南京大学

本 科 毕 业 设 计

院　　系 软件学院

专　　业 软件工程

题　　目 神经网络的自动化数值解析技术

年　　级 2014 级　学 号 141250016

学生姓名 陈之威

指导教师 汤恩义　职 称 讲师

提交日期

南京大学本科生毕业论文（设计）中文摘要

毕业论文题目： 神经网络的自动化数值解析技术

软件学院 院系 软件工程 专业 2014 级本科生姓名： 陈之威

指导教师（姓名、职称）： 汤恩义 讲师

摘要：

随着机器学习以及深度学习在学术界以及企业界的普及，这二者的关键技术——人工神经网络也成为了研究学者们和互联网巨头们的重点研究对象。虽然人工神经网络的应用在今天已经无处不在，但对于它内部的工作原理人们却知之甚少。加深该原理的理解能让人工神经网络更可靠、降低网络冗余以及提高其在移动端的应用广度。当前对于人工神经网络内部运行原理的研究有对网络权重及激活函数的可视化。此外，纽约大学提出用阻断实验来分析神经网络对图像各个部分的“兴趣程度”。这些方法都为人们一窥人工神经网络内部提供了很好的切入点。

在本次毕业论文中，本文提出了一个新的思路：基于数值解析技术去解析以 ReLU 函数为激活函数的 BP 神经网络的运算逻辑，并设计运用一种可视化的方法来对网络进行分析。本文的主要工作包括如下三个方面：

首先，在充分了解 BP 神经网络及 ReLU 函数的情况下，结合二者特性，完成自动化数值解析系统的设计，并且合理划分系统模块，然后借助 Mathematica 强大的科学计算能力，用 Java 语言实现了这些模块。

然后，根据自动化数值解析系统的性能限制，构建出简洁、合理且具有数学意义的 BP 神经网络，并且依据网络结构设计并生成了以线性函数为关系基础的数据集，然后使用该数据集去训练神经网络。

最后，运用自动化数值解析系统对 BP 神经网络进行解析及可视化，获得结果，然后结合解析出的数学公式及呈现出的函数图像，对该神经网络进行分析，了解其对输入数据的处理方式内部的数学意义。

关键词：机器学习 深度学习 人工神经网路 数值解析

南京大学本科生毕业论文（设计）英文摘要

THESIS: Automatic Numerical Analysis on Neural Network

DEPARTMENT: Software Institute

SPECIALIZATION: Software Engineering

UNDERGRADUATE: Zhiwei Chen

MENTOR: Enyi Tang

ABSTRACT:

With the popularization of machine learning and deep learning in both academic circle and business circle, Artificial Neural Networks (ANN), the key technology of them, is attached much importance on by researchers and Internet firms. Even though it is being applied almost everywhere nowadays, little do people know about its inner working principles. Deeper understandings can help make ANN more reliable, reduce its redundancy and apply it to mobile devices more widely. So far there are some studies about ANN's inner work. Visualization on ANN's weights and activation functions are put up with. Besides, in New York University, occlusion experiments are designed to plot the interest of Convolutional Neural Network (CNN) in graphs. All of these approaches are good entry points to learn about ANN's interior.

In this thesis, a new idea has been put forward, which is to use numerical analysis to analyze the arithmetic logic of Backpropagation Neural Network (BPNN), whose activation function is Rectified Linear Unit (ReLU), and to visualize the network in a certain way. The main work includes:

First, with the thorough understandings of BPNN and ReLU, an automatic numerical analysis system has been designed. By dividing the system into modules appropriately and with the help of Mathematica, these modules are developed in Java.

Secondly, under the performance limitation of the system, a simple, reasonable and mathematically significant BPNN has been constructed. Accordingly, the dataset, which is used to train the network, has been designed and generated based on linear function relationship.

Last, the system has analyzed the BPNN and visualized the result. With the analyzed mathematical formulas and the presented functional graphs, the original network could be analyzed somehow to learn the mathematical meaning about how it processed the dataset.

KEY WORDS: machine learning, artificial neural network, numerical analysis

目 录

图目录

表目录

第一章 引言

1.1 背景

自 20 世纪 80 年代兴起以来，人工神经网络就为人们解决了各式各样的问题，尤其是其强大的学习能力，为人工智能的发展做出了不可磨灭的贡献，甚至为大数据时代下的深度学习奠定了基础。但是，虽然对于人工神经网络的应用已经历经了数十年的时间，且衍生出数十种类型，但人们对于人工神经网络内在的理解仍然不够深刻与透彻。随着神经网络被应用到各种生命攸关领域，如自动驾驶[1]，对于神经网络的理解紧密关系着人们对于它的信任程度。当其内部的原理逐渐被人们所认知时，它的可靠性也同时会随之受到认可，这对于它能被应用到更多关键领域是意义深远的。

在建立人工神经网络时，最重要的地方在于应该如何去确立神经网络的结构以及其基本的训练参数，使得其能在测试数据上达到最好的表现，也就是高准确率以及高效率。而这一步在现在的机器学习领域，最好的、也可以说是唯一的方法便是由研究者根据每一次测试之后的结果，对其进行手动调整，而应当如何进行调整，除了一些约定俗成的标准之外，在实际使用中，更多的是实验者本人的经验，这就使得人工神经网络的建立具备高度的不确定性。

并且，在当今的大数据时代背景下，人工神经网络所要进行训练以及测试的数据量都非常庞大，每一次训练以及检验神经网络都要花费短则以时计、长则以天计的时间，因此，在实际建立人工神经网络的过程中，实验者们往往会浪费大量的时间和资源在错误方向上，也就是说通过调整其结构以及参数，人工神经网络的最终表现不仅没有上升，反而降低了，这使得人工神经网络的建立过程无法保证始终走在最短的正确的轨迹上。

此外，对于已经完成了训练数据训练、测试数据测试的人工神经网络，其内部仍然可能、也往往会存在着大量冗余的神经元节点，由于权重等因素的影响，这些节点对人工神经网络的输出的影响几乎可以忽略不计，但在人工神经网络运行时，对这些节点的计算会耗费一部分的时间与资源。在如语音识别、自动驾驶、计算机视觉这样数据量庞大且对于即时性要求较高的应用领域，多余的时间与资源的浪费会极大的影响软件或者应用最终的表现。

受移动端设备普及的影响，机器学习与深度学习的应用逐渐也有着向移动端发展的趋势，而人工神经网络作为这两者的交叉模型，自然无法避免[2]。但与传统的桌面端不同，受限于移动端设备的存储空间以及性能，移动端应用对于大小以及其运行效率会有更多的限制。而现在常见的机器学习与深度学习模

型，以微软公司开发的深度残差网络[3]为例，其模型大小差不多在 500MB 以上更有甚者其模型可能以 GB 为单位。因此，如何将人工神经网络模型的体积合理缩小化，这也需要考虑到人工神经网络的冗余节点。

人们对于人工网络神经节点对比不了解，归根结底在于，人工神经网络对于其使用者以及开发者来说就像是一个黑盒子，人们只知道它的结构如何、有多少层、层与层之间如何连接、每层拥有多少个节点、每个节点的权重是多少，但对于其数学含义以及逻辑含义的理解却少之又少。这来源于人们无法仅仅通过对模型的观察，就能够直观地看出其处理数据的原理。

本文就着眼于利用数值解析的技术，从数学的角度去剖析人工神经网络内部的运行原理。

1.2 人工神经网络的研究现状

对于人工神经网络是如何“思考”，即它的内部究竟是如何工作的，一直都要很多专家和学者致力于这一问题的研究。因为对这一点更深刻的理解可以提升人工神经网络的综合表现，而这些改进可以被灵活运用到其他的应用中。

截至目前，研究者们提出了许多新颖有效的方法[4]用于分析各种人工神经网络的内部工作原理。但是在网络内部，仍然有很多东西是研究者们所没有发现、没有理解的。而随着机器学习以及深度学习的广泛应用，了解人工神经网络的内部的意义愈发重要。其目的不仅仅在于提升人工神经网络的效率和准确率，还可以帮助研究者们设计出更加合理的网络结构。当前主要的研究方法有如下几种：

1.2.1 激活函数与权重的可视化

深度学习系统提供了以二维网格的形式对卷积网络层进行可视化的便捷的方法[5]。其目的便在于帮助研究者去理解当想神经网络中传入某一数据时，神经网络究竟在做什么，或者说，神经网络在图像中看到了什么，使得它能够在工作中表现得几乎不亚于人类水准。这项技术可以被用来确定卷积网络在每一层学习到了什么样的特征。

此外，也有研究更进一步，实现了对于神经网络中每一层每一个神经元节点进行可视化的工作[6]，包括池化层与归一化层。它希望研究者能洞察一个神经元节点学习到了怎样的特征，这有助于对深度神经网络工作原理的理解。

除了对激活函数进行可视化，对权重进行可视化也是一种非常常见的研究方法。神经网络的第一层通常最容易理解，因为它就是原始的数据，但对于更深的层，也可以通过对权重可视化的方式来间接帮助对其的理解。

1.2.2 阻断实验

阻断实验的原理则更加直接，其核心思路是移除一组数据中的一部分，以研究移除部分对于神经网络最终输出的影响。该方法主要应用在卷积神经网络分析图片时，通过将图片的一部分用灰色区域遮挡住，观察网络对遮挡后的图片的分析结果。

更进一步，研究者可以对图片各个区域进行阻断，然后根据网络对这些处理后的图片的分析结果，绘制相对应的图片各个区域的关键度图像。

在纽约大学的研究[7]中，就使用了该方法，对一副图像的所有区域进行迭代阻断，而后绘制出了该图像的关键度的二维热力图，如图 1.1。

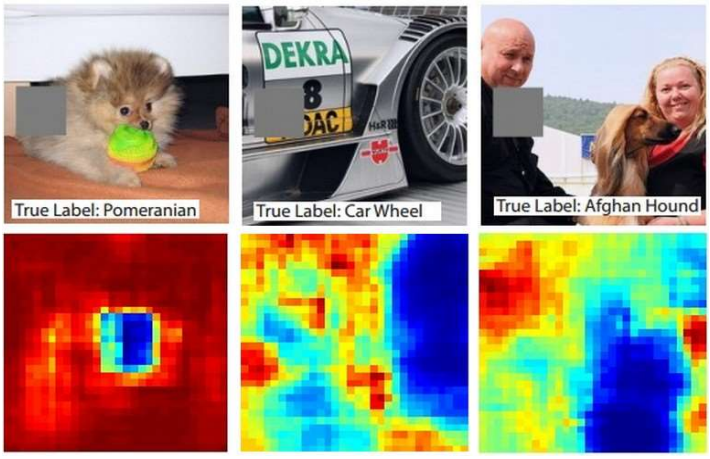

图 1.1 神经网络眼中的图像关键度热力图

从该实验可以分析出，神经网络拥有识别关键信息的能力。

1.2.3 反卷积

另一种用于分析卷积神经网络内部工作原理的方法是反卷积[8]，即去计算相对于整个图像的梯度，而不是去计算相对于权重的梯度。具体做法是选择一个卷积层，将一个神经元节点的梯度设置为 1，然后将其他神经元节点的梯度都设置为 0，然后反向传播图像进行计算。

1.2.4 其他研究

研究者对于不同的人工神经网络的内部工作原理的研究都有涉及，除了以上提及的卷积神经网络，还有深度学习的其他常用网络，如递归神经网络[9]。

1.3 本文工作

针对当前大部分对于人工神经网络的内部工作原理的研究都是基于可视化分析的思想，本文提出结合使用数值分析的方法去解析人工神经网络，将特定类型的人工神经网络转化为等价的一系列分别受各自线性不等式方程组约束的线性不等式，再对其进行数值分析以及图像分析，以两者结合的方式去窥探人工神经网络的内部工作原理。

该思路的创新点在于将人工神经网络的复杂计算通过数值分析方法解析为简洁易懂的线性不等式，从而大大降低了人工神经网络运算逻辑表达形式的复杂程度。在这一基础上，再运用可视化技术，将解析结果与可视化结果进行一一对应，从而更加直观形象地理解这些运算逻辑在该人工神经网络中所发挥的作用。

