毕 业 论 文

题　　目　面向深度学习应用的
FPGA 加速器原型设计与实现

姓　　名　杨　韵

学　　号　14073226

指导教师　蔡　旻

日　　期　2018 年 6 月 4 日

北京工业大学

毕业设计（论文）任务书

题目　面向深度学习应用的 FPGA 加速器原型设计与实现

专业　物联网工程　　学号　14073226　　姓名　杨韵

主要内容、基本要求、主要参考资料等：

主要内容：

（1）通过实验分析深度学习应用的基本计算和数据访问特征。

（2）掌握 FPGA HLS（高层次综合）技术，了解基本的 FPGA 加速器设计方法与技巧。

（3）在选定的 FPGA 开发板中设计实现面向深度学习应用的 FPGA 加速器原型，并对其性能与功耗进行分析比较。

基本要求：

1、参与本课题的同学将根据用户需求，进行系统分析、系统设计、系统实现。

2、系统分析、设计、实现过程应遵循系统开发规范。

3、课题进行期间，每周保证不少于 40 学时从事课题研究工作；每周至少一次到校汇报课题进度及接受指导。

4、课题结束应整理出系统相应文档。

时间安排：

寒假：毕设准备，阅读相关资料，了解系统需求。

第二周：提交开题报告。

第五周：系统分析与设计。

第八周：系统基本实现，中期检查。

第十一周：系统完善。

第十三周：测试。

第十四周：完成设计文档。

第十五周：撰写论文。

参考文献：

[1] Yakun Sophia Shao and David Brooks, “Research Infrastructures for Hardware Accelerators,” in Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2015.

[2] Yu-Ting Chen, Jason Cong, Michael Gill, Glenn Reinman, and Bingjun Xiao, “Customizable Computing,” in Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2015.

完成期限：2018 年　月　日

指导教师签章：

专业负责人签章：

2018 年　月　日

声 明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京工业大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签名：　　日期：

关于论文使用授权的说明

本人完全了解北京工业大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

（保密的论文在解密后应遵守此规定）

签名：　　导师签名：　　日期：

摘要

随着计算机硬件和理论的不断发展，深度学习成为一个研究热点。深度学习应用程序具有计算量大、访存密集且不规则等特点，传统的 CPU 和 GPU 等多核体系结构受自身限制，无法有效满足深度学习应用的硬件需求。FPGA 作为一种灵活高效的半定制化硬件技术，成为被学术界广泛应用的加速器平台技术。发展基于 FPGA 的深度学习应用加速器能有效改善深度学习应用的性能与功耗，从而提高深度学习技术的可用性和可扩展性。

分析 FPGA 及 Zynq-7000 的一些特性，阐述其作为常用加速器的优势。分析所选激活函数 Sigmoid 的相关特性，以及与其他激活函数的区别，并说明在 FPGA 中如何用查找表实现 Sigmoid 函数。分析经典的三层感知机网络模型，在 MATLAB 中实现一个神经网络，利用 MNIST 数据集训练出神经网络的最优权重矩阵。在此基础上用 Chisel 设计并实现神经元模块、层模块以及一个两层的神经网络模块。同时为各个模块分别设计测试单元，以检测其功能性。

关键词 ：感知机；深度学习；FPGA ；Chisel；Zynq

Abstract

With the continuous development of computer hardware and theory, deep learning has become a research hotspot. Deep learning applications have the characteristics of large amount of calculation, intensive and irregular access, and traditional multi-core architectures such as CPUs and GPUs are limited by themselves and cannot effectively meet the hardware requirements of deep learning applications. As a flexible and efficient semi-customized hardware technology, FPGA has become an accelerator platform technology widely used by academia. The development of FPGA-based deep learning application accelerators can effectively improve the performance and power consumption of deep learning applications, thereby improving the usability and scalability of deep learning technologies.

Analyze some features of FPGA and Zynq-7000, and explain its advantages as a common accelerator. Analyze the relevant features of the selected activation function Sigmoid, and distinguish it from other activation functions, and explain how to use a lookup table to implement Sigmoid function in an FPGA. Analyze the classical three-layer sensor network model, implement a neural network in MATLAB, and use the MNIST data set to train the optimal weight matrix of the neural network. Based on this, use Chisel to design and implement neuron modules, layer modules, and a two-layer neural network module. At the same time, test units are designed for each module to test its functionality.

