以下的形式:

$$Y_i = \beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2} + \dots + \beta_p X_{ip} + \epsilon_i$$

$$i = 1, \dots, n.$$

使用最小二乘法 (Least Square Method) 需要做矩阵的逆运算, 下一章我们再介绍。而梯度下降法起点和学习率都非常重要, 顺着梯度 Δ 下降最快的方向迭代调整。若干次迭代之后就会落入局部最小点附近, 有可能来回震荡无法达到极值点。所以, 调整学习率就非常关键, 因此到极值点附近的时候收敛速度也会变慢。

4.4.1 反向传播 (Back Propagation)

前述几节, 我们利用偏导获得梯度, 然后逐步调节参数, 朝着误差越来越小的方向迭代。这还算不上神经元, 实际比这还要复杂一些, 通常还有一个激活函数 (Activation Function), 只有对神经元的刺激足够强才会前向传递。典型的深度神经网络, 至少包含: 输入层、隐藏层、输出层。

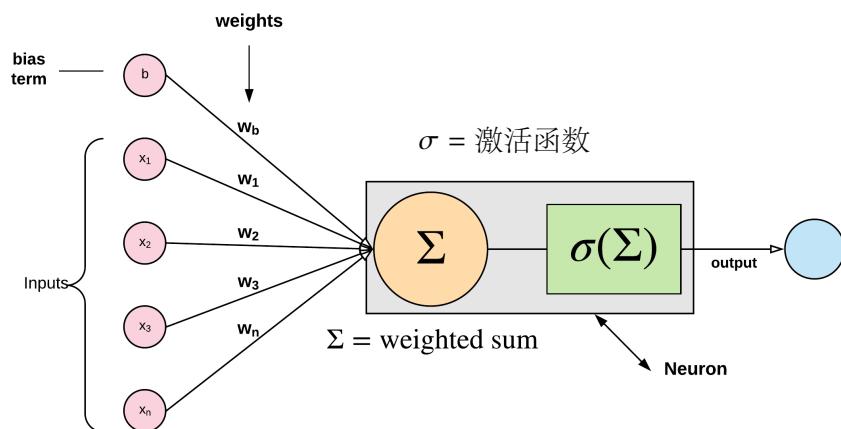


图 4.8: 神经网络

缺少激活函数的线性模型, 甚至都无法解决异或问题。如果选用 Linear 函数 $g(x) = x$ 作为激活函数就无法划分下图的区域, 很显然隐藏层的混入也非常关键。通过隐藏层引入了更多线段, 以便合成一个封闭的多边形, 恰好能对数据分类。激活函数可以在隐藏层引入更多特征, 产生非线性结果以解决线性不可分问题。

BP 神经网络是一种多层的前馈神经网络, 分为两个阶段: 前向传播和反向传播。前向传播从输入层经过隐含层, 最后到达输出层。而反向传播从输出层到隐含层, 最后到

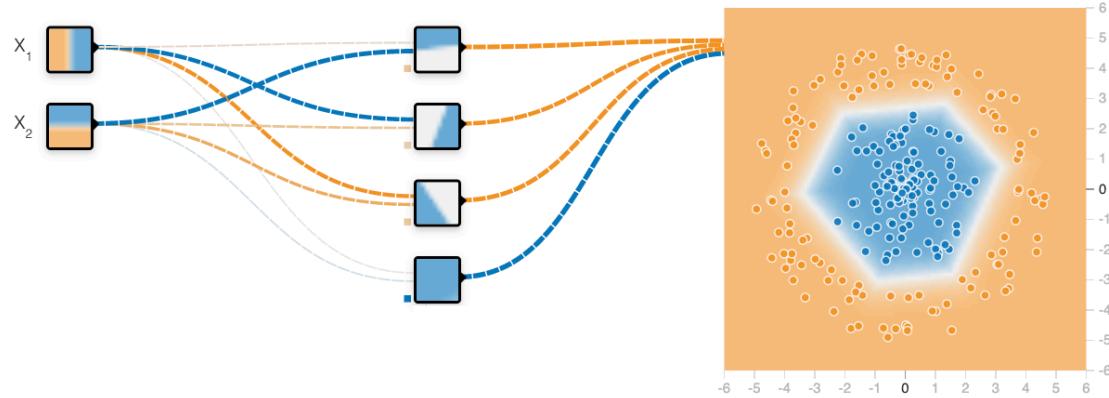
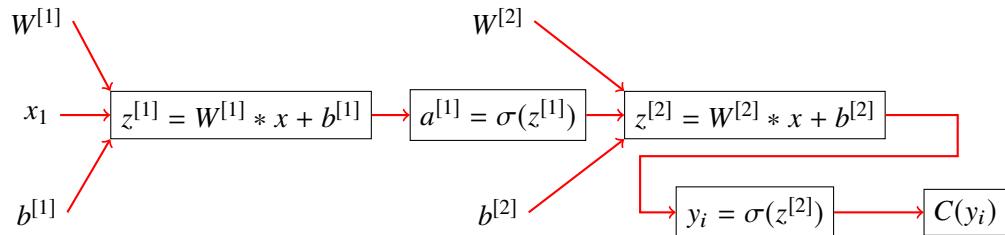


图 4.9: 使用激活函数，合成多边形区域

输入层，逐步调节隐藏层（Hidden Layer）到输出层的权重和偏置（bias），输入层到隐含层的权重和偏置。经由反向传播，把误差反馈给神经网络用于调节参数。此处不作严格证明，

假设有一个两层深的网络（1个隐藏层），并且每一层只有一个神经元， σ 为 sigmoid 函数。相应的数学模型如下：



从输入到输出，函数都是平滑的可求偏导的，由误差估计 $C(\hat{y})$ 使用梯度下降法，反向调节参数。其中， $a^{[i]}$ 作为下一个神经元的输入，也就是 x_i 。

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_1} = \frac{\partial C}{\partial y_i} \cdot \sigma'(z_2) \cdot W_2 \cdot \sigma'(z_1)$$

如图 4.8，神经网络的输入层就是我们要分类的样本特征，而隐藏层和输出层每个节点都代表了一个 sigmoid 单元。计算的时候，通常把 b 转化为 x_0 和 w_0 ，可把 sigmoid 单元简化为 $net = \sum_{i=0}^n w_i x_i$ 。求各输入的和用 sigmoid 函数得到输出 $out = \sigma(net) = \frac{1}{1+e^{-net}}$ ，这便是一个单元的计算过程。

整个网络便是前一层作为输入与后一层的每各单元连接，计算出各单元的输出后，再把这一层做为输入传递到后一层。算法本身并不复杂，隐含在其中的数学推理比较晦涩，包含的过程主要是梯度下降和函数迭代。

第5章 概率与分布

日常生活中，我们差不多一直在做各种判断，这样或者那样。俗话说，选择大于努力。如果你的认知或者价值观是错误的，将不会得出正确的结果。这种价值观的偏离，需要“误差分析”保证在正确的轨道上。然而，误差很难线性（任意曲线）吻合的，这取决于个人的生活经验，



手写数字：4 还是 9？

如上图，手写数字很不容易判断，只能根据大多数人的情况猜测。如果，你能拿到该人之前的手稿了解一下，就能有 99.999% 概率做出正确判断。

有很多情形，我们也很难正确判断，只需要人工智能输出 TOPN 可能性。譬如，仅根据病人是否发烧、咳嗽难以确诊是否肺炎。进一步，借助沟通和血液化验，才更加确诊病人的实际病症。这也只能说是 99.99% 的准确率，即使是非常有经验的医生也有误诊的概率，只是非常小而已。

人工智能研究的多是不确定性问题，需要你掌握概率和统计学的基础知识。现实生活中，很多不确定事件都服从某种分布，譬如公交地铁站客流量服从泊松分布，而 1 小时内到银行办理业务的客户数都也服从泊松分布（Poisson distribution）。泊松分布具有以下特点：

1. 无限分隔为若干小时间段，在这接近于零的小时间段里，发生 1 次的概率与时间段的长度成正比。
2. 在每一个极小时间段内，该事件发生两次及以上的概率恒等于零。
3. 在不同的小时间段里，发生与否相互独立。

$$P(x) = \frac{e^{-\lambda} \lambda^x}{x!}$$

泊松分布服从上述公式，很显然它所有事件的概率和等于 1。

$$\frac{e^{-\lambda} \lambda^x}{x!} = e^{-\lambda} \left(\frac{\lambda^x}{x!} \right) = e^{-\lambda} * e^{\lambda} = 1$$

在医疗质量上，美国误诊率是 30-40% 之间，因医疗差错而死亡的人，仅次于心血管疾病与肿瘤。

个别疾病的误诊率高得使你不敢想象，达到 70% 以上。

我国的误诊率尚没有公认的数据，但与美国相比只会高不会低。

5.1 全概率公式

全概率公式是概率论中重要的公式。他的意义在于直接计算 $P(A)$ 比较困难时，可以将事件 A 分解为几个小事件 B_i ，通过计算这些小事件的概率和，从而达到简化问题的效果。

全概率公式可以用如下的公式表示：

$$P(A) = \sum_{i=1}^{\infty} P(B_i) P(A|B_i) \quad (5.1)$$

其中，事件 $B_1, B_2, B_3, \dots, B_n$ 需要满足完备事件组，即两两之间不能有交集，它们的和为全集，且概率大于零。这样，事件 A 就被分解成了小事件 $AB_1, AB_2, AB_3, \dots, AB_n$ ，由概率的加法公式即可得出事件 A 的概率。

与全概率公式相反，贝叶斯公式是基于 $P(A)$ 已知的前提下，寻找事件发生的原因 $P(B_i|A)$ 。贝叶斯公式可以用以下公式表示：

$$P(B_t|A) = \frac{P(B_t) P(A|B_t)}{\sum_{j=1}^n P(B_j) P(A|B_j)}$$

其中事件 $B_1, B_2, B_3, \dots, B_n$ 须满足完备事件组。

5.2 最大似然

最大似然估计是一种统计方法，用来求一个样本集的相关概率密度函数的参数。通俗地来讲，就是通过已知的样本结果，反推最有可能出现这样结果的参数值。

例如，假设一个口袋中同时存在黑球和白球，我们从中随机抽取十个球，得到了 8 个黑球和 2 个白球。在求解最有可能的黑白球比例时，我们就会采用最大似然法：假设从中抽到黑球的概率为 p ，那么得到 8 次黑球 2 次白球的概率为：

$$P(A) = p^8 * (1 - p)^2$$

在这个公式中，使 $P(A)$ 达到最大的 p 值即为我们要求的结果。这就是最大似然问题的基本过程。

5.3 期望与方差

期望也称数学期望，在概率论与数理统计中指的是一个离散型随机变量在实验中每次可能的结果乘上各自的概率的总和。在机器学习中，期望值是衡量一组数据离散程度的重要度量。

期望值的计算用以下公式表达 $E[X] = \sum_i p_i x_i$ ，其中 $E[X]$ 代表期望值， x_i 和 p_i 分别代表每次随机变量实验的可能结果与出现的概率。

与期望值类似，方差是概率论中衡量随机变量离散程度的度量。它具体指每个样本值与全体样本值平均数之差的平方值的平均数。一般情况下，方差的公式可以用如下公

式定义： $\sigma^2 = \frac{\sum(X-\mu)^2}{N}$ ；其中， σ^2 为总体方差， X 为单个样本值， μ 为总体样本的平均值， N 为总体样本的个数。

同时，方差还可以用期望值来求出 $\text{Var}(X) = E[X^2] - (E[X])^2$ ，由此可见期望值与方差之间紧密的联系。

5.4 常见概率分布

本书不打算把所有的概率知识进行讲解，主要为后续章节做铺垫。生活中，常见的概率事件都是离散的，譬如泊松分布。

$$P(X = x_i) = p_i, i = 1, 2, \dots, n \quad (5.2)$$

且概率 P_i 满足 $\sum_{i=1}^n P_i = 1$ 。因此，离散型随机变量 X 的概率分布函数为，其中 x_i 为可能的状态。

$$F(x) = \sum_{x_i < n} P_i. \quad (5.3)$$

这是离散型概率的特征，具备以下特征：

1. $P(x_i) = 0$ 代表不会发生， $P(x_i) = 1$ 表示一定会发生。
2. 总概率不会大于 1，也就是 $F(x) \leq 1$

概率分布函数是概率论的重要概念，在实际应用中常用的有正态分布函数、泊松分布函数、二项分布函数等。对于离散型随机变量，分布函数是“0-1 分布”、“二项式分布”、“泊松分布”等；而连续型随机变量有“均匀分布”、“正态分布”、“瑞利分布”等。

5.4.1 正态分布

正态分布，也称常态分布，高斯分布，是一种概率分布模型。正态分布在数学，工程与物理领域有着重要的意义。在机器学习中，正态分布的统计模型应用非常广泛。

若一个随机变量 X 服从位置参数为 μ ，尺度参数为 σ 的概率分布，且公式为

$$X \sim N(\mu, \sigma^2)$$

则称变量 X 满足正态分布。

正态分布的概率密度函数则表示为

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

值得注意的是正态分布的数学期望值即为位置参数 μ ，标准差即为尺度参数 σ 。

5.5 矩估计

矩估计又称矩法估计，指的是利用样本矩来估计总体所对应的参数。矩估计在概率论中有着广泛的应用，主要的理论基础有大数定理和中心极限定理。



5.5.1 大数定理

大数定理是描述实验次数很大时所呈现的概率性质的定律。在随机事件大量重复出现的情况下，事件往往呈现出近乎必然的概率，这就是大数定律最简单的解释。

大数定律在高等数学上有以下两种数学解释：切比雪夫大数定理和伯努利大数定理。切比雪夫大数定理假设有 $x_1, x_2, x_3, \dots, x_n, \dots$ 这样一列相互独立的随机变量，那么对于任意小的正数 ξ ，有公式：

$$\lim_{n \rightarrow \infty} P \left\{ \left| \frac{1}{n} \sum_{k=1}^n x_k - \frac{1}{n} \sum_{k=1}^n E x_k \right| < \varepsilon \right\} = 1$$

切比雪夫大数定理可以有以下理解：当样本容量 n 不断增大，样本的平均值将不断接近总体的平均值。

伯努利大数定理则对大数定理进行了概率的解释，假设 μ_n 是独立事件中事件 A 发生的次数，则对于任意小的正数 ξ ，满足公式：

$$\lim_{n \rightarrow \infty} P \left(\left| \frac{\mu_n}{n} - p \right| < \varepsilon \right) = 1$$

伯努利大数定理可以这样解释：当事件发生次数 n 足够大时，事件 A 发生频率将接近其发生的频率。

5.5.2 中心极限定理

中心极限定理与大数定理一样，都是描述试验次数很大时，所呈现的概率性质定理。但与大数定理不同的是，中心极限定理描述的是随机事件大量重复，结果服从正态分布的情况。

假设有随机变量 $X_1, X_2, \dots, X_n, \dots$ 相互独立，并具有方差和期望值，则对任意 x 有 $F_n(x)$

$$F_n(x) = P \left\{ \frac{\sum_{i=1}^n X_i - n\mu}{\sigma\sqrt{n}} \leq x \right\}$$

利用中心极限定理和大数定理，我们可以利用少量数据对总体进行精确的预测，从而达到矩估计的效果。

第6章 线性代数

6.1 矩阵和向量

矩阵是从许多实际问题中抽象得来的数学概念，是整个线性代数学科的基础，在自然科学，工程数学和国民经济中有着广泛的应用。在机器学习领域，大部分计算方法都是以矩阵的形式进行，因此学习矩阵的性质和计算显得尤为重要。

矩阵的定义是 $m * n$ 个数 a_{ij} ($i = 1, 2, \dots, m; j = 1, 2, \dots, n$) 排成 m 行 n 列的矩阵数表。这样的一个数表称为 $m * n$ 矩阵，简称为矩阵， a_{ij} 也被称为矩阵的第 i 行第 j 列元素。