1.4 论文的组织结构

本文主要介绍了人工神经网络的相关理论知识，设计并用 Java 语言实现了一个能够自动化数值解析以 ReLU 函数为激活函数的 BP 神经网络的系统，并用 Java 语言实现了 BP 神经网络，而后用该神经网络去验证系统的正确性以及有效性，并结合实验结果合理分析该神经网络在训练完成后运行时的内部数学逻辑与意义。

第一章：引言。主要介绍了背景、人工神经网络的研究现状、本文所使用的研究方案，并描述了本文的主体结构。

第二章：相关理论技术概述。主要介绍人工神经网络的相关理论知识以及在开发和实现过程中所使用的工具的相关知识。

第三章：自动化数值解析系统的设计与实现。本章是本文的重点，先介绍了该系统的体系结构，将该系统分为三大模块。第一个模块的职责是将训练完成的 BP 神经网络分解为一系列独立的受不等式方程组约束的线性多项；第二个模块的职责是利用 Mathematica 对这些不等式方程组进行约简；第三个模块的职责是对不等式方程组所表示的范围在二维和三维时进行可视化图像呈现。然后分别介绍了这三个模块中所设计的算法以及模块的实现。

第四章：实验过程与结果。构建并训练合适的 BP 神经网络，使用自动化数值解析系统进行解析，根据解析结果检验自动化数值解析系统的正确性和有效性，同时根据结果去分析该神经网络内部的数学逻辑的意义。

第五章：总结与展望。总结该实验已实现的功能与成果，探讨实验的缺点和不足，并指出该实验进一步的发展方向。

第二章 相关理论技术概述

2.1 人工神经网络的相关介绍

2.1.1 人工神经网络

人工神经网络是机器学习[1]的一部分，同时它也是深度学习[2]的基础。人工神经网络是建立起基于动物大脑的生物神经网络[10]，将生物学理论与计算机科学应用相结合产生的研究模型，如图 2.1 所示。它模拟了生物大脑内的神经传递结构，设置了大量的神经元节点相互联接，以节点为单位构成神经网络层，以神经网络层为单位构成整个网络。同时，还模仿了生物神经元对于电刺激的响应机制，为每个神经元设置了激活函数，进而能使其充分逼近任意的非线性关系，从而使得整个神经网络模型都类似于真实生物的大脑，在大量数据的调试下对神经元之间的参数进行训练，达到一种类似于生物习得后天神经反射的状态，使得该模型能够在面临同样问题、不同数据的时候，达到类似于学习的机器行为。

图 2.1 生物神经网络

人工神经网络的建立模式如图 2.2 所示：

[1] 机器学习是计算机科学的一个领域，其特点是通过数据训练来赋予计算机系统“学习”的能力，而不用通过显示编程来实现。

[2] 深度学习是更广泛的机器学习方法的一部分，其特点是基于学习数据特征，而不是面向任务的算法。

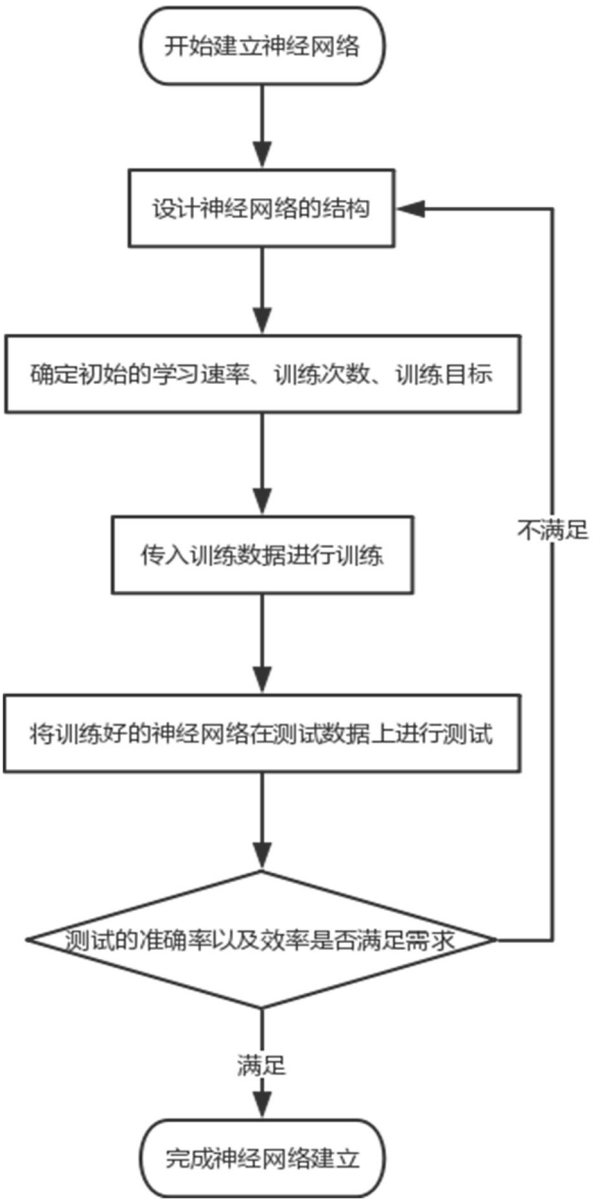

图 2.2 神经网络建立流程图

该模型的优势[11]在于：一是相对来说需要较少的正式测试数据进行训练；二是可以隐性地学习到独立或者相关的变量之间的非线性关系；三是有能力检测到预测变量之间的交互；四是可以使用多种不同的训练算法来进行开发。

人工神经网络的种类也非常繁多，从最为简单的感知机[12]逐渐演化，以适应不同的数据特征以及实际需要。对于目前已有的数十种神经网络模型，按网络结构进行划分可以划分为三大类：前馈神经网络、反馈神经网络和自组织神经网络。

2.1.2 BP 神经网络

本文主要讨论的是前馈神经网络中的 BP 神经网络[13]，如图 2.3 所示，其简洁、灵活的结构特征，使其被广泛应用于简单的预测、文字识别等等问题，并且它作为深度学习的根基，深度学习中的经典模型卷积神经网络便是在 BP 神经网络的基础上演化而来。

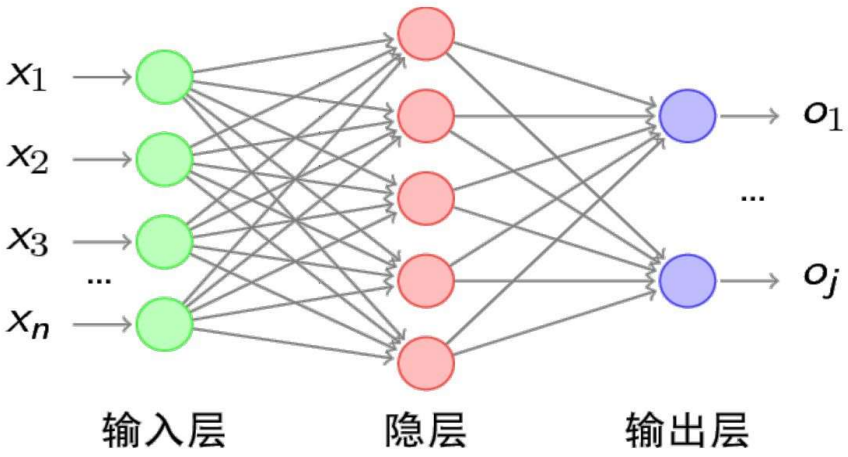

图 2.3 BP 神经网络，人工神经网络的一种

因为 BP 神经网络是当前应用最为广泛的人工神经网络，因此本实验便选择它作为研究对象。

2.1.3 ReLU 激活函数

为了使神经网络能够学习非线性关系，人们引入了激活函数的概念，即在每个神经元节点上对输出的数据进行非线性变换，再作为下一个神经元节点的输入，通过这种方式，神经网络拥有了模拟非线性运算逻辑的能力。常用的激活函数有 Sigmoid 函数、TanH 函数，如图 2.4、图 2.5。它们的共同点在于，对于中央区域的信号都会更加的敏感，而对于边缘区域的信号会呈现抑制态，从而可以将重点特征维持在中央区域，而把不重要的特征维持在边缘区域[14]。

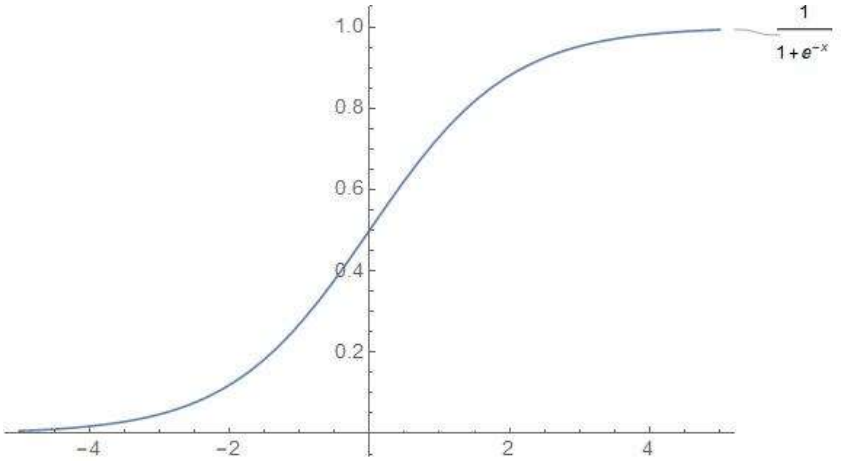

图 2.4 Sigmoid 函数图像

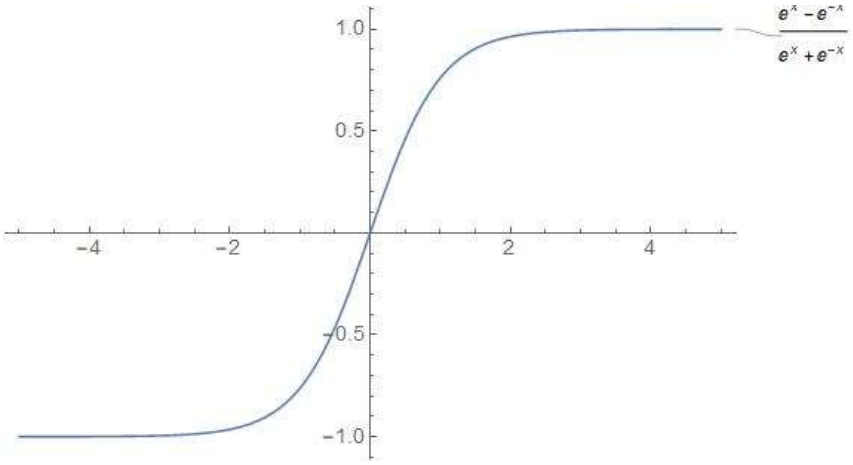

图 2.5 TanH 函数图像

而之后，人们从生物学角度出发，提出了更加拟合脑神经元接受信号的激活模型，而后被称为线性整流函数，简称为 ReLU 函数，如图 2.6。

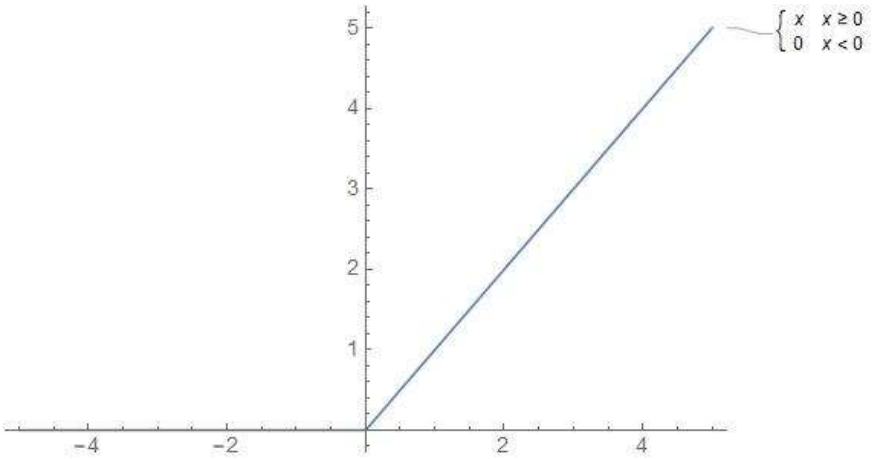

图 2.6 ReLU 函数图像

相比于 Sigmoid 系的激活函数，ReLU 函数拥有三个特点[15]：一是单侧抑制；二是更加宽阔的兴奋边界；三是稀疏激活性。

在 AlexNet[16]中，ReLU 函数被验证其会使学习周期大大缩短，如图 2.7 所示。这也是为什么在深度学习中，大部分激活函数都会使用 ReLU 函数。

此外，相比于 Sigmoid 函数以及 TanH 函数，ReLU 函数的计算难度相当简单，会大大简化本实验的难度，非常适合作为本研究早期实验的人工神经网络的激活函数。