Keywords ： Perceptron ; Deep Learning ; FPGA Chisel ;Zynq

目录

1. 绪论

1.1 选题目的及意义

人工神经网络（简称神经网络）是最初受到神经科学启发的机器学习技术的一个大家族，日前一直在向更深更大的结构发展。深度神经网络在人工智能界占据着统治地位，凡是有关人工智能的产业报道，都必然离不开深度学习。近年来，深度学习的网络规模在不断增加，数据以及计算的复杂性也随之剧增。

神经网络的深度学习概念源自于对人工神经网络的研究。建立、模拟人脑进行分析学习的神经网络，模仿人脑的机制来解释诸如图像，声音和文本这些数据。基于深信度网(DBN)提出非监督贪心逐层训练算法，为解决深层结构相关的优化难题带来希望，随后提出多层自动编码器深层结构。卷积神经网络是第一个真正多层结构学习算法，它利用空间相对关系减少参数数目以提高训练性能。

目前,卷积神经网络主要基于通用处理器实现,但基于软件方式无法充分挖掘卷积神经网络的并行性，在实时性和功耗方面都不能满足应用的需求。

FPGA 又称为现场可编程门阵列，通过硬件编程来实现并行运算。由于 FPGA 计算资源丰富、灵活可配、开发周期短，越来越多研究者喜欢采用 FPGA 开发基于卷积神经网络的应用。作为一种灵活高效的半定制化硬件技术，被学术界广泛选作面向特定应用的加速器平台技术。发展基于 FPGA 的深度学习应用加速器能有效改善深度学习应用的性能与功耗，从而提高深度学习技术的可用性和可扩展性。用 FPGA 实现神经网络，可大大加快计算速度，满足实时性要求，降低功耗，且有利于将神经网络学习推广至嵌入式领域，同时减小系统体积。

1.2 国内外文献综述

高性能低能耗的实现深度学习相关算法，已成为当下的研究热点。虽然计算成本很高，但[1,3～5]所示的深度学习神经网络技术已经成为了广泛应用的先进技术（如[7]中提到的模式识别和[8]中的网络搜索）,有些甚至在特定的任务上取得了人性化的表现，比如[13]中的 ImageNet 识别和[18]中的 Atari 2600 视频游戏。

现有的神经网络技术已经在其网络拓扑和学习算法中表现出显着的多样性。例如，[32]中的深度信任网络（DBN）由一系列层组成，每个层完全连接到相邻的层。相比之下，[3]中的卷积神经网络（CNN）使用卷积/合并窗口来指定神经元之间的连接，因此连接密度远低于 DBN。DBN 和 CNN 的连接密度均低于[33]中的玻尔兹曼机器（BM），它们将所有的神经元彼此完全连接起来。针对不同神经网络的学习算法也可能存在差异，如[35]中用于训练多层感知器（MLP）的反向传播算法，[33]中用于训练受限玻尔兹曼机器（RBM）的吉布斯采样算法，以及[34]中用于训练自组织映射（SOM）的非监督学习

算法。尽管采用高级、复杂和信息性的指令可能是支持一小组类似神经网络技术的加速器的可行选择，但显着的多样性和大量现有神网络技术使得构建单个加速器不可行，它使用大量的高级指令来涵盖范围广泛的神经网络。此外，没有一定程度的通用性，即使现有的成功加速器设计可能很容易因为神经网络技术的发展而变得可用。

[9,10]中前馈深度神经网络（DNNs）在语音和模式识别应用中表现出相当好的性能。前馈深层神经网络的实时实现需要大量的算术和存储器访问操作，因此 DNN 通常使用 GPU（图形处理单元）[11,12]来实现。基于 GPU 的实现消耗超过 100 瓦的大功率。另外，基于 GPU 的系统需要占用大量空间的 PC，这可能不适合需要小型印刷单元的嵌入式应用。

有些关于基于 VLSI 和 FPGA 的 DNN 或 CNN（卷积神经网络）的实现方法。[6]中的系统将权重存储在外部 DRAM 中，并且可以相当灵活地配置算法。但是，这个系统需要大量的外部存储器访问，并且吞吐量相当有限。在[7]中开发了一个完全定制的 VLSI，它采用数千个处理单元并将权重存储在片上存储器中。这种基于自定义 VLSI 的系统可以实现非常高的吞吐量并消耗小功率，但不够灵活。

使用量化或修剪降低神经网络的复杂性已经被研究了很多[19～21]。[22～24]发展了反向传播再训练，而不是直接加权量化。所设计的网络通常使用 2～8 位的权重，并且使用多于 7 位的模拟或高精度固定点表示信号。

神经网络技术的计算量非常大，传统上是在 CPU 和 GPGPU 组成的通用平台上执行的，这些通用平台对于神经网络技术通常不是节能的[16]，因为他们投入了过多的硬件资源来灵活地支持各种工作负载。在过去的十年中，如[25，26，28]中已经有许多定制化为神经网络的硬件加速器，在 FPGA 上实现。[16]中 Chen 等人提出了一个小型的神经网络加速器，称为 DianNao，它的指令直接对应于 CNN 中的不同层类型。DaDianNao 采用了类似的指令集，但通过保持所有的网络参数在芯片上，达到了更高的性能和能源效率，这是加速器体系结构而不是 ISA 的创新[1D5]。所以 DaDianNao 的应用范围仍然受到 ISA 的限制，与 DianNao 的情况相似。[31]中 Liu 等人设计了可以容纳 7 种经典机器学习技术的 PuDianNao 加速器，其控制模块只提供 7 种不同的操作码（每种操作码对应一种特定的机器学习技术）。因此，PuDianNao 只允许对七种机器学习技术进行微小的改动。总而言之，指令集缺乏灵活性，阻碍了以前的加速器灵活有效地支持各种不同的神经网络技术。