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad (6.1)$$

特殊的，当 $m=1$ 时，矩阵 $A = (a_1, a_2, \dots, a_n)$ 被称为行矩阵，也叫 n 维行向量；同样的，当 $n=1$ 时，矩阵 $A = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$ 被称为列矩阵，也称为 m 维列向量；当 $m=n$ 时，矩阵 $A = (a_{ij})_{n*n}$ 称为 n 阶矩阵或 n 阶方阵，也可记为 A_{n*n} 。

向量指的是既包含大小，又包含方向的一类量，例如位移，速度，加速度，力等，又称为矢量。在线性代数中，向量可以用矩阵的形式进行表示，例如一个一维的向量可以用向量 $A = (a_1)$ 表示，二维向量可以用 $A = (a_1, a_2)$ 表示，如图6.1和图6.2所示。同理，我们可以推出，包含 n 个分量的向量可以称为 n 维向量，用 $A = (a_1, a_2, a_3, \dots, a_n)$ 表示。

在 DL4J 中，普通矩阵的创建用 `create` 方法实现，源码如图6.3所示。首先我们需要传入两个数组 `data` 和 `shape`，作为源数据组并决定矩阵的形状与维度。然后，系统会调用

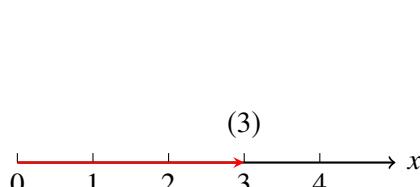


图 6.1: 一维向量示意图

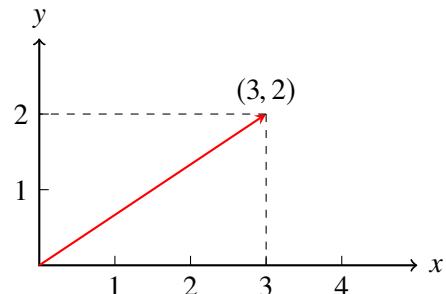


图 6.2: 二维向量示意图

checkShapeValues 方法来确保输入 data 数组的长度与 shape 数组相符。最后，利用这些数据创建一个 INDArray 并返回，这就完成了矩阵的创建。

特殊的，我们可以用简便的方法来创建部分简单的矩阵，例如 zeros() 方法可以创建一个全零矩阵，仅需要传入行数和列数即可。同理还有 ones(), valueArrayof() 等方法。

6.1.1 基本运算

矩阵有四种运算：加减、数乘、乘法和转置。下面我们将用例子展示矩阵的运算，假设我们有两个矩阵 A 和 B ，如图6.4与图6.5所示。

6.1.1.1 矩阵的加减

A 和 B 是两个 $m * n$ 型矩阵，那么存在矩阵 C ，记 $C = A + B$ ，称 C 为矩阵 A 和 B 的加法运算。需要注意的是，进行加法运算的两个矩阵行数和列数必须相等，否则无法进行运算。

矩阵的加法在 DL4J 中由 add 方法和 addi 方法实现。代码会先检测输入的数组是否为数字矩阵，以此判断是否可进行加法操作，最后将两个矩阵进行加法操作。

6.1.1.2 矩阵的数乘

矩阵 A 是一个 $m * n$ 型矩阵， k 是一个数，则称矩阵 D 为矩阵 A 与数 k 的乘积，称为数乘，记为 kA 。

矩阵的数乘与加法基本一致，都是先检验是否为数字矩阵的操作，随后对矩阵中的每个元素进行乘法操作。

矩阵的加法和数乘运算统称为矩阵的线性运算。矩阵的线性运算满足交换律，分配律和结合律。

6.1.1.3 矩阵的乘法

设 $A = (a_{ik})_{m*s}$, $B = (b_{kj})_{s*n}$, 定义 A 与 B 的乘积 AB 是一个 $m * n$ 矩阵 $C = (C_{ij})_{m*n}$, C 的第 i 行第 j 列元素等于 A 的第 i 行元素与 B 的第 j 列对应元素乘积的代数和，如图6.8:

```
public static INDArray create(double[] data, long[] shape) {  
    checkShapeValues(data.length, shape);  
  
    INDArray ret = INSTANCE.create(data, shape, Nd4j.getStrides(shape, Nd4j.  
        order()), DataType.DOUBLE, Nd4j.getMemoryManager().getCurrentWorkspace  
       ());  
    return ret;  
}
```

图 6.3: NDArray 的创建

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

图 6.4: 矩阵 A

$$\begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix}$$

图 6.5: 矩阵 B

DL4J 中，矩阵乘法的源码先对进行乘法操作的两个矩阵判断是否为同一数据类型，之后再得出乘法计算后矩阵的维度和形状，最后计算后将结果写入结果矩阵，返回结果，如图 6.9。

6.1.1.4 矩阵的转置

设存在 $m * n$ 矩阵 A ，则称 $n * m$ 矩阵 B 为 A 的转置矩阵，记为 A^T 。

在 DL4J 中，矩阵的转置调用了 `permute` 方法，先将原矩阵的形状，源数据，排列规则等属性提出并进行转置，再利用新属性生成一个新的 `INDArray` 并返回，在实际使用中我们需要调用 `transpose()` 方法即可完成矩阵的转置。

6.1.2 多维空间

多维空间一般指多维向量空间。一般来说，我们将实数域上的全体 n 维向量构成的集合称为 n 维向量空间 R^n 。

在此基础上，我们设 $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_r$ 是向量空间中 V 中的向量，若它们满足：(1) $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_r$ 线性无关；(2) V 中的任何一个向量都可以用 $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_r$ 线性表示，则称 $\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_r$ 为向量空间 V 的一个基， r 称为向量空间 V 的维数，并称 V 为 r 维向量空间。

显然，向量空间的基不是唯一的。同时，我们应该注意到向量的维数和向量空间的维数并不是同一个概念。

$$\begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots & a_{mn} + b_{mn} \end{pmatrix}$$

图 6.6: 矩阵 C

```
public INDArray add(INDArray other) {
    validateNumericalArray("add", false);
    return addi(other, Nd4j.createUninitialized(Shape.pickPairwiseDataType(
        this.dataType(), other.dataType()), this.shape(), this.ordering()));}
```

$$(ka_{ij})_{m \times n} = \begin{pmatrix} ka_{11} & ka_{12} & \cdots & ka_{1n} \\ ka_{21} & ka_{22} & \cdots & ka_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ ka_{m1} & ka_{m2} & \cdots & ka_{mn} \end{pmatrix} \quad (6.2)$$

图 6.7: 矩阵 D

```
public INDArray mul(Number n) {
    validateNumericalArray("mul", false);
    return muli(n, Nd4j.createUninitialized(Shape.pickPairwiseDataType
(this.dataType(), n), this.shape(), this.ordering()));}
```

$$AB = \begin{pmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n b_1 & a_n b_2 & \cdots & a_n b_n \end{pmatrix}$$

图 6.8: 矩阵 E

```
public INDArray mmul(INDArray other) {
    Preconditions.checkState(this.dataType() == other.dataType(), "Matrix_
multiplication:
arrays must have same dtype: %s vs. %s", this.dataType(), other.
dataType());
    long[] shape = {rows(), other.rank() == 1 ? 1 : other.columns()};
    INDArray result = createUninitialized(this.dataType(), shape, 'f');
    if (result.isScalar())
        return Nd4j.scalar(Nd4j.getBlasWrapper().dot(this, other)).reshape(1, 1)
    ;
    return mmuli(other, result);
}
```

图 6.9: 矩阵乘法源码

6.1.3 特征值与特征向量

设 A 是一个 n 阶矩阵，如果存在数 λ 和非零列向量 α ，使得 $A\alpha = \lambda\alpha$ ，则称 λ 为矩阵 A 的一个特征值，称 α 为 A 属于特征值 λ 的一个特征向量。

代码块展示了简单的二维矩阵计算特征值与特征向量的方法。在此过程中，我们首先求出矩阵 A 的特征多项式，并求出 A 的所有特征值；然后再根据特征值求出齐次线性方程组 $(\lambda * I - A)x = 0$ 的基础解系，进而得出特征向量。

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad (6.3) \qquad \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{pmatrix} \quad (6.4)$$

图 6.10: 矩阵 A

图 6.11: 矩阵 B

```

public INDArray permute(int... rearrange) {

    checkArrangeArray(rearrange);
    int[] newShape = doPermuteSwap(shapeOf(), rearrange);
    int[] newStride = doPermuteSwap(strideOf(), rearrange);

    char newOrder = Shape.getOrder(newShape, newStride, 1);

    INDArray value = create(data(), newShape, newStride, offset(), newOrder
        );
    return value;
}

```

```

private static void getEigenValue(int[] A){
    int a = 1;
    int b = -(A[0] + A[3]);
    int c = A[0]* A[3] - A[1] * A[2];
    double t = b*b - 4 * (a * c);
    if (t >= 0) {
        double lambda1 = (Math.sqrt(t) - b) * 1.0 / (2 * a);
        double lambda2 = (-(Math.sqrt(t))-b) * 1.0 / (2 * a);
        int[] resultArray1 = {1, (int)((A[2])/(lambda1 - A[3]))};
        System.out.print("矩阵的第一个特征值为" + lambda1 + "，特征向量为" + "[" +
            resultArray1[0]
            + "," + resultArray1[1] + "]");
    }
}

```

6.2 雅克比矩阵

雅克比矩阵是一阶偏导数以一定方式排列组成的矩阵。在线性代数中，雅克比矩阵体现了一个可微方程与给定点的最优线性逼近，有着重要的意义。

雅克比矩阵的定义为假设存在 n 个 $y_n = f(x_1, x_2, \dots, x_n)$ 型函数。如果这些函数的偏导数存在，则可组成一个 m 行 n 列的矩阵，这个矩阵就是雅克比矩阵，用符号表示为 $J_F(x_1, \dots, x_n)$ ，如图6.12所示。

$$J_F(x_1, \dots, x_n) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \quad (6.5)$$

图 6.12: 雅克比矩阵

6.3 Hessian 矩阵

Hessian 矩阵又称黑塞矩阵，海瑟矩阵。与雅克比矩阵相对应，是多元函数的二阶偏导数构成的方阵，描述了函数的局部曲率情况。

Hessian 矩阵的定义为设有一个 n 元实函数 $f(x_1, x_2, \dots, x_n)$ 在定义域内有二阶连续偏

导，这些二阶偏导组成的矩阵被称为 Hessian 矩阵，如下图所示。

$$\begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix} \quad (6.6)$$

图 6.13: Hessian 矩阵

6.4 泰勒展开和牛顿法

在数学中，泰勒展开式主要指利用函数在一点的各阶导数值，来构建一个多项式近似函数值，同时还能给出近似值与实际值的误差。如果一个函数在点 x_0 处可以计算 n 阶导数，则我们有 Taylor 展开：

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n + R_n(x)$$

其中 $R_n(x)$ 称为拉格朗日余项，值为 $R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}$ ，当 $x_0 = 0$ 时，我们能得到 Taylor 的麦克劳林公式：

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + o(x^n)$$

6.5 向量与矩阵求导

利用向量和矩阵求导可以让原本有许多变量的单个函数的偏导值收集到可以视为单个个体的矩阵和向量当中，这大大简化了我们寻找多元函数的最大值或最小值，微分方程组操作等问题。

在上面的两节里，我们介绍了雅克比矩阵和 Hessian 矩阵，它们的本质都是特殊的矩阵求导。在这里我们将给出一般的形式。对于一个向量 $y = (y_1, y_2, \dots, y_m)$ 对 x 求导，我们可以写成 $\frac{\partial y}{\partial x} = \left(\frac{\partial y_1}{\partial x}, \frac{\partial y_2}{\partial x}, \dots, \frac{\partial y_m}{\partial x} \right)$ 。在求解梯度下降的加速度问题时，向量求导能比较好地解决问题。矩阵的求导与雅克比矩阵的形式十分类似，都是对矩阵中的元素逐个求导。

同理，矩阵的求导也与此类似，设有函数矩阵 Y ，其对于 x 的求导结果可以表示为：

$$J_F(x_1, \dots, x_n) = \begin{pmatrix} \frac{\partial y_{11}}{\partial x} & \cdots & \frac{\partial y_{1n}}{\partial x} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_{m1}}{\partial x} & \cdots & \frac{\partial y_{mn}}{\partial x} \end{pmatrix} \quad (6.7)$$

图 6.14: 矩阵的求导

第 7 章 数学基础-DEMO

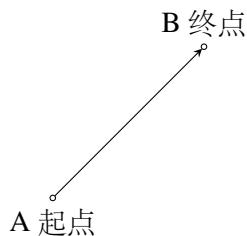
AI 实在是一个多学科综合性应用，涉及到的理论不仅多而且深。限于笔者知识水平以及篇幅要求，不会对此进行长篇巨幅地解释。

7.1 向量运算

单独的一个数字，称为标量（scalar），而向量通常用数组的形式表示。一个向量就是一列（行）数字：

$$x = [x_1 x_2 \dots x_n], x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad (7.1)$$

向量是一种带方向指示性的量，代表空间中的一个点。一维向量 $\vec{a} = [4]$ 代表 a 点在原点右侧距离为 4 的位置。而二维向量 $\vec{b} = [3 4]$ 就代表了在第一象限坐标 (3,4) 有一个点 b。所以，向量是一个方向性的偏移量。向量可以在坐标系中自由平移，选定一个起点就确定了它的终点。



向量支持的运算有内积（点乘）和外积（叉乘）运算，以及加减运算。向量 A、B 的运算过程，设 $\vec{A} = (a_1, a_2, a_3), \vec{B} = (b_1, b_2, b_3)$

1. 点乘，结果是一个标量： $A \cdot B = a_1 * b_1 + a_2 * b_2 + a_3 * b_3$
2. 叉乘，结果还是个向量： $A \times B = (a_2 * b_3 - a_3 * b_2, a_3 * b_1 - a_1 * b_3, a_1 * b_2 - a_2 * b_1)$
3. 标量，用于每一个元素： $A + 2 = (a_1 + 2, a_2 + 2, a_3 + 2)$

通常使用数组来表示向量，但这样扩展性不够好。咱们使用 Java 面向对象的方法，把数组和函数封装进一个向量类型里面去。定义一个 Vector 类，如图 7.1：

```
public IntVector(int... array) {
    this.array = array;
}

public IntVector add(IntVector vector) {
    foreach(this.array, vector.array, (a,b)->a+b);
```

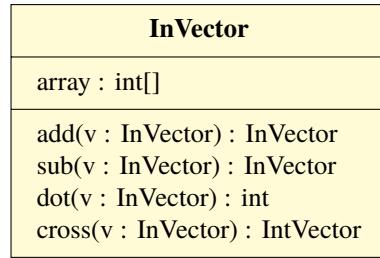


图 7.1: 类图: IntVector

```

    return this;
}

public int dot(IntVector vector) {
    return foreach(this.array, vector.array, (a,b)->a*b);
}

```

如上代码, 构造函数用到了可变参数, 你可以理解成一个int[]。在编译的时候, Java会自动地把函数声明和调用代码都转换成数组。*add* 运算返回this, 用来支持链式运算: v1.add(v2).add(v3)...。容易看出, *add* 和 *dot* 运算, 是对应元素相加、相乘的过程。使用foreach 从相加的 2 个向量中, 逐个地取出数据并执行 λ 操作。

Listing 7.1: 叉乘运算

```

public IntVector cross(IntVector vector) {
    int[] v1 = this.array;
    int[] v2 = vector.array;

    int length = vector.array.length;
    if (length == 2) {
        return new IntVector(v1[0]*v2[1]-v1[1]*v2[0]);
    }

    int[] result = IntStream.range(0, length).map(
        i->v1[(i+1)%length]*v2[(i+2)%length] -v1[(i+2)%length]*v2[(i+1)%length]
    ).toArray();

    return new IntVector(result);
}

// sum用于dot运算
private static int foreach(int[] src, int[] dst, IntBinaryOperator op) {
    int sum = 0;
    for (int i = 0; i < src.length; i++) {
        src[i] = op.applyAsInt(src[i], dst[i]);
        sum += src[i];
    }
}

