西安建筑科技大学

本科毕业设计（论文）

题　　目　基于负载预测的反向传播神经网络算法研究

学生姓名　吉彬

学　　号　140607117

院（系）　信息与控制工程学院

专　　业　软件工程

指导教师　边根庆

2018 年 6 月 13 日

西安建筑科技大学

本科毕业设计（论文）任务书

题　　目：基于负载预测的反向传播神经网络算法研究

院（系）：信息与控制工程学院

专业班级：软件工程1401班

学生姓名：吉彬

学　　号：140607117

指导教师（签名）：

主管院长（主任）（签名）：

时　　间：2018 年 4 月 2 日

一、毕业设计（论文）的主要内容（含主要技术参数）

人工神经网络是由众多连接权值可调的神经元连接而成，具有大规模并行处理、分布式信息存储、良好的自组织自学习能力等特点，能够完成模式识别、机器学习以及预测趋势等任务，具体内容如下：

1、围绕课题查阅相关资料和文献；

2、了解和掌握集群系统结构以及负载预测算法；

3、分析负载特性以及反向传播神经网络算法性能；

4、在负载均衡策略和反向传播神经网络算法分析基础上，实现基于负载预测的反向传播神经网络算法；

5、设计要求：基于负载预测的反向传播神经网络算法，考虑服务器的实际负载状态对算法性能进行优化，提高算法运行效率，并通过实验论证；

6、根据上述设计方法，撰写毕业论文。

二、毕业设计（论文）应完成的具体工作（含图纸数量）

1、学习关于神经网络方面的理论知识；

2、学习和掌握 BP 算法；

3、了解和掌握集群系统结构；

4、学习相关的软件，并进行编程验证。

三、毕业设计（论文）进程的安排（起讫日期：2018年3月20日至2018年6月16日）

序号	设计（论文）各阶段任务	日　期	备　注
1	根据任务书，完成开题报告	第4-5周	
2	学习神经网络方面的理论知识	第6-7周	
3	学习BP算法	第8-9周	
4	了解和掌握集群系统结构	第10-12周	
5	总结设计与研究报告，撰写毕业论文	第13-15周	
6	毕业答辩	第16周	

四、主要参考资料及文献阅读任务（含外文阅读翻译任务）

[1] Fabio M. Soares，Alan M. F Souza，神经网络算法与实现，人民邮电出版社，2017.09

[2] 朱大奇.人工神经网络研究现状及其展望[N].江南大学学报,2004:103～108.

[3] 董军,胡上序.混沌神经网络研究进展和展望[J].信息与控制,1997 (5):360～368.

[4] 韩力群.人工神经网络理论、设计及应用[M].北京:化学工业出版社,2002.

[5] 马锐.人工神经网络原理[M].北京:机械工业出版社.2010.

[6] 胡守仁,余少波,戴葵.神经网络导[M].长沙:国防科技大学出版社,1992.

[7] 胡金滨,唐旭清.人工神经网络的 BP 算法及其应用[J].信息技术,2004 (4):1～4.

[8] 宋桂荣.改进 BP 算法在故障诊断中的应用[N].沈阳工业大学学报,2001 (3):252～254.

[9] 智会强,牛坤,田亮等.BP 网络和 RBF 网络在函数逼近领域内的比较研究[J].科技通报,2005 (2).

[10] Wu Yong-wei, Hwang Kai, Yuan Yu-lai, et al.. Adaptive workload prediction of grid performance in confidence windows[J]. IEEE Transactions on Parallel and Distributed Systems, 2010, 21(7): 925-938.

[11] Gmach D, Rolia J, Cherkasova L, et al.. Workload analysis and demand prediction of enterprise data center applications[C]. Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization, Washington, 2007: 171-180.

[12] Ganapathi A, Chen Y, Fox A, et al.. Statistics-driven workload modeling for the cloud[C]. Proceedings of Workshop on Self-Managing Database Systems (SMDB 2010), California, 2010: 87-92.

[13] Khan A, Yan Xi-feng, Tao Shu, et al.. Workload characterization and prediction in the cloud: a multiple time series approach[C]. Proceedings of 3rd International Workshop on Cloud Management (CloudMan 2012), HAWAII, 2012: 1287-1294.

基于负载预测的反向传播神经网络算法研究

摘要

现如今随着网络的快速发展，基本上全民都会上网，那么对于一些购物、医疗等网站可能在某一时刻，支撑网站的服务器要接收很多的请求并进行处理，即高并发问题。单一的服务器已不能满足需求，于是便有了集群服务器系统的诞生，它是将应用程序部署在多台单一服务器上，这多个单一服务器通过网络互联、数据共享形成集群服务器系统，这样就可以将某一时刻的并发请求分配到该集群中的某些或全部单一服务器上，从而解决高并发问题。

为了更好的对集群系统进行管理，可能需要了解集群系统在未来的负载情况，于是便有了本篇论文，在本篇论文中介绍了集群技术和集群服务器系统，通过 BP（Back Propagation）神经网络对集群系统进行训练学习。将影响服务器负载的因素作为神经网络的输入，负载情况作为神经网络的输出，对现已有的数据进行学习训练，得到输入与输出之间的映射关系，并用各层间的连接权值矩阵记录保存，得到训练好的神经网络，这样对未来集群服务器系统的负载情况根据输入可以预测出负载情况。

神经网络因其并行处理、分布式存储、良好的鲁棒性和容错性、可塑性与自适应性、自组织与自学习等特点已被应用于诸多领域，如：模式识别、信号处理、图像处理、自动控制、工程技术、医学、金融等方面，而本篇论文则是将神经网络应用于预测方面。

通过研究发现标准的 BP 神经网络在进行负载预测时，训练学习的速度较为缓慢，除了学习训练本身就是一个缓慢的过程之外，标准的 BP 神经网络的学习速率在训练过程中一直保持不变也是训练学习缓慢的一个原因。因为学习速率仅表示每次训练时各层间连接权值的调整幅度，所以可以根据前后两次的训练误差的相对大小关系对学习速率进行动态调整。当前误差相对小于上一次误差时，表明神经网络的训练在沿着误差减小的方向发展，此情况下可以适当的提高学习速率的值进而提高训练学习的速度；当前误差相对小于上一次误差，表明神经网络的训练在沿着误差增大的方向发展，此情况下可以适当降低学习速率的值；当两者误差相对相等时，表明神经网络在稳定的进行学习训练。

目前神经网络已能够训练成功并且进行预测，但预测结果不稳定。其重要原因在于各层神经元节点数目的选择上，而神经元个数的确定需要大量的实验去进行尝试确定，目前还没有什么类似于数学公式的东西对神经元个数进行确定。

关键字：高并发，神经网络，负载预测，神经元，集群服务器系统。

Research on the Back-Propagation Neural Network Algorithm Based on Load Prediction

Abstract

Nowadays with the rapid development of network, and basically people on the Internet, so for some shopping, medical and other sites may at some point, support the site server to process the request and to receive a lot of, namely high concurrency issues. Single server cannot meet the demand, then with the birth of the cluster server system, it is will be deployed in a single application server, the multiple single server through the network interconnection form cluster server system, data sharing, so that it can be a moment of concurrent requests assigned to some or all of the cluster on a single server, thereby solving the problem of high concurrency.

In order to better manage the cluster system, may need to be aware of the load of the cluster system in the future, so this paper, clustering technology is introduced in this paper and the cluster server system, using BP (Back Propagation) neural network to cluster system to carry on the training study. Will affect the factor of the load on the server as the input of neural network load as the output of neural network, the existing data are learning training, get the mapping relationship between input and output, and between the layers of the connection weight matrix of record keeping and get trained neural network, so the load of cluster server system in the future based on the input can estimate the load.

Neural network because of its parallel processing, distributed storage, good robustness and fault tolerance, plasticity and adaptability, self-organization and self-learning, etc have been used in many fields, such as: pattern recognition, signal processing, image processing, automatic control, engineering, medicine, finance, etc., and this paper is applying neural network to predict.

Through the study found that the standard BP neural network in load forecasting, the training of learning speed is relatively slow, in addition to the training itself is a slow process, the standard BP neural network's learning rate remains the same in the process of training is also a reason for the slow learning training. Because every time learning rate only said training connection weights between layers of adjustment, so can according to the relations between the two times the relative size of the training error before and after to dynamically adjust the learning rate. Relative to the current error is less than the last error, show that the neural network training error decreases along the direction of development, this situation can appropriate increase the value of learning rate and improve the speed of training to learn; The current error is relatively less than the last error, which indicates that the training of neural network develops along the direction of error increase. When the error is equal to each other, it indicates that the neural network is in stable learning training.

Currently, neural networks have been able to train successfully and make predictions, but the results are not stable. The important reason is that the choice of neurons in each layer node number, and the determination of number of neurons needed a lot of experiments are conducted to try to determine, there is no something similar to a mathematical formula to determine the number of neurons.

Key Words: high concurrent, neural network, load prediction, neuron, Cluster server systems.

目录

1.绪论

1.1 课题背景及研究意义

随着计算机网络技术的不断发展和广泛应用，Internet 已成为社会各个方面必不可少的交流手段。由于客户机/服务器模型的简单性、易管理性和易维护性，客户机／服务器模式在网络上被广泛应用。万维网(World Wide Web)以其简单操作方式将图文并茂的网上信息带给广大用户，使用户可以通过简单的图形界面访问互联网上的各种信息和服务，这促使 Internet 用户剧烈增长，服务器系统对任务处理变得越来越繁重，网络服务器不堪重负，特别对于访问数量多的站点，服务器经常在短时间内过载。解决该问题的方法通常有两个：一是单服务器策略，不断升级服务器，使其具有更高的性能。然而网络的急剧扩展必然经常升级服务器系统，造成这种升级成本提高。另一种方法是多服务器策略。当负载增大的时候，只需在服务器群上增加一个或多个服务器即可，这种技术就是服务器集群技术。与单服务器策略相比，集群技术可以利用各档次的计算机作为节点，不仅系统的造价低，还可以实现很高的运算速度，提高性能，能够满足目前网络服务要求。当有很多的请求同时发送到集群服务器时，根据负载均衡将这些请求相对平均的分配给集群服务器中的每个单独服务器，而负载预测是对未来负载情况的一种推测，根据推测结果利用负载均衡技术分配负载。

在科学计算领域，高性能服务器集群由于其较好的可扩展性、容错性和高性价比而成为目前使用最为广泛的计算基础设施，为了提高集群系统的资源利用率和性能，现有的集群管理系统根据系统当前负载状况，采取了按照一定的作业调度策略分派计算任务到各计算节点的方法。但该方式仅考虑了当前系统的资源使用状况，缺乏对系统未来一段时间运行状况的了解，可能会导致较差的资源分配和作业调度方案，继而引起系统性能下降等问题。因此，利用预测技术对集群系统资源使用状况进行实时预测具有重要意义。

负载平衡是网络并行计算中的一个重要问题。负载平衡牵涉到把一个大任务划分为一系列能够并行处理的小任务,并把每个小任务分配到合适的计算资源上,如一个处理器在多处理器系统中或者是一台计算机在计算机网络中。通过研究一些可以将负载平衡分配到各个处理器计算机上的策略,使得各处理器计算机能同时完成并行任务,减小并行任务的执行时间,提高并行效率,是并行程序设计人员关心的热点问题

现有的基于时间序列的预测技术包括：中值和均值预测法、遗传算法预测及小波分析预测等，但这些方法由于自身固有的缺陷不适合于实时的集群资源预测。而人工神经网络(ANN)具有自组织、自学习、并行处理信息以及能够逼近任意非线性等特点，较之其他预测技术更适合于集群资源预测。且ANN允许数据中带有较强的噪声，这也是其他方法所不能比拟的。文献表明对于稳态的时间序列预测，ANN要优于其他预测技术。然而，针对Dinda总结出的负载突变性现象，由于已训练好的网络模型的训练样本不包含突变后的负载特征而不得不重新训练，故在训练期间无法提供负载预测，而这不能满足实时预测的要求。

基于此，本文提出了一种利用ANN技术中的反传(BP)神经网络对集群系统负载进行预测的方法，并针对突变情况，提出了一种基于通知的BP神经网络和动态滑动窗口均值(DSWA)混合预测方法(BPDSWA)。

1.2 国内外研究情况

对集群服务器系统来说,进行动态负载平衡的最终目的是使该系统获得较小的工作时延。准确地衡量并行任务在系统中各个计算结点上的执行时间,从而选择合适的结点来分配任务,是实现高性能并行计算的关键所在。结点上的任务执行时间与该结点上的负载有很密切的关系,如果能够预测结点上的负载,那也能预测该结点上的任务执行时间,从而为并行计算系统上实现高效的负载平衡提供坚实、可靠的信息基础。

目前关于负载的收集方法主要有实测法和预测法两种。实测法通过直接或间接调用系统函数或访问注册表来测量结点负载,时间开销大,而预测法是通过运行预测模拟程序获得结点的负载,时间开销相对较小,采用神经网络预测负载时,实验证明预测法的时间开销比实测法低一个数量级。另外,实测法是对结点机负载进行实时的测量,所测出的负载情况具有一定的滞后性,因此在负载波动大的机群系统中并不适用。预测法是通过对结点机以前的负载情况进行学习,以此来预测下一时刻的结点机负载状况,具有预测功能,是实测法无法比拟的。在实际应用中可以根据预测所得结果进行并行计算任务的合理分配,从而提高并行计算的效率和计算资源利用率。

关于负载预测的研究主要分为网络流量负载预测和结点机负载预测两个领域,目前国内外主要集中在网络流量负载预测方面。网络流量负载预测研究始于上世纪九十年代初期,目前已经有了比较成熟的理论。网络流量负载预测关心的是流量大小和延迟。结点机负载预测的研究起步较晚,国外关于这方面的系统研究开始于上世纪九十年代后期,结点机负载预测研究的是结点机的平均负载量,其表现为计算结点机所耗费资源的综合体,一般包括利用率、就绪队列长度、可用内存空间、磁盘可利用空间等。在这方面具有代表性的人物是美国的 Dinda,她从年开始收集了大量负载样本,并由此提出了基于时间序列预测的预测理论,建立了主机资源负载系统。国内在这方面的研究较少,典型的有文献[27]和[28]提出的基于人工神经网络的预测算法,文献[29]应用滤波理论预测负载,提出基于滤波理论的预测算法,文献[34]提出对分时UNIX系统的 CPU 利用率的预测,均有较高的创新和理论借鉴价值。但文献[27]未对负载的特性作出任何解释,文献[29]只是笼统地分析了一般信号序列的简单特征,因而其算法的可靠性、普适性值得怀疑。另外,文献[27]和[28]未涉及预测系统模型及实现策略,都没有形成完整的预测体系,其应用性不得而知。文献[34]提出了负载的有关特性和预测利用率的实现方法,但未提出一个完整的负载预测模型。

1.2.1 神经网络的发展

神经网络的研究始于 19 世纪末期，1890 年专著《心理学原理》问世，这也是第一步详细论述人脑结构及功能的研究专著，W.James[35]对相关学习及联想记忆的基本原理做了开创性的研究，他指出，大脑皮层上任意一点的刺激量是其他所有发射点进入该点的刺激量的总和。

1943 年，美国生物学家W.S.McCulloh[36]和数学家W.Pitts[36]用逻辑数学工具研究客观事件在形成神经网络中的数学模型表达，提出形式神经元的数学模型，简称 M-P 模型，并且还证明了任意有限的逻辑表达式都可以由 M-P 模型组成的人工神经网络来实现。

1949 年，加拿大心理学家、神经生物学家D.O.Hebb[37]提出了 Hebb 学习规则，他认为，在神经网络中，信息式存储在连接权值中的。

1958 年，美国计算机学家 F.Rosenblatt 在原始的 M-P 模型的基础上引入了学习机制，提供了一种具有学习能力的“感知器”模型，这种模型是具有三层网络特性的神经网络结构。至此，人们完成了从单个神经元到三层神经网络的过渡，得到了第一个完整的人工神经网络模型。

1960 年，美国电机工程师B.Widrow[38]和M.Hoff[38]提出了一种连续取值的线性加权求和阈值网络，它被成功地应用于自适应信号处理和雷达天线控制等连续可调过程。他们在计算机上设计了仿真的人工神经网络，并在之后利用硬件电路实现了该设计。

1994 年，我国著名神经网络及智能控制专家廖晓昕教授奠定了细胞神经网络的数学理论与基础，并取得了一系列丰富的成果。

1995 年，美国波士顿学者 B.K.Jenkins 将光学方法引入神经网络中，利用光学强大的互联与并行处理能力来提高神经网络的实现规模，从而增强神经网络的自适应与学习能力。同年我国清华大学教授阎平凡研究了此类型神经网络的容量、泛化能力、学习性能以及计算复杂度。