[2]中提出了一种新的用于神经网络加速器的领域专用轻量级指令集架构（Instruction Set Architecture,ISA），称为 Cambricon，是一种集标量、向量、矩阵、逻辑、数据传输和控制指令于一体的加载-存储架构。与[15]中最新的最先进的神经网络加速器设计 DaDianNao（其仅能适应 3 种神经网络技术）相比，[2]中基于 Cambricon 的加速器原型在 TSMC 65nm 技术中实现的延迟、功耗和面积管理费用（分别为 4.5%/4.4%/1.6%），涵盖 10 个不同的神经网络基准。这个原型加速器可以容纳所有十个基准神经网络，而最先进的神经网络加速器 DaDianNao 只能支持其中的三个。即使在对三个基

准神经网络进行测试的时候，可以实现与最先进的加速器相媲美的性能/能效，而且开销可以忽略不计。

1.3 本论文所研究的内容

通过实验分析深度学习应用的基本计算和数据访问特征；分析深度神经网络的预测过程和训练过程的算法共性和特性，在此基础上设计多层神经网络。实现神经元以及神经网络的模块设计。

掌握 FPGA HLS（高层次综合）技术，了解基本的 FPGA 加速器设计方法与技巧；在选定的 FPGA 开发板中设计实现面向深度学习应用的 FPGA 加速器原型，并对其性能与功耗进行分析比较。最终实现面向深度学习应用的 FPGA 加速器原型设计。

为了验证设计的功能性，需采用 MINST 数据集作为系统验证。用训练集 (training set)在 MATLAB 中将最优权重矩阵训练出来，然后用测试集(test set)进行功能性验证。

1.3.1 感知机网络

多层感知机网络是神经网络中应用最广、研究最多的网络之一。其结构为一个输入层、一个输出层、若干个隐含层，每层由多个神经元组成。神经元之间互相连接构成的一个非循环的图，也就是说一些神经元的输出会作为其他神经元的输入；环路不能存在是因为这会使得神经网络的前向传播陷入无止尽的循环中。当然，神经元之间的排列是有规律的，通常情况下被构建成层层连接的形式，每一层中又有多个神经元。通俗的来讲，神经网络就是一个学习器，给它一组输入，它会得到一组输出，神经网络里的节点相互连结决定了输入的数据在里面经过怎么样的计算。每个神经元到另一个神经元的连接权(后者对前者输出的反应程度)是可以接受外界刺激而改变的，这构成了学习机能的基础。

在设计神经网络时，输入层的节点数需要与特征的维度匹配；输出层的节点数要与目标的维度匹配；而隐藏层的节点数是由设计者指定的。节点也就是神经元。多层网络，顾名思义就是有多个节点层结构组成的网络系统，它的每一层都是由若干神经元节点构成，该层的任意一个节点都和上一层的每一个节点相连，由上一层的节点来提供输入，经过计算产生该节点的输出并作为下一层节点的输入。

多层神经网络的本质就是复杂函数拟合。多层神经网络中，输出是按照一层一层的顺序来计算。从输入层开始，之后的层在算出所有单元的值以后，再继续对更深一层进行计算。只有当前层的所有单元的值都计算好以后，才会算下一层。计算向前不断推进，这个过程就叫做“正向传播”。