```

```
    }  
    return sum;  
}
```

以上，仅仅只能处理 int 类型的向量。结合 IntVector 的实现代码，相信大家也能理解实数类型的向量。也许你也意识到，使用 add 太麻烦了，但 Java 目前还不支持运算符重载。如果确实想使用，可以借助 Java-OO 开源插件¹

Listing 7.2: 运算符重载

```
@Test  
public void add_operator_override() {  
    IntVector a = new IntVector(1,2,3);  
    IntVector b = new IntVector(10,20,30);  
  
    IntVector c = a + b; // 看上去运算符重载了  
    int[] expected = new int[]{11, 22, 33};  
  
    assertArrayEquals(expected, c.array);  
    assertEquals(new IntVector(expected), c);  
}
```

实际上，很少有自己实现的，并且也较难保障准确性和稳定性。开源的 ND4J 作为 DL4J 的张量计算库，提供了丰富的运算接口，支持几乎所有的数学操作，相当于 Python 界的 numpy。以下代码 7.3

Listing 7.3: dl4j 例子

```
INDArray vec1 = Nd4j.create(new float[]{1,2,3});  
  
// vector add  
INDArray vec2 = Nd4j.create(new float[]{10,20,30});  
  
// [[11.0000, 22.0000, 33.0000]]  
System.out.println(vec1.add(vec2));  
  
// scalar add  
INDArray vec3 = vec1.add(10);  
  
// [[11.0000, 12.0000, 13.0000]]  
System.out.println(vec3);
```

ND4J 在开源、分布式、支持 GPU 的库内，为 Java 带来了符合直觉的，类似 Python 编程人员所用的科学计算工具。它具有：

¹一种使用 APT 在编译过程中替换运算符为相应函数的方法。

1. 多用途多维数组对象
2. 多平台功能，包括 GPU
3. 线性代数和信号处理功能

本节之后都将使用 ND4J 来演示算法，有兴趣的同学可以研究它的实现代码。

7.2 矩阵运算

矩阵是一个 $m \times n$ 个数组成的 m 行 n 列的矩形表格。特别地，向量可以看作矩阵的特殊形式。对于 $1 \times n$ 或 $n \times 1$ 的矩阵，就是一个行向量或列向量。实际上，矩阵的加法/减法运算，也可适用于向量。尽管矩阵是多维度数据，却很少采用多维数组表示。如果向量和矩阵都使用一维数组表示的话，*add* 运算当然可以用与矩阵。

对于这样一个数组：int [] numbers={1,2,3,4,5,6}。

1. 可能是一个 1×6 的矩阵，或是一个向量。
2. 可能是一个 2×3 的矩阵
3. 可能是一个 3×2 的矩阵

IntNDArray
array : int[] shape : int[]
add(v : IntNDArray) : IntNDArray mul(v : IntNDArray) : IntNDArray rank() : int isVector() : boolean

图 7.2: 类图：IntNDArray

为了让 IntVector 升级为 IntNDArray，势必要加入维度信息才行，记为 *shape*。而矩阵只有 2 个维度，如果不考虑更多维度的情况下，使用 int [2] 就可以。但你立即就会认识到，改成 int [] 有更好的可扩展性。使用 IntNDArray 创建向量就是创建 $1 \times n$ 或 $n \times 1$ 的矩阵。

Listing 7.4: 创建 NDArry

```
int[] array;
int[] shape;

// 默认: 1 x n
public IntNDArray(int... array) {
    this.array = array;
    this.shape = new int[]{1, array.length};
}

// 可以把1x6转换成2x3或者3X2的矩阵
```

```
public IntNDArray reshape(int[] shape) {
    this.shape = shape;
    return this;
}
```

根据数学定义，只有当矩阵 A 的列数等于矩阵 B 的行数时，A 与 B 才可以相乘。乘积 C 的第 m 行第 n 列的元素等于矩阵 A 的第 m 行的元素与矩阵 B 的第 n 列对应元素乘积之和。

$$c_{ij} = \sum_{k=1}^p a_{ik} * b_{kj} \quad (7.2)$$

其中，A 为 $m \times p$ 的矩阵，B 为 $p \times n$ ，结果为 mxn 的矩阵 C。相应地，实现算法如下：

Listing 7.5: 矩阵乘法

```
public IntNDArray mul(IntNDArray ndArray) {
    final int ROWS = this.shape[0];
    final int COLS = ndArray.shape[1];

    int[] shape = new int[]{ROWS, COLS};
    int[] data = new int[ROWS*COLS];

    for(int i=0; i<ROWS; i++) {
        for (int j=0; j<COLS; j++) {
            for (int k=0; k<ndArray.shape[0]; k++) {
                int a_i_k = this.array[i*this.shape[1]+k];
                int b_k_j = ndArray.array[k*ndArray.shape[1]+j];
                data[i*COLS+j] += a_i_k * b_k_j;
            }
        }
    }
    return new IntNDArray(data).reshape(shape);
}
```

现在，我们就可以用一维数组记录矩阵了，不管维度如何变化 `reshape` 之后不需要变更 `array` 的内容。对于向量的点乘和叉乘，也可以借助矩阵处理。点乘比较容易解决，但叉乘需要反对称矩阵（anti-symmetric）辅助才能计算出结果。不过叉乘主要应用于几何和动量计算，在此就不再详细解释。

以下代码是 ND4J 的矩阵示例：

```
INDArray v1 = Nd4j.create(new float[]{1,2,3}).reshape(1,3);
INDArray v2 = Nd4j.create(new float[]{10,20,30}).reshape(3,1);
System.out.printf("v1v2=[%s]\n", v1.mmul(v2));
```



7.3 梯度下降

在机器学习里，梯度下降虽然不是什么高深的算法，但它却是机器学习的关键。经常会用到梯度下降法来进行训练，常见的有：批量梯度下降法 BGD，随机梯度下降法 SGD，小批量梯度下降法 MBGD。

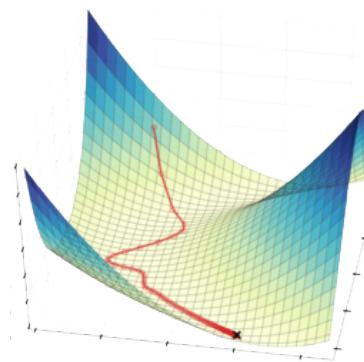


图 7.3: 梯度下降

梯度是方向上升/下降最快的方向，它的幅值代表陡峭的程度。所以，在最小值的地方，曲面轮廓几乎是平坦的，理论上最小值是梯度为 0。但事实上，我们又可能无法达到最小值，只能在最小值附近的平坦区域来回震荡。当在这个区域震荡时，损失（Loss）值几乎是我们能达到的最小值了，并且不会有很大的变化，因此我们是在真实的最小值附近跳动。通常，当损失值在预定的数字内没有改善的时候就会停止迭代，例如 10 次或者 20 次迭代。当这种情况发生时，就说训练已经收敛了，或者说收敛已经完成了。

7.3.1 BGD

使用 BGD(Batch Gradient Descent, 批量梯度下降)，在目标函数为凸函数时，虽然可以找到全局最优解，但收敛速度慢，需要用到全部数据，因此内存消耗也大。因此，BGD 不适用于大数据集。其公式如下：

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

如公式所示，BGD 的策略就是朝着当前所在位置的坡度最大的方向前进。但它也有缺点，它在面对峡谷型函数时，效率会变得很低，呈现出震荡的姿态。

7.3.2 SGD

而 SGD 相当于 BGD 的升级版。SGD 被称为随机梯度下降 (Stochastic Gradient Descent)。正如它的名字所说，它首先通过 mini-batch 学习，意思就是从训练数据中随机选择一部分数据 (称为 mini-batch)，将这些 mini-batch 作为对象，使用梯度法进行学习。其代码如下所示：

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .weightInit(WeightInit.XAVIER)
    .activation("relu")
```

```
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.updater(new Sgd(0.05))
// ... other hyperparameters
.list()
.backprop(true)
.build();
```

7.3.3 Momentum

Momentum 也是一种常见的优化算法, 也被称为 SGD with Momentum。恰如其名, 它为了抑制 SGD 的震荡, 在梯度下降过程中加入惯性。简单来说, 它就是在梯度越陡时, 下降越快。较平缓时, 下降较慢。其公式如下:

$$\nu \leftarrow \alpha \nu - \eta \frac{\partial L}{\partial W}$$

$$W \leftarrow W + \nu$$

如上式所示, W 表示要更新的权重参数, $\frac{\partial L}{\partial W}$ 表示损失函数关于 W 的梯度, η 表示学习率。而 ν 这一变量便是用来模拟惯性的。其通过在 SGD 的基础上引入一阶动量, 这代表着现在下降方向不仅由当前点的梯度方向决定, 而且由此前累积的下降方向决定。

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
.weightInit(WeightInit.XAVIER)
.activation("relu")
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.updater(new Nesterovs(0.05))
// ... other hyperparameters
.list()
.backprop(true)
.build();
```

Momentum 的更新路径就如同小球在碗中滚动一样。与 SGD 相比, 其大大缓解了震荡问题, 原因就是它加入的一阶动量。即使在某一方向“受力”很小, 但因为其一直在同一方向受力, 所以它会朝着同一方向有一定的加速。通俗地讲, 就是它所加入的惯性, 抵消了试图改变它的力。

7.3.4 AdaGrad

在梯度下降中, 学习率的值很重要 (记为 η), 而在学习率的相关研究中, 有一种被称为 **学习率衰减** (learning rate decay) 的方法, 随着机器学习的过程, 使学习率逐渐减小。它具体表现为, 在一开始与其他方法类似, 进行参数学习, 但在学习的过程中, 当准确率



越来越高时，便减少学习率。其公式如下：

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial \mathcal{L}}{\partial \mathbf{W}} \odot \frac{\partial \mathcal{L}}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial \mathcal{L}}{\partial \mathbf{W}}$$

如公式所示，AdaGrad 会为参数的每个元素适当地调整学习率，与此同时进行参数的学习。AdaGrad 的公式比起之前的，新出现了一个变量 \mathbf{h} ，它代表着之前所有梯度值的平方和，在更新参数时，通过乘以 $\frac{1}{\sqrt{\mathbf{h}}}$ ，AdaGrad 便可以为每个元素调整适宜它的学习率。其代码如下所示：

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .weightInit(WeightInit.XAVIER)
    .activation("relu")
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .updater(new AdaGrad(0.05))
    // ... other hyperparameters
    .list()
    .backprop(true)
    .build();
```

AdaGrad 比起 Momentum 更好地抑制了 SGD 的震荡，函数的取值高效地向着最小值移动。刚开始时，也许还会有些震荡，但越接近最小值，几乎是呈直线般向着目标前进。

7.3.5 Adam

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .weightInit(WeightInit.XAVIER)
    .activation("relu")
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .updater(new Adam(0.05))
    // ... other hyperparameters
    .list()
    .backprop(true)
    .build();
```

7.4 概率分布

7.5 损失函数

在机器学习中，机器的学习需要某个指标来表示现在的状态，然后，以这个指标为基准，寻找最优权重参数。而在机器学习中所用的指标便被称为损失函数（loss function）。对于损失函数，我们主要使用均方误差和交叉熵误差。



首先，我们介绍一种常用的函数，交叉熵误差（cross entropy error）。

$$E = - \sum_k t_k \log y_k \quad (7.3)$$

在上式中， y_k 代表着机器学习的输出， \log 表示以 e 为底数的自然对数 (\log_e)。 t_k 代表着正确解标签，仅当解标签为正确时， t_k 的索引才为 1。其余情况都为 0。因此，E 所代表的实际为解标签为正确时所输出的自然对数。

如上图所示，当输出结果 y_k 越发趋向于 1.0 时，得到的 E 越发增大而趋向于 0。交叉熵误差通过 E 所表示的负值，来表明该权重参数与正确结果之间的偏差程度。

下面，我们用代码来实现交叉熵误差：

Listing 7.6: 交叉熵误差

下面我们介绍另一种，在损失函数中最有名的均方误差（mean squared error）。它的公式如下所示：

$$E = \frac{1}{2} \sum_k (y_k - t_k)^2 \quad (7.4)$$

在上式中 k 表示着数据处于哪一个维度， t_k 表示着各维度的监督数据。通常情况下，监督数据仅将正确解标签表示为 1，而其他非正确解则表示为 0。而机器学习输出的结果 y_k ，则是会在各个维度都显示该维度是正确解标签的可能性。均方误差通过计算机器学习的输出和正确解监督数据的各个元素之差的平方，再求总和。从而得到该权重参数的输出结果与正确结果之间的偏差。

Listing 7.7: 均方误差

7.6 拟合算法

很多时候，我们希望通过一些样本来反应总体的特征，因此我们需要拟合曲线来判断总体的情况。假设有如下这些个样本，看起来各点分布趋于一条直线，因此我们希望通过一条直线来描述该样本所在总体的一些特征，对总体进行预测。一般的方法就是先假设一条直线，如 $L=ax+b$ ，之后再根据前面这些样本，确定最优的 a,b，所谓最优就是通过这些点计算出合适的 a,b，使各个点对直线上垂直距离的平方和最小（最小二乘法）。具体的方法是通过迭代来计算的。

第8章 机器学习

8.1 分类问题

分类问题就是将数据以一定的分类标准分为几簇，在本节中，我们将介绍几种分类方法。第一种分类方法是 SVM (Support Vector Machine, 支持向量机)，是机器学习中经典的算法。我们以简单的逻辑分类为例，如图8.1所示，我们需要找到一条直线，这样就能将所有的数据分为两类。

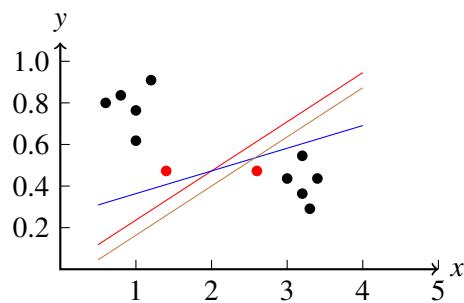


图 8.1：支持向量机示意图

如果不加以限制，这样的分类直线将有无数条。为了在众多的直线中找到最优的分类直线，我们要遵循间隔最大化原则，即分类直线离数据越远就是最优的，因为如果不是这样，分类直线会紧贴边界的数据；这时如果出现一个与边界数据略有不同的新数据，就很容易被分到错误的一边。同时，过于接近分类直线的数据点不具有太大的实际意义。想要达到分类直线离数据最远的效果，我们需要找到离分类直线最近的数据点，用它们与直线的距离来训练神经网络。这些数据点被称为支持向量，这也是支持向量机名字的由来。

还是以图8.1为例，图中两个红色的点为支持向量，也就是离分类直线最近的两个点。蓝线因为离两个支持向量都不是最远，所以不是最佳分类直线；棕线离上方的支持向量足够远，然而最佳分类直线应与所有支持向量保持最远距离，即所有支持向量离直线等距且最远，所以棕线也不符合要求。红线符合上述所有要求，时这个数据集的最佳分类直线。

分类问题的第二种分类方法是 RBM (Restricted Boltzmann Machine, 受限波兹曼机)。RBM 是一种无向图模型，具有两层结构：可见层 (V 层)，隐藏层 (H 层)，这两层的节点相互全部连接，但是每一层各自的神经元之间没有连接，因此被称为受限波兹曼机。波兹曼机是允许同一层神经元连接的。

另外，受限波兹曼机是二值化的，也就是说，其神经元的输出只有两种状态：激活和未激活，一般情况下我们分别用 1 和 0 表示。在图8.2所示的例子中， n_v 和 n_h 分别表示可见层和隐藏层的神经元数目；前面我们提到，受限波兹曼机的神经元只有激活和