1.2.2 当前预测方法

负载预测研究是最近国际上的研究热点，国外一些专家已经对此进行了研究，但还不太成熟，目前国内在该领域的研究非常少，因此，深入研究负载预测具有非常重要的意义。国内外在该领域的研究主要集中在网络流量的预测。目前已有的预测方法大多是基于时间序列的历史信息来预测。主要有以下几种预测方法：

1） 神经元网络预测

人工神经网络不需要建立精确的数学模型，并且有强大的自学习功能，通过适当的训练就能准确获得负载的特征，进行预测。神经元网络预测本质上具有并行分布式处理结构，适用于多信息融合，可同时综合定量、定性信息，具有鲁棒性。该方法在国外的相应研究较多，但是神经网络模型过程相对比较复杂，模型拟合速度较慢，而且神经网络需要大量的自我学习数据，数据不足会导致不好的预测结果。目前应用较多的是BP算法（Back Propogation），即误差反向传播神经网络。还有很多将模糊理论或灰色系统理论预测技术与人工神经网络相结合的方法。

李庆华[5]提出的基于人工神经网络的BP预测算法，但未对负载的特性做出任何解释。我国国家自然科学基金委员会、国防基金会、“863”计划和国家“攀登”计划等对神经网络研究工作的关注也逐年增加。神经网络的研究不仅其本身正在向综合性发展，而且越来越与其他领域密切结合起来，在只能控制、图像处理、模式识别、非线性优化、自适应过滤波器和信号处理、自动目标识别、时间序列分析、连续语言识别、知识处理、生物医学、机器人控制等方面取得了很大的进展，发展出性能更强的结构。目前国内在很多方面都已使用到BP预测算法。诸如宏观经济预测、市场预测、软件缺陷预测、网络流量预测等。

2） 线性时间序列模型预测

主要通过Box-Jenkins模型进行预测。Peter August Dinda认为，基于历史信息的负载预测策略，如线性时间序列模型，是比较符合实际的。但是，从模型的角度来看，这种模型依赖于对连续负载信息的测量和捕获。采用Box-Jenkins线性时间序列模型和分数ARFIMA(Auto Regressive Fractionally Integrated Moving Average)模型，对处理机的负载进行短期的预测。同时，比较了 AR (Auto Regressive)、MA (Moving Average)和ARIMA(Auto Regressive Integrated Moving Average)等各种算法的性能，发现AR的算法性能比较高。在此基础上，实现了在负载预测中使用线性时间序列模型，把结点负载的周期性样本看作是一个随机过程的实现，这个过程可以被模型化为驱动线性滤波器的白噪音源。滤波器的参数可以从以前的观测序列中来估计。其预测算法主要是用Box-Jenkins经典的时间序列预测算法。

NWS系统也采用了经典的 Box-Jenkins 模型，通过时间序列和历史信息，建立相应的线性模型来进行性能预测。然而预测周期不能根据实际应用的变化而动态调整，适应性还不够，预测算法的评价机制还不够完善。

Luis使用处理机的负载变化率对处理机进行动态负载预测。这种预测方法是线性预测的一种，其优点是简单易行，但是对于处理机负载的变化很快，随机性很强，没有平稳的、连续性的状态来说，这种预测算法的准确度就会降低。

Wei JIE等利用线程在CPU的执行时间片作为参数，来预测处理机的行为；Warren Smith和Parkson Wong利用作业的执行时间和队列等待时间进行预测，实现资源的选择；Rich Wolski提出了对分时UNIX系统的CPU利用率的预测，提出了CPU负载的有关特性和预测CPU利用率的实现方法，但就主机负载而言，不具有普遍意义。

3） 卡尔曼滤波预测

根据状态的连续性变化，来预测将来的状态。卡尔曼（Kalman）滤波是一种比较先进的数

据处理方法，是以60年代Kalman提出的滤波理论为基础的。该方法成功应用于交通、军事等预测领域，预测精度较高。Kalman滤波法是针对线性回归分析模型的一种矩阵迭代式参数估计方法，具有预测因子选择灵活，精度较高的优点。

许建峰提出了基于滤波理论的预测算法PAA。该算法只笼统分析了一般信号序列的简单特征，因而其可靠性、普适性值得怀疑。

4) 其他预测方法

遗传算法预测：根据遗传和进化的特性，预测未来的负载状况。优点：预测准确，自适应性强；缺点：计算量，速度慢，不适合实时强的应用。

小波分析预测：根据事件发生的周期性，来预测将来要发生的事件。优点：对周期性的行为预测性能较好；缺点：不适合随机性及实时性强的应用。

Petri网预测：利用Petri 网技术建立相应的模型，以预测将来状态。优点：若对具体的应用能建立相应的准确的模型，预测性能会很高；缺点：随着应用的不断变化，模型难以确定和调整，适应性不强。

1.3 负载预测方法的问题

综上所述，目前对于集群主机负载预测研究的问题主要集中在以下几个方面：

1) 负载的预测主要集中在网络流量的预测上，关于主机负载预测的研究较少。网络流量预测关心的是 TCP/IP 流量大小和延迟，而主机负载预测研究的是主机的平均负载量（Average Load），其表现为机器所耗费资源的综合体，包括：CPU 的利用率（CPU Availability）、自由内存的大小（Free Memory）、磁盘可利用空间（Disk Utility）等。

2) 负载预测系统的适应性不强。多数研究都集中在预测的算法上，并企图建立一个通用的预测算法以适应各种应用，然而这是很困难的。不同的应用，由于性质的差异，需要相对应的、适合的预测方法来进行负载预测。我们需要建立多种预测算法，分别应用于各种典型的领域，并实现对各个预测算法的内部调整（如参数调整等）。考虑到系统在根据预测结果实现负载均衡的同时，其实际负载状况已经发生改变，所以无法准确计算出当前预测算法的预测准确率，需要寻求一种有效的评价机制来评价各个预测算法，以选用相对较佳的预测算法进行预测服务。

3) 负载指数单一。大多数的负载预测算法主要依据主机上单一的一个负载指数进行负载预测。单一的负载指数不能完全反映系统的负载状态，因此不能提供更加精确的负载历史信息，则其预测的负载值的准确度也会大大的降低，负载预测也就失去了原有的目的，白白消耗了系统资源。

4) 负载预测系统没有很好的解决预测准确率和预测产生的系统额外开销的矛盾。预测的准确率与系统性能之间并不是简单的正比例关系，而是在一定的范围内，随着预测准确率的提高，系统性能会随之提高，但预测本身会占用一定的系统额外开销（预测准确率越高，额外开销越高），因此不能盲目的通过追求预测准确率的方法来提高系统的整体性能。那么，如何在尽量少增加系统额外开销的前提下，调整预测的方法，来获得较高的预测准确率或降低预测误差率，从而获得较高的系统性能，就成了我们要重点研究和解决的问题。

5) 大多数负载预测研究只专注于负载预测算法本身，但是都忽略了预测结束后进行负载平衡的问题。负载进行预测主要目的是为了更好的进行负载均衡，因此应对获得预测结果后的负载系统进行深入的均衡算法的研究。

从分析主机的负载特性入手，针对目前负载预测存在的问题，提出一个实用性较强的基于负载预测的负载均衡系统，尽量采用预测产生额外开销较小预测方法，建立预测模型及相关实现策略是十分必要的。

1.4 主要研究内容

论文的主要研究内容如下：

1）对现已有预测算法进行学习分析。

2）分析预测算法的性能并对其进行优化改进。

3）通过现有的集群系统负载相关数据对神经网络进行学习训练。

4）对未来集群系统负载情况进行预测估计。

1.5 论文的组织结构

论文的组织结构如下：

第 1 章 绪论。首先阐述了论文背景、国内外研究现状及神经网络发展史；其次论述了论文的主要研究工作和组织结构。

第 2 章 数学基础。因为在误差反向传播算法中使用到了一些基本的数学知识，所以在本章对所用到的数学知识进行简单的介绍。

第 3 章 人工神经网络的基本概念。在本章中简单的介绍了人工神经网络的一些概念及人工神经网络的结构。

第 4 章 BP 神经网络。本章对 BP 神经网络算法进行了介绍，提出了 BP 神经网络的缺陷并给出了算法改进方法，还给出了在设计神经网络时一些参数的选取规则。

第 5 章 集群系统结构。本章介绍了集群技术的分类、技术概念、技术层次和发展趋势，集群服务器的集群方法、优缺点和服务器性能指标，负载均衡常用的几种技术及分类。

第 6 章 模型可行性实验验证。在本章中以实验的方式来验证神经网络是否训练成功。

第 7 章 总结与体会。本章主要是对为何使用神经网络进行总结，通过神经网络的基本特征和功能来说明神经网络为何会应用于诸多领域。并通过本次研究来谈谈自己的体会。

2. 数学基础

因为在后面的 BP 神经网络中要使用到一些数学上的函数、向量、矩阵、导函数等知识，所以本章首先普及一下数学方面的基础知识，以方便后面对 BP 算法的理解。

2.1 向量

在线性代数中，标量（Scalar）是一个实数，而向量（Vector）是指 n 个实数组成的有序数组，称为 n 维向量。如果没有特别说明，一个 n 维向量一般表示列向量，即大小为 n × 1 的矩阵。

$$\mathrm{a}=\begin{bmatrix}\mathrm{a}_1\\ \mathrm{a}_2\\ \vdots\\ \mathrm{a}_n\end{bmatrix} \tag{2-1}$$

其中，a_i称为向量 a 的第 i 个分量。有时会加上转置符号 T 的行向量（大小为 1× n的矩阵）表示列向量。

$$\mathrm{a}=[\mathrm{a}_1 \quad \mathrm{a}_2 \quad \cdots \quad \mathrm{a}_n]^{\mathrm{T}} \tag{2-2}$$

2.1.1 向量的模

向量 a 的模|a|为

$$|\mathrm{a}|=\sqrt{\sum_{\mathrm{i}=1}^{\mathrm{n}}\mathrm{a}_{\mathrm{i}}^2} \tag{2-3}$$

2.1.2 向量的范数

在线性代数中，范数是一个表示长度概念的函数，为向量空间内的所有向量赋予非零的正长度或大小。对于一个 n 维的向量 a，其常见的范数有：

L_1范数：

$$|\mathbf{a}|_1=\sum_{i=1}^{n}|\mathbf{a}_i| \tag{2-4}$$

L_2范数：

$$|\mathbf{a}|_2=\sqrt{\sum_{i=1}^{n}\mathbf{a}_i^2}=\sqrt{(\mathbf{a}^T\mathbf{a})} \tag{2-5}$$

2.2 矩阵

一个大小为 m×n 的矩阵是一个由 m 行 n 列元素排列成的矩形阵列。矩阵里的元素可以是数字、符号或者数学式。矩阵一般默认是指数字矩阵。一个矩阵 A 从左上角数起的第 i 行第 j 列上的元素称为第 i，j 项，通常记为A_{ij}。

2.2.1 矩阵的基本运算

1) 如果 A 和 B 都是 m×n 的矩阵，则 A+B 和 A-B 也是 m×n 的矩阵，其每个元素是 A 和 B 相应元素相加或者相减。

$$(A+B)_{ij}=A_{ij}+B_{ij} \tag{2-6}$$

$$(A-B)_{ij}=A_{ij}-B_{ij} \tag{2-7}$$

2) 一个标量 c 与矩阵 A 的乘积为矩阵 A 的每一个元素与 c 相乘。

$$(cA)_{ij}=cA_{ij} \tag{2-8}$$

3) m×n 的矩阵 A 的转置是一个n × m的矩阵，记为A^T，A^T矩阵的第 i 行第 j 列的元素是原矩阵第 j 行第 i 列的元素。

$$(A^T)_{ji} = A_{ij} \quad (2\text{-}9)$$

4) A 和 B 两个矩阵相乘时，当且仅当矩阵 A 的列数和矩阵 B 的行数相等时才能定义。如 A 是m × k的矩阵，B 是k × n的矩阵，则 AB 乘积是一个 m×n 的矩阵。

$$(AB)_{ij} = \sum_{p=1}^{k} A_{ip}B_{pj} \quad (2\text{-}10)$$

矩阵的乘法满足结合律和分配律：

- 结合律：

$$(AB)C = A(BC) \quad (2\text{-}11)$$

- 分配律：

$$(A+B)C = AC + BC, \quad (2\text{-}12)$$

$$C(A+B) = CA + CB \quad (2\text{-}13)$$

2.2.2 常见的矩阵

1) 对称矩阵：是指转置后的矩阵与原矩阵相等，即满足$A = A^T$。

2) 对角矩阵：是一个主对角线之外的元素皆为 0 的矩阵，对角线上的元素可以为 0 或者其他值。一个n × n的对角矩阵 A 满足：

$$A_{ij} = 0 \ (i \neq j) \quad \forall i,j \in \{1,2,\cdots,n\} \quad (2\text{-}14)$$

3) 单位矩阵：是一种特殊的对角矩阵，其主对角线元素全为 1，其余元素全为 0，n 阶单位矩阵I_n是一个n × n的方形矩阵。一个矩阵和单位矩阵的乘积等于其本身。

$$AI = IA = A \quad (2\text{-}15)$$

4) 零一矩阵：矩阵的元素由 0 或者 1 组成的矩阵。单位矩阵是一个特殊的零一矩阵。

2.3 函数的导数

如果函数 f(x)在(a,b)中每一点处都可导，则称 f(x)在(a,b)上可导，则可建立 f(x)的导函数，简称导数，记为 f’(x)。

如果 f(x)在(a,b)内可导，且在区间端点 a 处的右导数和端点 b 处的左导数都存在，则称 f(x)在闭区间[a,b]上可导，f’(x)为区间[a,b]上的导函数，简称导数。

2.3.1 导数的基本法则

1) 加法法则：[f(x) + g(x)]' = f'(x) + g'(x)

2) 减法法则：[f(x) - g(x)]' = f'(x) - g'(x)

3) 乘法法则：[f(x)g(x)]' = f'(x)g(x) + f(x)g'(x)

4) 除法法则：[f(x)/g(x)]' = [f'(x)g(x) - f(x)g'(x)] / [g(x)^2]

2.3.2 导数的极值

一般地，设函数y = f(x)在x = x_0及其附近有定义，如果$f(x_0)$的值比x_0附近所有各点的函数值都大，我们说$f(x_0)$是函数y = f(x)的一个极大值;如果 f(x_0)的值比x_0附近所有各点的函数值都小，我们说$f(x_0)$是函数y = f(x)的一个极小值。极大值与极小值统称极值。

在定义中，取得极值的点称为极值点，极值点是自变量的值，极值指的是函数值。请注意以下几点:

1) 极值是一个局部概念。由定义，极值只是某个点的函数值与它附近点的函数值比较是最大或最小，并不意味着它在函数的整个的定义域内最大或最小。

2) 函数的极值不是唯一的。即一个函数在某区间上或定义域内极大值或极小值可以不止一个。

3) 极大值与极小值之间无确定的大小关系。即一个函数的极大值未必大于极小值。

4) 函数的极值点一定出现在区间的内部，区间的端点不能成为极值点。而使函数取得最大值、最小值的点可能在区间的内部，也可能在区间的端点。

5) 在函数取得极值处，如果曲线有切线的话，则切线是水平的，从而有$f'(x)=0$。但反过来不一定。如函数$y=f(x)$，在$x=0$处，曲线的切线是水平的，但这点的函数值既不比它附近的点的函数值大，也不比它附近的点的函数值小。 若x_0满足$x_0=0$，且在x_0的两侧$f(x)$的导数异号，则x_0是$f(x)$的极值点，$f(x_0)$是极值，并且如果 在x_0两侧满足"左正右负"，则x_0是$f(x)$的极大值点，$f(x_0)$是极大值;如果 在x_0两侧满足"左负右正"，则x_0是$f(x)$的极小值点，$f(x_0)$是极小值。

6) 极值与最值的区别:极值是在局部对函数进行比较，最值是在整体区间上对函数值进行比较。

2.3.3 常用函数及导函数

在 BP 神经网络中使用最广泛的激活函数就是单极性 S 型激活函数，该函数经常用来将一个实数空间的数映射到（0,1）区间，记为$f(x)$。

其函数表达式为：

$$f(x)=\frac{1}{1+e^{-x}} \tag{2-16}$$

其导函数表达式为：

$$f'(x)=f(x)(1-f(x)) \tag{2-17}$$

2.4 本章小结

在后面的 BP 神经网络正向传播和误差反向传播中，对输入数据的处理以及根据误差调整各层间连接权值时，会使用到一些数学方面的知识，故在本章介绍下相关的数学知识，以便更好的理解 BP 算法。