综上所述，本实验选取 ReLU 函数作为 BP 神经网络的激活函数。

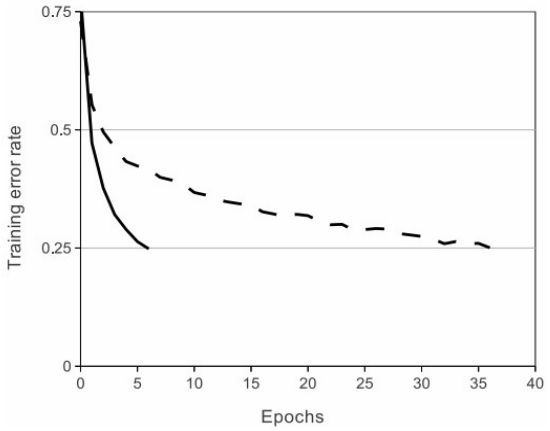

图 2.7 ReLU 函数与 Sigmoid 函数收敛速度对比

2.2 Mathematica 的相关介绍

2.2.1 Mathematica

Mathematica 是由 Wolfram 研究的一款科学计算软件，自从 1998 年发布以来，它已经对在科技、教育等各个领域运用计算机产生了巨大的影响，是当今世界通用计算系统中最强大的科学计算软件，与 MATLAB 和 Maple 并称为三大数学软件。

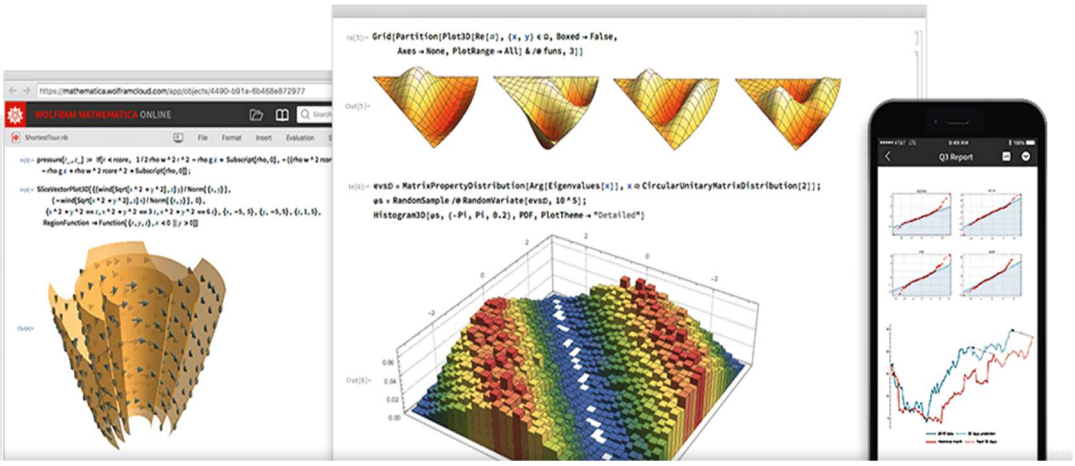

图 2.8 Mathematica 简介图

Mathematica 是一个全面集成的大型系统，其拥有将近5000个内置函数，在包括图像、几何、数据科学、机器学习等各个技术计算领域都有着卓越的表现。其所使用的 Wolfram 语言独树一帜，易于阅读、编写和学习。此外，它拥有着涵盖各个领域的高效算法，其并行计算、GPU 计算等功能使其可以面对各种各样的学术界以及产业界的问题。它使用 Wolfram 笔记本界面，使得可以快速编写文本、代码，并进行运行，甚至在运行结果上进行手动调整。除了拥有着强大的编辑环境以及用户图形界面，它可以与任意的内容连接从而进行构建，包括其他编程语言、数据库、物联网和设备等等。

2.2.2 J/Link

J/Link 是用来集成 Wolfram 语言与 Java 的工具，它使得 Wolfram 语言内核能够从 Java 程序内部被调用。对于 Java 语言的使用者来说，J/Link 将 Wolfram 语言转化成了一种脚本语言，使得它能在 Java 程序内部被编写、编译以及测试。它同时也让 Java 成为了编写 Wolfram 语言的科学计算服务的理想语言。

相对应的，J/Link 也能让 Wolfram 语言以一种完全透明的方式使用 Java 语言。

J/Link 功能主要有以下六点：一是从 Wolfram 语言中调用 Java 方法；二是编写使用 Wolfram 系统服务的 Java 程序；三是为 Wolfram 语言提供了一种前端；四是为 Wolfram 语言程序提供了对话框等弹出式用户界面元素；五是编写在客户端或者服务器端需要使用 Wolfram 语言内核的 Java 程序；六是编写让 Wolfram 系统服务能够被 HTTP client 使用的 servlets。

2.2.3 J/Link API

J/Link 使用 WSTP[1]来完成程序之间的数据与命令的传递。

WSTP 是用于程序间通信的独立与平台的协议。在本文的范围内，它主要用于发送与接收 Wolfram 语言表达式。

在 J/Link 中，接口与实现被完全分离。在实际运用中，使用者永远不需要直接创建一个对象，只需要调用工厂方法来创建一个链接类，然后通过接口进行调用。

其中，最为重要的两个接口是 MathLink 以及 KernelLink。

MathLink 是在 Java 语言中使用 WSTP 端口的必要接口，它封装了所有在连接上的操作，并且不关心连接的另一端究竟是什么样的程序。

KernelLink 继承了 MathLink 并在其上添加了许多重要的、便捷的、更高层次的方法，但这些方法需要保证连接的另一端一定是 Wolfram 语言内核。它是最为重要的接口，同时也是本研究中连接 Java 与 Wolfram 语言内核的接口。

在 Java 项目中使用 J/Link 需要将 JLink.jar 添加进 CLASSPATH 环境变量。

获取 KernelLink 实例只需要调用响应的工厂方法即可：

```
import com.wolfram.jlink.*;

public class JLinkExample {
    private static KernelLink kernelLink;
    /**
     * 两两一对，每对第一个为选项名称，第二个为选项内容
     * linkmode 是连接模式，launch 意味着启动 Wolfram 系统内核
     * linkname 为连接路径，其内容为可执行文件 mathkernel.exe 的路径
     */
    private static String[] argv = {"-linkmode", "launch", "-linkname",
path_of_mathkernel.exe};

    public JLinkExample (){
        kernelLink = null;
        try {
            kernelLink = MathLinkFactory.createKernelLink(argv);
            kernelLink.discardAnswer();
        } catch (MathLinkException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

图 2.9 获取 KernelLink 实例代码

[1] Wolfram Symbolic Transfer Protocol. Wolfram 符号传输协议。

2.3 Neuroph 框架的相关介绍

2.3.1 Neuroph 框架

Neuroph 是一个轻量级的、万全使用 Java 语言编写的、用于开发常见神经网络架构的神经网络框架。它拥有设计完好的、开源的 Java 库，这些库中会有一些基础类与基本神经网络相对应。同时它还拥有一个提供图形用户界面的神经网络编辑器，来快速创建神经网络的 Java 组件，如图 2.10 所示。

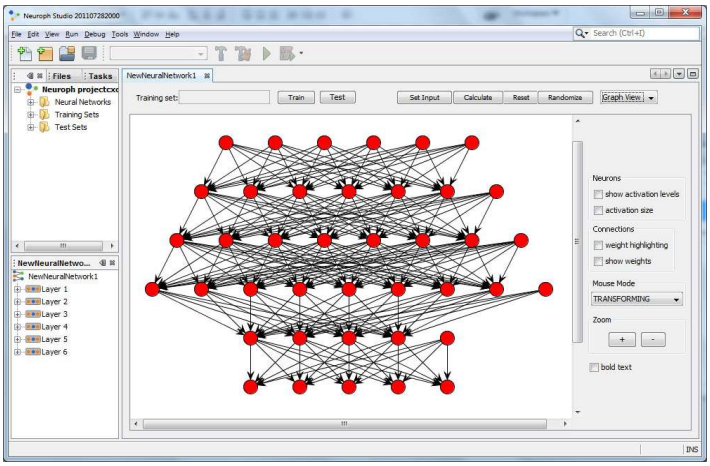

图 2.10 Neuroph 图像用户界面

2.3.2 Neuroph 框架的使用

在 Maven 项目中使用 Neuroph 框架只需要在 pom.xml 里添加相应的依赖即可：

```
<repositories>
    <repository>
        <id>neuroph.sourceforge.net</id>
        <url>http://neuroph.sourceforge.net/maven2/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.neuroph</groupId>
        <artifactId>neuroph-core</artifactId>
```