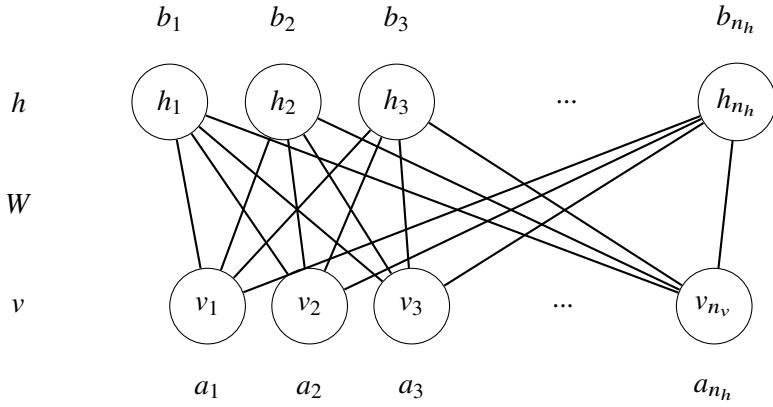


图 8.2: 受限波兹曼机模型结构

未激活两种状态，因此，我们可以用向量 \mathbf{v} 来表示可见层的状态向量，具体表达式为 $\mathbf{v} = (v_1, v_2, \dots, v_{n_v})^T$ ，其中 v_i 表示可见层中第 i 个神经元的状态。同理，隐藏层的状态向量也可以表示为 $\mathbf{h} = (h_1, h_2, \dots, h_{n_h})^T$ ，其中 h_i 表示可见层中第 i 个神经元的状态。可见层的偏置向量用 \mathbf{a} 来表示，具体表达式为 $\mathbf{a} = (a_1, a_2, \dots, a_{n_v})^T$ ，同样的，隐藏层的偏置向量为 \mathbf{b} ，表达式为 $\mathbf{b} = (b_1, b_2, \dots, b_{n_h})^T$ 。

最后，连接隐藏层和可见层的权值矩阵用 W 来表示，表达式为 $W = (w_{i,j})$ ， $w_{i,j}$ 代表隐藏层中第 i 个神经元与可见层中第 j 个神经元之间的连接权重。

有了这些前置参数，我们便可以探索受限波兹曼机的更多特性和用途。受限波兹曼机是一个能量模型 (Energy Based Model, EBM)，是由物理学能量模型演变而来；能量模型需要先定义一个合适的能量函数，然后基于这个能量函数得到变量的概率分布，最后基于概率分布去求解一个目标函数。受限波兹曼机的能量函数定义为：

$$E_\theta(\mathbf{v}, \mathbf{h}) = -\sum_{i=1}^{n_v} a_i v_i - \sum_{j=1}^{n_h} b_j h_j - \sum_{i=1}^{n_v} \sum_{j=1}^{n_h} h_j w_{j,i} v_i \quad (8.1)$$

如果写成矩阵的形式，则可改写为

$$E_\theta(\mathbf{v}, \mathbf{h}) = -\mathbf{a}^T \mathbf{v} - \mathbf{b}^T \mathbf{h} - \mathbf{h}^T W \mathbf{v} \quad (8.2)$$

相比于波兹曼机 (BM)，受限波兹曼机 (RBM) 因其具有快速学习的特点而被广泛地使用。与此同时，RBM 在分类、回归和图像特征提取上也得到了广泛的应用。

8.2 线性回归

线性回归是机器学习解决预测问题的重要途径之一。在进行线性回归算法时，我们通常会指定一个数据集，其中包括了正确答案。线性回归算法会根据给出的数据集去拟合一个函数，使之尽量符合所给出的数据集，同时达到预测更多正确答案的效果。（如图8.3所示）这种模式类似于老师给定多个参数和结果，让学生自己寻找函数并且不断地自我修正；同时老师也会监督并量化期望与实际输出间的误差。因此，线性回归算法也

被称为监督学习。

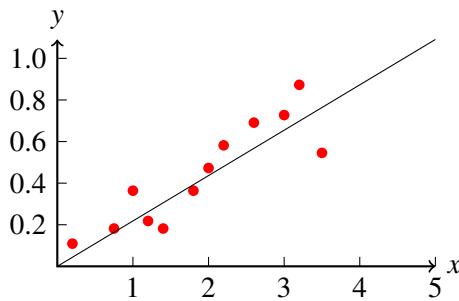


图 8.3: 根据所给的数据集拟合函数，并根据此函数预测将来的数值

如图8.3所示，自变量 x 会与要预测的目标变量 y 建立一个函数关系。为了使结果更精确，我们可以指定特定的模型，决定是使用一次函数还是二次函数来更好地贴合数据集，从而达到更好的预测效果。

在 DL4J 中，线性回归算法主要有三个部分组成，模型的建立，过程的监听和数据集的构造。建立模型需要一些超参数。在下面的代码块中指定了一些超参数：第一是随机种子，由于每次进行神经网络训练时，函数的偏置和权重都是随机的，我们需要用种子来确保初始情况大致一致，使结果具有可复现性。第二是 OptimizationAlgorithm 和 updater，代表了模型的优化算法，分别决定了学习的方向和学习率的大小。第三是对模型权重的初始化。第四是对神经元的设置，第六是设置是否进行反向传播。

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_
    GRADIENT_DESCENT)
    .weightInit(WeightInit.XAVIER)
    .updater(new Sgd(learningRate))
    .layer(0, new OutputLayer.Builder(LossFunctions.LossFunction.MSE)
    .backprop(true)
    .build
```

在进行训练的同时我们也要对模型进行监听，主要用于对函数的损失函数进行监听，从而更好地对算法进行改进。

最后是数据集的构造。我们说过线性回归的数据集包括数据和结果两个部分。在这里，我们把自变量 x 称为特征，因变量 y 称为标签，首先先创建两个长度为样本数量的数据组，分别代表输入和输出数据的数据组；接着根据特征的值随机生成输入和输出的数，最后将其包装成一个类返回给模型，代码如图8.4所示。

当然，这里的线性回归只含有一个自变量和一个因变量，因此被称为一次线性回归或单变量线性回归。这种模型比较简单，用二维的 xy 坐标轴即可表示出来。在实际的操作中还会遇到还有多变量线性回归，在面对更多维度的参数能更好地进行处理。

```

private static DataSetIterator getTrainingData(int batchSize, Random rand) {
    double [] output = new double[SampleNum];
    double [] input = new double[SampleNum];
    for (int i= 0; i< SampleNum; i++) {
        input[i] = MIN_RANGE + (MAX_RANGE - MIN_RANGE) * rand.nextDouble();
        output[i] = 0.5 * input[i] + 0.1;
    }
    INDArray inputNDArray = Nd4j.create(input, new int[]{nSamples,1});

    INDArray outPut = Nd4j.create(output, new int[]{nSamples, 1});
    DataSet dataSet = new DataSet(inputNDArray, outPut);
    List<DataSet> listDs = dataSet.asList();
    return new ListDataSetIterator(listDs,batchSize);
}

```

图 8.4: 数据集的构造: 包括特征和标签

8.2.1 过拟合

对于机器学习来说, 过拟合是一个十分常见的问题。过拟合具体是指经过机器学习所获得的权重参数, 可以拟合训练数据, 却在面对不包含在训练数据中的实际数据时, 不能很好的拟合他们。机器学习的目的就是获得一个更加抽象, 以及泛化的模型。即使是对未包含在训练数据中的数据, 我们也希望能获得较好的拟合结果。所以我们需要通过一定的方法来抑制过拟合。

过拟合产生的原因, 主要有两点: 1. 模型具有大量的参数, 表现力强 2. 训练数据的数量很少

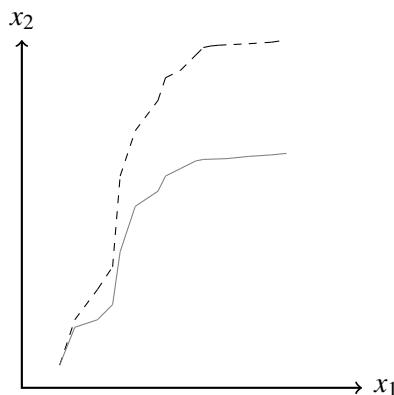


图 8.5: 过拟合情况

如图8.5所示, 对于训练用数据来说, 大约 100 个 epoch 之后, 精度便可以达到接近 100%。但是, 对于测试用数据来说, 识别精度确只能达到 70% 左右, 这并非我们所想要的结果。从上面的分析中可知, 模型对训练时没有使用的数据拟合得并不好。

8.2.2 正则化

过拟合作为机器学习中一种常见的问题, 自然也有解决的方法。基于本书主要介绍关于入门方面的知识, 在此便不深入介绍。仅简单介绍一下权值衰减这一方法。

权值衰减是一种常见的用来抑制过拟合的方法。其通过在学习过程中对大的权重进行一定比例的惩罚，从而达到过拟合的目的。过拟合往往就是因为权重参数取值过大才发生的。因此权值衰减不失为一种简单而有效的办法。

实现权值衰减具体方法是为损失函数加上权重的平方范数（L2 范数）。机器学习的目的是减少损失函数的值，使用权值衰减后，便可以对变大的权重进行惩罚，从而达到抑制的作用。具体来说，如果将权重记为 \mathbf{W} ，L2 范数的值便是 $\frac{1}{2}\lambda\mathbf{W}^2$ 。权值衰减便是将 L2 范数加到损失函数中。在 L2 范数中， λ 是控制正则化程度的参数。 λ 越大，对数值较大的权重施加的惩罚也就越重。下面我们用图片来具体解释：

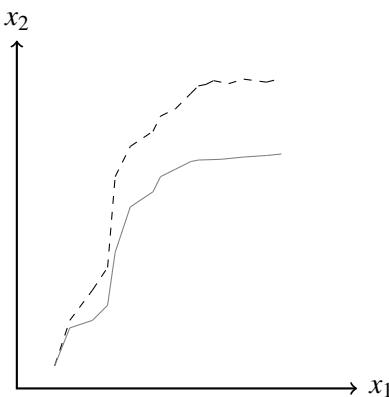


图 8.6：权值衰减方法

如上图所示，测试数据的识别精度没有太大变化。但训练数据和测试数据的识别精度之间的差距明显缩小了。这证明，过拟合收到了抑制，而训练数据和测试数据的整体识别精度的增加需要依靠其他的方法，并非权值衰减的作用范围。

8.3 逻辑回归

逻辑回归与线性回归相似，同样用于处理数据的分类问题。他们的不同之处在于逻辑回归的数据集中只包含数据，神经网络会自我学习，自行生成一套数据结构模式。在这种模式中，我们只需要提供数据，不需要提供数据集地的正确答案，其余部分均由神经网络完成对数据的分类。因此，逻辑回归也被称为无监督学习。

在下图的例子中，我们给出一个数据集（如图8.7）。模型并不知道这些数据的意义和用途，只会尝试着找出其中潜藏的数据结构。在这个例子中，逻辑回归算法会判定这个数据集从属于两个数据簇 a , b ，从而完成数据分类的任务。

在 DL4J 中，逻辑回归与线性回归配置类似，主要区别在于神经网络的最后一层输出层：在线性回归中，最后一个隐藏层是 Activation.IDENTITY，即不对数据作处理。而逻辑回归的最后一层为 Activation.SOFTMAX，指定了对当前数据进行分类任务。代码如下：

8.4 正向传播

正向传播也叫前向传播，是指数据从输入层到隐藏层，再到输出层的单向过程。数据会从输入层输入，经过隐藏层的一层层地变换，最终得出结果数据。如图8.8是一个典型的三层神经网络，我们便以此为例解释正向传播。

隐藏层 H_1 的值由输入层的值和权重所决定， $W_{(1,1)}$, $W_{(2,1)}$, $W_{(3,1)}$ 分别代表输入数据 I_1 , I_2 , I_3 对于隐藏层 H_1 的权重。公式如下：

$$H_1 = I_1 W_{(1,1)} + I_2 W_{(2,1)} + I_3 W_{(3,1)}$$

同理， H_2 也可表达为相似的公式。经由隐藏层激活函数计算后得到该层的激活值。同时，输出层的值由隐藏层的值和权值决定， W_1 , W_2 分别代表各个隐藏层数据对于输出层的权重。 b_1 表示偏置，公式如下：

$$O_1 = H_1 W_1 + H_2 W_2 + b_1$$

这样一个由输入层至隐藏层，再由隐藏层至输出层的过程就被称为正向传播。如图8.8所示，只含有正向传播的神经网络可以用一个有向无环图来表示。只含有正向传播的神经网络被称为前馈神经网络，是现今发展较为成熟的一种神经网络。

8.5 反向传播

在正向传播中，神经网络中的数据单向向前传播；因此，我们无法准确知道每一步造成的误差大小，也无法根据误差对神经网络进行调整更新；反向传播就提供了另一种

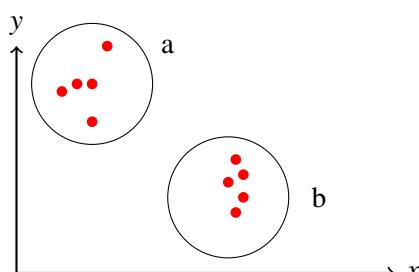
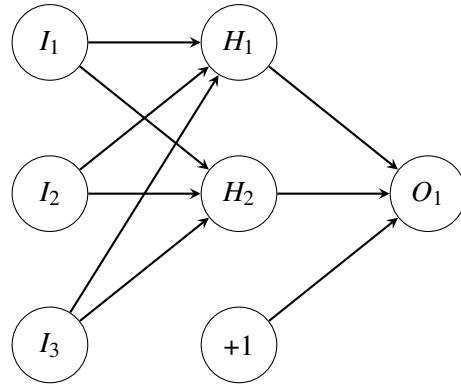


图 8.7：逻辑回归注重数据的分类

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .updater(new Nesterovs(learningRate, 0.9))
    .layer(0, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.SOFTMAX)
        .build())
    .build();
```



输入层 隐藏层 输出层

图 8.8: 经典的三层神经网络

途径来量化误差并更新权值。反向传播用于将代价函数最小化，将每一层的误差反向传播给上一层，由此将误差最小化，如图8.9。

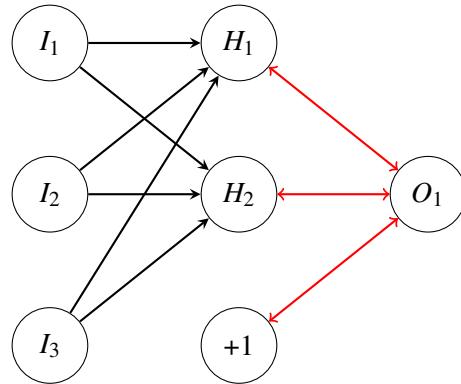


图 8.9: 具有反向传播的神经网络

为了计算每一层神经元所产生的误差，我们引入 $\delta_j^{(l)}$ ，代表第 l 层的第 j 个节点所产生的误差。由此可得最后一层输出层的误差公式为 $\delta_j^{(3)} = a_j^{(3)} - y_j$ ，其中 $a_j^{(3)}$ 代表神经网络计算出的值，而 y_j 代表数据对应的真实值。

误差向前一层传播时，公式需更改为 $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} * g'(z^{(2)})$ ， $(\Theta^{(n)})^T$ 代表第 n 层代价函数的转置， $z^{(n)}$ 等于 $\Theta^{(n-1)} a^{(n-1)}$ ，最后 $g(z)$ 代表该层的激活函数。依据这个公式，我们就而可以将误差逐层上传，并实现对神经网络参数的修改。

反向传播实际上与正向传播很相似。他们的计算方法类似，区别在于正向传播是从前向后计算，而反向传播是从后向前计算。其实， $\delta_j^{(l)}$ 即是该层代价函数关于该层对应的所有输入单元加权和的偏导。以图8.9为例， $\delta_1^{(3)}$ 可表达为 $\delta_1^{(3)} = \frac{\partial}{\partial z_1^{(3)}} \text{cost}(i)$ ，其中 $\text{cost}(i)$ 代表该层的代价函数， $z_1^{(3)}$ 等于 $a_1^{(2)} * W_{(1,2)} + a_2^{(2)} * W_{(2,2)}$ 。 $\delta_j^{(l)}$ 衡量了神经网络计算中的中间值，代表了神经网络权重改变的程度大小，使我们对整个网络有更深入的了解。