3. 人工神经网络的基本概念

在人工神经网络中，“处理单元”即为神经元。从网络的角度来说，也可称为“节点”。人工神经元形式化地对生物神经元进行了描述，抽象了生物神经元的信息处理程序，通过具体的数字及语言进行表述，模拟生物神经元的功能、结构，并用模型图进行表达。

3.1 人工神经元

人工神经元是构成人工神经网络的最基本单元。它简化模拟了生物神经元。图 3-1 是一种简化的人工神经元结构模型。一个神经元可以接受一组来自系统中其他神经元的输入信号，并且每个输入信号对应一个相应的权值，所有输入的加权和决定该神经元的激活状态。神经元模型包括三个基本元素：连接权值、求和单元及激活函数。

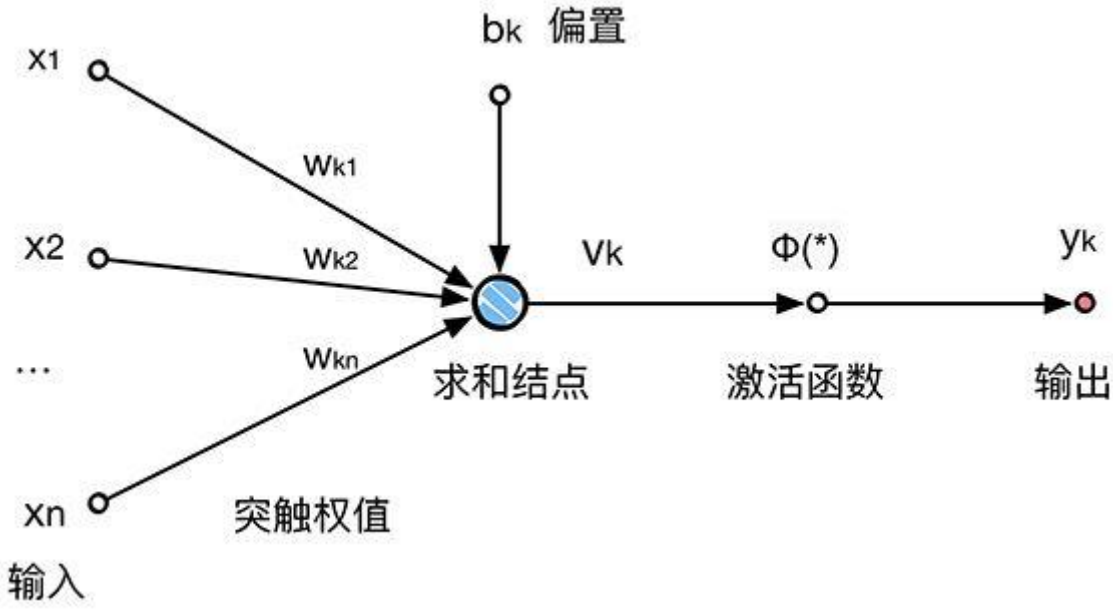


图 3-1 人工神经元结构模型

3.1.1 连接权值

各个神经元之间的连接强弱由连接权值表示，此部分对应于生物神经元的突触，权值就相当于突触的“连接强度”。权值为正时，表示激活状态；权值为负时，表示抑制状态。

3.1.2 求和单元

用于求取各输入信号与相应连接权值的加权和。此操作构成一个线性组合。

3.1.3 激活函数

一般多为非线性形式，具有非线性映射的作用，用来限制神经元的输出振幅，经信号限制在一定的允许范围之内。激活函数通常有如下性质：

1） 非线性

即导数不是常数。这个条件前面很多答主都提到了，是多层神经网络的基础，保证多层网络不退化成单层线性网络。这也是激活函数的意义所在。

2） 可微性

可微性保证了在优化中梯度的可计算性。传统的激活函数如 S 型激活函数等满足处处可微。

对于分段线性函数比如符号函数和阶跃函数，只满足几乎处处可微（即仅在有限个点处不可微）。对于 BP 算法来说，由于几乎不可能收敛到梯度接近零的位置，有限的不可微点对于优化结果不会有很大影响。

3） 计算简单

非线性函数有很多。极端的说，一个多层神经网络也可以作为一个非线性函数，但激活函数在神经网络前向的计算次数与神经元的个数成正比，因此简单的非线性函数自然更适合用作激活函数。

4） 非饱和性

饱和指的是在某些区间梯度接近于零（即梯度消失），使得参数无法继续更新的问题。最经典的例子是 S 型激活函数，它的导数在 x 为比较大的正值和比较小的负值时都会接近于 0。更极端的例子是阶跃函数，由于它在几乎所有位置的梯度都为 0，因此处处饱和，无法作为激活函数。在 x>0 时导数恒为 1，因此对于再大的正值也不会饱和。但同时对于 x<0，其梯度恒为 0，这时候它也会出现饱和的现象。

5） 单调性

即导数符号不变。这个性质大部分激活函数都有，除了诸如 sin、cos 等。单调性使得在激活函数处的梯度方向不会经常改变，从而让训练更容易收敛。

6） 输出范围有限

有限的输出范围使得网络对于一些比较大的输入也会比较稳定，但这导致了前面提到的梯度消失问题，而且强行让每一层的输出限制到固定范围会限制其表达能力。因此现在这类函数仅用于某些需要特定输出范围的场合，比如概率输出等。

7） 接近恒等变换

这样的好处是使得输出的幅值不会随着深度的增加而发生显著的增加，从而使网络更为稳定，同时梯度也能够更容易地回传。这个与非线性是有点矛盾的，因此激活函数基本只是部分满足这个条件。

8） 归一化

使样本分布自动归一化到零均值、单位方差的分布，从而稳定训练。

3.2 常见的激活函数

激活函数是神经元模型的三个基本要素之一，通过对神经元所获得的网络输入进行函数变换，从而得到适当的神经元输出，也称为变换函数或激励函数。激活函数可以对神经元的输出进行放大处理或者将其限制在一个适当的范围内。各神经元选取不同的激活函数，就会具有不同的数学模型与输出特性。神经元常用的五种激活函数：线性函数、阶跃函数、符号函数、单极性 S 型函数和双极性 S 型函数。

3.2.1 线性激活函数

线性激活函数可以达到对神经元的网络输出进行适当线性放大的目的，其函数形式如下：

$$f(x) = kx + c \tag{3-1}$$

式中：k，c 均为常数，分别表示放大系数和位移。该函数的变化曲线如图 3-2 所示：

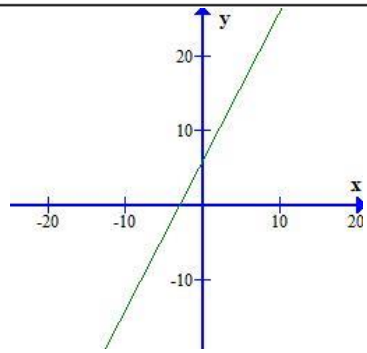


图 3-2 线性激活函数曲线图

3.2.2 阈值型激活函数

阈值型激活函数包括两个常用的函数形式，阶跃函数和符号函数。

1) 阶跃函数函数形式如下：

$$f(x)=\begin{cases}1, & x>0\\ 0, & x\le 0\end{cases} \tag{3-2}$$

该函数的变化曲线如图 3-3 所示：

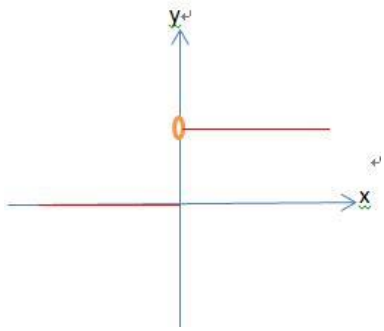


图 3-3 阶跃函数曲线图

2) 符号函数函数形式如下：

$$f(x)=\begin{cases}1, & x>0\\ -1, & x\le 0\end{cases} \tag{3-3}$$

该函数的变化曲线如图 3-4 所示：

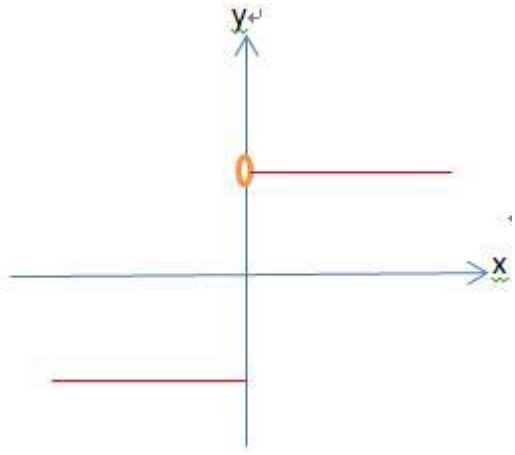


图 3-4 符号函数曲线图

由以上两种函数形式可以看出，两者非常类似，输出均为两种状态：输出为 1 时，神经元

为兴奋状态；输出为 0 或-1 时，神经元为抑制状态。

3.2.3 S 型激活函数

S 型激活函数又被叫做 Sigmoid 函数。该函数的本身与其导数都是连续的，所以比较容易处理。它分为单极性 S 型激活函数和双极性 S 型激活函数（双曲正切函数）两种函数形式。

1) 单极性 S 型激活函数函数形式如下：

$$f(x)=\frac{1}{1+e^{-x}} \tag{3-4}$$

该函数的变化曲线如图 3-5 所示：

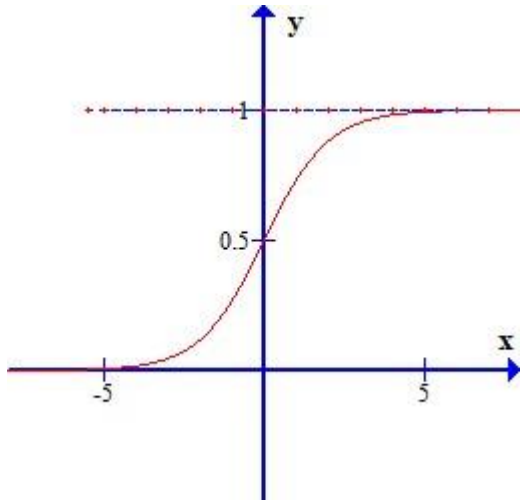


图 3-5 单极性 S 型激活函数曲线图

2) 双极性 S 型激活函数函数形式如下：

$$f(x)=\frac{1-e^{-x}}{1+e^{-x}} \tag{3-5}$$

该函数的变化曲线如图 3-6 所示：

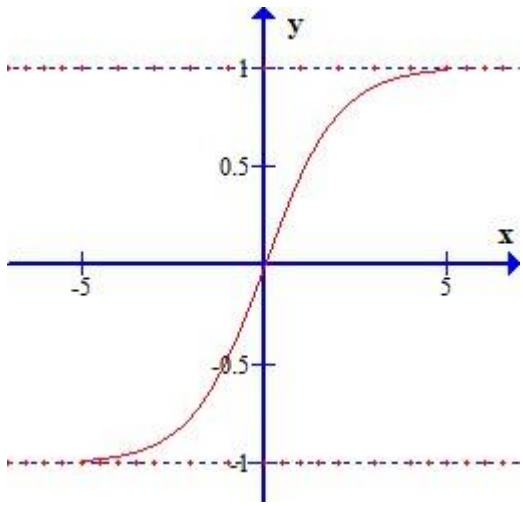


图 3-6 双极性 S 型激活函数曲线图

由上面两种 S 型激活函数的曲线图可以看出其不足有以下几点：

1) 容易过饱和并且造成梯度消失。S 型激活函数的导数等于其本身乘以 1 减去其本身，那么当 S 型激活函数的值为 0 或 1 时，就会导致梯度为 0 的情况出现，考虑到梯度传播时需要与本地的梯度相乘，那么梯度通过该函数后几乎没有信号流出，很容易使大

部分的神经元过饱和。

2) S 型激活函数包含指数函数，而计算指数函数的开销是很大的。

3.3 人工神经网络模型

人工神经网络通过不同的网络模型对人脑系统进行模仿和简化，神经网络强大的功能与它具有的大规模并行互连、非线性问题处理以及可塑的互连结构等特性密切相关。人们根据不同的需求按照不同的规则将神经元连成网络，构成不同拓扑结构的神经网络。

神经元之间的连接形式可以是任意的，按照不同的形式构成的网络模型具有不同的特性。不同的神经网络模型能够从不同的角度，按照不同的方法对各种网络模型进行分类。至今已有十余种不同的网络模型，其中的两种典型网络结构类型为前馈神经网络和反馈神经网络。在此详细说明一下前馈神经网络，因为后面的 BP 神经网络和前馈神经网络有着微妙的关系。

3.3.1 前馈神经网络

前馈神经网络又称为前向神经网络，它是指网络中信息处理的方向是从输入层到隐含层再到输出层，逐层进行。在前馈神经网络中，各神经元接收前一层的信息并处理，将处理结果传递给下一层，无反馈过程。前馈神经网络还可以细分为单层前馈神经网和多层前馈神经网络。如果网络中只有输入层和输出层，并且输入层的输入节点向量直接投射到输出层，没有反向作用，此类型的网络被称为单向前馈神经网络；如果网络中除了输入层和输出层外，还有一个或者多个隐含层，这样的网络被称为多层前馈神经网络。两种前馈神经网络结构图如图 3-7 和图 3-8 所示。

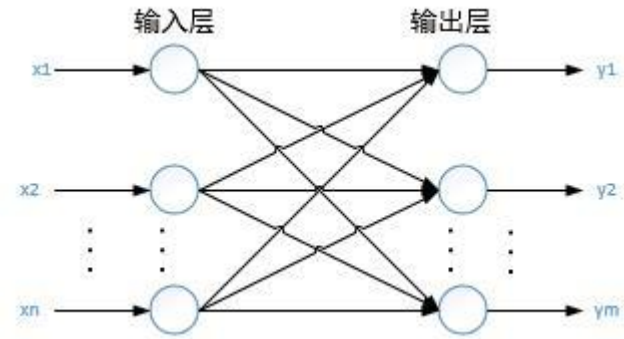


图 3-7 单层前馈神经网络结构图

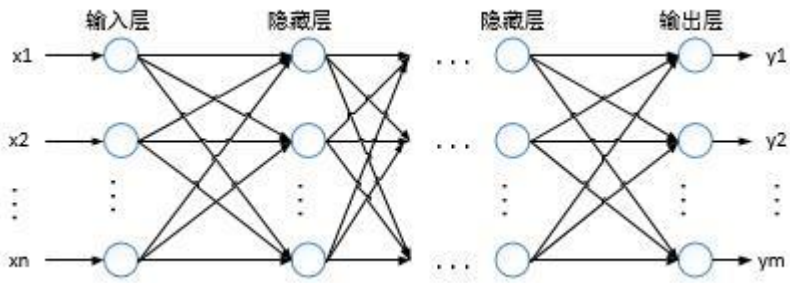


图 3-8 多层前馈神经网络结构图

以学习的角度来看，前馈神经网络是一种强有力的学习系统，其结构简单且容易编程实现；以系统的角度来看，前馈神经网络是一种静态非线性映射，只要经过简单地非线性复合映射的处理，就可拥有能够处理复杂非线性问题的能力；从计算的角度来看，缺少丰富的动力学行为。如今，前馈神经网络的大部分都属于学习网络，前馈神经网络在分类能力和模式识别能力上一般都比反馈神经网络强，经典的前馈神经网络有感知器网络、误差反向传播（BP）神经网络。

3.3.2 典型的神经网络结构模型

到目前为止，神经网络的模型已有几十种，其典型的网络模型有：①BP 神经网络②径向基函数神经网络等。这些网络模型具有函数逼近、数据聚类、分类模式、优化问题等能力。

1) 误差反向传播（Back Propagation）神经网络模型

误差反向传播（Back Propagation）神经网络模型被称为 BP 神经网络模型，是 1986 年由美国认知心理学家 D.E.Rumelhart 和 D.C.McCelland 等提出，是神经网络中重要之一。该网络模型通常采用基于 BP 神经元的多层前馈神经网络的结构形式，它由输入层、隐藏层、输出层组成，隐藏层可以是一层或者多层。BP 神经网络的学习过程分为两部分：①正向传播②反向传播。正向传播是信息从输入层经隐藏层处理后传递给输出层，每一层神经元的状态只能够对下一层的神经元状态造成影响。如果在输出层得到的输出不是期望值，则传播转入反向传播，误差信号会通过传播过来的原神经元连接通路返回。在返回的同时也会同步修改各层神经元连接的权值。这种过程不断迭代，最后使得误差信号在允许的范围之内。