```
        <version>2.94</version>
    </dependency>
</dependencies>
```

图 2.11 配置 Neuroph 代码

2.3.3 选择 Neuroph 框架的理由

Neuroph 是一个轻量级的、万全由 Java 语言编写的框架，这使得它在 Java 项目中非常容易使用；并且其提供了不同程度的可变性，以及详尽的文档。非常适合于 Java 项目的人工神经系统研究。

2.4 本章小结

本章介绍了本实验所研究的主要对象——人工神经网络——的概念，并介绍了其中的应用最为广泛的模型 BP 神经网络，并将其作为了本实验的研究对象，还说明了人工神经网络的激活函数的概念以及选择 ReLU 函数作为本实验中人工神经网络的激活函数的原因。

此外，本章还介绍了本实验所使用的两个工具：Mathematica 和 Neuroph 框架。前者是一个强大的科学计算工具，并提供了优秀的 Java 调用支持，后者则是一个 Java 库，封装了丰富的关于人工神经网络的实现并可根据需要进行修改。在接下来的章节里，笔者会详细说明这些工具如何应用在了本实验中。

第三章 自动化数值解析系统的设计与实现

本章首先介绍了自动化数值解析系统的算法的理论基础。然后此基础上将该系统设计为三个模块，第一个模块用于将 BP 神经网络运算改写为由一系列线性不等式方程组进行约束的线性不等式，第二个模块用于解线性不等式方程组，第三个模块用于对二维输入或三维输入进行图像可视化呈现。最后设计并完成各个模块的开发。

3.1 算法理论基础

人工神经网络运算归根结底就是进行一组矩阵运算，对于第i层的第j个神经元节点$node_{i,j}$，其计算公式为：

$$node_{i,j} = \emptyset(layer_{i-1})Weight_{i-1,j} \tag{3.1}$$

其中$\emptyset(x)$为激活函数，$layer_{i-1}$为第$\mathrm{i}-1$层的神经元节点构成的行矩阵，$Weight_{i-1,j}$为第$\mathrm{i}-1$层的权重矩阵中第j列的列举阵，其对应着计算$node_{i,j}$的权重参数。

因为在每一层非输出层神经元节点中都会添加一个值为1的截距项神经元节点，该公式可以展开为：

$$node_{i,j} = \sum_{k=1}^{n-1} \emptyset\left(node_{i-1,k}\right) weight_{i-1,j,k} + weight_{i-1,j,n} \tag{3.2}$$

ReLU 函数作为在深度学习中应用最为广泛的激活函数，其公式为：

$$\emptyset(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{3.3}$$

当采用 ReLU 函数作为激活函数时，相当于对于结果为正数的神经元节点输出，不作任何处理直接作为下一层神经元节点的输入，而对于结果为负数的输出，将其值记为0作为下一次节点的输入，引入到公式 3.2 中即变为：

$$node_{i,j} = \sum_{node_{i-1,k} \geq 0} node_{i-1,k} weight_{i-1,j,k} + weight_{i-1,j,n} \tag{3.4}$$

由于将线性表达式进行常数乘除再进行加减后的结果仍然为线性表达式，因此$x_{i,j}$的值可以看作是一组可能的线性表达式，对于每一种可能的$layer_{i-1}$取值，都存在着不同且唯一的线性表达式作为其最终的计算公式。若$layer_{i-1}$中一

共有n个神经元节点，则$node_{i,j}$一共有2^{n-1}种可能的线性表达式（因为$layer_{i-1}$中包含一个截距项神经元节点，而其取值不可能小于0）。

同理，对于 BP 神经网络最终的输出Y，其可能的线性表达式的数量为2^N，其中N为隐藏层所有的非截距项神经元节点的数目。在实际情况中，其数量通常会小于2^N，这是因为部分线性表达式所答应的线性不等式方程组不存在解。

通过这种方式，以 ReLU 函数作为激活函数的 BP 神经网络可以改写成一系列可能的线性表达式，其中的每一个线性表达式都有一组线性不等式方程组作为其输入的值域的约束。

3.2 自动化数值解析系统的模块设计

根据上节的算法基础，该系统可以被分解为三个主要模块，如图 3.1。

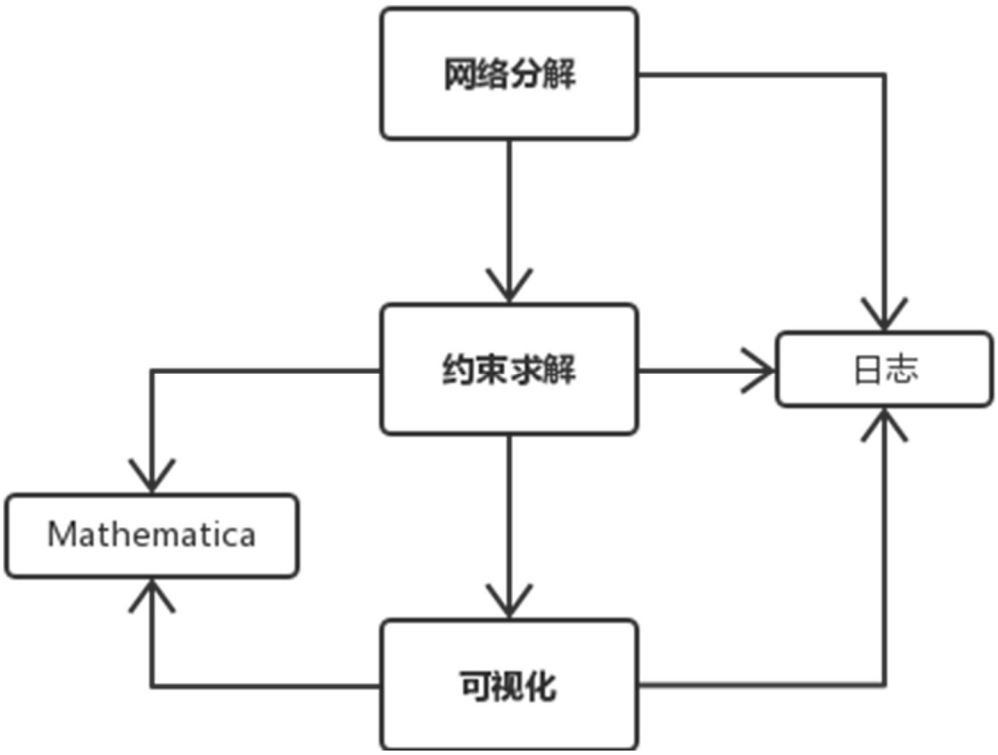

图 3.1 系统架构图

3.3 网络分解模块

3.3.1 模块职责

该模块负责的职责是对于已经完成训练的 BP 神经网络，根据其权重参数矩阵，将之分解为2^N个线性表达式，且对于每一个表达式，都记录相对应的约束它的线性不等式方程组。然后将这些线性不等式方程组传递给下一模块进行求解。

3.3.2 算法设计

该算法采用动态规划[1]的方法，其基本思想为：在已经确定第i层的所有神经元节点的情况下，再去计算第$\mathrm{i}+1$层的神经元节点。

根据公式 3.2

$$node_{i,j}=\sum_{k=1}^{n-1}\emptyset\left(node_{i-1,k}\right)weight_{i-1,j,k}+weight_{i-1,j,n}$$

第i层的第j个神经元节点$x_{i,j}$的值是由第$\mathrm{i}-1$层的神经元节点计算而出的。在 ReLU 函数作为激活函数的情况下，当$node_{i-1,k}\geq 0$时，该项保留，当$node_{i-1,k}<0$时，该项从求和中移除。

若第$\mathrm{i}-1$层的所有神经元节点线性表达式的结果均大于等于0，此时$node_{i,j}$的计算公式为：

$$node_{i,j}=\sum_{k=1}^{n-1}node_{i-1,k}weight_{i-1,j,k}+weight_{i-1,j,n} \tag{3.5}$$

若第$\mathrm{i}-1$层的所有神经元节点线性表达式的结果均小于0，此时$node_{i,j}$的计算公式为：

$$node_{i,j}=weight_{i-1,j,n} \tag{3.6}$$

然后计算出其他$2^{n-1}-2$种的第$\mathrm{i}-1$层的一个或多个神经元节点线性表达式结果小于0的排列组合情况，再将每种组合情况作为条件，确定$node_{i,j}$的线性

[1] 动态规划是指将一个复杂问题分解成一系列更简单的子问题，对于每个子问题都只解决一次，然后存储解决的结果，当下次同样的子问题出现时，直接获取结果，而不用再重新去解决，从而降低了解决时间。

不等式，而在该情况下的第$\mathrm{i}-1$层的各个神经元节点线性表达式与0的大小关系所构成的线性不等式方程组即为$node_{i,j}$为该线性不定式下的约束。

记隐藏层所有的非截距项神经元节点的数目为N，该算法的复杂度为$O(2^N)$。

3.3.3 数据结构

存储神经元节点的线性表达式

当该神经网络的输入为n维时，对任意神经元节点，其线性表达式始终可以表示为：

$$node = \sum_{i=1}^{n} a_i x_i + b \tag{3.6}$$

其中$node_i$为神经网络的原始输入，a_i与b均为常量。对于输入层的$node_i$，即x_{i_0}，其参数情况为$a_i = 0 \ \ i \neq i_0$且$b = 0$。

因此，可以用$n+1$维的数组来表示神经元节点的线性不等式，如图 3.2：所示

a_1	a_2	a_3	……	a_n	b

图 3.2 存储线性表达式的数组示意图

记录每一层神经元节点线性不等式与 0 的大小关系

对于每一层的2^{n-1}中可能出现的排列组合情况，显然不可以对每一种情况进行重新运算，因为这样会出现大量的重复计算。

由上文可知在计算$node_{i,j}$时，当$node_{i-1,k} < 0$时，相当于将该项从求和中移除，因此可以先计算出所有$node_{i-1,k} weight_{i-1,j,k}$项，将所有的结果缓存在二维数组内。

再用二进制数的方法表示在某个组合情况下，各项是否会出现在求和中。例如，对于4维输入，当该二进制数为 1011 时，其含义是上一层的第1、3、4个神经元节点的线性表达式大于等于0，第2个神经元节点的线性表达式小于0。然后将其所有可能出现的二进制数循环，即可罗列出某一层的所有的线性不等式组情况。

存储输出的线性表达式及其约束

在输出层，对于存储每一个节点的线性表达式的数组以及代表其约束的线性不等式方程的数组，将其还原成字符串形式，再分别存储在数组中。

```
public class Solution {
    private String[] objectives;
    private String[] constraints;

    /**
     * 构造器
     * @param obs 存储线性表达式组的数组
     * @param cons 存储约束的线性不等式方程组的数组
     */
    public Solution(double[][] obs, List<double[]> cons){
        objectives = new String[obs.length - 1];
        for (int i = 0; i < obs.length - 1; ++i){
            objectives[i] = convertObjective(obs[i]);
        }
        constraints = new String[cons.size()];
        for (int i = 0; i < constraints.length; ++i){
            constraints[i] = convertConstraints(cons.get(i));
        }
    }
}
```

图 3.3 Solution 关键部分代码

3.3.4 关键代码

动态规划计算网络逐层节点的线性表达式

```
/**
 * 动态规划计算网络逐层节点的线性表达式
 * @param input_layer 上一层所有节点的线性表达式
 * @param weights 节点权重
 * @param layer_index 指明当前的层数
 * @param layer_nubmer 指明最大的层数
 * @param condition 已经记录的约束的线性不等式方程组
 */
private void calculateNet(double[][] input_layer, List<double[][]> weights, int
layer_index, int layer_nubmer, List<double[]> condition){
```

```
        //如果已经计算到输出层，则将结果记录后返回
        if(layer_index == layer_nubmer) {
            Solution solution = new Solution(input_layer, condition);
            solutions.add(solution);
            return;
        }

        //获取当前层的所有权重
        double[][] weight = weights.get(layer_index);
        //获取当前一层节点线性表达式全部大于 0 的计算结果
        double[][][][] cache = attainNodeCache(input_layer, weight);

        if(layer_index++ == 0){
            //当该层为输入层时
            //递归调用实现动态规划
            calculateNet(calculateLayer(cache,0), weights, layer_index, layer_nubmer,
condition);
        }else {
            //当该层为隐藏层时
            //获得当前层所有节点大于 0 小于 0 的排列组合，递归调用计算下一层
            int relu_tag = (int) Math.pow(2, input_layer.length - 1) - 1;
            for (int i = relu_tag; i >= 0; --i){
                //记录当前已有的所有约束
                List<double[]> tmp_condition = new ArrayList<>(condition);
                tmp_condition.addAll(attainCondition(input_layer, i));
                //递归调用实现动态规划
                calculateNet(calculateLayer(cache, i), weights, layer_index, layer_nub-
mer, tmp_condition);
            }
        }
}
```

图 3.4 解析网络线性表达式动态规划部分代码

calculateNet 函数是该算法的核心函数，它实现了算法动态规划部分的逻辑。在 calculateNet 函数中，一共处理三种情况的输入：当所要处理的网络层为输入层时，计算出下一层所有神经元节点的线性表达式，然后递归进行下一层的运算；当所要处理的网络层为隐藏层时，排列组合出该层所有可能出现的神经元节点大于0或是小于0的情况，并在每一种条件下分别计算出下一层所有神经元节点的线性表达式，并递归进行下一层的计算；当所要处理的网络层为输出层时，存储最终的线性表达式以及将网络计算换算成该线性表达式所要满足的线性不等式方程组约束，然后返回。

值得说明的是 relu_tag 以十进制数的方式保存了各个神经元节点大于0或是小于0的情况，将其转化为二进制数后，该二进制数的每一位都分别代表了一个神经元节点与0的大小关系情况，当某位为1时，代表其对应的神经元节点线性表达式计算结果大于等于0，当某位为0时，代表其对应的神经元节点线性表达式计算结果小于0。

计算每一层节点的线性表达式

```
/**
 * 计算每一层节点的线性表达式
 * @param cache 存储当前一层所有节点线性表达式大于 0 时该层节点的计算结果
 * @param relu_tag 前一层各个节点线性表达式与 0 的大小关系
 * @return 该层各个节点的线性表达式
 */
private double[][] calculateLayer(double[][][][] cache, int relu_tag){
    double[][] output_layer = new double[cache.length + 1][cache[0][0][0].length];

    for (int i = 0;i < output_layer.length - 1; i++){
        output_layer[i] = calculateNode(cache, i, relu_tag);
    }
    //添加值为 1 的截距项节点
    output_layer[output_layer.length - 1][output_layer[0].length - 1] = 1;

    return output_layer;
}
```

图 3.5 计算每层节点线性表达式部分代码

在 calculateLayer 函数中，通过调用 calculateNode 函数依次计算出各个神经元节点的线性表达式，并加上值为1的截距项神经元，完成整层神经元节点的线性表达式的计算。

计算节点的线性表达式

```
/**
 * 计算节点的线性表达式
 * @param cache 存储当前一层所有节点线性表达式大于 0 时该层节点的计算结果
 * @param output_node_index 指明节点序号
 * @param relu_tag 该层各个节点的线性表达式
 * @return 该节点的线性表达式
 */
```