神经网络又分为前馈神经网络和后馈神经网络。若把结构图看作是有向图。神经元代表其中的顶点，连接则代表有向边。那么对于前馈神经网络，这个有向图是没有回路

的。而对于反馈神经网络，这个有向图是有回路的。后馈神经网络也是一类重要的神经网络。深度学习中的 RNN 就属于一种反馈神经网络。

一个三层的神经网络结构如图 1.1。

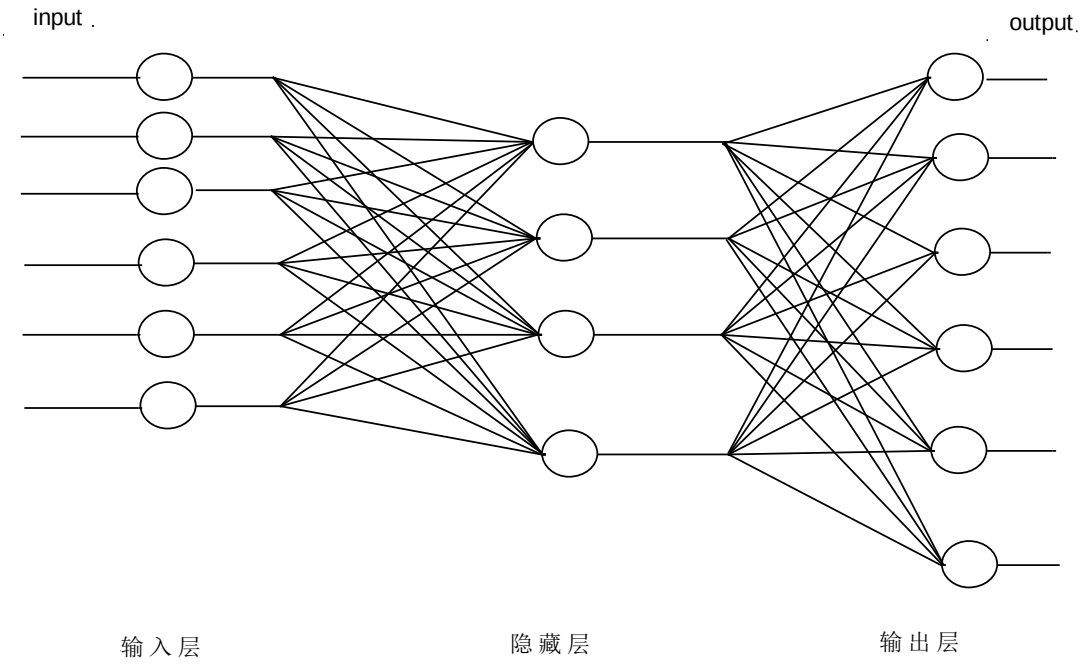

图 1.1 三层神经网络结构图

该神经网络结构为一个输入层、一个输出层、一个隐含层。输入层有 2 个神经元，隐藏层有 5 个神经元,输出层有 3 个神经元。只在隐藏层和输出层的各突触上设置权值，就是说输入层与隐藏层之间，各隐藏层之间以及隐藏层与输出层之间的个神经元之间都有一个连接，每个连接上都有一个权重值。那么图 1.1 中的输入层与隐藏层之间的连接有 5*2 个，需要设置权重值 10 个；隐藏层与输出层之间的连接有 5*3 个，需要设置权重值 15 个。

输入层负责接收输入，然后传递给隐藏层；隐藏层将上一层的输出作为其输入，然后进行逻辑运算，再输出到下一层；输出层也将上一层的输出作为其输入，进行逻辑运算，输出层的输出作为整个神经网络的输出。

神经网络具有这个功能：通过大量的输入，神经网络调整自身的连接情况，其实主要就是调整权重矩阵，然后给出我们的预期输出。

1.3.2 MNIST 数据集

本论文中所用的 MNIST 是一个手写数字数据库，是 NIST 数据库的一个子集。来自美国国家标准与技术研究所,它有 60000 个训练样本集(training set)和 10000 个测试样本集(test set)，共 70000 张手写数字的灰度图片。

文件不是标准的图像格式，图像数据都保存在解压后得到的二进制文件中。每个样本图像的宽高为 28*28，也就是说每一张图片包含 28*28 个像素点，可以用一个数组来表示这张图片。训练数据集标签是介于 0 到 9 的数字,用来描述给定图片里表示的数字。

在 MATLAB 中设计实现一个神经网络，其整体模块结构如图 1.2 所示。

图 1.2 MATLAB 中设计的神经网络模块结构示意图

定义各层神经元、权重、激活函数（第 2 章详细描述）以及学习率，然后在初始化函数里进行初始化。在负责训练和测试的函数里，将整个神经网络的输入作为输入层的输入；然后输入层的输出作为隐藏层的输入；隐藏层的输出作为输出层的输入；输出层的输出作为整个神经网络的输出。这样就能将整个神经网络连接起来。然后定义一个存储权重矩阵的函数。

由于每张图都有 784 个像素点,将各像素作为特征,所以该神经网络的输入层有 784 个神经元，隐藏层设置为 100 个神经元，输出层有 10 个神经元。将训练数据集读入，在测试时依次调用训练、测试和存储权重的函数，将训练得到的最优权重矩阵存储下来以供后续实验使用。

训练算法的目的就是让整个神经网络权重的值调整到最佳，以使得整个网络所达到的预测效果最好。前馈网络中，不论是离散还是连续，一般都不考虑输入和输出之间在时间上的滞后性，而只是表达两者间的映射关系。反馈就是每个神经元同时将自身的输出信号作为输入信号反馈给其他神经元，目的是为了让输出能够受到所有神经元的输出的控制，从而使得各个神经元的输出相互制约。

通过后向反馈算法，将输出依次传递到上一层，根据实际值与期望值的对比，调整网络中的各个权值，多次训练和多次修改，使整个网络的预测效果最好。

1.3.3 FPGA 加速神经网络

FPGA(Field-Programmable Gate Array)，即现场可编程门阵列，作为常用的加速手段之一。它是作为专用集成电路(ASIC)领域中一种半定制电路而出现的，解决了定制电路的不足,同时克服了原有可编程器件门电路数有限的缺点。以硬件描述语言(Verilog 或 VHDL）完成的电路设计，可以经过简单的综合、布局，烧录至 FPGA 上进行测试。