2) 径向基函数（Radial Basis Function）神经网络

径向基函数（Radial Basis Function）神经网络是由英国的 D.Broomhead 和 D.Lowe 教授于 20 世纪 80 年代末提出的一种以函数逼近理论为基础的前馈神经网络，这类网络的学习等同于在多为空间内，找出能够训练数据的最有拟合面。该网络中的各隐藏层神经元激活函数组成拟合面的基函数。径向基函数神经网络是一种局部的逼近网络，也就是说存在于输入空间的某一局部区域的少量神经元被用来决定网络的输出。而 BP 神经网络则是典型的全局逼近网络，也就是说在各输入、输出数据对中，所有网络参数都被调整。构造本质的不同决定了径向基函数网络通常比 BP 神经网络的规模大、学习速度快，并且网络的函数逼近能力、模式识别能力、分类能力都强于后者。

3.4 本章小结

本章介绍了一些神经网络中的名词概念，三种激活函数的使用场景及函数特性，着重介绍了前馈神经网络和典型的神经网络结构模型。

4. BP 神经网络算法

BP（Back Propagation）神经网络是一种根据误差反向传播算法训练的多层前馈神经网络，是目前应用最广泛的神经网络模型之一。BP 神经网络能够学习和存储大量的输入-输出模式，而不需要确定描述这种关系的数学方程。BP 神经网络在模式识别、函数拟合、优化计算、系统辨识、最有预测和自适应控制等领域有着较为广泛的应用。

4.1 BP 神经网络的优缺点

BP 网络的误差反向传播算法因有中间隐含层和相应的学习规则，使得它具有很强的非线性映射能力，而且网络的中间层数、各层神经元个数及网络的学习系数等参数可以根据实际情况设定，有很大的灵活性，且能够识别含有噪声的样本，经过学习能够把样本隐含的特征和规则分布在神经网络的连接权上。

4.1.1 BP 神经网络的优点

1) 非线性映射能力

BP 神经网络实质上实现了一个从输入到输出的映射功能，数学理论证明三层的神经网络就能够以任意精度逼近任何非线性连续函数。这使得其特别适合于求解内部机制复杂的问题，即 BP 神经网络具有较强的非线性映射能力。

2) 自学习和自适应能力

BP 神经网络在训练时，能够通过学习自动提取输出、输出数据间的“合理规则”，并自适应的将学习内容记忆于网络的权值中。即 BP 神经网络具有高度自学习和自适应的能力。

3) 泛化能力

所谓泛化能力是指在设计模式分类器时，即要考虑网络在保证对所需分类对象进行正确分类，还要关心网络在经过训练后，能否对未见过的模式或有噪声污染的模式，进行正确的分类。也即 BP 神经网络具有将学习成果应用于新知识的能力。

4) 容错能力

BP 神经网络在其局部的或者部分的神经元受到破坏后对全局的训练结果不会造成很大的影响，也就是说即使系统在受到局部损伤时还是可以正常工作的。即 BP 神经网络具有一定的容错能力。

5) 其他方面

网络实质上实现了一个从输入到输出的映射功能，而数学理论已证明它具有实现任何复杂非线性映射的功能。这使得它特别适合于求解内部机制复杂的问题；网络能通过学习带正确答案的实例集自动提取“合理的”求解规则，即具有自学习能力；网络具有一定的推广、概括能力。

4.1.2 BP 神经网络的缺点

1) 局部极小化问题

从数学角度看，传统的 BP 神经网络为一种局部搜索的优化方法，它要解决的是一个复杂非线性化问题，网络的权值是通过沿局部改善的方向逐渐进行调整的，这样会使算法陷入局部极值，权值收敛到局部极小点，从而导致网络训练失败。加上 BP 神经网络对初始网络权重非常敏感，以不同的权重初始化网络，其往往会收敛于不同的局部极小，这也是很多学者每次训练得到不同结果的根本原因。

2) BP 神经网络算法的收敛速度慢

由于 BP 神经网络算法本质上为梯度下降法，它所要优化的目标函数是非常复杂的，因此，必然会出现“锯齿形现象”，这使得 BP 算法低效；又由于优化的目标函数很复杂，它必然会在神经元输出接近 0 或 1 的情况下，出现一些平坦区，在这些区域内，权值误差改变很小，使训练过程几乎停顿；BP 神经网络模型中，为了使网络执行 BP 算法，不能使用传统的一维搜索法求每次迭代的步长，而必须把步长的更新规则预先赋予网络，这种方法也会引起算法低效。以上种种，导致了 BP 神经网络算法收敛速度慢的现象。

3) BP 神经网络结构选择不一

BP 神经网络结构的选择至今尚无一种统一而完整的理论指导，一般只能由经验选定。网络结构选择过大，训练中效率不高，可能出现过拟合现象，造成网络性能低，容错性下降，若选择过小，则又会造成网络可能不收敛。而网络的结构直接影响网络的逼近能力及推广性质。因此，应用中如何选择合适的网络结构是一个重要的问题。

4) 应用实例与网络规模的矛盾问题

BP 神经网络难以解决应用问题的实例规模和网络规模间的矛盾问题，其涉及到网络容量的可能性与可行性的关系问题，即学习复杂性问题。

5) BP 神经网络预测能力和训练能力的矛盾问题

预测能力也称泛化能力或者推广能力，而训练能力也称逼近能力或者学习能力。一般情况下，训练能力差时，预测能力也差，并且一定程度上，随着训练能力地提高，预测能力会得到提高。但这种趋势不是固定的，其有一个极限，当达到此极限时，随着训练能力的提高，预测能力反而会下降，也即出现所谓“过拟合”现象。出现该现象的原因是网络学习了过多的样本细节导致，学习出的模型已不能反映样本内含的规律，所以如何把握好学习的度，解决网络预测能力和训练能力间矛盾问题也是 BP 神经网络的重要研究内容。

6) BP 神经网络样本依赖性问题

网络模型的逼近和推广能力与学习样本的典型性密切相关，而从问题中选取典型样本实例组成训练集是一个很困难的问题。

4.2 BP 算法

BP 算法由输入信号的正向传播和误差信号的反向传播两个过程组成。正向传播时，输入信号经过输入层、隐藏层和输出层，前一层神经元只会影响到下一层神经元的状态。如果输出层的输出与期望输出之间的误差在不可接受范围内，则进行误差信号的反向传播。这两个处理过程交替进行，按照梯度下降算法迭代更新网络的权值向量，最终将误差锁定在可接受范围之内，从而完成信息的提取和记忆过程。

图 4-1 为标准的 BP 神经网络结构，包含输入层、隐藏层和输出层。其中输入层包含 n 个神经元节点$(i = 1,2,\cdots,n)$，输入向量为$\mathrm{x} = (x_1, x_2, \cdots, x_n)^T \in R$；隐藏层包含 p 个神经元节点$(j = 1,2,\cdots,p)$，$\mathrm{f}(x)$表示隐藏层神经元的激活函数；输出层包含 m 个神经元节点$(k = 1,2,\cdots,m)$，输出向量为$\mathrm{y} = (y_1, y_2, \cdots, y_m)^T \in R$，$\mathrm{F}(x)$表示输出层神经元的激活函数。$\mathrm{W}_{ij}$表示输入层神经元到隐藏层神经元之间的连接权值；$\mathrm{W}_{jk}$表示隐藏层神经元到输出层神经元之间的连接权值。

BP 神经网络学习训练过程如下：

1) 输入信号正向传播

2) 计算输出误差

3) 输出误差反向传播

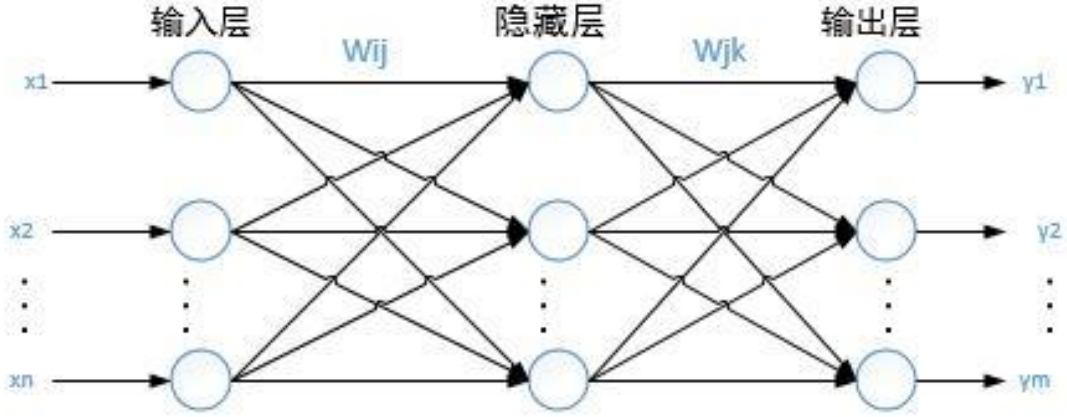


图 4-1BP 神经网络结构图

4.2.1 输入信号正向传播过程

输入信号首先输入到输入层，然后经过隐含层，最后到达输出层。分别计算隐藏层和输出层的输入输出信号。

1) 隐藏层第 j 个神经元节点的输入信号net_j为：

$$net_j = \sum_{i=1}^{n} W_{ij} x_i \tag{4-1}$$

2) 隐藏层第 j 个神经元节点的输出信号out_j为：

$$\mathrm{out}_j = f(net_j) = f\left(\sum_{i=1}^{n} W_{ij} x_i\right) \tag{4-2}$$

3) 输出层第 k 个神经元节点的输入信号net_k为：

$$net_k = \sum_{j=1}^{p} W_{jk} out_j = \sum_{j=1}^{p} \left(W_{jk} f\left(\sum_{i=1}^{n} W_{ij}\, x_i\right)\right) \tag{4-3}$$

4) 输出层第 k 个神经元节点的输出信号y_k为：

$$\mathrm{y}_k = F(net_k) = F\left(\sum_{j=1}^{p} \left(W_{jk} f\left(\sum_{i=1}^{n} W_{ij}\, x_i\right)\right)\right) \tag{4-4}$$

4.2.2 输出误差反向传播过程

误差反向传播，首先从输出层起反向逐层计算每一层的误差，根据梯度下降算法调整各层之间的连接权值，使网络的实际输出尽可能地接近期望输出，即使实际输出与期望输出之间的误差在可接受范围之内。

训练样本中包含输入层的输入向量 $\mathrm{x} = (x_1, x_2, \cdots, x_n)^T \in R$ 和期望输出向量 $\mathrm{d} = (d_1, d_2, \cdots, d_k)^T \in R$。隐藏层神经元到输出层神经元之间的连接权值修正量为$\Delta \mathrm{W}_{jk}$，输入层神经元到隐藏层神经元之间的连接权值修正量为ΔW_{ij}，学习速率为μ。

1) 隐藏层神经元到输出层神经元的连接权值修正量W_{jk}为：

$$\Delta \mathrm{W}_{jk} = \mu \sum_{k=1}^{m} (d_k - y_k)\, F'(net_k) out_j \tag{4-5}$$

2) 隐藏层神经元到输出层神经元的连接权值W_{jk}为：

$$\mathrm{W_{jk}} = \mathrm{W_{jk}} + \Delta \mathrm{W_{jk}} \tag{4-6}$$

3) 输入层神经元到隐藏层神经元的连接权值修正量ΔW_{ij}为：

$$\Delta W_{ij} = \mu \sum_{k=1}^{m} W_{jk}\, (d_k - y_k) F'(net_k) f'(net_j) x_i \tag{4-7}$$

4) 输入层神经元到隐藏层神经元的连接权值W_{ij}为：

$$W_{ij} = W_{ij} + \Delta W_{ij} \tag{4-8}$$

4.2.3 BP 算法学习流程

BP 学习算法的流程图如图 4-2 所示，标准 BP 学习算法的基本步骤如下：

步骤 1：参数初始化。随机初始化神经网络的各层间连接权值矩阵；初始化训练误差 E=0，最小训练误差 E(m)为一个小的正数，设置训练样本数为 P，学习速率取值范围为 0~1。

步骤 2：输入训练样本，计算神经网络各层的输出值。

步骤 3：计算神经网络的输出误差 E。

步骤 4：根据反向传播计算各层的误差信号。

步骤 5：计算各层间连接权值矩阵的误差，调整各层间的连接权值矩阵。

步骤 6：检查是否完成一次训练，若 p<P，p=p+1，返回步骤 2；否则，转向步骤 7；

步骤 7：检查神经网络总体输出误差是否满足精度要求，若满足 E<E(m)，则训练结束；否则，置 E 为“0”，p 为“1”，返回步骤 2。

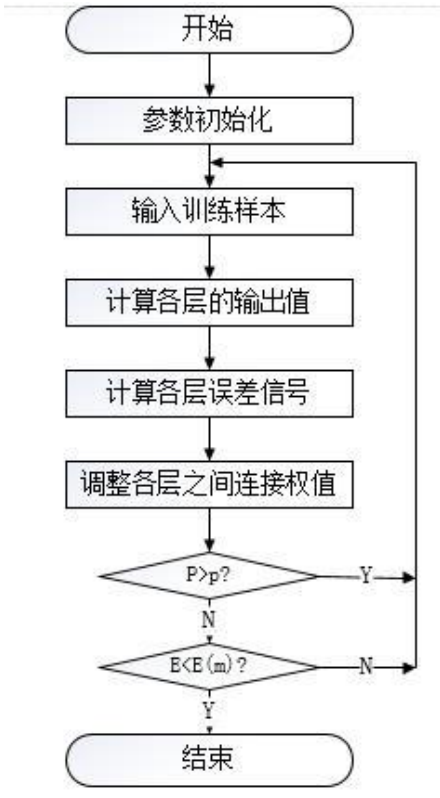


图 4-2BP 学习算法程序流程图

4.3 BP 算法改进

与早期的人工神经网络模型相比，BP 神经网络具有坚实的数学理论基础、严谨的证明推导过程和清晰的动态运行算法流程，使其成为实际应用中最普遍的神经网络之一。其突出优点是具有很强的通用性和非线性映射能力。但是前辈们通过非线性函数的拟合实验，再结合误差函数曲线图和在不同学习速率下的实验结果，发现 BP 神经网络仍存在缺陷。主要体现在两方面：

1) 学习速度慢。

2) 训练失败可能性大。

4.3.1 BP 算法的不足

1) BP 算法的学习速度很慢，其原因主要有：

①由于 BP 算法本质上为梯度下降法，而它所要优化的目标函数又非常复杂，因此，必然会出现“锯齿形现象”，这使得 BP 算法低效；

②存在麻痹现象，由于优化的目标函数很复杂，它必然会在神经元输出接近 0 或 1 的情况下，出现一些平坦区，在这些区域内，权值误差改变很小，使训练过程几乎停顿；

③为了使网络执行 BP 算法，不能用传统的一维搜索法求每次迭代的步长，而必须把步长的更新规则预先赋予网络，这种方法将引起算法低效。

2) 网络训练失败的可能性较大，其原因有：

①从数学角度看，BP 算法为一种局部搜索的优化方法，但它要解决的问题为求解复杂非线性函数的全局极值，因此，算法很有可能陷入局部极值，使训练失败；

②网络的逼近、推广能力同学习样本的典型性密切相关，而从问题中选取典型样本实例组成训练集是一个很困难的问题。

③难以解决应用问题的实例规模和网络规模间的矛盾。这涉及到网络容量的可能性与可行性的关系问题，即学习复杂性问题；

④网络结构的选择尚无一种统一而完整的理论指导，一般只能由经验选定。为此，有人称神经网络的结构选择为一种艺术。而网络的结构直接影响网络的逼近能力及推广性质。因此，应用中如何选择合适的网络结构是一个重要的问题；

⑤新加入的样本要影响已学习成功的网络，而且刻画每个输入样本的特征的数目也必须相同；

⑥网络的预测能力（也称泛化能力、推广能力）与训练能力（也称逼近能力、学习能力）的矛盾。一般情况下，训练能力差时，预测能力也差，并且一定程度上，随训练能力地提高，预测能力也提高。但这种趋势有一个极限，当达到此极限时，随训练能力的提高，预测能力反而下降，即出现所谓“过拟合”现象。此时，网络学习了过多的样本细节，而不能反映样本内含的规律

4.3.2 BP 算法的改进

1) 动态学习速率法

标准 BP 学习算法的收敛速度缓慢，一个主要原因就是学习速率设置不当。学习速率太小会导致学习时间太长；学习速率太大可能会在误差减小过程中产生振荡，导致学习过程不收敛。学习速率表示各层间连接权值调整的幅度，而标准 BP 学习算法在学习训练时，学习速率的值都是固定不变的。我们可以根据前后两次输出误差之间的相对大小关系来适当的修改学习速率的值，调节学习速率的原则是：检验调整后的权值是否降低了目标函数值，如果降低了，则说明设置的学习速率较小，应该适当增大学习速率的值；否则说明设置的学习速率较大，应该适当减小学习速率的值。动态调整学习速率的公式如下所示：