```
private double[] calculateNode(double[][][][] cache, int output_node_index, int
relu_tag){
    int length = cache[0].length;

    //考虑多个输入的情况，node 元素个数应该为每组输入数据元素个数加 1
    double[] node = new double[cache[0][0][0].length];

    node = addDoubleArray(node, cache[output_node_index][length - 1][1]);
    //将十六进制的 relu_tag 转变为二进制
    for (int i = 0; i < length - 1; ++i){
        boolean isOne = (relu_tag % 2 == 1);
        relu_tag /= 2;
        if(isOne){
            //该位为 1 时，代表节点的线性表达式大于 0
            node = addDoubleArray(node, cache[output_node_index][i][0]);
        }else{
            //该位为 0 时，代表节点的线性表达式小于 0
            node = addDoubleArray(node, cache[output_node_index][i][1]);
        }
    }

    return node;
}
```

图 3.6 计算节点线性表达式部分代码

在 calculateNode 函数中，由于已经分别计算出上一层各个神经元节点大于 0 以及小于 0 时与权重相乘的结果并存储在了 cache 中，因此只需要将代表上一层各个神经元节点大于0或是小于0的 relu_tag 进行转化，然后选择适当的计算结果进行相加，即可得到该神经元节点在当前 relu_tag 条件限定下的最终的线性表达式。

3.4 约束求解模块

3.4.1 模块职责

该模块负责的职责是对等价于 BP 神经网络运算的各个线性表达式的约束的线性不等式方程组进行求解。

若约束方程组了无解，则代表该网络的输入在任何情况都不可能满足于该约束，其 BP 神经网络运算也不可能换算为该约束对应的线性表达式。

若约束方程组有解，将其进行求解，以获取各约束情况下输入之间的大小关系，为下一模块的可视化以及为网络工作原理的数学分析做铺垫。

3.4.2 模块实现

该模块中对于代表约束的线性不等式方程组的求解通过使用科学计算工具 Mathematica 实现。本节主要介绍本实验项目通过 J/Link 与 Mathematica 的实现数据传递。

向 Mathematica 传递线性不等式方程组

通过 J/Link，Wolfram 语言可以像脚本语言一样编写在 Java 项目中。

Wolfram 语言中使用$Reduce$函数来实现不等式方程组的求解，其形式为

$$Reduce[exp, vars, dom]$$

其中exp为不等式方程组表达式，$vars$为变量，dom为变量的域。例如：

$$Reduce[x + y > 1 \&\& x - y > 1, \{x, y\}, Reals]$$

其返回的解为

$$x > 1 \&\& 1 - x < y < -1 + x$$

因此，将 Solution 中的局部变量 constraints 通过&&进行连接，将其包装成 Wolfram 语言形式，而后调用 J/Link API 即可。

从 Mathematica 接收方程组求解结果

对于$Reduce$函数，Mathematica 的返回结果为一个由&&和||连接的不等式，而 J/Link 会将该中缀不等式转化为一棵线索树，在程序请求获取数据时，它会返回对该线索树中序遍历的下一个节点。

在本实验中，数据一共有 4 中类型，分别为：Function 函数、Symbol 符号、Integer 整型和 Double 浮点数。其中 Symbol、Integer、Double 都为基本数据类型，也就意味着是树的叶节点，Function 为操作符，且在获取 Function 时，J/Link 同时会返回一个数字，代表该操作符操作的数据数目，即该节点子节点的数目。Function 的可能取值如表 3.1 所示：

表 3.1 Function 可能取值表

名称	符号	优先级	操作数据数目
Times	$*$	5	2
Plus	$+$	4	≥ 2
Greater	$>$	3	2
Less	$<$	3	2
GreaterEqual	$\geq$	3	2
LessEqual	$\leq$	3	2
Equal	$=$	3	2

Or	||	2	≥ 2
And	&&	1	≥ 2

因此用栈来存储 Function，其算法示意图如图 3.7 所示：

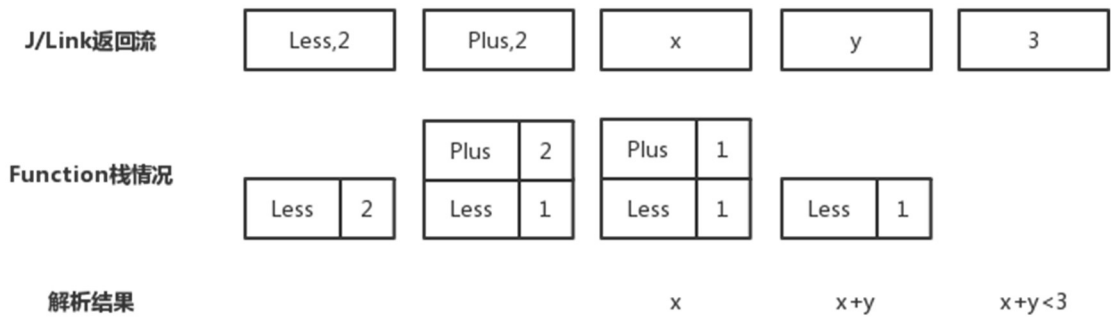

图 3.7 算法示意图

通过该算法，就可以将 J/Link 返回的线索树还原为不等式，即完成了线性不等式方程组的求解。

3.4.3 关键代码

通过 J/Link 解线性不等式方程组

```
/**
 * 求解线性不等式方程组
 * @param formula 已经包装成 Wolfram 语言形式的约束不等式方程组
 * @return 方程组的解
 */
public String solveInequality(String formula){
    //stack 为缓存 Mathematica 返回值的堆栈
    stack.clear();
    StringBuilder sb = new StringBuilder();
    try {
        //向 Mathematica 传递公式
        kernelLink.evaluate(formula);
        kernelLink.waitForAnswer();
        //当 stack 再次为空时，说明返回值已经全部解析完成
        do {
            int type = kernelLink.getType();
            if(type == 70){
                //返回值类型为操作符
```

```
                    sb.append(processFunction(kernelLink.getFunction()));
                }else if(type == 35){
                    //返回值类型为符号
                    sb.append(processSymbol(kernelLink.getSymbol()));
                }else if(type == 43){
                    //返回值类型为整型
                    sb.append(processNumerical(kernelLink.getInteger()));
                }else if(type == 42){
                    //返回值类型为浮点数
                    sb.append(processNumerical(kernelLink.getDouble()));
                }else{
                    //返回值类型错误
                    kernelLink.discardAnswer();
                    stack.clear();
                    System.out.println("Unexpected Type: " + type);
                }
            }while (!stack.empty());
        } catch (MathLinkException e) {
            System.out.println("MathLinkException occurred: " + e.getMessage());
        }

        //当 sb 为空时说明该方程组无解
        return sb.length() == 0 ? "Impossible" : sb.toString();
    }
```

图 3.8 通过 J/Link 解不等式方程组部分代码

在 solveInequality 函数中，首先向 Mathematica 传递转换好的 Wolfram 语言格式的字符串，等待 Mathematica 求解完成后，依次从 Mathematica 获取对代表最终结果不等式的线索树进行中序遍历的一个节点，并根据每一个节点的类型分别进行处理，并根据处理结果扩充结果不等式字符串。由于 Mathematica 不会返回终止符号，因此需要根据存储操作符的栈是否为空这一条件来判断该线索树是否解析完成。待线索树解析完成后返回最终的代表结果不等式的字符串。

处理 J/Link 返回值 （以基础类型为例）

```
/**
 * 处理基础类型返回值
 * @param name 基础类型
 * @return 解析出来的字符串
 */
```

```
private String process(String name){
    StringBuilder sb = new StringBuilder(name);
    while(!stack.empty()){
        if (--stack.peek().remain_operator_num > 0){
            //当该数据不是 Function 最后一个操作数时
            sb.append(stack.peek().name);
            break;
        }else {
            //当该数据是 Function 最后一个操作数时
            Function function = stack.pop();
            //判断操作符优先级以完成括号添加
            if (!higherPriority(function))
                sb.append(")");
        }
    }

    return sb.toString();
}
```

图 3.9 处理函数类型返回值部分代码

process 函数用于处理基础类型的返回节点，经过对符号、整型和浮点数类型返回数据做适当的处理，它们最终都被转化成了基础类型。由于每一个基础类型都是其上一级操作符的参数，因此需要将栈中最上层的操作符的剩余参数的数目减去1，当剩余参数为0时，则弹出该操作符，并对更新后的栈的最上层操作符做相同操作。当栈为空时，说明整个线索树已经被解析完成。为了最终字符串的易读性，同时也需要根据操作符之间的优先级合理加上括号。

3.5 可视化模块

3.5.1 模块职责

对于二维输入，该模块负责将其可视化，即在约束不等式组所限定的各个范围内，分别绘画出其对应的输出与输入的函数图像，以此来更加直观地展现神经网络输出与输入的对应关系，为而后的分析神经网络内部工作的数学原理打下基础。

3.5.2 模块实现

该模块中对于约束不等式组中各个约束以及输出的不等式所构成的分段函数的可视化通过使用科学计算工具 Mathematica 实现。由于高于三维的图像无法呈现，因此只有在输入为二维，输出作为该二元线性不等式的值，一共三维时，才能对最终结果进行可视化。因此本模块的功能均只局限于当输入数据为二维的情况。

J/Link 中提供了类 MathCanvas 用于绘制 Mathematica 所返回的图像，其使用方法与 Java Swing 的普通组件相同，而对于 Wolfram 脚本形式指令的传递，只需要调用 MathCanvas.setCommand(String)，将指令作为参数传入，J/Link 即会将指令传递给 Mathematica 进行计算以及绘图，而后将成像显示在 MathCanvas 所在的组建上：

```
public class MyFrame extends JFrame {
    /**
     * MathCanvas 示例
     * @param kernelLink  链接接口
     * @param command  需要显示的 Wolfram 脚本指令
     */
    public MyFrame(KernelLink kernelLink, String command){
        this.setLayout(null);
        this.setSize(500, 500);

        MathCanvas mathCanvas = new MathCanvas(kernelLink);
        mathCanvas.setBounds(0, 0, 500, 500);
        this.add(mathCanvas);
        mathCanvas.setMathCommand(command);