反向传播只是正向传播的逆向过程。我们还是以图8.9为例，由上可知 $\delta_1^{(3)} = a_1^{(3)} - y_1$ ，而逆向到 $\delta_2^{(2)}$ 的算式可表达为 $\delta_2^{(2)} = (W_{(2,2)})^{-1} * \delta_1^{(3)}$ 。这样的逆过程与正向传播的计算过

程区别不大；所以说，反向传播与正向传播在形式和计算方法上都很相似。

反向传播广泛地应用于循环神经网络，例如机器翻译、语言理解等前沿应用。这种结构的网络往往存在记忆单元和注意力机制，要求复杂的求导计算，例如 Google 语音助手的上下文理解功能。反向传播促进了记忆神经网络的发展，目前正在探索和发展中。

8.6 全连接网络

全连接层网络（MLP, Multilayer Perceptron）又叫多层感知机，特点是上一层的所有神经元都与下一层的所有神经元相连接，因此又叫全连接层。如图8.8就是一个简单的全连接层，最底层是输入层，中间是隐藏层，最后是输出层。在计算第 n 层的某个节点的时候，输入的参数是第 $n-1$ 层的所有节点的加权。

从图8.10可以看出，全连接网络的所有参数就是各个层的权重和偏置。确定并最优化这些参数就是一个最优化的问题，最简单的就是利用梯度下降（SGD），首先随机初始化所有参数，然后进行迭代训练，不断地更新梯度和参数，直到满足某个条件为止。全连接网络应用广泛，例如 MNIST 手写数字识别等。

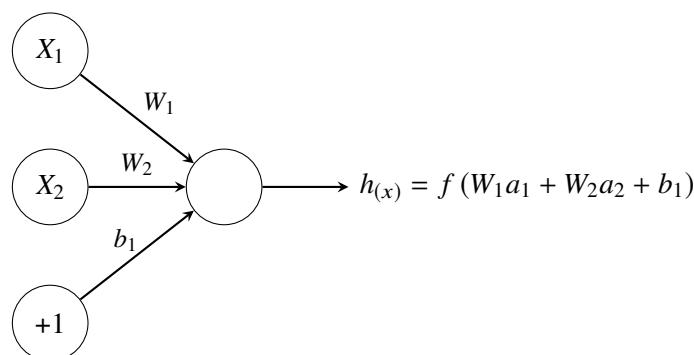


图 8.10：全连接层中的一个节点

第9章 有监督学习

机器学习分为：监督学习，无监督学习。本章先介绍监督学习（supervised learning），它从给定的训练数据集中学习出一个函数，用于对新数据的预测。监督学习的训练集包括输入和输出，也即是特征和目标。训练集中的目标是由人标注的，利用这些已知数据和其输出训练得到一个最优模型（目标函数）。监督学习是训练神经网络和决策树的常见方法。最典型的算法是 KNN 和 SVM，用于回归分析和统计分类。

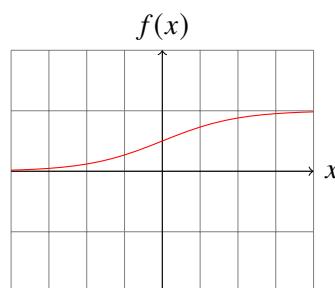
9.1 训练过程

有监督学习最常见的就是：回归（Regression）和分类（Classification）。一般来说，定量输出称为回归，或是对连续变量预测；定性输出称为分类，就是对离散变量进行预测。K-近邻（KNN）是一种分类算法，而之前提到的线性回归，就是利用已知数据集求线性函数，使其尽可能地拟合数据，也就是让损失函数最小。

9.2 回归问题

回归分析分为：线性回归、逻辑回归（Logistic Regression）。实际上，逻辑回归就是对线性回归的输出又进行了一次特殊函数的处理，使其输出一个分类可能性的概率，这个特殊的函数通常是 sigmoid 函数：

$$y = \frac{1}{1 + e^{-x}}$$



Sigmoid 函数是机器学习非常重要的一个函数，当 $x > 0$ 时，Sigmoid 函数大于 0.5；当 $x < 0$ 时，Sigmoid 函数小于 0.5。因此，把拟合曲线的函数值带入 Sigmoid 函数，比较 $f(x)$ 与 0.5 的大小就可判断其分类。总的来说，Sigmoid 函数具有以下几个良好性质：

1. 单调可微，具有对称性。
2. 便于求导。
3. 定义域为 $[-\infty, +\infty]$ ，值域为 $(0, 1)$ ，可将任意值映射为概率。

9.2.1 线性回归

之前章节介绍过线性回归的数学模型，并尝试用梯度下降法，找到线性函数的参数，来拟合数据集。所以实现线性回归的代码，得首先实现梯度的算法函数，分为：SGB、BGD 和 MBGD。

因为批梯度下降是使用所有的数据样本拟合出来的假设函数，所以计算梯度也就会涉及到所有的样本数据，这种情况就是所谓的 BGD (Batch Gradient Descent)。结合最小二乘法的定义，定义损失函数 $J(\theta) = \frac{1}{2} \sum_i^m (y_i - h_\theta(x^i))^2$ 现在问题就转化为求解最优的 θ ，使损失函数 $J(\theta)$ 取最小值。

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

我们先随便给 θ 一个初始化的值(多默认为 1)，然后改变 θ 值让 $J(\theta)$ 的取值变小，不断重复改变 θ 使 $J(\theta)$ 变小的过程直至 $J(\theta)$ 约等于最小值。此算法也称为最小均方算法 (Least mean square, LMS 算法)。

$$\theta_j := \theta_j + \frac{1}{\alpha} \sum_i^m (y_i - h_\theta(x^i)) x_j^i$$

使用伪代码表示如下，：

```
repeat{
     $\theta_j := \theta_j + \frac{1}{\alpha} \sum_i^m (y_i - h_\theta(x^i)) x_j^i$ 
    for every j=0, ... , n)
}
```

其中 α 称为步长 (learning rate)，控制 θ 每次向 $J(\theta)$ 变小的方向迭代的幅度。由以上伪代码，也容易发现 BGD(批梯度下降) 每次迭代都得把所有样本计算一次，计算量很大。把整个过程分解一下，包含：初始化、求偏导和梯度下降。

Java 代码，大致如下。

```
//  $h(x) = \theta_0 * x_0 + \theta_1 * x_1 + \theta_2 * x_2$ 
public class LinearRegression {
    public LinearRegression(double[][] data, double alpha, int iteration){}

    private void initialize_theta(){}
    public void train(){}
    private double[] partial_derivative(){}
    private double partial_derivative_of_theta(int j){}
    private double h_theta_x_i_minus_y_i_times_x_j_i(int i, int j){}
}
```

9.2.2 逻辑回归

Sigmoid 的其中一个优点就是容易求解导数，利于编码实现。推导过程略，相应的实现代码如下：

$$y = \frac{1}{1 + e^{-x}}, y' = y * (1 - y)$$

```
public static double sigMoid(double value) {
    double e_x = Math.pow(Math.E, -value);
    double result = 1 / (1 + e_x);
    return result;
}

public static double sigMoidDerivative(double value) {
    double A = sigMoid(value);
    double B = 1 - sigMoid(value);
    double result = A * B;
    return result;
}
```

9.3 SVM

在机器学习章节中，我们提到过利用 SVM（支持向量机）作为数据分类的一种方法，并提到了 SVM 的实质其实就是正确分类数据的直线，并且分类直线与支持向量的间隔较远。在这样的基础上，我们有必要对坐标系内点与直线的距离进行研究。假设 \mathbf{w}_i 为分类直线的函数， δ_i 代表坐标与分类直线的距离， $\|\mathbf{w}\|$ 代表向量 \mathbf{w} 的范数¹，由此我们可以得到一般性的点到直线距离公式：

$$\delta_i = \frac{1}{\|\mathbf{w}\|} |g(x_i)| \quad (9.1)$$

在二维坐标中，利用解析几何的知识，我们可以将公式简化为更容易理解的形式： $D = (Ax + By + c)/\sqrt{A^2 + B^2}$ ，其中 $Ax + By + c$ 就相当于 $g(x_i)$ ， $\sqrt{A^2 + B^2}$ 相当于 \mathbf{w} 的范数。这个公式的代码实现如下：

$$ax + by + c = 0$$

```
public double getDistance(double x, double y){
    double denominator = a * x + b * y + c;
    double numerator = Math.sqrt(a * a + b * b);
    double result = denominator / numerator;
    return result;
```

¹范数是对向量长度的度量，如果一个向量可以表示为 $\mathbf{W} = (w_1, w_2, w_3, \dots, w_n)$ ，那么该向量的范数则为 $\|\mathbf{w}\|_n = \sqrt[n]{m_1^P + m_2^n + \dots + m_n^n}$

}

在界定了分隔直线后，空间就被分成了 $g(x_i) > 0$ 和 $g(x_i) < 0$ 两个部分，以此便可以判断数据类型的不同。与此同时，根据我们在机器学习章节中提出的要求，最佳直线应该与所有支持向量等距且最远；因此，我们需要对最佳直线与支持向量间的距离进行研究。

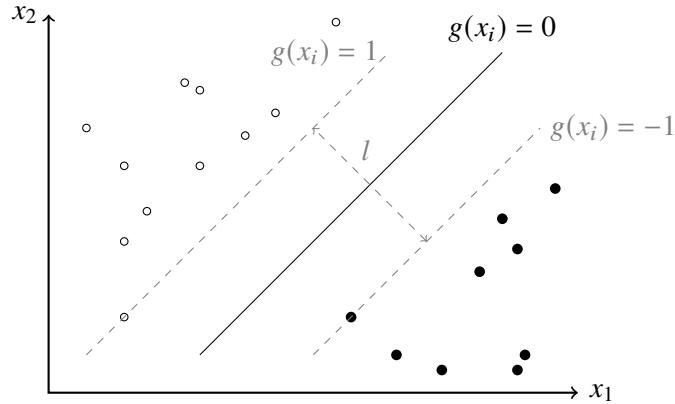


图 9.1: 二维向量的线性分类器

在这里，我们构建了名为“线性分类器”的模型来解决这一问题，如图9.1。这个问题的关键在于比较支持向量与分类直线的距离，那么我们不妨设当 $g(x_i) > 0$ 时，直线上方的支持向量经过直线 $g(x_i) = 1$ ；同理， $g(x_i) < 0$ 时，直线下方的支持向量经过直线 $g(x_i) = -1$ ，这样假设方便我们进行进一步的处理。我们可以得出 $g(x_i) = 1$ 和 $g(x_i) = -1$ 之间的距离，即 $l = 2/\|w\|$ 。同时，SVM 的目标是在对数据进行正确分类的情况下，最大化分类直线和支持向量的距离；由此，我们可以得到以下关系：

$$y_i (wx_i - b) \geq 1$$

$$\max(2/\|w\|)$$

可以看到最大化 $2/\|w\|$ ，也就是最小化 $\|w\|$ ；同时，利用高等数学的知识，我们能对其进行等价转化，这样就得到了 SVM 的基本形式：

$$y_i (wx_i - b) \geq 1$$

$$\min \frac{1}{2} \|w\|^2$$

最后，通过拉格朗日对偶，我们能够将上面的式子进一步转化为拉格朗日对偶性问题。在这个问题中我们通过添加拉格朗日乘子 $\alpha_i \geq 0$ ，写出拉格朗日函数和对应的目标函数：

$$L(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^n \alpha_i (y_i (wx_i - b) - 1)$$

$$\min_{w,b} \max_{\alpha \geq 0} L(w, b, \alpha)$$

在实际的 SVM 应用中，向量的维度往往不止二维，会出现三维甚至更多维度。线性函数在不同维度的空间中有不同的形式，例如在一维空间中是一个点，二维空间中是一条直线，三维空间是一个平面，如果不关注维数，线性函数又统称为超平面；此时我们研究函数的工具可能有变化，但线性分类器的基本思路仍然不变。我们依然可以通过基本形式和拉格朗日函数进行求解，这种方法具有普遍性。

9.4 卷积神经网络

卷积神经网络（Convolutional Neural Network，简称 CNN）属于前馈神经网络。与全连接神经网络不同，卷积神经网络只会计算上一层神经网络的部分单元，而全连接网络会计算所有单元。在大型图像处理上，卷积神经网络具有极大的优势；以常见的图像识别为例，神经网络的输入就是图像中的各个像素点，全连接网络的做法是将所有像素点不加以区分地全部输入下一层计算；然而图像识别的关键在于找到图像中的相邻或成群的像素点集，从而找到图像中各部分的“边缘”，全连接网络囫囵吞枣的做法显然效率不高。卷积神经网络则利用滤波器（Fliter）预先将图像中相邻像素之间的“边缘”过滤出来，从而大大提高了效率。

9.4.1 卷积层

9.4.2 池化层

9.4.3 全连接层

9.4.4 LeNet

9.5 手写识别示例

第 10 章 无监督学习

机器学习主要分为三类：监督学习（supervised Learning）、增强学习（reinforcement learning）、无监督学习（unsupervised learning）。也存在半监督学习（semi-supervised learning）这种情况，但在此不予讨论。

简单来说，区分监督学习和无监督学习的方法就是输入数据是否有标签（label）。如若没有标签则为无监督学习。我们以机器学习中的分类（classification）算法来举例，对于分类算法来说，输入的训练数据有特征（feature），有标签（label）。所谓的学习，其本质就是找到特征和标签间的关系（mapping）。而去判断一个分类算法的成功与否的方法，便是当我们输入有特征无标签的未知数据时，能否通过已知关系（参数）得到未知数据标签。

举个简单的例子，有监督学习相当于刷算法题的时候你知道答案，而无监督学习相当于你根本就没有答案，只能靠自己摸索。所以，与我们的常识相符，无监督学习的准确度往往比起有监督学习要低得多。那也许你会问了，既然无监督学习准确度那么低，为什么我们还要用它呢？那是因为在实际情况下，标签的获取需要极大的人工量，还不能保证变迁的准确性。所以，无监督学习也是十分重要的。

10.1 聚类算法

聚类是数据挖掘中的概念，其定义为按照某个标准把一个数据集分隔成不同的类或簇，使得相同类里的数据相似程度尽可能大。反之，不在同一个类里的数据，其差异也要尽可能大。“物以类聚，人以群分”就能大概指代聚类算法的意思。

讲到这里，我们需要区分聚类（clustering）与分类（classification）之间的区别。对于聚类来说，我们在聚类时，并不关心其中某一类是什么，我们需要实现的目标只是把相似的东西分到一起。因此，对于聚类算法来说，只要知道相似度衡量的标准就可以开始训练了。与之相对的，分类算法需要你告知它，它需要将数据分成哪些具体的类别，所以分类算法需要从训练集中进行学习，从而明白如何对未知数据进行分类。这就是聚类和分类的区别。

聚类的步骤主要如下：1. 数据准备：将数据转换为标准输入形式，使用特征标准化等方法；2. 特征选择：从最初的特征中选择最有效的特征，并将其存储于向量中；3. 特征提取：通过对所选择的特征进行转换，得到新的重点特征；4. 聚类：首先选择合适特征类型的某种距离函数进行接近程度的度量，而后执行聚类或分组；5. 聚类结果评估：对聚类结果进行评估，评估主要分为3种：外部有效性评估、内部有效性评估和相关性测试评估。

聚类主要分为层次化聚类算法，划分式聚类算法，基于密度的聚类算法，基于网格的聚类算法，基于模型的聚类算法等。下面我们就挑出几个重要的算法为大家进行讲解。

10.1.1 K-means 算法

K-Means 算法的特点是类别的个数是人为给定的，其假设数据之间的相似度可以使用欧氏距离度量，如果不能使用欧氏距离度量，要先把数据转换到能用欧氏距离度量。

接下来，我们简单介绍一下流程：首先，我们有在 n 维向量当中的一堆点，这里以二维空间为例。

接着我们随机生成 k 个聚类中心点，相当于将其分为几个类别。然后分别计算每一个数据点到这些中心的距离，把距离最短的那个当成自己的类别。这样就可以发现每个点都已经被分类了(有一个中心点)，但是并不准确。