其中，$\mu(t)$和$\mathrm{E}(t)$分别表示第 t 次训练的学习速率和误差平方和。

$$\mu(t+1)=\begin{cases}1.05\mu(t); E(t+1)<E(t)\\0.7\mu(t); E(t+1)>1.04E(t)\\\mu(t); 其他\end{cases} \quad (4\text{-}9)$$

2) 附加动量法

附加动量法是在 BP 学习算法的基础上，在每一个权值的变化上加一项正比于前一次权值变化量的值，并根据误差反向传播算法产生新的权值变化。采用附加动量法的权值调整公式为：

其中 t 表示训练次数，m_c表示动量因子。

$$\Delta W_{jk}(t+1)=(1-m_c)\Delta W_{jk}+m_c\Delta W_{jk}(t) \quad (4\text{-}10)$$

$$\Delta W_{ij}(t+1)=(1-m_c)\Delta W_{ij}+m_c\Delta W_{ij}(t) \quad (4\text{-}11)$$

附加动量法的实质是将最后一次权值变化的影响，通过动量因子传递到下一次权值的更新。当动量因子$m_c=0$时，权值的变化仅根据标准 BP 学习算法产生；当动量因子$m_c=1$时，权值的变化则设置为最后一次权值的变化，而标准 BP 学习算法产生的变化部分则被忽略。当增加了动量项后，促使网络权值的调整向误差曲面底部的平均方向变化，当网络权值进入误差曲面底部的平坦区时，$\Delta W(t+1)\approx m_c\Delta W(t)$，从而防止$\Delta W=0$的出现，有助于使网络从误差曲面的局部极小值点中跳出。

附加动量法的设计原则：当修正的权值使误差有较大增长时，新的权值应该被取消，并使动量的作用停止，以避免网络进入较大的误差曲面；当新的误差变化率超过一个事先设定的最大误差变化率时，也应该取消所计算的权值变化。因此可以采用如下的判断条件确定动量因子的大小：$E(t)$为第 t 次训练的误差平方和

$$m_c=\begin{cases}0; & E(t)>1.04E(t-1)\\0.95m_c; & E(t)<E(t-1)\\m_c; & 其他\end{cases} \quad (4\text{-}12)$$

4.3.3 改进后 BP 学习流程

1) 改进后 BP 学习算法的流程图如图 4-3 所示，改进后 BP 学习算法的基本步骤如下：

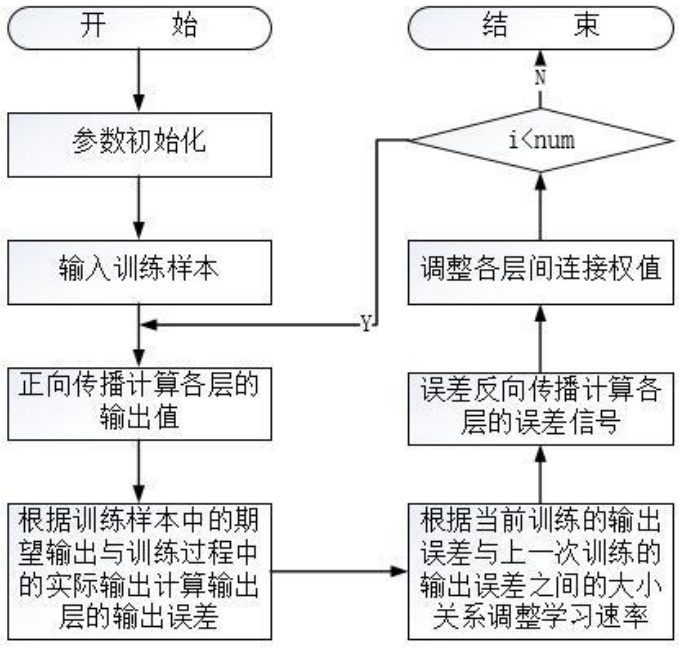


图 4-3 改进后 BP 学习算法流程图

步骤 1：参数初始化。随机初始化神经网络各层间的连接权值矩阵；初始化上一次输出误差为“0”；初始化学习速率范围为(0，1)；初始化训练次数（要训练多少次）num；初始化当前训练次数（第几次训练）i。

步骤 2：输入训练样本。训练样本为 n 个输入—输出对。

步骤 3：根据输入层的输入通过正向传播算法计算神经网络各层的输出值。

步骤 4：根据训练样本中的期望输出与训练过程中的实际输出计算输出层的输出误差。

步骤 5：根据当前训练的输出误差和前一次训练的输出误差的大小关系调整网络的学习速率。

步骤 6：将当前训练的输出误差反向传播计算神经网络各层的误差信号。

步骤 7：计算各层间连接权值矩阵的修正量。

步骤 8：修正各层间的连接权值矩阵。

步骤 9：判断训练次数是否达标。若未达标，则返回步骤 3 继续训练；否则训练结束。如果训练结束后，误差在不可接受范围，则应该适当调整训练总次数与初始学习速率的值。

2) 学习速率调整流程图如图 4-4 所示，学习速率调整规则如下：

规则 1：将前后相邻的两次训练误差进行对比。

规则 2：将误差平方和 E 作为误差。

规则 3：当 E(t+1)<E(t)时，即误差在减小，我们可以适当的提高学习速率。

规则 4：当 E(t+1)>mE(t)时，m 取值在 1.02~1.06 之间即可，说明误差在增大或者误差减小的幅度不大，我们可以适当降低学习速率。

规则 5：其他情况下，尽可能保持学习速率不变。

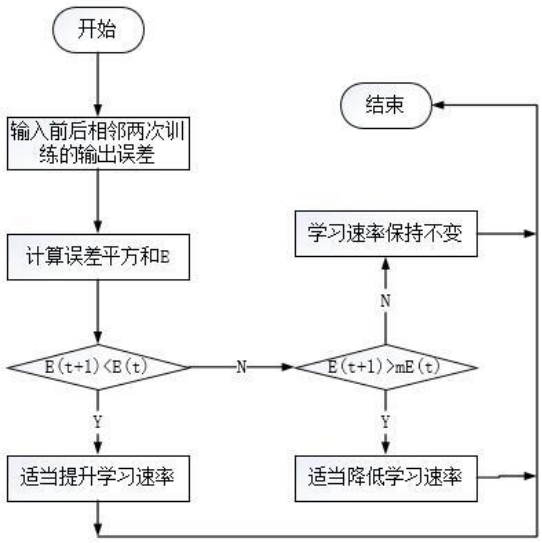


图 4-4 调整学习速率流程图

4.4 BP 神经网络的设计

神经网络的合理设计更有利于神经网络的训练学习以及以后的预测功能，一般以三层网络结构作为通用的函数逼近器。在设计神经网络时，先选择三层网络结构，如果在训练学习过程发现该网络结构不能很好的解决问题，再考虑多加一层隐藏层。

4.4.1 网络层数的设计

数学上已证明：具有一个输入层、具有 Sigmoid 激活函数的一个或者多个隐藏层和一个输出层的神经网络，能够以任意精度逼近所有连续函数。只有一个隐藏层的网络是通用的函数逼近器，可以实现 BP 神经网络的一些基本功能。但是只有一层隐藏层的神经网络并不是 BP 神经网络的最有结构，有时采用多个隐藏层会更好地解决问题。在设计 BP 神经网络时，一般先考虑设置一个隐藏层，当一个隐藏层的神经元节点数很多仍不能改善网络的性能时，可以考虑再增加一个隐藏层。增加隐藏层个数可以提高网络的非线性映射能力，能够进一步降低误差，提高训练精度，但是加大隐藏层个数必将使得训练过程复杂，训练时间延长。

4.4.2 网络的神经元个数选择

增加神经元个数可以有效提高 BP 神经网络的训练精度，其学习效果比增加网络层数更容易调整和观察。在结构实现上，增加隐藏层神经元节点个数比增加隐藏层层数要简单的多，比较常用的方法是通过对不同神经元数的 BP 神经网络进行对比，然后选择训练精度较高的网络。在 BP 神经网络中，隐藏层神经元节点个数的多少对神经网络的性能影响很大，因此选择恰当的节点个数是非常重要的。

输入层是外界环境与网络连接的纽带，其节点个数取决于输入数据的维数和特征向量的选取。选择恰当的特征向量，必须要考虑该特征是否能准确而完全地描述事务自身的本质特征，如果所选特征不能很好地表达事物本身的这些特征，那么网络学习和训练后得到的输出可能与期望输出有很大的误差。因此在进行网络训练之前，应该全面收集要进行仿真的系统的样本数据，并且在对数据处理之前要进行必要的相关性分析，以剔除无关特征和冗余特征，确定特征向量的维度。

对输出层神经元节点的选取，通常需要根据实际情况来确定。当 BP 神经网络用于模式识别时，往往模式自身的特性就已经决定了网络输出层神经元节点的个数。当神经网络作为分类器时，输出层神经元节点个数就等于所需信息类别数。

如果隐藏层神经元节点个数太少，则神经网络从训练样本中学习的能力就不足，网络很容易陷入局部极小值点，有时甚至可能得不到稳定的结果；而如果隐藏层神经元节点个数太多，则导致网络训练的时间延长，还有可能出现“过度拟合”现象，而且误差也不一定是最小的。虽然隐藏层神经元节点个数存在最优值，但是精确的找到这个最优值难度很大。

4.4.3 初始权值的选取

由于系统是非线性的，初始权值的选取对于学习是否达到全局最优、是否能够收敛以及训练时间的长短影响很大。如果初始权值太大，使得加权后的输入落在激活函数的饱和区，从而导致其导数非常小，进而影响权值修正量过小，使得网络的权值调整过程停滞。一般希望每个神经元通过初始设定权值后的输出值接近于 0，以保证神经元权值能够在激活函数最大变化处调整。因此，通常初始权值在（-1,1）之间选取。

选取不同的初始权值，不会影响网络的收敛精度，但有可能造成网络陷入某一局部极小值点。曾有人将初始权值锁定在[-0.5,0.5]和[-1.5,1.5]，结果发现：

1) 对于不同的训练数据和选用不同的网络结构，存在不同的最佳初始权值取值范围。最佳初始权值可以比其他初始权值取得更小的初始误差。

2) 最佳的初始权值不可以保证网络最终得到最好的收敛效果及最短的训练时间。

3) 初始权值也不能取的太大，取值太大意味着初始权值之间的差异很大，在网络训练过程中权值的更新容易出现跳跃现象。此外，网络初始权值太大也会导致神经网络陷入饱和状态。

从全局来看，能够影响网络学习过程的因素除了初始权值意外，还有一些其他的因素，如网络结构、学习速率的设定、动量项及激活函数的选择等。

4.4.4 确定学习速率

标准 BP 神经网络在学习过程中，学习速率始终保持不变。学习速率太小，则网络权值每次的调整幅度小，收敛速度慢，学习训练时间延长；学习速率太大，则网络权值每次调整幅度大。而较大幅度的调整网络权值可能导致神经网络在迭代更新过程中围绕误差最小值来回跳动，从而产生振荡，网络会变得发散而不能收敛到稳定值。对于标准 BP 神经网络，学习速率的初始值贯穿整个网络学习过程，决定每一次循环训练中产生的权值变化量。大的学习速率可能导致系统的不稳定；小的学习速率会导致网络收敛速度慢、训练时间延长，但是能保证网络的误差值不跳出误差表面的低谷而最终趋于最小误差值。因此，在 BP 神经网络中，倾向于选取较小的学习速率以保证系统的稳定性，取值范围在0.01~0.80。

4.5 本章小结

本章主要介绍了 BP 神经网络的输入正向传播、误差反向传播算法、BP 神经网络算法的缺陷及改进方法，还有 BP 神经网络在设计时，网络层数、神经元节点个数、各层间初始连接权值和学习速率的选取。

5. 集群系统结构

5.1 集群技术基础

集群技术是一种较新的技术，通过集群技术，可以在付出较低成本的情况下获得在性能、可靠性、灵活性方面的相对较高的收益，其任务调度则是集群系统中的核心技术。它将多个服务器连接到一起，使多台服务器能够像一台机器那样工作或者看起来好像一台机器。采用集群系统通常是为了提高系统的稳定性和网络中心的数据处理能力及服务能力。

集群系统是一组相互独立的、通过高速网络互联的计算机，它们构成了一个组，并以单一系统的模式加以管理。一个客户与集群相互作用时，集群像是一个独立的服务器。集群配置是用于提高可用性和可缩放性。

5.1.1 使用集群技术的目的

1) 提高性能

一些计算密集型应用，如：天气预报、核试验模拟等，需要计算机要有很强的运算处理能力，现有的技术，即使普通的大型机器计算也很难胜任。这时，一般都使用计算机集群技术，集中几十台甚至上百台计算机的运算能力来满足要求。提高处理性能一直是集群技术研究的一个重要目标之一。

2) 降低成本

通常一套较好的集群配置，其软硬件开销要超过 100000 美元。但与价值上百万美元的专用超级计算机相比已属相当便宜。在达到同样性能的条件下，采用计算机集群比采用同等运算能力的大型计算机具有更高的性价比。

3) 提高可扩展性

用户若想扩展系统能力，不得不购买更高性能的服务器，才能获得额外所需的 CPU 和存储器。如果采用集群技术，则只需要将新的服务器加入集群中即可，对于客户来看，服务无论从连续性还是性能上都几乎没有变化，好像系统在不知不觉中完成了升级。

4) 增强可靠性

集群技术使系统在故障发生时仍可以继续工作，将系统停运时间减到最小。集群系统在提高系统的可靠性的同时，也大大减小了故障损失。

5.1.2 集群的分类

1) 负载均衡集群

负载均衡集群为企业需求提供了更实用的系统。负载均衡集群使负载可以在计算机集群中尽可能平均地分摊处理。负载通常包括应用程序处理负载和网络流量负载。这样的系统非常适合向使用同一组应用程序的大量用户提供服务。每个节点都可以承担一定的处理负载，并且可以实现处理负载在节点之间的动态分配，以实现负载均衡。对于网络流量负载,当网络服务程序接受了高入网流量，以致无法迅速处理，这时，网络流量就会发送给在其它节点上运行的网络服务程序。同时，还可以根据每个节点上不同的可用资源或网络的特殊环境来进行优化。与科学计算集群一样，负载均衡集群也在多节点之间分发计算处理负载。它们之间的最大区别在于缺少跨节点运行的单并行程序。大多数情况下，负载均衡集群中的每个节点都是运行单独软件的独立系统。

但是，不管是在节点之间进行直接通信，还是通过中央负载均衡服务器来控制每个节点的负载,在节点之间都有一种公共关系。通常，使用特定的算法来分发该负载。

2) 科学集群

科学集群是并行计算的基础。通常，科学集群涉及为集群开发的并行应用程序，以解决复杂的科学问题。科学集群对外就好像一个超级计算机，这种超级计算机内部由十至上万个独立处理器组成，并且在公共消息传递层上进行通信以运行并行应用程序。

3) 高可用性集群

当集群中的一个系统发生故障时，集群软件迅速做出反应，将该系统的任务分配到集群中其它正在工作的系统上执行。考虑到计算机硬件和软件的易错性，高可用性集群的主要目的是为了使集群的整体服务尽可能可用。如果高可用性集群中的主节点发生了故障，那么这段时间内将由次节点代替它。次节点通常是主节点的镜像。当它代替主节点时，它可以完全接管其身份，因此使系统环境对于用户是一致的。

高可用性集群使服务器系统的运行速度和响应速度尽可能快。它们经常利用在多台机器上运行的冗余节点和服务，用来相互跟踪。如果某个节点失败，它的替补者将在几秒钟或更短时间内接管它的职责。因此，对于用户而言,集群永远不会停机。

5.1.3 集群技术的层次

1) 网络层：网络互联结构、通信协议、信号技术等。

2) 节点机及操作系统层高性能客户机、分层或基于微内核的操作系统等。

3) 集群系统管理层：资源管理、资源调度、负载平衡、并行 IPO、安全等。

4) 应用层：并行程序开发环境、串行应用、并行应用等。

集群技术是以上四个层次的有机结合，所有的相关技术虽然解决的问题不同，但都有其不可或缺的重要性。

集群系统管理层是集群系统所特有的功能与技术的体现。在未来按需计算的时代，每个集群都应成为业务网格中的一个节点，所以自治性（自我保护、自我配置、自我优化、自我治疗）也将成为集群的一个重要特征。自治性的实现，各种应用的开发与运行，大部分直接依赖于集群的系统管理层。此外，系统管理层的完善程度，决定着集群系统的易用性、稳定性、可扩展性等诸多关键参数。正是集群管理系统将多台机器组织起来，使之可以被称为“集群”。