        this.setVisible(true);
    }
}
```

图 3.10 MathCanvas 使用部分代码

Wolfram 语言中提供了数十种函数用于绘制各种各样的图像，对于三维函数图像，需要使用函数$Plot3D$，其形式为

$$Plot3D[f,\{x_1,x_{1min},x_{1max}\},\{x_2,x_{2min},x_{2max}\},\dots]$$

其中f为为函数表达式，对于分段函数，有专用的函数$Piecewise$表示，其形式为：

$$Piecewise[\{\{val_1,cond_1\},\{val_2,cond_2\},\dots\}]$$

该函数在范围域$cond_1$表达式为val_1，在范围域$cond_2$表达式为val_2，以此类推。

在$Plot3D$中，$\{x_1, x_{1min}, x_{1max}\}$和$\{x_2, x_{2min}, x_{2max}\}$是图像中$x_1$和$x_2$的显示范围，最后的...是图像的一些配置选项。

例如，对于分段函数

$$f(x_1, x_2) = \begin{cases} x_1 - x_2, & x_1 + x_2 < -100 \\ 2x_1 - 2x_2, & -100 \le x_1 + x_2 < 0 \\ 4x_1 - 4x_2, & 0 \le x_1 + x_2 < 100 \\ 8x_1 - 8x_2, & x_1 + x_2 \ge 100 \end{cases} \tag{3.7}$$

对x_1和x_2均在$(-100,100)$范围上做出函数图，其写法为

$$Plot3D[Piecewise[\{$$
$$\{x_1 - x_2, x_1 - x_2 < -100\}, \{2x_1 - 2x_2, -100 \le x_1 - x_2 < 0\},$$
$$\{4x_1 - 4x_2, 0 \le x_1 - x_2 < 100\}, \{8x_1 - 8x_2, x_1 - x_2 \ge 100\},$$
$$\}], \{x_1, -100,100\}, \{x_2, -100,100\}]$$

3.5.2 可视化结果

对于公式 3.7 的分段函数，程序运行截图如图 3.11 所示：

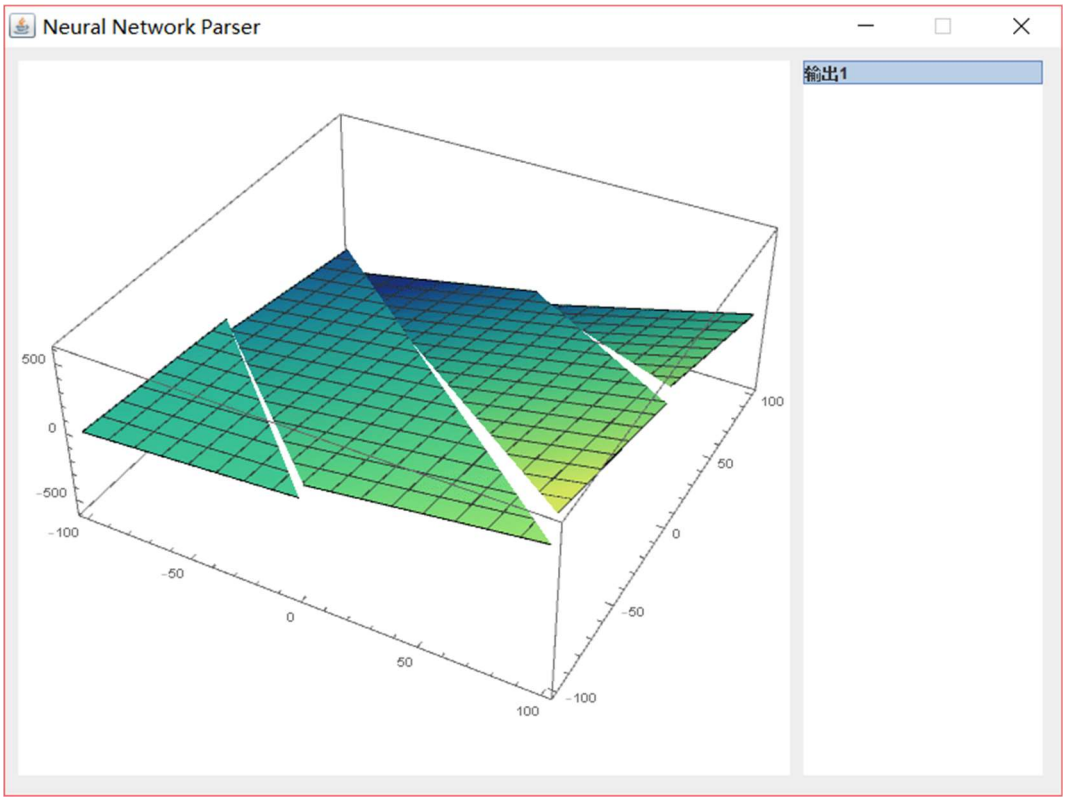

图 3.11 可视化模块运行截图

3.6 性能分析

由前文可知，在网络分解模块中，将 BP 神经网络分解为一系列由线性不等式方程组进行约束的线性表达式的算法的复杂度为$O(2^N)$，其中N为网络所有隐藏层的非截距项神经元的数目。在约束求解模块中，需要对这些不等式组进行简化求解，因此复杂度也为$O(2^N)$。而在可视化模块中，经过约束求解模块的筛选会过滤掉不存在解的情况，并且最终与 Mathematica 进行数据传递以及绘制图像的数目在一次界面交互中只会发生一次，因此其复杂度为$O(1)$。

因此该系统的整体复杂度是$O(2^N)$。

通过运行程序，对N各不相同的 BP 神经网络使用自动化数值解析系统进行分析，其中网络分解模块运行时间如表 3.2 所示：

表 3.2 网络分解模块运行时间表

网络结构	N	网络分解模块所需时间（秒）
2 10 1	10	0.306
2 11 1	11	0.519
2 12 1	12	0.631
2 13 1	13	0.832
2 14 1	14	1.716
2 15 1	15	3.536
2 16 1	16	7.902
2 17 1	17	17.130
2 18 1	18	36.776
2 19 1	19	76.108
2 20 1	20	158.203
2 12 2 1	14	1.510
2 13 2 1	15	2.896
2 14 2 1	16	6.618
2 15 2 1	17	13.735
2 16 2 1	18	30.585
2 4 7 3 1	14	2.398
2 4 8 3 1	15	3.018
2 4 9 3 1	16	5.889
2 4 10 3 1	17	13.329
2 4 11 3 1	18	28.655

由此可以验证，网络分解模块的运行效率与网络的层数以及每层无关，而仅仅与隐藏层非截距项神经元节点数目N有关，其运行时间随着N的增大以2^N的速率呈指数上升，且由于算法设计的原因，程序运行所需内存空间的增长速率也呈指数上升。因此可以得出结论，该系统只能够处理隐藏层非截距项神经元较少的网络，当其数目增多时，系统所需要的增长的时间以及空间代价都太过昂贵。

而对于约束求解模块，因为其处理的基本单位是由一个线性不等式方程组进行约束的线性表达式，而该表达式的数目仅仅与隐藏层非截距项神经元节点数目N有关，而与神经网络的结构无关，因此其运行时间也与神经网络的结构无关。其运行时间如表 3.3 所示：

表 3.3 约束求解模块运行时间表

N	网络分解模块所需时间（秒）
5	0.385
6	0.901
7	2.401
8	6.405
9	17.347
10	50.941
11	133.649
12	332.662
13	891.655
14	2387.92

根据实际运行情况，随着隐藏层非截距项神经元节点数目N的增加，约束求解模块的运行时间以$2.6^N \sim 2.9^N$的速率呈指数上升。经分析，其速率之所以大于2^N，其原因在于随着N的上升，增加的不仅仅是线性表达式的个数及其相对应的作为约束的线性不等式方程组的组数，还有每组线性不等式方程组中不等式方程的数目，而当该数字增加时，Mathematica 处理每组线性不等式方程组的时间也会增加，这就导致了最终该模块的运行时间增长速率会大于2^N。而通过对模块的分析可知，该模块所需的内存空间增长率也因为同样的原因略大于2^N。

最后，对于可视化模块，其运行时间同样与网络结构无关，其实际运行情况如表 3.4 所示：

表 3.4 可视化模块运行时间表

N	网络分解模块所需时间（秒）

5	2.534
6	2.771
7	2.619
8	2.849
9	2.873
10	3.054
11	3.214
12	3.489
13	3.721
14	3.910

虽然在一次界面交互中，该模块仅仅只会与 Mathematica 发生恒定次数的数据传递，但与约束求解模块的情况类似，在每次数据传递中，其传递的数据变得更加复杂，分段函数的段数增加，同时分段函数的约束范围也更加复杂，这增加了 Mathematica 进行运算以及处理的复杂度，使得其需要更多的时间才能返回结果，因此随着隐藏层非截距项神经元节点数目N的增加，该模块运行时间呈常数级上升趋势。

因此，将隐藏层非截距项神经元节点数目记为N，对于整个自动化数值解析系统，其复杂度略微大于2^N，处于$2.6^N \sim 2.9^N$的范围内。根据这一原因，在接下来的 BP 神经网络的设计中，需要合理控制N的大小，以保证系统对于该神经网络是可以计算的。

3.7 本章小结

本章介绍了本实验最重要的部分——自动化数值解析系统的设计与实现。

首先介绍了该系统设计的核心算法理论。然后该理论上上将该系统设计为三个主要模块：网络分解模块、约束求解模块、可视化模块。而后分别展示了这三个模块的职责、设计思路、实现以及关键代码。之后，对该系统在模块和整体上分别进行了性能分析，分析出该系统的复杂度处于$2.6^N \sim 2.9^N$的范围内，明确了该系统的限制，为后续研究打下基础。

第四章 实验过程与结果

前一章完整介绍了自动化数值解析系统的设计以及实现，本章将会使用该系统进行实验，验证该系统的有效性，同时也对神经网络的内部工作原理进行适当的分析。

针对系统的限制，本实验搭建并训练了满足该限制的神经网络，然后通过使用自动化数值解析系统进行分析，得到了系统的解析结果以及可视化结果，再结合两者进一步分析，从而探究该神经网络在处理数据时所运用的数学原理。

4.1 实验准备

4.1.1 BP 神经网络搭建

由第三章可知，本实验所设计与开发的自动化数值解析系统所能处理的神经网络有两个关键性的特征：一是神经网络类型为 BP 神经网路；二是该网络将 ReLU 函数作为激活函数。此外，由章节 3.5 可知，因为图像可视化的维度限制，若需要将网络输入数据与输出结果结合进行可视化呈现，对于网络输入数据的维度需要控制在二维。因此，也需要设计出一种符合这一条件的数据集以方便可视化模块的功能。此外，由于自动化数值解析系统当前对所能分析的网络的隐藏层非截距项神经元节点数N有一定的限制，其运行的复杂度处于$2.6^N \sim 2.9^N$的范围内，当N增大时，系统进行分析所需要的时间以及空间都呈指数上升。根据章节 3.6 性能分析可知，当N为18时，系统的整体运行时间便会达到36小时左右，考虑到实际设备以及时间成本的限制，需要将N控制在16以内。

综上所述，该 BP 神经网络的结构需要尽可能的简单，考虑到N的限制，其隐藏层数目应该不多于3层。在此基础上，该 BP 神经网络要实现自身的功能，并将准确率保持到较高范围，从而使下一步的神经网络内部工作原理的分析具备可行性。

最终，本文将 BP 神经网络结构设计为3层，其中隐藏层为1层，拥有3个神经元节点，其结构如图 4.1 所示：

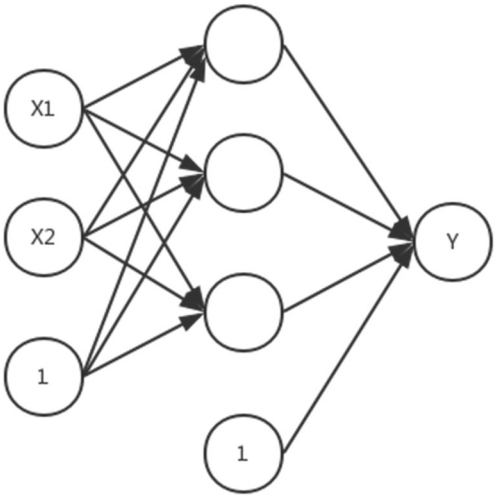

图 4.1 神经网络模型图

实验所用到的神经网络的搭建的实现借助了 Neuroph 框架。Neuroph 框架使用了类 NeuralNetwork 作为神经网络整体框架，其子类 MultiLayerPerception 作为多层神经网络，在其构造器中，除了需要以数组的形式传入神经网络各层的神经元节点数目，还需要传入预设的激活函数的枚举类型，作为该神经网络的激活函数。

在创建完成神经网络基本结构后，可以设置其学习规则，对于反向传播算法，其使用了类 BackPropagation 完成了该算法的逻辑。此外，还可以设置合适的学习速率与目标损失函数。这样便完成了 BP 神经网络的搭建。其代码如下所示：