接下来就是无监督学习，使得分类变得更加准确的时候。我们一开始随机确定的分类点，这时候就要变化了。而它变化的标准就是“收复”附近的点，所以它将往所有它这一类别的点的坐标平均值移动，也就是移向中心。而到达中心后，将再一次判断各个点到 k 个中心点的距离，选取离每个点最近的中心点作为它的类别，以此类推。

伪代码流程如下所示：

```
public static double K-Means(输入数据, 中心点个数K){  
    获取输入数据的维度Dim和个数N  
    随机生成K个Dim维的点  
    while(算法未收敛){  
        对N个点: 计算每个点属于哪一类。  
        对于K个中心点:  
            1, 找出所有属于自己这一类的所有数据点  
            2, 把自己的坐标修改为这些数据点的中心点坐标  
    }  
    return 结果  
}
```

接下来，我们来说明一下 k-means 算法使用过程中有可能会遇到的问题。1. 测量距离的方法并非一定要使用欧氏空间这个方法，只需满足以下条件都可以用：首先有个分类两个点的方法的算符记作

$$\langle \vec{a}, \vec{b} \rangle$$

， 并且其具有交换性 $\langle \vec{a}, \vec{b} \rangle = \langle \vec{b}, \vec{a} \rangle$ 。其次需要可以求一堆点的平均值的算法(求中心点): $\vec{\mu} = \text{Mean}(\vec{a}_1, \dots, \vec{a}_n)$ 求出后只需满足: $\sum_{i=1}^n (\vec{\mu} - \vec{a}_i)^2$ 。

2. 如何知道是否收敛? 使用代价函数: $J = \sum_{i=1}^C \sum_{j=1}^N r_{ij} \times v(x_j, \mu_i)$ 。其中: $v(x_j, \mu_i) = \|x_j - \mu_i\|^2$ 。代价函数的差分值小于一定数值的时候(N 次越不过最小值点) 即可认为是收敛了。

3. 代价函数不收敛，怎么办? 首先说一下什么时候容易发生震荡：在数据点个数比较少而且比较稀疏的时候容易发生这种事情，发生的原因大约有两种常见的：1、陷入某个环里，然后开始震荡，它将会绕着中心点进行低频振荡。2、两个点互相交换，每次交换不改变 J 的值就收敛了，如果交换以后不幸影响了其它的点，就出现了高频振荡。这个时候给出一种简单的解决方案：阻尼。简而言之，就是更新自己位置的时候考虑一下

原来的位置，一般阻尼比（在 0 1 之间取值）决定收敛速度，收敛的慢了也就不容易震荡，也就越容易陷入局部极小值，也就是说，不震荡的情况下我们应该把阻尼比尽可能取小一点 $\vec{C}^{upd} = \vec{C}^{new} \times (1 - \xi) + \vec{C}^{old} \times \xi$ \vec{C}^{upd} 是最后中心点的取值， \vec{C}^{new} 是当前集合的中心点， \vec{C}^{old} 是原来的中心点坐标。

10.1.2 DBScan 算法

10.2 DL4J 示例



第 11 章 神经网络

11.1 评价标准

准确率、召回率、精准率

第 12 章 深度学习

由神经元提出的感知机模型，只适用于解决二分类问题，对于多分类却无能为力，而多层次感知机却能解决这个问题。

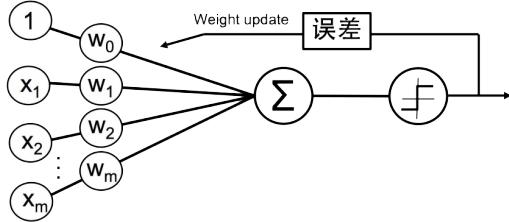


图 12.1：感知机

我们知道感知机可以实现 AND、OR、NAND 逻辑，而 XOR 正好可以由它们推算得出：
 $x_1 \oplus x_2 = (x_1 | x_2) \& (x_1 \bar{\&} x_2)$

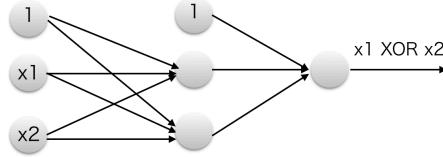


图 12.2：异或

这就是深度学习的魅力，但为什么在感知机提出来之后却停滞发展二十年呢？对于普通的二分类问题，使用的损失函数是基于点线距离的，但毕竟不是一个通用的解决方案。用于解决多分类问题的时候，就会显得非常笨重，越来越不好用。实际上，感知机和 SVM 非常相似，主要差别在于超平面（分割线）是否唯一，是否满足间隔最大化。感知机是不要求间隔最大的，只要能分开数据就算解决了。

12.1 多层感知机

本节以 XOR 为例，作为深度学习的入门铺垫。已知，单层感知机是可以实现 AND、OR、NAND 逻辑功能的。如下图：

感知机的参数如上，我们定义了 AND 和 OR 的运算模型，而 NAND 是 AND 相反运算，只要把上图 AND 的参数都取反即可： $(0.5, 0.5, 0.7) \Rightarrow (-0.5, -0.5, 0.7)$ 。示意图，如下。

感知机的最大缺点是误差计算是基于经验的（点到直线距离），而不是基于输出结果。如果误差的调节，能反应到 w 和 b ，势必要求整个调整过程是可微的，也就是一个平滑的过程。这个平滑的调节就是基于偏导数实现的，也就是梯度。因此，这也要求从输入到输出，从误差到输入是一个可以微调的平滑过程，这样才有利于机器学习。实际上，这

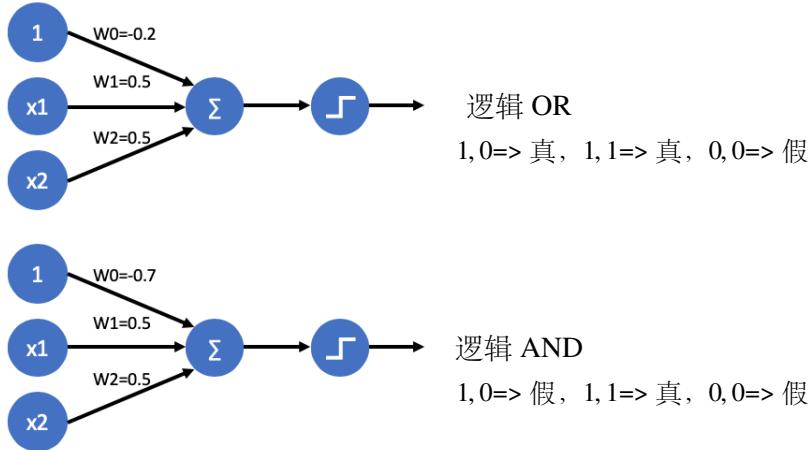
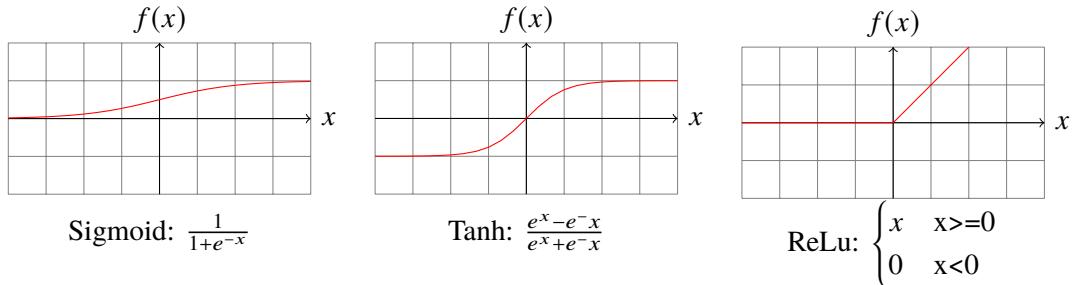


图 12.3: 感知机模型: AND、OR



图 12.4: 多层感知机求解异或

就是一个迭代求极值点的过程，直到误差最小或者迭代次数到达。所谓的前向传播和反向传播，就是这个道理。



感知机的缺点来源于激活函数是阶跃函数 *Sign*，它不是连续可导的函数，无法微调误差。换成 *ReLU*, *Sigmoid*, *TanH* 就可以让感知机重获新生。输出结果不再是 0 和 1 而是 (0, 1) 之间的概率值，代表更应该是哪个结果。

12.2 PlayGround

PlayGround 是一个在线演示神经网络的平台，是一个入门神经网络非常直观的网站。它图形化地展示神经网络的训练过程，非常有利于初学者获得感性认识。PlayGround 演示页面由 DATA (数据)、FEATURES (特征)、HIDDEN LAYERS (隐含层)、OUTPUT (输出层)，四个部分组成。

在 DATA 一栏里提供了 4 种不同形态的数据，分别是圆形、异或、高斯和螺旋。还可以对这些数据进行配置：噪声、训测比、批大小 (Batch)。平面内的数据分为蓝色和黄

色两种。



第三部分

人工智能应用



在人工智能的早期，那些对人类智力来说非常困难、但对计算机来说相对简单的问题得到迅速解决，比如，那些可以通过一系列形式化的数学规则来描述的问题。人工智能的真正挑战在于解决那些对人来说很容易执行、但很难形式化描述的任务，如识别人们所说的话或图像中的脸。对于这些问题，我们人类往往可以凭借直觉轻易地解决。

线性回归是机器学习中最基本的一个算法。回归分析中，又依据描述自变量与因变量之间因果关系的函数表达式是线性的还是非线性的，分为线性回归分析和非线性回归分析。

第 13 章 ND4J

13.1 ND4J 介绍

Nd4j 是一个基于 JVM 的，接口和 numpy 接近的张量运算库。目前流行的数据分析框架，大多是基于 Python 或 MatLab 开发的，很难移植到 JVM 运行环境。虽然有 Colt 这样的 Java 类库，但它对商业不友好。

ND4J 主要特点：（摘自官网）

1. 多用途多维数组对象
2. 支持多平台扩展，包括 GPU
3. 提供线性代数和信号处理功能

使用 ND4J 创建多维向量和矩阵非常容易，用起来和 numpy 非常类似，甚至还提供了 ND4S 这种更加接近 numpy 的类库。

创建 2X2 的 NDArray:

```
INDArray arr1 = Nd4j.create(new float[]{1,2,3,4}, new int[]{2,2});  
System.out.println(arr1);
```

输出： [[1.0 ,3.0] [2.0 ,4.0]]

13.2 使用 ND4J

在学习使用 ND4J 之前，有必要了解数组在内存中的组织方式。数组通常是一段连续的内存，但 Java 并不能保证整个维度是连续的，这也很容易证明：

Listing 13.1: Java 二维数组

```
int[][] numbers = new int[2][];  
int[] N1 = {1,2,3};  
int[] N2 = {2,3,4};  
  
numbers[0] = N1;  
numbers[1] = N2;
```

代码 13.1，展示了 N1 和 N2 是 2 个不同的 int[] 对象，但无法确定数据在内存中的组织是否连续。Java 这种分散的数据组织范式，非常不利于批量运算和 GPU 加速。为了更有效率的使用数据，ND4J 没采用 Java 的多维数组形式来表示张量（标量、向量、矩阵），而是在 JVM 之外申请的内存。不仅有更好的性能，还可以结合更多的 Native 加速

线性代数库，譬如 BLAS。¹

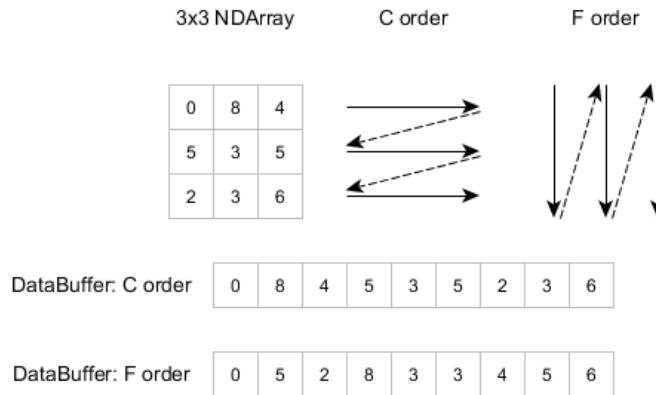


图 13.1: C 序和 F 序

图 13.1展示了这种不同。创建 NDArray 的时候，可以指定使用哪种方式。

13.2.1 创建和更新

与前面 IntVector 不同的是，NDArray 的数据只有 double 或 float 类型，并且所有的 NDArray 共享相同的 dtype。默认情况下，NDArray 的数据都是 float 类型的。创建 NDArray 有很多方式，ND4J 还为全 0 和全 1 数组提供了便利接口：

```
Nd4j.zeros(int...)
Nd4j.ones(int...)
```

其中，int... 是维度信息。

创建一般的 NDArray 则可以用章节一开始时提到的 create() 方法，传入两个数组作为数据组和维度即可。

另外，对于元素服从正态分布的 NDArray，或是元素属于(0,1)区间的 NDArray，ND4J 都提供了相应的接口：

```
Nd4j.rand(int, int);
Nd4j.randn(int, int);
```

在 DL4J 中，更新操作也很便利，利用 putScalar 和 putRow 方法，传入需要改变元素的位置与值，即可完成操作，例如下面的代码，putScalar 方法将矩阵下标为(1,3)的元素修改成 100，putRow 方法则将第一行的所有元素重新设置成传入的 INDArray 中对应的元素，这样就完成了元素的更新。

```
arr1.putScalar(1, 3, 100);
arr1.putRow(1, INDArray);
```

¹BLAS: Basic Linear Algebra Subprograms，目前最快的是 Intel 的 MKL

13.2.2 切片操作

对于 **NDArray**，我们往往需要其中的一部分来进行操作，这就需要用切片操作来完成。切片操作的代码如下图所示，我们传入一个或多个行数信息，就能获得对应行数的 **NDArray**。但是需要注意的是这两种方法只适用于二维矩阵。

```
arr1.getRow(int);  
arr1.getRows(int...);
```

但在日常使用中，我们对 **NDArray** 的切片要求往往更加复杂，仅仅使用上述的两个方法会比较困难。例如我们对一个 **NDArray** 进行取前三行，前三列的操作时，仅仅用 **getRow** 和 **getRows** 方法就显得比较麻烦。这时我们就需要用 **get** 方法进行操作：

```
arr1.create(new float[]{1,2,3,4,5,6,7,8,9}, new int[]{3,3});  
INDArray arr2 = arr1.get(NDArrayIndex.interval(0,2), NDArrayIndex.interval(0,2)  
);  
System.out.println(arr2);
```

输出结果为 **[[1.0,2.0] [4.0,5.0]]**，这就完成了对 **NDArray** 的切片操作。

13.2.3 基本运算

NDArray 的基本运算包括张量操作和元素操作。其中张量操作包括加减乘除四则运算，操作对 **NDArray** 中的所有元素进行，在调用时传入一个数即可，例如：

```
double myDouble = 1.0;  
INDArray arr1 = Nd4j.create(new float[]{1,2}, new int[]{1, 2});  
NDArray arr2 = arr1.add(myDouble);  
arr1.sub(myDouble);  
arr1.mul(myDouble);  
arr1.add(myDouble);  
System.out.println(arr2);
```

输出结果为 **[[2.0,3.0]]**。

元素操作与张量操作类似，都是对 **NDArray** 中的元素进行运算操作，不同的是张量操作传入的是一个数，所有元素执行同一个操作；而元素操作传入一个 **NDArray**，每个元素与另一个 **NDArray** 中对应的元素进行操作。因此，元素操作要求进行操作的两个矩阵形状一致。

```
INDArray arr1 = Nd4j.create(new float[]{1,2}, new int[]{1, 2});  
INDArray arr2 = Nd4j.create(new float[]{2,3}, new int[]{1, 2});  
System.out.println(arr1.add(arr2));
```