5.2 集群服务器

集群技术是将若干个松散连接的独立的服务器架构成具有高可靠性和可扩展的集群服务器。集群的内部结构对客户是透明的，客户端只看到一个高性能的服务器端。当集群中一个节点发生故障时，集群软件迅速做出反应，将故障节点移出系统，直到故障解除。当总体的工作流量超出集群的承受能力时，可以向集群中透明地增加节点。实际服务器是通过高速的 LAN 或物理上分散的 WAN 相连，其前端是负载均衡器。由负载均衡器将客户端的请求分发给后面实际的服务器,每个实际的服务器都有自己的物理地址,但客户端只能看到负载均衡器的 IP 地址。这样多个实际的服务器提供的服务在外界看起来好像是由一个高性能的服务器提供的一样。

在集群服务器中，所有的计算机拥有一个共同的名称，集群内任意系统上运行的服务可被所有的网络客户所使用。集群必须可以协调管理各分离组件的错误和失败，并可透明的向集群中加入组件。用户的公共数据被放置到了共享的磁盘柜中，应用程序被安装到了所有的服务器上，也就是说，在集群上运行的应用需要在所有的服务器上安装一遍。当集群系统在正常运转时，应用只在一台服务器上运行，并且只有这台服务器才能操纵该应用在共享磁盘柜上的数据区，其它的服务器监控这台服务器，只要这台服务器上的应用停止运行（无论是硬件损坏、操作系统死机、应用软件故障，还是人为误操作造成的应用停止运行），其它的服务器就会接管这台服务器所运行的应用，并将共享磁盘柜上的相应数据区接管过来。

5.2.1 集群的方法

服务器集群系统通俗地讲就是把多台服务器通过快速通信链路连接起来，从外部看来，这些服务器就像一台服务器在工作，而对内来说，外面来的负载通过一定的机制动态地分配到这些节点机中去，从而达到超级服务器才有的高性能、高可用。采用集群技术进行服务器集群时有两种方法：

1) 将备份服务器连接在主服务器上，当主服务器发生故障时，备份服务器才投入运行，把主服务器上所有任务接管过来。

2) 将多台服务器连接，这些服务器一起分担同样的应用和数据计算任务，改善关键大型应用的响应时间。同时，每台服务器还承担一些容错任务，一旦某台服务器出现故障时，系统可以在系统软件的支持下，将这台服务器与系统隔离，并通过各服务器的负载转嫁机制完成新的负载分配。

5.2.2 服务器集群的好处

1) 高可伸缩性

服务器集群具有很强的可伸缩性。随着需求和负荷的增长，可以向集群系统添加更多的服务器。在这样的配置中，可以有多台服务器执行相同的应用和数据库操作。

2) 高可用性

高可用性是指在不需要操作者干预的情况下，防止系统发生故障或从故障中自动恢复的能力。通过把故障服务器上的应用程序转移到备份服务器上运行，集群系统能够把正常运行时间提高到大于 99.9%，大大减少服务器和应用程序的停机时间。

3) 高可管理性

系统管理员可以从远程管理一个、甚至一组集群，就好像在单机系统中一样。

5.2.3 服务器集群的不足

如果集群中的应用只在一台服务器上运行，且刚好这个应用出现故障，其它的某台服务器会重新启动这个应用，接管位于共享磁盘柜上的数据区，进而使应用重新正常运转。整个应用的接管过程大体需要三个步骤：侦测并确认故障、后备服务器重新启动该应用、接管共享的数据区。因此在切换的过程中需要花费一定的时间，原则上根据应用的大小不同切换的时间也会不同，越大的应用切换的时间越长。

5.2.4 影响服务器负载的因素

1) 通用服务器性能指标

表 5-1

指标	说明
ProcessorTime	服务器 CPU 占用率，一般平均达到 70%时，服务就接近饱和
Memory Available Mbyte	可用内存数，如果测试时发现内存有变化情况也要注意，如果是内存泄露则比较严重
Physicsdisk Time	物理磁盘读写时间情况

2) Web 服务器性能指标

表 5-2

指标	说明
Requests Per Second(Avg Rps)	平均每秒钟响应次数＝总请求时间 /秒数
Avg time to last byte per test(mes)	平均每秒业务脚本的迭代次数 有人会把上面那个混淆
Successful Rounds	成功的请求
Failed Requests	失败的请求
Successful Hits	成功的点击次数
Failed Hits	失败的点击次数
Hits Per Second	每秒点击次数
Successful Hits Per Second	每秒成功的点击次数
Failed Hits Per Second	每秒失败的点击次数
Attempted Connections	尝试链接数

3) 数据库服务器性能指标

表 5-3

指标	说明
User 0 Connections	用户连接数，也就是数据库的连接数量
Number of deadlocks	数据库死锁
Butter Cache hit	数据库 Cache 的命中情况

服务器性能指标即影响服务器负载的因素。将这些因素作为神经网络的输入，负载情况作为神经网络的输出对神经网络进行训练，然后就可以使用已训练的神经网络对服务器负载进行预测估计。将未来某时刻的这些因素作为神经网络的输入，得到的输出就是对未来该时刻服务器负载的预测。

5.3 负载均衡

负载均衡技术基于现有网络结构，提供了一种扩展服务器带宽和服务器吞吐量的方法，加强了网络数据的处理能力，提高了网络的灵活性和可用性。负载均衡的应用，能够有效地解决网络拥塞问题，能够就近提供服务，还能实现与地理位置无关的异地负载均衡。同时，这项技术还能提高服务器的响应速度，提高服务器及其它资源的利用效率，避免网络关键部位出现单点失效，从而为用户提供更好的访问质量。

5.3.1 负载均衡的技术

1） DNS 负载均衡

它是最早的负载均衡技术，通过 DNS 来实现，在 DNS 中为多个地址配置同一个名字，因而查询这个名字的客户机将得到其中某一个地址，从而使得不同的客户访问不同的服务器，达到负载均衡的目的。DNS 负载均衡是一种简单而有效的方法，但是它不能区分服务器的差异，也不能反映服务器的当前运行状态。

2） 代理服务器负载均衡

使用代理服务器，可以将请求转发给内部的服务器，使用这种加速模式显然可以提升静态网页的访问速度。然而，也可以考虑这样一种技术，使用代理服务器将请求均匀转发给多台服务器，从而达到负载均衡的目的。

3） 地址转换网关负载均衡

支持负载均衡的地址转换网关，可以将一个外部 IP 地址映射为多个内部 IP 地址，对每次 TCP 连接请求动态使用其中一个内部地址，达到负载均衡的目的。

4） 协议内部支持负载均衡

除了这三种负载均衡方式之外，有的协议内部支持与负载均衡相关的功能，例如 HTTP 协议中的重定向能力等，HTTP 运行于 TCP 连接的最高层。

5） 反向代理负载均衡

普通代理方式是代理内部网络用户访问 internet 上服务器的连接请求，客户端必须指定代理服务器，并将本来要直接发送到 internet 上服务器的连接请求发送给代理服务器处理。反向代理（Reverse Proxy）方式是指以代理服务器来接受 internet 上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给 internet 上请求连接的客户端，此时代理服务器对外就表现为一个服务器。

6） 混合型负载均衡

在有些大型网络，由于多个服务器群内硬件设备、各自的规模、提供的服务等的差异，可以考虑给每个服务器群采用最合适的负载均衡方式，然后又在这多个服务器群间再一次负载均衡或群集起来以一个整体向外界提供服务（即把这多个服务器群当做一个新的服务器群），从而达到最佳的性能。此种方式有时也用于单台均衡设备的性能不能满足大量连接请求的情况下。

5.3.2 负载均衡的分类

负载均衡从应用的地理结构上分为本地负载均衡（Local Load Balance）和全局负载均衡（Global Load Balance）。

1） 本地负载均衡

能有效地解决数据流量过大、网络负荷过重的问题，并且不需花费昂贵开支购置性能卓越的服务器，充分利用现有设备，避免服务器单点故障造成数据流量的损失。其有灵活多样的均衡策略把数据流量合理地分配给集群内的服务器共同负担。即使是再给现有服务器扩充升级，也只是简单地增加一个新的服务器到集群中，而不需改变现有网络结构、停止现有的服务。它关注一个地理位置上的设备群，是微观的。

2） 全局负载均衡

主要用于在一个多区域拥有自己服务器的站点，为了使全球用户只以一个 IP 地址或域名就能访问到离自己最近的服务器，从而获得最快的访问速度。它关注的是一个网络的整体，是对放在不同位置、为完成同一个任务的设备群体做负载均衡，是宏观的。全局负载均衡有以下的特点：

①实现地理位置无关性，能够远距离为用户提供完全的透明服务。

②除了能避免服务器、数据中心等的单点失效，也能避免由于 ISP 专线故障引起的单点失效。

③解决网络拥塞问题，提高服务器响应速度，服务就近提供，达到更好的访问质量。

5.4 本章小结

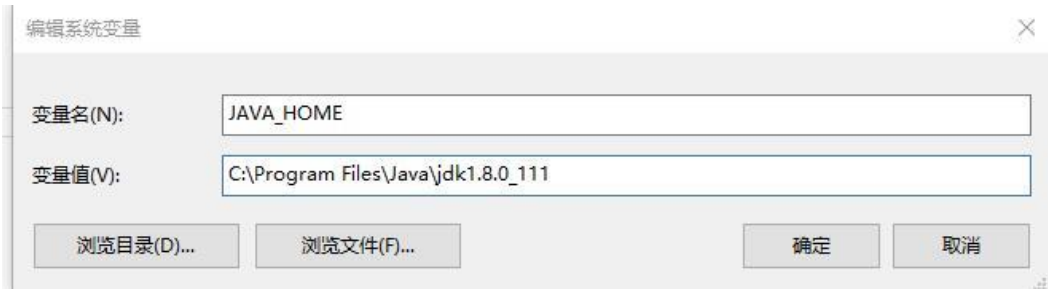
本章介绍了集群技术的一些基础知识，如集群技术的概念、分类、技术层次和发展趋势。另外还介绍了集群服务器即集群系统的集群方法、优缺点和服务器的性能指标。集群服务器通过负载均衡技术将整体负载相对平均的分配给每个独立服务器，使这些独立的服务器共同对数据进行处理。介绍了几种较为常用的负载均衡技术以及负载均衡的分类。

6. 模型可行性实验验证

6.1 搭建实验环境

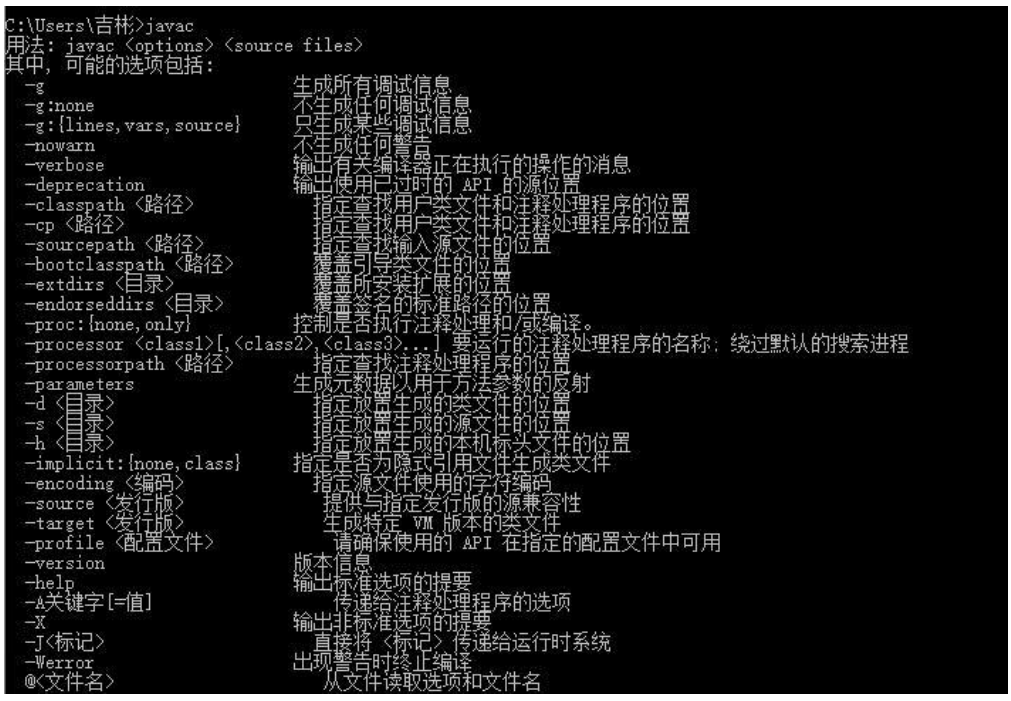
在 Windows10 64 位操作系统上安装 JDK1.8，默认安装到系统盘即可。并通过计算机的属性配置环境变量。

1） 首先新增系统变量



2） 修改系统变量里面的 path 变量，将下图中两项加入到 path 变量中

3） 在命令行中验证 java 环境是否搭建成功以及查看 JDK 版本



由上图可以看出 JDK 版本是 1.8.0_162。javac 是 java 文件编译指令，由上图可看出 java 编译指令可用，说明 java 环境搭建成功。

4） 在以上成功的基础上安装 IDEA 或 Eclipse 等 java 编译器。

5） 将第 4 章的算法进行代码实现。

6.2 实验过程

训练数据 1 如下：

```
double[][] in = {
        {1,1,1}, {1,0,0}, {0,1,0},
        {0,0,1}, {1,1,0}, {1,0,1}};
double[] out = {0.6,0.3,0.2,0.1,0.5,0.4};
```

测试数据 1 如下：

```
double[][] testIn = {
        {0,1,1}
};
double[] testOut = {0.3};
```

针对于训练数据 1 和测试数据 1 的实验结果如下：

```
第100次训练error:0.002891141754364827
实际输出：
0.565710619360186 0.3202282273519116 0.2124590692362083 0.13265424138309442 0.505646039357038 0.39273410012848375
------------------------------------------------------------
第200次训练error:4.0432460151476265E-4
实际输出：
0.591686648673651 0.2943635775785608 0.1912940646569926 0.11042944167225872 0.5100593194611697 0.3957944011469219
------------------------------------------------------------
第300次训练error:2.821147601389264E-4
实际输出：
0.5935994072181997 0.2957209623295968 0.19336085113523835 0.10893501885113825 0.5097346269534733 0.3979601011412785
------------------------------------------------------------
第400次训练error:1.8890397790609175E-4
实际输出：
0.5943445358410315 0.2961961698766064 0.19469255842721214 0.10709994831097111 0.5078643412984873 0.3985771460654844
------------------------------------------------------------
第500次训练error:1.1544020912236853E-4
实际输出：
0.5952011493012579 0.2967358144818791 0.1959146863787293 0.10534909362955204 0.5059550000002196 0.3990041491314606
------------------------------------------------------------
第600次训练error:6.353117284330857E-5
实际输出：
0.5963233592956372 0.297486053715907 0.19710316958483043 0.10387606304387238 0.5044788543655143 0.3995331676390666
------------------------------------------------------------
第700次训练error:3.402288942700614E-5
实际输出：
0.596867149538641 0.29780050381564105 0.19781270559401964 0.10253638565617168 0.5028024682672306 0.3994531759472572
------------------------------------------------------------
第800次训练error:1.7724369458730108E-5
实际输出：
0.5974784843741681 0.2981951263108133 0.1984103931552492 0.10158132055685602 0.5016794942697745 0.3994894769869265
------------------------------------------------------------
第900次训练error:8.276057355286922E-6
实际输出：
0.5982102476780617 0.29870951716505667 0.19896293025443854 0.10098979884204358 0.5011319027348139 0.399733393870716
------------------------------------------------------------
第1000次训练error:3.3690693866300593E-6
实际输出：
0.5989378838423344 0.29924072808443003 0.19943941983050226 0.10065829884016325 0.500954845002438 0.4000717519644828
------------------------------------------------------------
```

```
第19800次训练error:7.954090670319284E-30
实际输出:
0.5999999999999983 0.29999999999999893 0.20000000000000026 0.0999999999999989 0.500000000000001 0.40000000000000135
------------------------------------------------------------
第19900次训练error:7.954090670319284E-30
实际输出:
0.5999999999999983 0.29999999999999893 0.20000000000000026 0.0999999999999989 0.500000000000001 0.40000000000000135
------------------------------------------------------------
第20000次训练error:7.954090670319284E-30
实际输出:
0.5999999999999983 0.29999999999999893 0.20000000000000026 0.0999999999999989 0.500000000000001 0.40000000000000135
------------------------------------------------------------
预测输出:
0.2993326702352521
期望输出:
0.3
```