```
//设置神经网络层数，其输出层有 2 个节点，有 2 个隐藏层，分别有 4 个和 3 个节点，其输出层有 1 个节点
int[] layers_config = {2, 4, 3, 1};
//构造神经网络，并将 ReLU 函数作为其激活函数
NeuralNetwork nn = new MultiLayerPerceptron(TransferFunctionType.RectifiedLinear, layers_config);
//将其学习规则设置为反向传播算法，配置 BP 神经网络
BackPropagation learningRule = new BackPropagation();
nn.setLearningRule(learningRule);
```

图 4.2 BP 神经网络搭建部分代码

4.1.2 BP 神经网络训练

因为在 BP 神经网络中，隐藏层第一层的神经元节点数通常要大于输入层的神经元节点数，并且各个隐藏层之间神经元节点数相差不能过大，因此隐藏

层的神经元总节点数会随着输入层的神经元节点数的增加而增大。而在自动化数值解析系统中，其运行的时间复杂度与空间复杂度与隐藏层神经元节点的数目呈指数正比关系，因此需要数据集的数据维度不能过大。另外，又由于可视化呈现的问题，因此对于某些数据集，甚至需要将其输入的维度限定在二维。

综合以上两点原因，本实验所选取的数据集其输入数据的维度应该偏小，并且输入数据所包含的特征也应该较少，输入数据与输出数据之间的关系应该较为简单。这样的数据集更适合作为本研究的初始研究数据，当输入数据与输出数据之间的关系较为简单时，处理该数据集的神经网络的内部的工作原理会更为简单与直接，更易于分析。

本文提出以线性函数作为基础来设计本实验所进行处理的目标数据集。例如，将输入数据设置为三维：x_1、x_2和x_3，将输入数据设置为一维y，y与x_1、x_2和x_3的对应关系以线性函数呈现：

$$y = \begin{cases} 0, & x_1 + x_2 + x_3 < 1.5 \\ 1, & x_1 + x_2 + x_3 \geq 1.5 \end{cases} \tag{4.1}$$

其中x_1、x_2和x_3均为 0 至 1 之间的浮点数。

选取这样的线性函数作为数据集的数学逻辑关系的好处有以下几点：一是可以灵活控制输入数据的维度，且维度能够控制在较低的层次；二是可以灵活控制输出数据的维度，因为输出数据各维数据之间没有对应关系，因此可以自由组合组成输出数据；三是输出数据仅有两种取值：0 与 1，因此可以非常直观地在测试阶段分析神经网络对于测试数据集的测试表现；四是输入数据与输出数据之间的数学逻辑关系足够简单直接，呈线性关系；五是输入数据与输出数据之间呈线性关系，在章节 3.1 中分析过，经过激活函数 ReLU 函数处理过后的 BP 神经网络，其运算始终等价于一种线性关系，这可以帮助我们更清楚地分析经过自动化数值解析系统解析而出的一系列线性表达式与原本的输入数据与输出数据之间的线性函数之间的关系，从而帮助我们分析 BP 神经网络内部的工作原理。

综合以上原因，本文以线性函数作为基础，对数据集进行设计，完成对神经网络的训练与测试，并将该网络作为自动化数值解析系统的分析对象神经网络。

实际实验中，本文用来对输出数据进行分类的线性函数定义如下：

$$y = \begin{cases} 0, & x_1 + x_2 < 1 \\ 1, & x_1 + x_2 \geq 1 \end{cases} \tag{4.2}$$

其中x_1和x_2都在[0,1]范围内。随机生成5000条数据，作为 BP 神经网络的训练数据。其代码实现如下：

```
Random rd = new Random();
double[][] inputs = new double[5000][];
double[][] outputs = new double[inputs.length][];
//人工模拟 5000 条训练数据 ， 分界线为 x1+x2+x3=1.54
for(int i = 0; i < inputs.length; i++){
    double x1 = rd.nextDouble();//随机产生一个分量
    double x2 = rd.nextDouble();//随机产生一个分量
    double x3 = rd.nextDouble();//随机产生一个分量
    inputs[i] = new double[]{x1, x2, x3};
    outputs[i] = new double[]{x1 + x2 + x3 > 1.5 ? 1 : 0};
}
```

图 4.3 测试数据生成部分代码

在 Neuroph 框架中，其提供了专用的类 DateSet 用来存储神经网络数据集。DataSet 由若干条 DataSetRow 构成，其中每一条 DataSetRow 都是一条训练数据，即包括了输出数据以及输出数据。将原始数据格式传入 DataSetRow 中，再调用 NeuralNetwork 的相应函数进行训练即可：

```
//设置每组数据集输入数据和输出数据的维数
DataSet set = new DataSet(inputs[0].length, outputs[0].length);
for (int i = 0; i < inputs.length; ++i){
    set.addRow(new DataSetRow(inputs[i], outputs[i]));
}
//对神经网络进行训练
nn.learn(set);
```

图 4.4 BP 神经网络训练部分代码

经过训练，第1层与第2层之间的权重参数为：

$$\begin{bmatrix} -3.714 & -3.632 & 3.786 \\ 0.993 & 0.926 & -0.826 \\ 0.391 & 0.504 & -0.368 \end{bmatrix}$$

第2层与第3层之间的权重参数为：

$$[-2.523 \quad 0.137 \quad 0.041 \quad 0.941]$$

对其进行测试，选取10条测试数据，如表 4.1 所示：

表 4.1 神经网络测试结果表

输入数据	预期输出	实际输出
[0.0938,0.1795]	[0]	[0.0]
[0.1432,0.3342]	[0]	[0.0]

[0.5038,0.0171]	[0]	[0.0]
[0.3489,0.4992]	[0]	[0.0]
[0.0878,0.8970]	[0]	[0.45]
[0.8673,0.1577]	[1]	[0.9683]
[0.2028,0.9268]	[1]	[0.9808]
[0.7151,0.5460]	[1]	[1.0026]
[0.9837,0.6017]	[1]	[1.0517]
[0.9385,0.9609]	[1]	[1.0979]

4.1.3 BP 神经网络持久化

对于已经完成训练的神经网络，将其进行持久化以便于自动化数值解析系统读取。Neuroph 框架提供了相应的方法对其进行持久化，其保存与读取的方法分别为：

```
//将神经网络保存到指定路径
nn.save(Utility.NETWORK_PATH);
//从指定路径读取神经网路并构造网络对象
nn = NeuralNetwork.createFromFile(Utility.NETWORK_PATH);
```

图 4.5 BP 神经网络持久化部分代码

由于自动化数值解析系统需要单独分析 BP 神经网络的神经元节点的权重参数，因此需要从 Neuroph 框架从提取出训练好的各个神经元节点的权重参数并输出到文件中，供自动化数值解析系统分析：

```
/**
 * 提取神经网络各神经元节点的权重参数
 * @param nn 训练完成的神经网络
 * @return 整个神经网络权重参数，以字符串形式返回
 */
public List<String> outputWeights(NeuralNetwork nn){
    List<String> weights = new ArrayList<>();
    //获得神经网络层，其中第 1 层是输入层，没有权重参数
    List<Layer> layers = nn.getLayers();
    for(int i = 1; i < layers.size(); ++i){
        //获得某层的神经网络节点
        List<Neuron> neurons = layers.get(i).getNeurons();
        for (int j = 0; j < neurons.size() - 1; ++j){
            //获得某节点与前一层各个神经元联接的权重参数，其中最后
```

```
                //一个神经元是截距项神经元，不存在权重参数，因此排除
                String weight = Arrays.toString(neurons.get(j).getWeights());
                weights.add(weight.substring(1, weight.length() - 1) + "\n");
            }
            weights.add("\n");
        }