输出结果为 **[[3.0,5.0]]**。

13.3 线性回归示例



第 14 章 DL4J

14.1 DL4J 介绍

Deeplearning4j(简称 DL4J) 是为 Java 和 Scala 编写的首个商业级开源分布式深度学习库。DL4J 还可与 Hadoop 和 Spark 集成, 为商业化提供了很好的支持, 而不仅仅是学术研究工具。而在使用 Deep Learning 之前, 我们还要先确定问题的类型, 是有监督学习还是无监督问题? 对于有监督学习, 你需要先提供有标签的数据进行训练; 对于无监督学习, 需要能自主检测数据的相似性和异常状况。

还要考虑处理的特征数量有多少? 特征数量越多, 需要的内存也越大。就图像而言, 第一层的特征数量等于图像所包含的像素数。所以 MNIST 数据集中的 28 x 28 像素的图像有 784 个特征。医疗诊断中的图像则可能有 14 兆像素。



主流的深度学习框架有 TensorFlow、Caffe、Keras、Theano 等, 它们大多在单节点服务器通过 GPU 加速完成模型训练。随着大数据时代的来临, 使用分布式计算极大提高了计算性能。因此将分布式计算与深度学习结合成为必然趋势。DL4J 就是为此而生, 利用 Spark 在多服务器多 GPU 上进行分布式的深度学习模型训练, 让模型跑得更快。运用 DL4J 高效的训练一个完整神经网络模型包括:

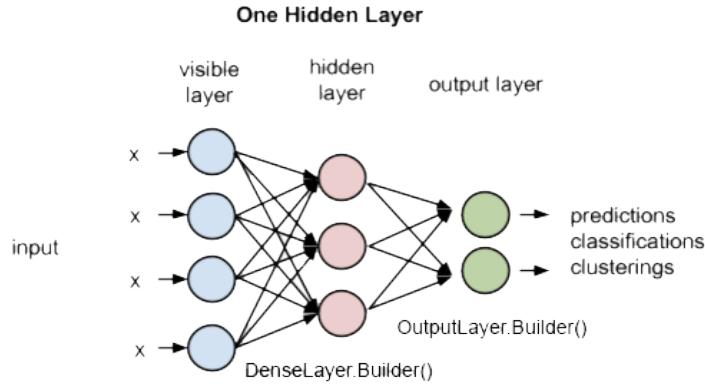
1. 预加载数据, 对数据进行预处理。DL4J 提供了 *DataVec* 简化数据采集过程。
2. 超参数配置, 各种优化算法和学习率、激活函数等。
3. 构建神经网络, 从单层到多层神经网络的构建。
4. 模型训练, 借助 ND4J 和分布式计算, 高效地训练模型。
5. 模型评估和保存。这正是 DL4J 的核心算法部分。

所以, 在开始训练之前, 要先花点时间学习如何使用 DL4J 的构建神经网络以及加载数据。由于加载数据的过程 (reshape、正则化、归一化等) 很容易引入意想不到的错误, DL4J 提供了 *DataVec* 类对数据加工预处理, 不仅可以减少错误, 还支持多种数据源, 譬如图片、CSV、ARFF 和纯文本等。

14.2 构造网络

DL4J 的神经网络都是用 `NeuralNetConfiguration` 构建的, 用它可以设定网络的层 (layer) 以及超参数。典型的神经网络包含: 输入和输出层, 以及隐藏层。对于全连接

层，在DL4J中也称为 *DenseLayer*。



1. 设置超参数

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(rngSeed)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .iterations(1)
    .learningRate(0.006)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .regularization(true).l2(1e-4)
    .list()
```

超参数 (Hyperparameter) 决定了神经网络学习方式的变量，包括模型的权重、激活函数、学习率和优化算法等。

2. 构建神经网络层

```
.layer(0, new DenseLayer.Builder()
    .nIn(numRows * numColumns) // Number of input datapoints.
    .nOut(1000) // Number of output datapoints.
    .activation("relu") // Activation function.
    .weightInit(WeightInit.XAVIER) // Weight initialization.
    .build())
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
    .nIn(1000)
    .nOut(outputNum)
    .activation("softmax")
    .weightInit(WeightInit.XAVIER)
    .build())
.pretrain(false).backprop(true)
.build();
```

而超参数(Hyperparameter)决定了神经网络学习方式的变量，包括模型的权重、激活函数、学习率和优化算法等。

14.3 准备数据

再好的训练模型，如果没有数据支撑，得出的结果也不是你想要的。本节将详细介绍常规数据使用方法，以及规范化手段。

14.3.1 MNIST 数据集

MNIST (Mixed National Institute of Standards and Technology database) 是一个计算机视觉数据集，包含 60000 张手写数字的灰度图片训练集(training set)，以及 10000 图片测试集(test set)。由来自 250 个不同人手写的数字构成，其中 50% 是高中生，50% 来自人口普查局(the Census Bureau)的工作人员。测试集也是同样比例的手写数字数据。其中每一张图片都包含 28×28 个像素点，并配有对应的标签，也即是图片对应的数字。

图 14.1，展示了测试集的组织形式：

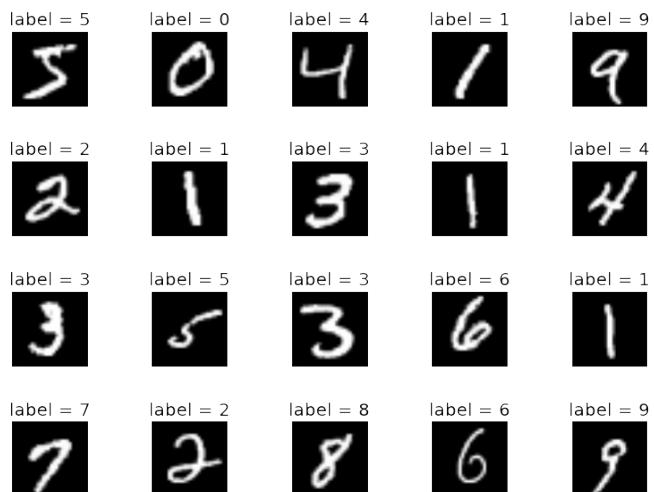


图 14.1: MNIST 数据

MNIST 官网提供了以下 4 个下载链接：

1. train-images-idx3-ubyte.gz: training set images (9912422 bytes)
2. train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
3. t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
4. t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

看到这里，大家最关心的还是下载的 MNIST 数据集，是如何定义的。这可以在官网查到，格式如下：

训练集标签文件 (train-labels-idx1-ubyte):

[偏移]	[数据类型]	[值]	[描述]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....			
xxxx	unsigned byte	??	label

标签的值: 0 ~ 9

16进制数据: 训练集-标签集合

```
0000 0801 0000 ea60 0500 0401 0902 0103
0104 0305 0306 0107 0208 0609 0400 0901
0102 0403 0207 0308 0609 0005 0600 0706
....
```

训练集数据文件 (train-images-idx3-ubyte):

[偏移]	[数据类型]	[值]	[描述]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....			
xxxx	unsigned byte	??	pixel

16进制数据: 训练集-图片集合

```
0000 0803 0000 ea60 0000 001c 0000 001c
0000 0000 0000 0000 0000 0000 0000 0000
.....
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0312 1212 7e88 af1a
a6ff f77f 0000 0000 0000 0000 0000 0000
1e24 5e9a aafd fdfd e1ac fdf2 c340
....
```

DL4J 对 MNIST 数据集支持得很好，提供有 deeplearning4j-datasets-xxxx.jar，其中有很多便利的 API 接口。

14.3.2 IRIS 数据集

Iris 也称鸢尾花卉数据集，是常用的分类实验数据集，由 Fisher, 1936 收集整理。包含 150 个数据集，分为 3 类，每类 50 个数据，每个数据包含 4 个属性。可通过花萼长度，花

萼宽度, 花瓣长度, 花瓣宽度 4 个属性预测鸢尾花卉属于 (Setosa, Versicolour, Virginica) 三个种类中的哪一类。



图 14.2: 鸢尾花

该数据集由 3 种不同类型的鸢尾花的 50 个样本数据构成。其中的一个种类与另外两个种类是线性可分离的, 后两个种类是非线性可分离的。Iris 以鸢尾花的特征作为数据来源, 常用在分类操作中。该数据集包含了 5 个属性:

1. Sepal.Length (花萼长度), 单位是 cm;
2. Sepal.Width (花萼宽度), 单位是 cm;
3. Petal.Length (花瓣长度), 单位是 cm;
4. Petal.Width (花瓣宽度), 单位是 cm;
5. 种类: Iris Setosa (山鸢尾)、Iris Versicolour (杂色鸢尾), 以及 Iris Virginica (维吉尼亚鸢尾)。

14.3.3 波斯顿房价数据集

波士顿房价数据集 (Boston House Price Dataset), 该数据集是美国人口普查局收集的有关波士顿马萨诸塞州住房的信息, 发表于 1978 年美国某经济学杂志。该数据集包含波士顿房屋的价格及其各项数据, 每个数据项有 14 个数据, 常用于回归分析。

- CRIM - 城镇人均犯罪率
- ZN - 占地面积超过 25,000 平方英尺的住宅用地比例。
- INDUS - 城镇非零售商用土地的比例。
- CHAS - Charles River 虚拟变量 (如果是河道, 则为 1; 否则为 0)。
- NOX - 一氧化氮浓度 (每千万)
- RM - 住宅平均房间数。
- 年龄 - 1940 年之前建成的自用房屋比例。
- DIS - 到波士顿五个中心区域的加权距离。
- RAD - 径向高速公路的可达性指数
- 税 - 每 10,000 美元的全额物业税率
- PTRATIO - 城镇的学生与教师比例
- B - $1000Bk - 0.63^2$, 其中 Bk 是城镇黑人的比例
- LSTAT - 人口中地位低下者的比例。
- MEDV - 自有住房的中位数价值 1000 美元

波士顿房价数据集，共有 506 个观察，有 13 个输入变量和 1 个输出变量。在回归分析中，如果有两个或两个以上的自变量，就称为多元回归。使用多个自变量的共同来预测，比只用一个自变量进行预测更符合实际，因此多元线性回归比一元线性回归的实用意义更大。实际上，经常抽取波士顿房价数据的 1-2 个特征，做线性回归分析的演示。

14.4 配置 DL4J

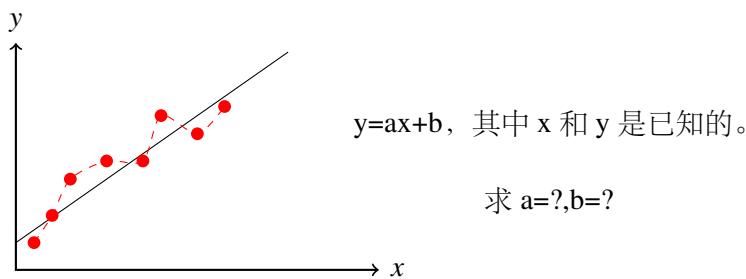
使用 IntelliJ IDEA 开发环境，建议创建 Maven 工程，根据官网的说明配置 pom.xml 联网下载依赖就可以。如果你比较熟悉 Gradle，官网也有相应的说明。

Listing 14.1: pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>deeplearning4j-core</artifactId>
    <version>1.0.0-beta2</version>
  </dependency>
  <dependency>
    <groupId>org.nd4j</groupId>
    <artifactId>nd4j-native-platform</artifactId>
    <version>1.0.0-beta2</version>
  </dependency>
</dependencies>
```

14.5 线性回归

线性回归可分为一元线性回归和多元线性回归。只包括一个自变量和一个因变量的线性回归，结果近似一条直线。



上图，根据已知点如何

预测下一个位置呢？若选用线性模型 $y = a_n * x_n + a_{n-1} * x_{n-1} + \dots + a_0$ 可以拟合所有点，但 $y = a * x + b$ 简单的只是一条直线。选用哪个预测模型更好呢？更高维度的线性模型，实际上也就是多元线性回归：

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p + \epsilon \quad (14.1)$$

其中, ϵ 代表一些未知因素引入的随机误差(残差), 譬如丢失一些关键特征。这个误差会导致有些点偏离预测的函数曲线。在机器学习中, 线性回归的模型描述通常使用 $\hat{y} = w^T x + b$ 的形式。其中, \hat{y} 是模型预测 y 应该取的值。

已知数据							
1	2	3	4	5	6	7	8
(0.2,0.3)	(0.4,0.6)	(0.6,1.0)	(1.0,1.2)	(1.4,1.2)	(1.6,1.7)	(2.0,1.5)	(2.3,1.8)

对于简单的线性回归问题, 使用 *ND4J* 和 *DL4J* 都可以求解。不同的是 *ND4J* 结合最小二乘法直接公式求解, 而 *DL4J* 根据已知数据训练的结果。

1、创建神经网络

```
OutputLayer outputLayer = new OutputLayer.Builder(LossFunctions.LossFunction.
    MSE)
    .activation(Activation.IDENTITY)
    .nIn(1)
    .nOut(1).build();

MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .weightInit(WeightInit.ZERO)
    .updater(new Sgd(0.01))
    .list()
    .layer(0, outputLayer)
    .pretrain(false)
    .backprop(true).build();

MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.init();
```

2、初始化数据

```
double [] input = {
    0.2, 0.4, 0.6, 1.0, 1.4, 1.6, 2.0, 2.3
};

double [] output = {
    0.3, 0.6, 1.0, 1.2, 1.2, 1.7, 1.5, 1.8
};
INDArray inputNDArray = Nd4j.create(input, new int[]{input.length,1});
INDArray outPut = Nd4j.create(output, new int[]{output.length, 1});
```

3、机器学习过程

```
for( int i=0; i<5000; i++ ) {
    net.fit(inputNDArray, outPut);
```

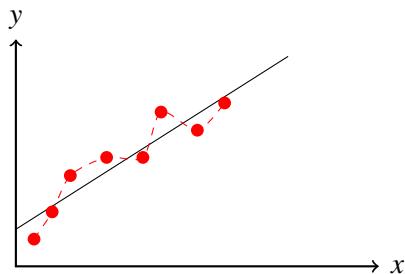


```
}
```

```
Map<String, INDArray> params = net.paramTable();
params.forEach((key, value) -> System.out.println("key:" + key +", value = " +
    value));
```

```
key:0_W, value = 0.6350
key:0_b, value = 0.4085
```

因此，拟合出的直线方程： $y = 0.635 * x + 0.4085$



很显然，机器学习预测的图像，比上一页凭我直观的绘制直线更加符合实际。

14.6 逻辑回归应用

14.7 多分类问题

14.8 聚类问题

14.9 卷积神经网络

14.10 循环卷积神经网络

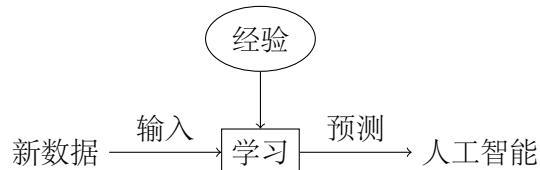
第四部分

综合运用



十年前，手写识别还只是专业技术人员的玩物，而如今手写识别已经变成深度学习的入门教程。而模式识别（Pattern Recognition）意味着需要专业人员提供很多特征。模式识别从 19 世纪 50 年代开始出现，在 20 世纪 80 年代左右曾经风靡一时，也是信息科学和人工智能的重要组成部分。它被应用于图像分析与处理、语音识别、数据挖掘等很多方面。尽管模式识别也很高大上，也有了较长时间的应用，但效果一直差强人意。

这种由人提取特征交给机器，再交给机器去判断的方法识别率一直差强人意。而机器学习可以从海量数据学习特征，使用已知经验数据（样本）自己提炼特征。另外，再强调一下概念，深度学习只是机器学习算法的一种，而人工智能也不限于机器学习一种。



大数据和深度学习，让我们能够摆脱人工特征提取的繁琐和束缚，让更多的人参与人工智能并为社会创造价值。互联网 + 没有过时，再加上分布式和云计算，使得人工智能 + 如鱼得水，再次迎来十年兴盛，相信接下来是数据为王的时代！