根据实验结果可以看出在训练过程中误差呈现递减规律，即实际输出在逐步的接近期望输出。在最后对神经网络给定一个输入，得到的输出也非常接近我们想要的期望输出，说明神经网络训练成功。接下来我们再训练一组数据并进行预测，用以说明改进后的反向传播神经网络算法的正确性。

训练数据 2 如下：

```
double[][] in = {
        {10,10,10}, {3,7,9}, {1,2,3},
        {6,4,2}, {4,0,6},{1,4,7},{5,1,3}};
double[] out = {0.90,0.69,0.22,0.28,0.34,0.48,0.23};
```

测试数据 2 如下：

```
double[][] testIn = {
        {2,5,8}
};
double[] testOut = {0.57};
```

针对于训练数据 2 和测试数据 2 的实验结果如下：

```
第100次训练error:0.40018235833031357
实际输出:
0.46418926979655484 0.4641881681509144 0.45622541008166695 0.46408013816154803 0.4627156055454425 0.464036441598670
------------------------------------------------------------
第200次训练error:0.4041493834101778
实际输出:
0.4618337031893006 0.46183384300154473 0.46175065426335404 0.46181726133259937 0.46175074937277233 0.461842763358152
------------------------------------------------------------
第300次训练error:0.6592309303521547
实际输出:
0.6444021895909905 0.6444021740693768 0.6411795430685262 0.6439254800112025 0.6420586949017468 0.6443957332690858 0.
------------------------------------------------------------
第400次训练error:0.2115007175226004
实际输出:
0.6088317314251008 0.6101814877398714 0.49730222788872147 0.43042657229407916 0.3767817152573589 0.5958654664320678 0
------------------------------------------------------------
第500次训练error:0.15121065919933308
实际输出:
0.6085688534952769 0.6109720649745344 0.42259207454625786 0.3760600359580159 0.31874579499513367 0.5680369534275399 0
------------------------------------------------------------
```

```
第600次训练error:0.12378614054073347
实际输出:
0.6320363618507758 0.6271979749366058 0.3883293197681765 0.37901033684981306 0.3248561188552049 0.5712126903019785 0.
------------------------------------------------------------
第700次训练error:0.07083548094727074
实际输出:
0.7037542771581525 0.6540657539626794 0.3545384757250012 0.36191469817840544 0.34335444529385856 0.5437698311493088 0
------------------------------------------------------------
第800次训练error:0.051187562312719405
实际输出:
0.7286368528268439 0.6487542106669428 0.349645733474067 0.32034366654334856 0.32536068241130306 0.5059251548290518 0.
------------------------------------------------------------
第900次训练error:0.043372223380002725
实际输出:
0.7566090718065764 0.6652007809981036 0.3535731542633473 0.3274561613344191 0.3410523604188423 0.498889191076312 0.27
------------------------------------------------------------
第1000次训练error:0.035515250330983544
实际输出:
0.7664874691552765 0.6681295893693562 0.34198761701462005 0.31539380864474204 0.33771869246725567 0.48760721729825923
------------------------------------------------------------
第19800次训练error:5.697924353421087E-4
实际输出:
0.8823457784662301 0.6996143805059288 0.22835979419188723 0.28263991076965544 0.34299091982244573 0.4713605163842652
------------------------------------------------------------
第19900次训练error:5.87211406033326E-4
实际输出:
0.8829883431206935 0.7014461983052656 0.22911257183303466 0.28392151789311293 0.34458706193383254 0.4732278543490208
------------------------------------------------------------
第20000次训练error:5.693723973565993E-4
实际输出:
0.8827305673427569 0.7004221246483374 0.22866361089307444 0.28321838489901435 0.34373353851732336 0.4722559191950668
------------------------------------------------------------
预测输出:
0.5697955046062063
期望输出:
0.57
```

根据上述两组数据的实验结果，我们能够充分的说明改进后的神经网络方向传播算法的正确性。

6.3 本章小结

本章通过实验的方式来验证改进后的神经网络反向传播算法的正确性。通过两组实验数据对应的实验结果分析，训练后的神经网络已经可以进行预测。

7. 总结与体会

7.1 总结

近年来，随着人工智能、机器学习的火热，神经网络被应用于模式识别、信号处理、图像处理、自动控制、工程、医学、金融等许多实际应用领域中取得了显著的成效。

经过研究发现，标准 BP 算法中的学习速率的含义是表示各层间连接权值的调整幅度，学习速率太低导致训练学习的速度缓慢、训练时间延长；学习速率太高则会导致神经网络在更新迭代过程中围绕误差最小值来回跳动，从而产生振荡，网络会变得发散而不能收敛到稳定值。所以在标准 BP 算法中一般倾向于选择较小的学习速率值，用训练时间来换取系统的稳定性。

因为学习速率仅仅表示神经网络各层间连接权值的调整幅度，所以在对标准 BP 算法进行改进时，我们可以动态的去修正学习速率的值，使得神经网络各层间的连接权值的调整幅度随着前后两次训练的输出误差的相对大小而改变。当前的训练误差小于前一次训练误差时，表示神经网络训练的发展方向是沿着我们想要的误差减小的方向发展的，此时我们可以适当的提升学习速率的值，进而提高训练的速度；当前的训练误差相对大于前一次训练误差时，表示神经网络训练的发展方向是沿着误差增大的方向发展的，这是我们不想看到的结果，此时我们可以适当的减小学习速率的值，使得在以后的训练中将训练方向拉回我们想要的误差减小的方向。当前的训练误差相对等于前一次的训练误差时，说明神经网络的发展处于稳定发展阶段，此时无需对学习速率进行调整，就让神经网络沿着稳定发展方向进行发展。

在神经网络的设计中，网络层数、每层神经元个数、初始权值和学习速率不是随便选取的。网络层数一般选取三层网络结构（输入层、隐藏层和输出层），只包含一个隐含层的网络是通用的函数逼近器，如果在训练过程中发现一个隐含层不足以很好的解决问题时，可以考虑适当的添加隐含层的个数。输入层是神经网络与外界连接的纽带，对于输入层神经元个数的选取取决于输入数据的维数，在进行负载预测时，我们将影响负载的因素作为神经网络的输入，因此输入层神经元的个数为影响负载的因素的个数。输出层神经元个数就等于我们训练预测时所需的信息类别数。隐含层的神经元个数太少导致神经网络从训练样本中学习的能力不足，网络很容易陷入局部极小值点，进而导致可能得不到稳定的结果；神经元个数太多导致出现“过拟合”现象、训练时间延长；虽然隐含层神经元个数存在最优值，但精确的找到最优值有很大难度，因此需要不断的进行试验去调整隐含层神经元的个数，进而确定最终的隐含层神经元个数。

7.2 体会

随着毕业日子的到来，毕业设计也接近了尾声。经过近一个学期的奋战我的毕业设计终于完成了。在没有做毕业设计之前，感觉毕业设计只是对大学这 4 年来所学知识的总结，但是通过这次毕业设计发现自己的看法有点太片面。毕业设计不仅是对前面所学知识的一种检验，而且也是对自己能力的一种提高。通过这次毕业设计使我明白了自己知识欠缺的还很多。要学习的东西还有很多，以前老是觉得自己什么都会，什么都懂，有点眼高手低。通过这次毕业设计，我才明白学习是一个长期积累的过程，在以后的工作、生活中都应该不断的学习，努力提高自己知识和综合素质。

在这次毕业设计中也使我和同学们的友谊更进了一步，同学之间互相帮助，有什么不懂的大家在一起讨论，听听不同的看法可以使我们更好的理解知识，促使大家共同进步，共同学习。

致谢

光阴似箭，日月如梭。大学四年的时间，在我们漫长的人生旅途中是那么的短暂。四年的大学生活将在这个季节划上一个句号，但对我而言这又是另一个生活的开始，我将带着学生时代的所学去闯荡社会，或许一帆风顺，或许道路崎岖，但不管如何，我都不会忘记那个曾经教育过我的地方和人。那里是我不想离开但又不得不离开的地方—校园，我之所以不想离开，那是因为在那里我有一群嬉戏打闹但又勤奋好学的同学、环境优雅安静无比的图书馆自习室、兢兢业业和蔼可亲的老师。

大学是一个学习知识，自我能力提升以及广交朋友的地方，大学里学生和老师不仅是师生关系，更是朋友关系。因为他们在学习和做人方面教给了我们很多东西，在生活方面像家人像朋友一样的关心我们。我感谢所有的恩师：是您赋予我们最有意义的收 获；是您带领我们走进知识殿堂，使我们不但丰富了知识；是您给我们一个全新 的角度去发现美、创造美、欣赏美，给我们美的眼睛去发现世界的美，感悟生活 的美；是你教会我们珍惜友谊和时间；是您给了我们看世界的眼睛，是你们用博 大的胸怀，给予我们最无私的关怀和奉献。我要感谢我的指导老师，还有我的班主任老师，以及任课老师，感谢他们的教诲，让我知道在社会上懂得怎样去做好自己，端正自己的位置，为社会贡献出我自己的力量。

参考文献

[1] Fabio M.Soares，Alan M.F Souza，神经网络算法与实现，人民邮电出版社，2017.09
[2] 朱大奇.人工神经网络研究现状及其展望[N].江南大学学报,2004:103～108.
[3] 董军,胡上序.混沌神经网络研究进展和展望[J].信息与控制,1997 (5):360～368.
[4] 韩力群.人工神经网络理论、设计及应用[M].北京:化学工业出版社,2002.
[5] 李庆华，柳笛，张凯丽. 基于 BP 神经网络和支持向量机的铝板表面缺陷分类方法:CN104766097A[P].2015.
[6] 马锐.人工神经网络原理[M].北京:机械工业出版社.2010.
[7] 胡守仁,余少波,戴葵.神经网络导[M].长沙:国防科技大学出版社,1992.
[8] 胡金滨,唐旭清.人工神经网络的 BP 算法及其应用[J].信息技术,2004 (4):1～4.
[9] 宋桂荣.改进 BP 算法在故障诊断中的应用[N].沈阳工业大学学报,2001 (3):252～254.
[10] 智会强,牛坤,田亮等.BP 网络和 RBF 网络在函数逼近领域内的比较研究[J].科技通报,2005 (2).
[11] 施彦，韩力群，廉小亲. 神经网络设计方法与实例分析［M］. 北京：北京邮电大学出版社，2009.
[12] 韩力群. 人工神经网络教程［M］. 北京：机械工业出版社. 2006.
[13] 薛正华，董小社，李炳毅等. 基于ＢＰ神经网络的集群负载预测器［Ｊ］. 华中科技大学学报：自然科学版，2007.
[14] 焦李成.神经网络系统理论[M].西安：西安电子科技大学出版社，1990.
[15] 沉清，胡德文，时春.神经网络应用技术[M].长沙：国防科技大学出版社，1993.
[16] 张立明.人工神经网络的模型及其应用[M].上海：复旦大学出版社，1993.
[17] 李学桥，马莉.神经网络工程应用[M].重庆：重庆大学出版社，1996.
[18] 蔡宗礼.人工神经网络[M].北京：高等教育出版社，2001.
[19] 吴微.神经网络计算[M].北京：高等教育出版社，2003.
[20] 韩力群.人工神经网络教程[M].北京：北京邮电大学出版社，2006.
[21] 马锐.人工神经网络原理[M].北京 L 机械工业出版社，2010.
[22] 王霜，修保新，肖卫东.Web 服务器集群的负载均衡算法研究[J].计算机工程与应用.2004(25).
[23] 刘健，徐磊，张维明.基于动态反馈的负载均衡算法[J].计算机工程与科学. 2003(25). 65-67
[24] 吴家祺.Web 服务器集群系统的设计与实现[D].南京航空航天大学论文.2005. 53-54
[25] 张洪武.服务器集群与均衡技术研究[D].重庆大学论文.2004
[26] 李长志.集群服务器系统负载均衡原理的分析与实现[J].重庆邮电学院学报.2004(6)
[27] 李庆华,郭志鑫.一种面向工作站网络的系统负载预测方法.华中科技大学学报(自然科学版),2002.
[28] 张建军,蒋廷耀,郭志鑫.PVM 中动态负载平衡的设计和实现[J].计算机工程,2005.
[29] 许建峰,朱晴波,胡宁等分布式实时系统中的预测调度算法[J].软件学报,2000.
[30] Rich Wolski et al. Predicting the CPU availability of time-shared Unix systems[C].In:HPDC'99
[31] Wu Yong-wei, Hwang Kai, Yuan Yu-lai, et al.. Adaptive workload prediction of grid performance in confidence windows[J]. IEEE Transactions on Parallel and Distributed Systems, 2010, 21(7): 925-938.
[32] Gmach D, Rolia J, Cherkasova L, et al.. Workload analysis and demand prediction of enterprise data center applications[C]. Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization, Washington, 2007: 171-180.
[33] Ganapathi A, Chen Y, Fox A, et al.. Statistics-driven workload modeling for the cloud[C]. Proceedings of Workshop on Self-Managing Database Systems (SMDB 2010), California, 2010: 87-92.
[34] Khan A, Yan Xi-feng, Tao Shu, et al.. Workload characterization and prediction in the cloud: a multiple time series approach[C]. Proceedings of 3rd International Workshop on Cloud Management (CloudMan 2012), HAWAII, 2012: 1287-1294.
[35] W.James.Principles of Psychology[C],New York,1890.
[36] W.S.McCulloh,W.Pitts.Bulletin of Mathematical Biophysic[C],1943.
[37] D.O.Hebb. The Organization of Behavior: a Neuropsycholodical Theory[N].1949.
[38] B.Widrow,M.Hoff.Adaptive Switching Circuits[M].1960.