以下是 FPGA 的几个主要特点：

（1）采用 FPGA 设计 ASIC 电路(专用集成电路)，用户不用投片生产，就可以得到合用的芯片。

（2）FPGA 可做其它半定制或全定制 ASIC 电路的中试样片。

（3）FPGA 内部有着丰富的 I/O 引脚和触发器。

（4）FPGA 是 ASIC 电路中设计开发费用最低、周期最短、风险最小的器件之一。

（5）FPGA 采用高速 CMOS 工艺，功耗低，可以与 CMOS、TTL 电平兼容。

简而言之就是高性能、低功耗、可编程。

可以说，FPGA 芯片是小批量系统提高系统集成度、可靠性的最佳选择之一。

FPGA 是由存放在片内 RAM 中的程序来设置其工作状态的，因此，工作时需要对片内的 RAM 进行编程。用户可以根据不同的配置模式，采用不同的编程方式。

加电时，FPGA 芯片将 EPROM 中数据读入片内编程 RAM 中，配置完成后，FPGA 进入工作状态。掉电后，FPGA 恢复成白片，内部逻辑关系消失，因此，FPGA 能够反复使用。FPGA 的编程无须专用的 FPGA 编程器，只须用通用的 EPROM、PROM 编程器即可。当需要修改 FPGA 功能时，只需换一片 EPROM 即可。这样，同一片 FPGA，不同的编程数据，可以产生不同的电路功能。因此，FPGA 的使用非常灵活。

FPGA 是一种高度密集型计算加速器件，以并行运算为主，可通过硬件描述语言完成算法实现，从而利用 FPGA 的硬件结构特性实现并行运算的加速。以硬件描述语言来实现。

卷积神经网络加速系统通常采用“主机＋FPGA” 架构，其中：主机包括处理器核和 DRAM 控制器，负责计算和数据传输的控制；而 FPGA 端用于实现数据缓存、数据处理单元以及控制电路等，主要负责对数据的缓存和处理。

卷积神经网络操作的数据相对比较规整，对数据传输的灵活性要求较低，因此采用 FPGA 实现数据传输的控制代价较低，同时能避免处理器和 FPGA 之间过多的通信操作。

2. 总体设计

2.1 神经网络设计

在设计神经网络时，输入层的节点数需要与特征的维度匹配；输出层的节点数要与目标的维度匹配；而隐藏层的节点数是可更改的。两层的各个神经元之间都有连接，同层的神经元互不相连。

由于需要利用 MNIST 数据集对神经网络进行训练和测试。其每张图都有 784 个像素点。需将各像素作为特征，所以该神经网络有 784 个输入，而要得到的目标为 0～9 这 10 个数字之一，所以输出层的神经元个数为 10。

整个神经网络的数据处理流程是先输入 784 个特征,让这些输入先在隐含层做逻辑运算（第 3 章第 2 节），再经过激活函数（第 2 章第 2、3 节）处理后得到结果。将隐藏层的结果作为输出层的输入，然后再做同样的处理，得到的结果作为该神经网络的输出。

在 MATLAB 中设计实现一个神经网络，具体设计思路见第 1 章第 3 节。然后训练得到最优的权重矩阵，使该神经网络能达到 97%左右的准确率。

经过训练分别得到隐藏层和输出层的权重矩阵。隐藏层权重矩阵为 100*784 的数组,既隐藏层共 100 个神经元，每个神经元有 784 的突触。输出层权重矩阵为 10*100 的数组，输出层共 10 个神经元，每个神经元有 100 的突触。

在“IntelliJ IDEA Community Edition”中用 Scala 实现神经网络的设计。所设计的神经网络以左到右的形式表达的结构图如图 2.1 所示。