第 15 章 大数据和分布式

15.1 云计算

云计算是基于互联网的一种全新的模式。通过“云”我们可以轻松地存储数据，运行应用，甚至进行强度很大的运算。云计算实质上实现了对计算机资源的完全掌控，我们可以很轻易地通过云对资源进行分配，例如分配内存、处理器等计算资源。

机器学习和云计算、大数据的关系十分密切。大数据技术具有大量的数据与资源，能促进机器学习的进一步发展。反过来，机器学习的发展也能促进数据挖掘，分析等方面的进步。

15.2 边缘计算

边缘计算的定义是在靠近数据生成的设备端，就近提供服务。相比于云计算需要将所有数据上传至云端的特性，边缘计算具有天然的优势。第一，由于边缘计算强调“边缘”，数据的处理相比于云计算更为实时，高效；第二，边缘计算处理的数据多为“小数据”，在数据存储和计算上成本都比较低；第三，边缘计算的出现降低了对网络带宽的需求。随着物联网的不断发展，越来越多的联网设备将出现在我们的生活中，这大大增加了网络传输的压力。边缘计算则通过数据本地处理解决了这一问题。

边缘计算是对云计算的一种补充与完善。思科预计到 2022 年，移动设备和连接数量将达到 123 亿¹。如果仅仅用云计算来解决这么多互联设备的数据量往往会造成资源浪费，及时性较差，以及隐私泄露等问题。边缘计算在一方面能减轻云端的负担，另一方面也能保护用户的隐私，具有重要的实际意义。

15.3 开源架构

随着云计算和大数据的不断发展，人们对于数据存储和数据处理的效率提出了越来越严格的要求，因此一批优秀的开源架构应运而生。

15.3.1 Hadoop

Hadoop 是基于分布式计算的大数据开源架构，它的特点在于能高效，可靠地对大量数据进行处理。Hadoop 由 HDFS 和 MapReduce 组成，分别负责数据的分布式储存和分布式计算。

HDFS 提供了大规模存储的分布式解决方案，可以横跨成千上百台机器，操作却像对本地文件进行操作一样简单，并能存储 TB 甚至 PB 级别的特大文件。这些优势来源于

¹数据来源思科官网《思科移动网络 VNI 预测（2017-2022 年）》

在 HDFS 中，文件的储存是以较大的抽象块为单位，这样做的好处在于能将文件存储于不同的硬盘中，使得文件的大小不受限于硬盘大小；同时较大的抽象块有利于对大型文件进行寻址操作，最小化寻址开销。

除了抽象块，HDFS 还包含了 NameNode 和 DataNode。用户在用 HDFS 进行存储时，客户端首先会将文件切块，再由 DataNode 储存在多个硬盘中。同时，NameNode 负责记录每一个文件的切块信息，以及每一块具体的存储机器。这样的系统就构成了 NameNode 为主服务器，多个 DataNode 为从服务器的 HDFS 文件系统。

MapReduce 是 Hadoop 的计算框架，能够以容错、可靠的方式并行处理海量的数据。MapReduce 主要包括了两个主要部分：Map 阶段和 Reduce 阶段。

15.3.2 Storm

Storm 是类似于 Hadoop 的实时数据处理框架，两者的区别在于 Storm 用于数据的实时计算，Hadoop 用于对离线数据进行处理。同时，Storm 将数据存储在内存中，这也有区别于 Hadoop 的 HDFS 存储方式。Storm 也提供了类似于 MapReduce 的简单编程模型，便于进行开发。

一个完整的 Storm 集群应该包括 Nimbus，Zookeeper 和 Supervisor，工作架构如图15.1所示。其中，Nimbus 负责接受客户端传来的拓扑代码并分拆 task，Zookeeper 负责 Nimbus 和 Supervisor 之间的通信和协调，Supervisor 接收并处理数据。这样高效的工作架构使得 Storm 具有实时处理大批量数据的能力，同时具有高扩展性，可以通过增加节点实现性能的线性提高。

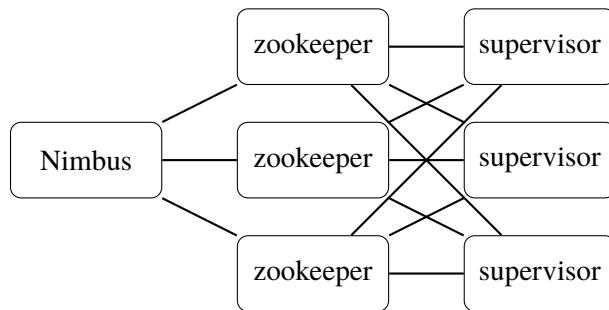


图 15.1: Storm 架构示意图

15.3.3 Spark

Spark 是一种通用的大数据计算框架，与 Hadoop 的 MapReduce，Storm 的流式实时计算引擎十分相似。同时 Spark 也提供了大数据和机器学习方面的框架，逐渐形成了大数据处理的一站式解决平台。

Spark 的特点是支持大型，低延迟的数据处理方式。拓展了常用的 MapReduce 计算模型。由于 Spark 支持在内存中进行计算，在必要时依赖硬盘进行复杂运算，因此 Spark 比 MapReduce 更高效。

Spark 是 MapReduce 的一种替代方案，兼容 HDFS，Hive 等存储方案，能够方便地融入 Hadoop 的生态环境，以弥补 MapReduce 的不足。

15.4 示例：人脸识别



第 16 章 车牌识别

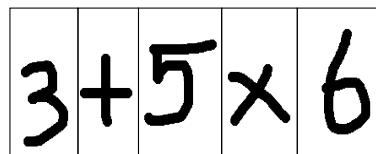
手写识别机器学习领域中的一个经典问题。自从深度学习的兴起，MNIST 已经是一个被“嚼烂”了的数据集。该问题要解决的是把 28×28 像素的灰度手写数字图片，识别为 0 到 9 的数字。而手写算式识别在数字识别的基础上，需要识别运算符号，并输出计算结果。

$$3+5 \times 6 \rightarrow \text{识别: } 3 + 5 \times 6$$

图 16.1: 手写算式识别

限于篇幅和笔者能力，本演示代码仅支持四则运算 ($+ - \times \div$)，其它运算符号留给同学们研究。利用深度学习算法，为用户提供手写算式识别，可以极大的提高用户体验。目前，有很多成熟的产品可以借鉴，相关的论文也非常多。

手写数字识别属于图像分类，可用的算法有：KNN、SVM、BP 和 CNN。其中 KNN 和 SVM 比较善于数据挖掘，用做手写数字识别的话，比较难达到实用目的；而 BP 使用全连接，难于构建深度网络，并存在梯度消失和过拟合问题。本章例子，选用 CNN 深度学习训练手写算式识别。



现在我们要识别一个算式，其实和 MINIST 数字识别没有太多差别，主要是增加了图像切割和四则运算符的识别任务。为了降低难度，约束字符之间最好有一些间距，不要字符连笔。

16.1 图像切割

手写算式的输入的形式，有 2 种：拍照、手绘。照片要经过图像处理之后，才能进行切割；而手绘可直接采集像素轨迹，就省掉了这一步。对于照片的预处理，需要先二值化和降噪把手写数字区域保留下来。如果考虑图片的背景干扰，可先使用 Canny 算子检测边缘，这借助 OpenCV 的 API 很容易实现。

我们这种要求手写数字不黏连的情况，把 Canny 输出的图像往 X 轴投影，就能得出每个数字的横向宽度坐标。虽然要求数字不黏连，但难免有横向重叠的情况，必须用类似范水法进行切割。

这种图像预处理，OpenCV¹非常擅长，不需要使用机器学习算法。

¹OpenCV 是一个开源的跨平台视觉库，实现了图像处理和计算机视觉方面的很多通用算法，支持 Linux、

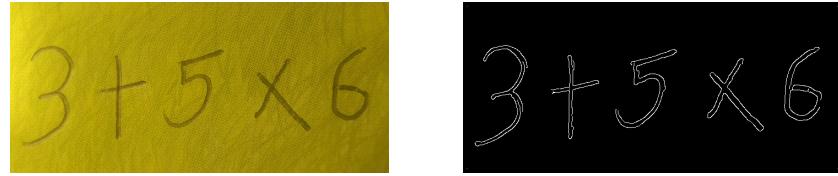


图 16.2: 图像处理: 二值化和边缘识别 (OpenCV)

Java 代码:

```
System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
Mat img = Imgcodecs.imread(src);
Imgproc.cvtColor(img, img, Imgproc.COLOR_BGR2GRAY);
Imgproc.Canny(img, img, threshold, threshold * 3, 3, true);
Imgcodecs.imwrite(dstImg, img);
```

Python 代码:

```
import cv2
from matplotlib import pyplot as plt

img = cv2.imread("handwriting_input.png", 0)
edges = cv2.Canny(img, 30, 60)
plt.subplot(1, 2, 1), plt.imshow(img, cmap='gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(1, 2, 2), plt.imshow(edges, cmap='gray')
plt.title('Edge Image'), plt.xticks([]), plt.yticks([])
plt.show()
```

实现 Canny 算子不在本书讨论范围, GitHub 上的一份 Java 实现接近 OpenCV 的效果 [Github 上获取](#)。而 CSDN 上的一份实现简化了很多处理细节, 有一定差距。有条件的同学, 建议使用 OpenCV 的接口。遇到的 OpenCV 问题, 也容易在网络上检索。

1. 转换为灰度图像
2. 高斯模糊处理
3. 根据梯度计算图像边缘幅值与角度
4. 非最大信号压制处理 (边缘细化)
5. 双阈值边缘连接处理
6. 二值化图像输出结果



经过 Canny 算子处理之后, 切割图像就非常容易了。把 Canny 输出的二值图像, 在水平方向上投影, 对数字进行竖向分割; 同理, 在纵向再做一次, 就能切割得紧凑一些。以下, 是在原图上切割的效果:

Windows、Android 和 Mac OS 操作系统上。它轻量且高效, 同时提供 C++、Python、MATLAB 等语言的接口。



Minist 数据集是白底黑字的，使用它训练出来的模型，就很难识别我们的手写算式。所以，有必要在二值图像上切割。



但 Canny 生成的是空心字，并且有很多断线，使用 FloodFill 算法是填不满的。笔者采用的是膨胀法，对于每一个白色点向周围扩张 6 个像素，这样的结果还不错，差不多和 Minist 数据集很接近了。



尺寸也要标准化，做成和 Minister 一样的 28×28 ，这个对图片缩放就可以实现。

16.2 算式识别

16.3 结果评估

第 17 章 写诗机器人

17.1 word2vec 介绍

word2vec 是 google 在 2013 年推出的一个 NLP 工具, 它的特点是将所有的词向量化, 这样词与词之间就可以定量的去度量他们之间的关系, 挖掘词之间的联系。

17.2 如何写诗

第五部分

常见问题和附录

第 18 章 常见问题集

问题 有没有办法章节用“第一章，第一节，(一)”这种？

解 你可以修改模板中对于章节的设置，利用 `ctex` 宏集的 `\zhnumber` 命令可以把计数器的数字形式转为中文。

问题 3.07 版本的 `cls` 的 `natbib` 加了 `numbers` 编译完了没变化，群主设置了不可更改了？

解 3.07 中在 `gbt7714` 宏包使用时，加入了 `authoryear` 选项，这个使得 `natbib` 设置了 `numbers` 也无法生效。3.08 版本中，模板增加了 `numbers` 和 `authoryear` 文献选项，你可以参考前文设置说明。

问题 大佬，我想把正文字体改为亮色，背景色改为黑灰色。

解 页面颜色可以使用 `\pagecolor` 命令设置，文本命令可以参考[这里](#)进行设置。

问题 ! LaTeX Error: Unknown option ‘scheme=plain’ for package ‘`ctex`’.

解 你用的 CT_EX 套装吧？这个里面的 `ctex` 宏包已经是 10 年前的了，与本模板使用的 `ctex` 宏集有很大区别。不建议 CT_EX 套装了，请卸载并安装 Te_X Live 2019。

问题 我该使用什么版本？

解 请务必使用[最新正式发行版](#)，发行版间不定期可能会有更新（修复 bug 或者改进之类），如果你在使用过程中没有遇到问题，不需要每次更新[最新版](#)，但是在发行版更新之后，请尽可能使用最新版（发行版）！最新发行版可以在 Github 或者 Te_X Live 2019 内获取。

问题 我该使用什么编辑器？

解 你可以使用 Te_X Live 2019 自带的编辑器 Te_Xworks 或者使用 Te_Xstudio，Te_Xworks 的自动补全，你可以参考我们的总结 [TeXworks 自动补全](#)。推荐使用 Te_X Live 2019 + Te_XStudio。我自己用 VS Code 和 Sublime Text，相关的配置说明，请参考 [LaTeX 编译环境配置：Visual Studio Code 配置简介](#) 和 [Sublime Text 搭建 LaTeX 编写环境](#)。

问题 您好，我们想用您的 ElegantBook 模板写一本书。关于机器学习的教材，希望获得您的授权，谢谢您的宝贵时间。

解 模板的使用修改都是自由的，你们声明模板来源以及模板地址（github 地址）即可，其他未尽事宜按照开源协议 LPPL-1.3c。做好之后，如果方便的话，可以给我们一个链接，我把你们的教材放在 ElegantLaTeX 用户作品集里。

问题 我想要原来的封面！

解 我们计划在未来版本加入封面选择，让用户可以选择旧版封面。

问题 我想修改中文字体！

解 首先，我们强烈建议你不要去修改字体！如果你一定坚持修改字体，请在 `newtxttext` 宏包加载前加入中文字体设置（`xeCJK` 宏包）。如果你选择自定义字体，请设置好 `\kaishu`, `\heiti` 等命令，否则会报错。如果你看不懂我现在说的，请停止你的字体自定义行为。

问题 请问交叉引用是什么？

解 本群和本模板适合有一定 LATEX 基础的用户使用，新手请先学习 LATEX 的基础，理解各种概念，否则你将寸步难行。

问题 定义等环境中无法使用加粗命令么？

解 是这样的，默认中文并没有加粗命令，如果你想在定义等环境中使用加粗命令，请使用 `\heiti` 等字体命令，而不要使用 `\textbf`。或者，你可以将 `\textbf` 重新定义为 `\heiti`。英文模式不存在这个问题。

问题 代码高亮环境能用其他语言吗？

解 可以的，ElegantBook 模板用的是 `listings` 宏包，你可以在环境之后加上语言，全局语言修改请使用 `\lstset` 命令，更多信息请参考宏包文档。

问题 群主，什么时候出 Beamer 的模板（主题），ElegantSlide 或者 ElegantBeamer？

解 这个问题问的人比较多，我这里给个明确的答案。由于 Beamer 中有一个很优秀的主题 `Metropolis`。我觉得在我们找到非常好的创意之前不会发布正式的 Beamer 主题，如果你非常希望得到 ElegantLATEX “官方”的主题，请在用户 QQ 群内下载我们测试主题 PreElegantSlide（未来不一定按照这个制作）。正式版制作计划在 2020 年之后。

问题 群主好棒，想嫁！

解 我取向正常！



附录 基本数学工具

本附录包括了计量经济学中用到的一些基本数学，我们扼要论述了求和算子的各种性质，研究了线性和某些非线性方程的性质，并复习了比例和百分数。我们还介绍了一些在应用计量经济学中常见的特殊函数，包括二次函数和自然对数，前 4 节只要求基本的代数技巧，第 5 节则对微分学进行了简要回顾；虽然要理解本书的大部分内容，微积分并非必需，但在一些章末附录和第 3 篇某些高深专题中，我们还是用到了微积分。

A.1 求和算子与描述统计量

求和算子是用以表达多个数求和运算的一个缩略符号，它在统计学和计量经济学分析中扮演着重要作用。如果 $\{x_i : i = 1, 2, \dots, n\}$ 表示 n 个数的一个序列，那么我们就把这 n 个数的和写为：

$$\sum_{i=1}^n x_i \equiv x_1 + x_2 + \dots + x_n \quad (\text{A.1})$$