附录

神经网络的代码实现（使用 java 语言编写）：

代码下载地址 https://github.com/pingyuangulang/BP

MyBP.java 文件，包含内容：模拟神经网络以及神经网络的训练和预测操作。

```
package com.Network;
import com.Exceptions.MatrixException;
import com.Functions.Function;
import com.Matrix.Matrix;
/**
 * 模拟 BP 神经网络，3 层结构（输入层、隐含层和输出层）。
 * @author JiBin
 * @date 2018/6/9 2:05
 */
public class MyBP {
    //训练时的实际输出。
    private double[] o;
    //训练时的输入。
    private double[][] trainInput;
    //期望输出。
    private double[] output;
   //输入层与隐含层之间的连接权值矩阵。
    private double[][] inputHiddenWeight;
    //隐含层与输出层之间的连接权值矩阵。
    private double[][] hiddenOutputWeight;
    //前一次的训练误差平方和。
    private double preErrorSum;
    //当前训练误差平方和。
    private double curErrorSum;
     //当前训练误差。
    private double[] curOutError;
    //学习速率。
    private double lr;
    /**
     * 带参构造器，初始化网络
     * @param trainInput 对网络进行训练学习时的输入。
     * @param output 训练输入对应的输出。
     * @param lr 学习速率。
     * @param hidden 隐含层神经元节点的个数。
     */
    public MyBP(double[][] trainInput, double[] output, double lr, int hidden) {
        this.trainInput = trainInput;
        this.output = output;
        this.lr = lr;
        //设置权值矩阵的行和列数
        this.inputHiddenWeight = new double[trainInput[0].length][hidden];
```

```
        this.hiddenOutputWeight = new double[hidden][1];
        //初始化前一次的训练误差平方和为0
        this.preErrorSum = 0.0;
        //初始化权值矩阵
        InitWeight();
    }
    /**
     * 初始化两个权值矩阵（输入--隐含权值矩阵和隐含--输出权值矩阵）。
     */
    private void InitWeight(){
        //初始化输入层与隐含层之间的连接权值矩阵，取值范围（-0.5,0.5]
        for (int i=0;i<inputHiddenWeight.length;i++){
            for (int j=0;j<inputHiddenWeight[0].length;j++){
                inputHiddenWeight[i][j] = 0.5 - Math.random();
            }
        }
        //初始化隐含层与输出层之间的连接权值矩阵，取值范围（-0.5,0.5]
        for (int i=0;i<hiddenOutputWeight.length;i++){
            for (int j=0;j<hiddenOutputWeight[0].length;j++){
                hiddenOutputWeight[i][j] = 0.5 - Math.random();
            }
        }
    }
    /**
     * 计算当前训练的输出误差及误差平方和。
     * @param curOutput 当前训练得到的训练输出。
     */
    private void calculateError(double[] curOutput){
        try {
            curOutError = Matrix.matrixSub(output,curOutput );
        } catch (MatrixException e) {
            e.printStackTrace();
        }
        curErrorSum = Matrix.quadraticSum(curOutError);
    }
    /**
     * 根据上一次训练的误差平方和与当前训练的误差平方和的关系调整学习速率。
     */
    private void updateLr(){
        if (curErrorSum > 1.04 * preErrorSum){
            lr = 0.7 * lr;
        }
        else if (curErrorSum < preErrorSum){
            lr = 1.05 * lr;
        }
        else {
```

```
                lr = lr;
            }
            preErrorSum = curErrorSum;
        }
        /**
         * 单次训练，对神经网络只训练一次。
         * @throws MatrixException
         */
        public void train()throws MatrixException{
            //隐含层的输入
            double[][] hiddenIn = Matrix.matrixMultiple(trainInput, inputHiddenWeight);
            //隐含层的输出
            double[][] hiddenOut = Function.sigmoid(hiddenIn);
            //输出层的输入
            double[] outIn = Matrix.change(Matrix.matrixMultiple(hiddenOut,
hiddenOutputWeight));
            //输出层的输出
            double[] outOut = Function.sigmoid(outIn);
            o = outOut;
            //计算当前训练的误差（输出层的输出误差）
            calculateError(outOut);
            //修正学习速率
            updateLr();
            //输出层的输入误差
            double[] outInError = Matrix.matrixMultiple(Function.deSigmoid(outIn),
curOutError);
            //隐含层与输出层之间的权值误差
            double[][] hiddenOutWeightError
=Matrix.matrixMultiple(Matrix.matrixTranspose(hiddenOut), outInError);
            hiddenOutWeightError = Matrix.matrixMultiple(hiddenOutWeightError, lr);
            //调整隐含层与输出层之间的权值
            hiddenOutputWeight = Matrix.matrixAdd(hiddenOutputWeight,
hiddenOutWeightError, 1.0);
            //隐含层的输出误差
            double[][] hiddenOutError =
Matrix.matrixMultiple(Matrix.change(outInError),
Matrix.matrixTranspose(hiddenOutputWeight));
            //隐含层的输入误差
            double[][] hiddenInError = Matrix.matrixMultiple(hiddenOutError,
Function.deSigmoid(hiddenIn), true);
            //输入层与隐含层之间的权值误差
            double[][] inputHiddenWeightError =
Matrix.matrixMultiple(Matrix.matrixTranspose(trainInput), hiddenInError);
            inputHiddenWeightError = Matrix.matrixMultiple(inputHiddenWeightError, lr);
            //调整输入层与隐含层之间的权值
            inputHiddenWeight = Matrix.matrixAdd(inputHiddenWeight,
```

```
inputHiddenWeightError, 1.0);
    }
    /**
     * 多次训练，对神经网络训练所给的参数值次。
     * @param num 训练的次数。
     */
    public void train(int num)throws MatrixException{
        for (int i=1;i<=num;i++){
            train();
            if (i%100 == 0){
                System.out.println("第"+i+"次训练 error:"+curErrorSum);
                System.out.println("实际输出：");
                for (int j=0;j<o.length;j++){
                    System.out.print(o[j]+" ");
                }
                System.out.println();

System.out.println("------------------------------------------------------------
");
            }
        }
    }
    /**
     * 神经网络训练学习完毕后，给定一个输入，对输出进行预测。
     * @param in 输入层神经元的输入信息。
     * @param out 该输入对应的期望输出。
     * @throws MatrixException
     */
    public void forecase(double[][] in, double[] out)throws MatrixException{
        //隐含层的输入
        double[][] hiddenIn = Matrix.matrixMultiple(in, inputHiddenWeight);
        //隐含层的输出
        double[][] hiddenOut = Function.sigmoid(hiddenIn);
        //输出层的输入
        double[] outIn = Matrix.change(Matrix.matrixMultiple(hiddenOut,
hiddenOutputWeight));
        //输出层的输出
        double[] outOut = Function.sigmoid(outIn);
        System.out.println("预测输出：");
        for (int i=0;i<outOut.length;i++){
            System.out.print(outOut[i]+"    ");
        }
        System.out.println();
        System.out.println("期望输出：");
        for (int i=0;i<out.length;i++){
            System.out.print(out[i]+"    ");
```

```
        }
    }
}
```

Matrix.java 文件，包含内容：对矩阵的操作

```
package com.Matrix;
import com.Exceptions.MatrixException;
/**
 * 封装有关矩阵运算的操作工具类。
 * @author JiBin
 * @date 2018/6/9 1:32
 */
public class Matrix {
    /**
     * 两个二维矩阵相乘。
     * @param matrix1 第一个二维矩阵。
     * @param matrix2 第二个二维矩阵。
     * @return 两个二维矩阵相乘的结果。
     * @throws MatrixException
     */
    public static double[][] matrixMultiple(double[][] matrix1, double[][] matrix2)
throws MatrixException {
        int matrix1Row = matrix1.length;
        int matrix1Col = matrix1[0].length;
        int matrix2Row = matrix2.length;
        int matrix2Col = matrix2[0].length;
        if(matrix1Col!=matrix2Row){
            throw new MatrixException("matrix1 的列数与 matrix2 的行数不相等，因此
两矩阵不能相乘！");
        }
        double[][] result = new double[matrix1Row][matrix2Col];
        for(int i=0;i<matrix1Row;i++){
            for(int j=0;j<matrix2Col;j++){
                double temp = 0;
                for(int k=0;k<matrix1Col;k++){
                    temp+= matrix1[i][k]*matrix2[k][j];
                }
                result[i][j] = temp;
            }
        }
        return result;
    }
    /**
     * 二维矩阵与数相乘，二维矩阵中的每个元素与数相乘为对应位置元素值。
     * @param matrix 二维矩阵。
     * @param lr 数。
     * @return 二维矩阵与数相乘的结果。
```

```
     */
    public static double[][] matrixMultiple(double[][] matrix,double lr){
        int matrixRow = matrix.length;
        int matrixCol = matrix[0].length;
        double[][] result = new double[matrixRow][matrixCol];
        for(int i=0;i<matrixRow;i++){
            for(int j=0;j<matrixCol;j++){
                result[i][j] = matrix[i][j]*lr;
            }
        }
        return result;
    }
    /**
     * 两个一维矩阵按位相乘。
     * @param matrix1 第一个一维矩阵。
     * @param matrix2 第二个一维矩阵。
     * @return 相乘之后的一维结果矩阵。
     * @throws MatrixException
     */
    public static double[] matrixMultiple(double[] matrix1, double[] matrix2)throws
MatrixException{
        if (matrix1.length != matrix2.length){
            throw new MatrixException("两个一维矩阵长度不相等，因此不能按位相乘！
");
        }
        double[] result = new double[matrix1.length];
        for (int i=0;i<matrix1.length;i++){
            result[i] = matrix1[i] * matrix2[i];
        }
        return result;
    }
    /**
     * 二维矩阵与一维矩阵相乘。
     * @param matrix1 二维矩阵(m*n)。
     * @param matrix2 一维矩阵(n)，可以看成(n*1)的二维矩阵。
     * @return 只有一列的二维矩阵(m*1)。
     * @throws MatrixException
     */
    public static double[][] matrixMultiple(double[][] matrix1, double[]
matrix2)throws MatrixException{
        if (matrix1[0].length != matrix2.length){
            throw new MatrixException("二维矩阵的列数不等于一维矩阵的行数，因此不
能相乘！");
        }
        double[][] result = new double[matrix1.length][1];
        for (int i=0;i<matrix1.length;i++){
```

```
            double temp = 0.0;
            for (int j=0;j<matrix2.length;j++){
                temp = temp + matrix1[i][j] * matrix2[j];
            }
            result[i][0] = temp;
        }
        return result;
    }
    /**
     * 两个二维矩阵按位相乘，相乘的结果放在新矩阵的对应位置。
     * @param matrix1 第一个二维矩阵(m*n)。
     * @param matrix2 第二个二维矩阵(m*n)。
     * @param flag 标识，传值为 true/false/null 都行，为区别矩阵相乘函数。
     * @return 按位相乘后的二维结果矩阵(m*n)。
     * @throws MatrixException
     */
    public static double[][] matrixMultiple(double[][] matrix1, double[][] matrix2,
boolean flag)throws MatrixException{
        int matrix1Row = matrix1.length;
        int matrix1Col = matrix1[0].length;
        int matrix2Row = matrix2.length;
        int matrix2Col = matrix2[0].length;
        if (matrix1Row != matrix2Row || matrix1Col != matrix2Col){
            throw new MatrixException("两个二维矩阵的行或列数不相等，因此不能按位
相乘！");
        }
        double[][] result = new double[matrix1Row][matrix1Col];
        for (int i=0;i<matrix1Row;i++){
            for (int j=0;j<matrix1Col;j++){
                result[i][j] = matrix1[i][j] * matrix2[i][j];
            }
        }
        return result;
    }
    /**
     * 两个二维矩阵相加减，对应位置元素相加减。
     * @param matrix1 第一个二维矩阵(m*n)。
     * @param matrix2 第二个二维矩阵(m*n)。
     * @param flag 传-1 表示相减，其他值表示相加。
     * @return 相加减的二维结果矩阵。
     * @throws MatrixException
     */
    public static double[][] matrixAdd(double[][] matrix1, double[][] matrix2,
double flag)throws MatrixException{
        int matrix1Row = matrix1.length;
        int matrix1Col = matrix1[0].length;
```

```
        int matrix2Row = matrix2.length;
        int matrix2Col = matrix2[0].length;
        if (matrix1Row != matrix2Row || matrix1Col != matrix2Col){
            throw new MatrixException("matrix1 的行列数与 matrix2 的行列数不相等，
因此两矩阵不能相加减！");
        }
        if (flag == -1)
            matrix2 = matrixMultiple(matrix2, flag);
        double[][] result = new double[matrix1Row][matrix1Col];
        for (int i=0;i<matrix1Row;i++){
            for (int j=0;j<matrix1Col;j++){
                result[i][j] = matrix1[i][j] + matrix2[i][j];
            }
        }
        return result;
    }
    /**
     * 两个一维矩阵相减，对应位置元素相减的结果放在新矩阵的对应位置。
     * @param matrix1 第一个一维矩阵
     * @param matrix2 第二个一维矩阵
     * @return 两个一维矩阵相减的一维结果矩阵
     * @throws MatrixException
     */
    public static double[] matrixSub(double[] matrix1, double[] matrix2)throws
MatrixException{
        int matrix1Length = matrix1.length;
        int matrix2Length = matrix2.length;
        if (matrix1Length != matrix2Length){
            throw new MatrixException("matrix1 的长度与 matrix2 的长度不相等，因此
不能相加减！");
        }
        double[] result = new double[matrix1Length];
        for (int i=0;i<matrix1Length;i++){
            result[i] = matrix1[i] - matrix2[i];
        }
        return result;
    }
    /**
     * 二维矩阵转置，第 i 行变成第 i 列，第 j 列变成第 j 行。
     * @param matrix 需要被转置的二维矩阵。
     * @return 转置后的二维矩阵。
     */
    public static double[][] matrixTranspose(double[][] matrix){
        int matrixRow = matrix.length;
        int matrixCol = matrix[0].length;
        double[][] result = new double[matrixCol][matrixRow];
```

```
        for (int i=0;i<matrixRow;i++){
            for (int j=0;j<matrixCol;j++){
                result[j][i] = matrix[i][j];
            }
        }
        return result;
    }
    /**
     * 求一维矩阵中元素的平方和。
     * @param matrix 一维矩阵。
     * @return 一维矩阵中所有元素的平方和。
     */
    public static double quadraticSum(double[] matrix){
        double result = 0.0;
        for (int i=0;i<matrix.length;i++){
            result = result + Math.pow(matrix[i], 2);
        }
        return result;
    }
    /**
     * 将列数为 1 的二维矩阵转化成一维行矩阵。
     * @param matrix 二维矩阵。
     * @return 转化后的一维矩阵。
     * @throws MatrixException
     */
    public static double[] change(double[][] matrix)throws MatrixException {
        if (matrix[0].length != 1){
            throw new MatrixException("该二维矩阵的列数不等于 1，因此不能转换成一
维矩阵！");
        }
        double[] result = new double[matrix.length];
        for (int i=0;i<matrix.length;i++){
            result[i] = matrix[i][0];
        }
        return result;
    }
    /**
     * 一维行矩阵转化成只有一列二维矩阵。
     * @param matrix 一维行矩阵
     * @return 只有一列的二维矩阵。
     */
    public static double[][] change(double[] matrix){
        double[][] result = new double[matrix.length][1];
        for (int i=0;i<matrix.length;i++){
            result[i][0] = matrix[i];
        }
```

```
        return result;
    }
}
```

Function.java 文件，包含内容：单极性 S 型激活函数及导函数

```
package com.Functions;
/**
 * 封装激活函数及激活函数导函数的函数工具类。
 * @author JiBin
 * @date 2018/6/9 1:20
 */
public class Function {
    /**
     * 激活函数,单个处理。
     * @param x 激活函数的参数值。
     * @return 激活函数运算后的函数值。
     */
    public static double sigmoid(double x){
        double result = 0.0;
        result = 1d/(1d+Math.exp(-x));
        return result;
    }
    /**
     * 激活函数，批处理,一维矩阵。
     * @param x 需要传入激活函数进行运算的一维参数矩阵。
     * @return 激活函数运算后的一维函数值矩阵。
     */
    public static double[] sigmoid(double[] x){
        double[] result = new double[x.length];
        for (int i=0;i<x.length;i++){
            result[i] = sigmoid(x[i]);
        }
        return result;
    }
    /**
     * 激活函数，批处理,二维矩阵。
     * @param x 需要传入激活函数进行运算的二维参数矩阵。
     * @return 激活函数运算后的二维函数值矩阵。
     */
    public static double[][] sigmoid(double[][] x){
        double[][] result = new double[x.length][x[0].length];
        for (int i=0;i<x.length;i++){
            for (int j=0;j<x[0].length;j++){
                result[i][j] = sigmoid(x[i][j]);
            }
        }
        return result;
```

```
    }
    /**
     * 激活函数导函数，单个处理。
     * @param x 激活函数导函数的参数值。
     * @return 激活函数导函数运算后的函数值。
     */
    public static double deSigmoid(double x){
        double result = 0.0;
        result = (Math.exp(x))/(Math.pow(Math.exp(x)+1, 2));
        return result;
    }
    /**
     * 激活函数导函数，批处理，一维矩阵。
     * @param x 需要传入激活函数导函数进行运算的一维参数矩阵。
     * @return 激活函数导函数运算后的一维函数值矩阵。
     */
    public static double[] deSigmoid(double[] x){
        double[] result = new double[x.length];
        for (int i=0;i<x.length;i++){
            result[i] = deSigmoid(x[i]);
        }
        return result;
    }
    /**
     * 激活函数导函数，批处理，二维矩阵。
     * @param x 需要传入激活函数导函数进行运算的二维参数矩阵。
     * @return 激活函数导函数运算后的二维函数值矩阵。
     */
    public static double[][] deSigmoid(double[][] x){
        double[][] result = new double[x.length][x[0].length];
        for (int i=0;i<x.length;i++){
            for (int j=0;j<x[0].length;j++){
                result[i][j] = deSigmoid(x[i][j]);
            }
        }
        return result;
    }
}
```

MatrixException.java 文件，包含内容：异常信息类

```
package com.Exceptions;
/**
 * 自定义异常类
 * @author JiBin
 * @date 2018/6/9 1:14
 */
public class MatrixException extends Exception{
```

```
    //异常信息
    private String message;
    /**
     * 无参构造器
     */
    public MatrixException() {}
    /**
     * 传入异常信息的构造器
     * @param message 异常信息
     */
    public MatrixException(String message) {
        super(message);
        this.message = message;
    }
    /**
     * 获取异常信息
     * @return 异常信息
     */
    @Override
    public String getMessage() {
        return message;
    }
    /**
     * 设置异常信息
     * @param message 要设置的异常信息
     */
    public void setMessage(String message) {
        this.message = message;
    }
}
```

Test.java 文件，包含内容：对神经网络进行测试

```
package com.Test;
import com.Exceptions.MatrixException;
import com.Network.MyBP;
/**
 * 对神经网络进行运行测试的测试类。
 * @author JiBin
 * @date 2018/6/9 2:23
 */
public class Test {
    /**
     * 主函数，java 应用程序的入口。
     * @param args JVM 调用时传入的相关参数。
     */
    public static void main(String[] args) {
        double[][] in = {
```

```
                {1,1,1}, {1,0,0}, {0,1,0},
                {0,0,1}, {1,1,0}, {1,0,1}};
        double[] out = {0.6,0.3,0.2,0.1,0.5,0.4};
        MyBP myBP = new MyBP(in, out, 0.11, 6);
        try {
            myBP.train(20000);
        } catch (MatrixException e) {
            e.printStackTrace();
        }
        double[][] testIn = {
                {0,1,1}
        };
        double[] testOut = {0.3};
        try {
            myBP.forecase(testIn, testOut);
        } catch (MatrixException e) {
            e.printStackTrace();
        }
    }
}
```