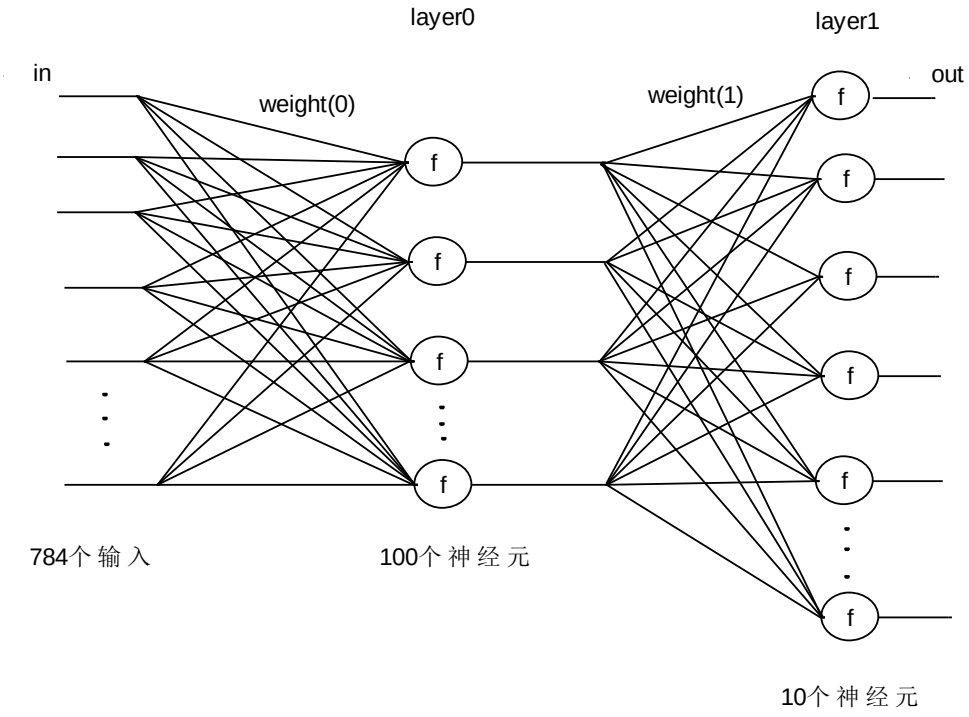

图 2.1 神经网络

本文设计的神经网络的结构只有两层：一个隐藏层（layer0）、一个输出层（layer1）。不设计输入层而是直接将输入给到 layer0 的每一个神经元。由于隐藏层和输出层的逻辑结构相同，且都与输入层不同，在实现时只需要实现一个层的模块设计。

图 2.1 中 in 表示神经网络的输入，out 表示神经网络的输出；layer0 表示第一层，layer1 表示输出层；weight（0）表示 layer0 的权重矩阵，weight（1）表示 layer1 的权重矩阵；f 表示神经元内部运算，由两部分组成：乘积累加函数（详细逻辑运算见第 3 章第 2 节）以及激活函数(见本章第 2 节)。

需要将神经网络的权重存储在外部，而不是先放入神经元。所以要先将权重值给神经元，再输入特征，然后才能进行逻辑运算。于是需要输入的数据有权重和特征。其中权重除了要包含权重值，还需包含其所属的突触的信息：第几层、第几个神经元以及第几个突触。

784 个输入直接给 layer0，layer0 的输出给 layer1，然后将 layer1 的输出作为神经网络的输出。实现神经网络首先要实现单个神经元，其次在单个神经元的基础上实现层，最后整合多个层实现神经网络。

神经元是神经网络中最基本的结构，也可以说是神经网络的基本单元，它的设计灵感完全来源于生物学上神经元的信息传播机制。一个神经元通常具有多个树突，主要用来接受传入信息；而轴突只有一条，轴突尾端有许多轴突末梢可以给其他多个神经元传递信息。轴突末梢跟其他神经元的树突产生连接，从而传递信号。这个连接的位置在生物学上叫做“突触”。

神经元模型是一个包含输入、输出与计算功能的模型。输入可以类比为神经元的树突，而输出可以类比为神经元的轴突，计算则可以类比为细胞核。连接是神经元中最重要的东西。每一个连接上都有一个权重。

一个经典的神经元模型如图 2.2 所示。

图 2.2 中 in_1、in_2 和 in_3 表示个输入，w_1、 w_2 和 w_3 表示连接（中间的箭头线）上的权重值，用 f 代表神经元的内部计算，out 表示输出。在别的绘图模型里，有向箭头可能表示的是值的不变传递。而在神经元模型里，每个有向箭头表示的是值的加权传递。

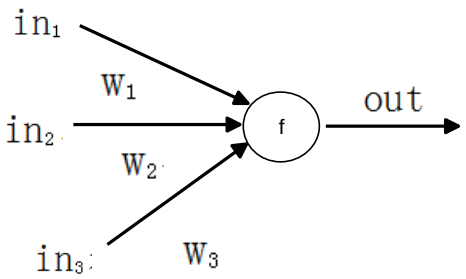

图 2.2 经典神经元模型

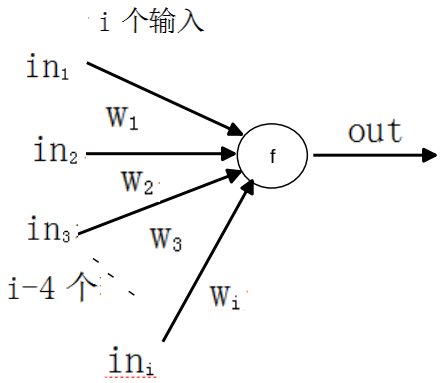

图 2.3 本文设计的神经元模型

本论文设计的神经元的结构模型如图 2.3 所示。

in_1、in_2、in_3直到 in_i分别表示 i 个突触的输入，w_1、 w_2、w_3直到 w_i分别表示 i 个突触上的权重值，f 代表神经元的内部逻辑运算，out 表示输出。每个神经元有 i 个输入，只有一个输出。