        return weights;
    }
```

图 4.6 BP 神经网络权重提取部分代码

4.2 实验结果

4.2.1 数值解析结果

使用自动化数值解析系统对该神经网络权重参数进行解析，其结果如表 4.2 所示：

表 4.2 解析结果表

序号	线性表达式	约束不等式
1	$9.37x_1 + 9.16x_2 - 8.61$	$x_1 \leq 0.55 \&\& x_2 < 0.73 - 0.78x_1 \|\|$ $x_1 > 0.55 \&\& x_2 < 0.89 - 1.07x_1$
2	$0.14x_1 + 0.13x_2 + 0.83$	$x_1 > 1.26 \&\& 1.04 - 1.02x_1 < x_2 < 0.73 - 0.78x_1$
3	$9.50x_1 + 9.29x_2 - 8.72$	$0.55 < x_1 \leq 1.26 \&\& 0.89 - 1.07x_1 < x_2 < 0.73 - 0.78x_1 \|\|$ $x_1 > 1.26 \&\& 0.89 - 1.07x_1 < x_2 < 1.04 - 1.02x_1$
4	$0.02x_1 + 0.02x_2 + 0.93$	$x_1 < -3.01 \&\& 1.04 - 1.02x_1 < x_2 < 0.89 - 1.07x_1$
5	$9.38x_1 + 9.18x_2 - 8.62$	$x_1 \leq -3.01 \&\& 0.73 - 0.78x_1 < x_2 < 1.04 - 1.02x_1 \|\|$ $-3.01 < x_1 < 0.55 \&\& 0.73 - 0.78x_1 < x_2 < 0.89 - 1.07x_1$
6	$0.15x_1 + 0.15x_2 + 0.81$	$x_1 \leq -3.01 \&\& x_2 > 0.89 - 1.07x_1 \|\|$ $-3.01 < x_1 \leq 1.26 \&\& x_2 > 1.04 - 1.02x_1 \|\|$ $x_1 > 1.26 \&\& x_2 > 0.73 - 0.78x_1$
7	$9.52x_1 + 9.31x_2 - 8.74$	$-3.01 < x_1 \leq 0.55 \&\& 0.89 - 1.07x_1 < x_2 < 1.04 - 1.02x_1 \|\|$ $0.55 < x_1 < 1.26 \&\& 0.73 - 0.78x_1 < x_2 < 1.04 - 1.02x_1$

由于x_1和x_2都在[0,1]范围内，因此这些线性表达式实际情况如表 4.3 所示：

表 4.3 解析结果实际可取值表

序号	线性表达式	约束不等式

1	$9.37x_1+9.16x_2-8.61$	$0\le x_1\le 0.55\&\&0\le x_2<0.73-0.78x_1\|\|$ $0.55\le x_1\le 1\&\&0\le x_2<0.89-1.07x_1$
2	$9.50x_1+9.29x_2-8.72$	$0.55<x_1\le 1\&\&0.89-1.07x_1<x_2<0.73-0.78x_1$
3	$9.38x_1+9.18x_2-8.62$	$0\le x_1<0.55\&\&0.73-0.78x_1<x_2<0.89-1.07x_1$
4	$0.15x_1+0.15x_2+0.81$	$0<x_1\le 1\&\&1.04-1.02x_1\le x_2\le 1$
5	$9.52x_1+9.31x_2-8.74$	$0<x_1\le 0.55\&\&0.9-1.07x_1<x_2<1.04-1.02x_1\|\|$ $0.55<x_1\le 1\&\&0.73-0.78x_1<x_2<1.04-1.02x_1$

4.2.2 可视化结果

使用自动化数值解析系统对解析后的一系列线性表达式所组成的分段函数进行可视化，其图像如图 4.7 所示：

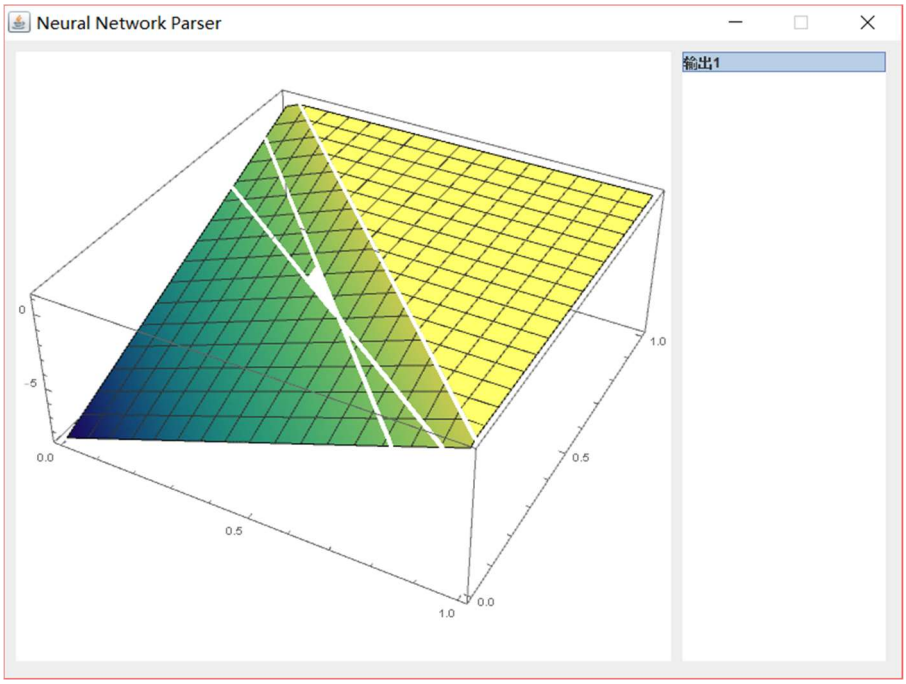

图 4.7 可视化结果截图

从图像中看到函数会以较大斜率延伸到负无穷，而在实际情况中，函数最小值为0，因此当该分段函数输出值小于0时，将其值取为0。在 Mathematica 中，记原分段函数为f，其处理后作图函数为：

$$Plot3D[Max[0,f],\{x_1,x_{1min},x_{1max}\},\{x_2,x_{2min},x_{2max}\}]$$

处理后函数图像如图 4.8 所示：

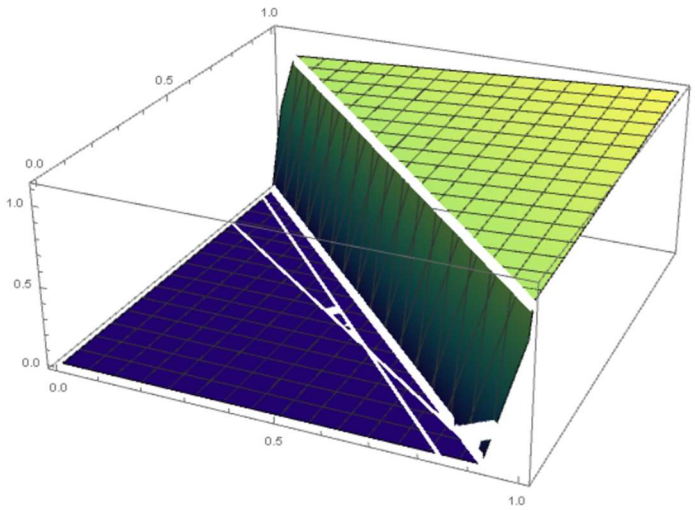

图 4.8 最小值取0后函数图像

同理，在实际情况中，函数最大值为1，因此当该分段函数输出值大于1时，将其取为1。在 Mathematica 中，其二次处理后作图函数为：

$$Plot3D[Min[1, Max[0, f]], \{x_1, x_{1min}, x_{1max}\}, \{x_2, x_{2min}, x_{2max}\}]$$

处理后函数图像如图 4.9 所示：

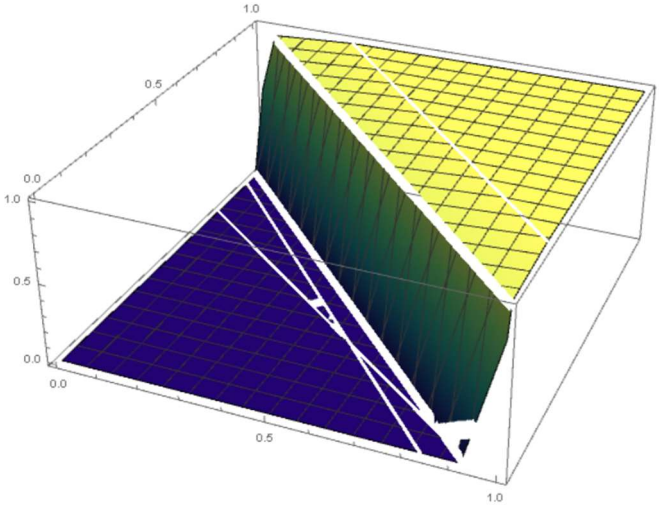

图 4.9 二次处理后函数图像

4.3 结果分析

经过分析，图 4.7 一共被分为5块，与表 4.3 的线性不等式一一对应，对应关系如图 4.10 所示：

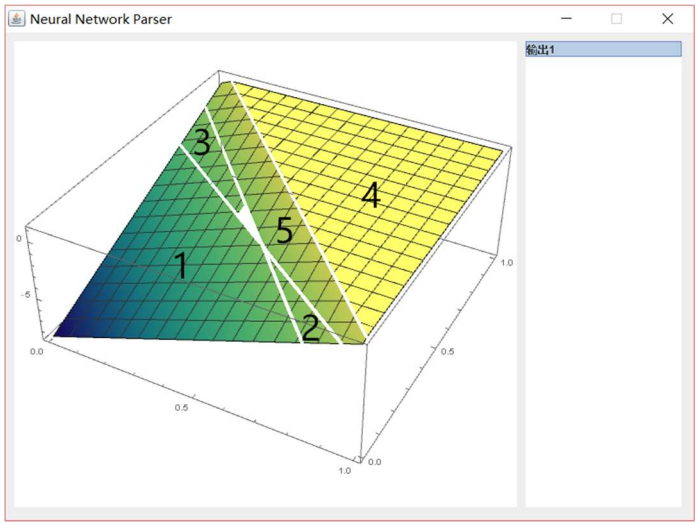

图 4.10 函数图像分析图

当x_1和x_2位于区域 4 时，由约束可知，该区域内的数据其和大于 1，对于该类型的数据，神经网络对其每个输入用较小系数相乘，相加后再加上较大常数，使得最终结果稳定在较高范围；

当x_1和x_2位于区域1、2、3、5时，由约束可知，该区域内的数据其和小于1，对于该类型的数据，神经网络对其输入用较大系数相乘，相加后减去常数，因此其输出结果会在[0,1]范围内波动，整体呈现陡峭的线性趋势，而对于区域1的数据，其和趋近于0，其中大部分区域计算结果都小于0，对于该部分的数据在处理后将其归整为0，因此得以拟合原始线性关系函数。

此外，根据对线性表达式及约束的分析可以看出，他们都并非对称函数或者对称区域，这说明神经网络在学习过程中会有细小的偏差，当该偏差越小时，神经网络的代价函数所计算出的值就会越小，其精确度就会越高。

但从整体上，这5个区域近似构成了对称区域，这说明对于该神经网络来说，x_1和x_2对其的意义基本是等价的，这也符合在原始线性函数中，x_1和x_2是相同系数这一数学含义。

4.4 本章小结

本章将自动化数值解析系统作为分析对象，对其进行分析实验，分别用数值和图表的方式展示了数值解析系统的输出结果，而后综合这两种输出结果，对其进行分析。更加直观而且深刻地剖析了神经网络的工作原理，即在以 ReLU 函数作为其激活函数时，BP 神经网络是如何根据输入数据的范围以及数值，将实际输出尽可能拟合预期输出的。

第五章 总结与展望

5.1 总结

本毕业论文首先分析了学术界与企业界对神经网络内部工作原理研究的现状，提出了使用数值分析的方法去解析神经网络的内部的运算的方法，结合可视化的手段去分析神经网络的内部工作原理。在当前已有的关于神经网络内部工作原理的研究中，有通过可视化去辅助研究者进行研究的工作，也有通过对输入数据进行处理去分析神经网络对各部分输入数据敏感度的工作。而本设计借助数学思维，将神经网络进行分解，可以说是一种全新的思路。

而后本文结合当前大数据背景进行分析，选择人工神经网络最为基础且最为常用的类型 BP 神经网络作为研究对象，并根据当前深度学习逐渐进入主流视野的趋势，选择了深度学习中最为常用的 ReLU 函数作为网络的激活函数，构成了本次设计的研究对象。之后，本文介绍了本设计所借助的两大工具：当前最为强大的科学计算工具 Mathematica 以及以 Java 编写灵活可变的神经网络框架 Neuroph。

之后，本文重点介绍了此次设计的主体——自动化数值解析系统。详细介绍了其支撑的算法理论以及设计，根据设计将系统分解为三个模块：网络分解模块、约束求解模块、可视化模块。再对这三个模块逐一进行讲解，分别说明其职责、内部逻辑以及关键代码。并且分析了该系统的性能，从而明确了该系统复杂度较高的缺点，对其所能处理的神经网络结构提出了限制。

最后，在满足该限制的基础上，本文构建了合理、简单且有效的 BP 神经网络，并设计生成了满足网络约束且尽可能符合 ReLU 函数特征的数据集去训练该 BP 神经网络。在试验中使用自动化数值解析系统去分析该网络，展示了系统数值解析的结果，同时也展示了系统对神经网络进行可视化后的函数图像，并且综合数据和图像分析，合理而深刻地剖析了 BP 神经网络所学习到的数据特征在其内部运算中的体现，取得了预期的成果。

5.2 展望

本次设计虽然设计并实现了能够解析特定神经网络的自动化数值解析系统，但通过对系统的理论分析以及实际的实验结果，可以明确该系统仍然只是一个雏形，其所能处理的问题非常有限。针对该系统所存在的不足，未来的研究方向主要有以下几个方面：

1. 该自动化数值解析系统当前只能处理以 ReLU 函数作为激活函数的 BP 神经网络，对于运算难度更大的激活函数以及结构更加复杂的神经网络，该系统仍然需要进一步的研究与优化；
2. 该网络将 ReLU 函数作为激活函数是为了适应当前深度学习应用日益广泛的趋势，而深度学习网络的结构通常都相当庞大，其节点数以千为单位计，该系统2^N的复杂度无法应用在这种网络上，因此需要对其算法进一步的优化；
3. 当前系统提供了可视化的功能，但仅仅局限于二维输入，对于更高维的输入尚未找到一个合理的方法去将其呈现为图像，在之后的工作研究中这也是一个可以改进的方面；
4. 对于系统所解析的结果，仍然需要人工干预去分析，而当数据集变得更为复杂、输入数据维度变得更高时，其必将对分析难度造成影响，因此可以设计出能够分析解析结果的自动化模块；
5. 通过对系统解析后产生的结果，可以将其与其它研究结合进行应用，例如可以根据分解出的线性表达式的有效性来判断神经网络中各个节点的重要程度，移除那些不重要的冗余节点，从而达到精简神经网络的目的。

参考文献

[1] D. A. Pomerleau, Efficient Training of Artificial Neural Networks for Autonomous Navigation, In *Neural Computation*, Vol. 3, No. 1, pp. 88-97, 1991.

[2] X.-W. Chen, X. Lin, Big Data Deep Learning: Challenges and Perspectives, In *IEEE Access*, Vol. 2, pp. 514-525, 2014.

[3] He, K., Zhang, X., Ren, S., And Sun, J., Deep Residual Learning for Image Recognition, In *IEEE Conference on Computer Vision and Pattern Recognition*, 2015

[4] Inceptionism: Going Deeper into Neural Networks. URL https://ai.google-blog.com/2015/06/inceptionism-going-deeper-into-neural.html.

[5] Visualizing Neural Network Layer Activation (Tensorflow Tutorial). URL https://medium.com/@awjuliani/visualizing-neural-network-layer-activation-tensorflow-tutorial-d45f8bf7bbc4.

[6] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, Hod Lipson, Understanding Neural Networks Through Deep Visualization, arXiv:1506.06579v1, 2015.

[7] Matthew D Zeiler, Rob Fergus, Visualizing and Understanding Convolutional Networks, arXiv:1311.2901v3, 2013.

[8] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, Rob Fergus, Deconvolutional networks, In *IEEE Conference on Computer Vision and Pattern Recognition*, 2010.

[9] The Unreasonable Effectiveness of Recurrent Neural Networks. URL http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[10] A.K. Jain, J. Mao, K.M. Mohiuddin, Artificial Neural Networks: A Tutorial, In *Computer*, Vol. 29, pp. 31-34, 1996.

[11] Jack V. Tu, Advantages and Disadvantages of Using Artificial Neural Networks versus Logistic Regression for Predicting Medical Outcomes, In *J Clin Epidemiol* Vol. 49, No. 11, pp. 1225-1231, 1996.

[12] F. Rosenblatt, The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, In *Psychological Review*, 1958.

[13] R. Hecht-Nielsen, Theory of the back-propagation neural network, In *Proceedings of the International Joint Conference on Neural Networks*, pp. 593-605, 1989.

[14] B. Karlik and A. Vehbi, Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Networks, In *Information Journal of Artificial Intelligence and Expert Systems*, Vol. 1, No. 4, p. 111, 2011.

[15] Glorot, X., Bordes, A., and Bengio, Y., Deep Sparse Rectifier Neural Networks, In *Porceeding of the Conference on Artificial Intelligence and Statistics*, 2011.

[16] Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet Classification with Deep Convolutional Neural Networks, In *Advances in neural information processing systems*, pp. 1097-1105, 2012.

致谢

感谢指导老师汤恩义老师的指导。
感谢其他所有为本实验提供帮助的老师与同学。