在所设计的神经元的基础上实现含有多个神经元的层，层输入就是该层神经元的输入，层输出就是该层各神经元的输出的集合，然后将该层的输出作为下一层的输入。如此便可实现整个神经网络。

需要在在 MATLAB 中设计实现一个神经网络(见第 1 章第 3 节第 2 部分),利用 MNIST 数据集的训练数据，在此神经网络中将神经网络的最优权重训练出来，以备使用。

实现整个神经网络后，把得到的两个最优权重矩阵输入到神经网络，存入到各神经元的存储单元中， 然后用 MNIST 数据集的测试数据测试该神经网络的功能性。

2.2 激活函数

2.2.1 激活函数定义及特性

在人工神经网络中需要对大量的非线性函数进行计算。因此，研究高速地处理非线性函数具有非常重要的意义。激活函数是神经网络的一个重要组成部分，代表了轴突接收到冲激信号的频率。它对于人工神经网络模型去学习、理解非常复杂和非线性的函数来说具有十分重要的作用。它将非线性特性引入到我们的网络中。其主要目的是将神经网络模型中一个节点的输入信号转换成一个输出信号。该输出信号现在被用作堆叠中下一个层的输入。在神经网络中的具体操作是这样的，我们做输入（X）和它们对应的权重（W）的乘积之和，并将激活函数 f（x）应用于其获取该层的输出并将其作为输入送到下一层。以模拟生物神经元对突触传来的电信号产生兴奋输出电脉冲的过程。

激活函数通常有如下一些性质：

（1）非线性。 当激活函数是线性的时候，一个两层的神经网络就可以逼近基本上所有的函数了。但是，如果激活函数是恒等激活函数的时候（即$f(x) \approx x$），就不满足这个性质了，而且如果 MLP 使用的是恒等激活函数，那么其实整个网络跟单层神经网络是等价的。

（2）可微性。当优化方法是基于梯度的时候，这个性质是必须的。

（3）单调性。当激活函数是单调的时候，单层网络能够保证是凸函数。

（4）$f(x) \approx x$。 当激活函数满足这个性质的时候，如果参数的初始化是 random 的很小的值，那么神经网络的训练将会很高效；如果不满足这个性质，那么就需要很用心的去设置初始值。

（5）输出值的范围。 当激活函数输出值是有限的时候，基于梯度的优化方法会更加稳定，因为特征的表示受有限权值的影响更显著；当激活函数的输出是无限的时候，模型的训练会更加高效，不过在这种情况小，一般需要更小的学习率。

在神经网络中，激活函数的作用是能够给神经网络加入一些非线性因素，使得神经网络可以更好地解决较为复杂的问题。如果不运用激活函数的话，则输出信号将仅仅是一个简单的线性函数。线性函数一个一级多项式。现如今，线性方程是很容易解决的，但是它们的复杂性有限，并且从数据中学习复杂函数映射的能力更小。一个没有激活函数的神经网络将只不过是一个线性回归模型（Linear regression Model）罢了，它功率有限，并且大多数情况下执行得并不好。同样是因为没有激活函数，神经网络将无法学习和模拟其他复杂类型的数据，例如图像、视频、音频、语音等。这就是为什么要使用人工神经网络技术，诸如深度学习（Deep learning），来理解一些复杂的事情，一些相互之间具有很多隐藏层的非线性问题，而这也可以帮助了解复杂的数据。

2.2.2 神经网络中常用的激活函数

常用的激活函数有 Tanh 函数、Sigmoid 函数和 ReLu 函数。

在人工神经网络中应用最为广泛的是 Sigmiod 函数。

目前对于 Sigmiod 函数实现技术的研究主要分为软件实现和硬件实现两个方面。由于软件相比硬件而言速度较慢并且并行程度很低，所以无法满足其快速处理的要求[2]。因此，在超大规模集成电路快速发展的当今时期，研究如何利用硬件快速处理 Sigmiod 函数显然更加有意义。

本论文选用的正是激活函数 Sigmiod，其表达式如公式 2.1 所示。

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.1}$$

其对应的图像如图 2.2：

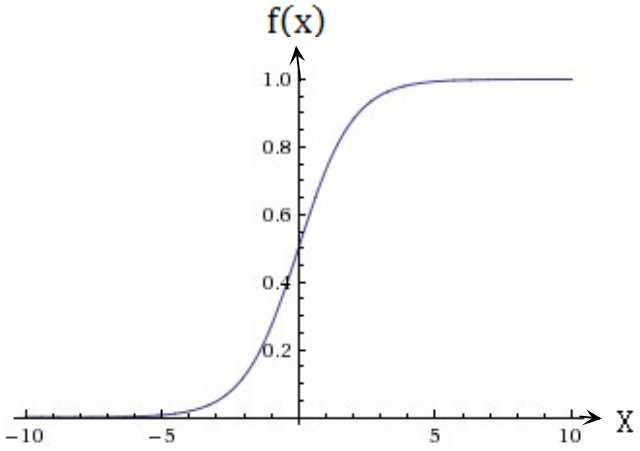

图 2.2 Sigmiod 函数图

输入一个实值的数，然后将其压缩到 0～1 的范围内，是一个 S 形曲线。特别地，大的负数被映射成 0，大的正数被映射成 1。它能够很好的表达“激活”的意思，未激活就是 0，完全饱和的激活则是 1。

Sigmiod 函数的输出映射在(0,1)之间，单调连续，输出范围有限，优化稳定，可以用作输出层；求导容易。但由于其软饱和性，容易产生梯度消失，导致训练出现问题；会导致后一层的神经元将得到上一层输出的非 0 均值的信号作为输入。且参数收敛速度很慢，影响了训练的效率。

Tanh 函数的表达式如公式 2.2 所示。

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{2.2}$$

X 对应的函数图像如图 2.4 所示。

Tanh 函数是 0 均值的，将一个实数输入映射到[-1,1]范围内。也存在梯度饱和问题。符合人脑神经饱和的规律 ，但比 Sigmoid 函数延迟了饱和期.

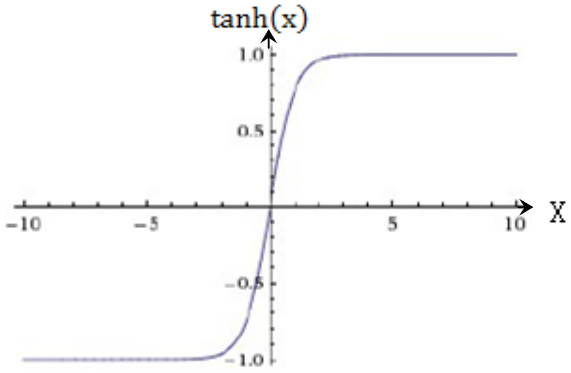

图 2.4 Tanh 函数图

ReLu 函数的表达式如公式 2.3 所示。

$$\varphi(x) = \max\ (0, x)。 \tag{2.3}$$

对应的函数图像如图 2.5 所示。

其优点是计算简单，只需要一个阈值就可以得到激活值，而不用去算一大堆复杂的运算；且收敛速度快。缺点是在训练的时候很容易出错。要注意设置 learing rate。

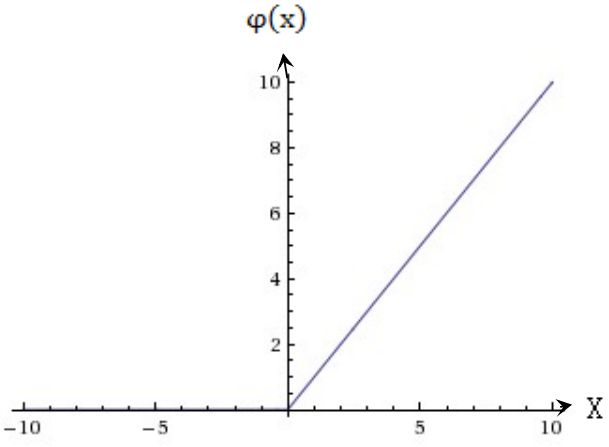

图 2.6 ReLu 函数图

2.3 激活函数在 FPGA 中的实现

大规模数字集成电路的发展，使采用 FPGA 实现神经网络成为可能，在 FPGA 设计中，神经元的激活函数的实现是很重要的环节。FPGA 凭借其可重构技术的灵活性，成为解决 Sigmiod 函数高速计算问题的有力工具。目前利用 FPGA 计算 Sigmiod 函数常用的方法有多种可选择。常用的有泰勒展开、多项式拟合法、分段拟合方法、查表法、CORDIC 算法等。

数字表示有两种类型，即定点表示和浮点表示。定点数克服整数表示法不能表示实数的缺陷，我们以通过将实数乘上一个分数来实现，若分数就是2^{-i}倍数，定点数表示法就会是精确的表示。但是自然界中的数并不是那么凑巧，所以定点数只能是近视地表示实数，浮点法也是这样的。具体的实现就是将数表示成 2 进制后，然后在左移 k 位，那么对于 N 位的定点数表示法中，就有（N-1-K）位用来表示整数部分，低 k 位则表示分数，最高位表示符号。数在定点化时，小数点位置可以随意定。最常用的还是定点化为（-1,1）之间，即最高位是符号位，小数点，后面全是数据位。

一般来说，神经网络计算会涉及大量的浮点数计算。虽然浮点数的安排很宽，但由于其算法的复杂性，FPGA 并不适合进行浮点数运算，它们通常需要过多的芯片面积，但 FPGA 中的资源总是有限的。与浮点相比，定点数的算术架构简单，可以通过归一化来控制固定数的数字排列。

在 FPGA 中浮点数表示法较为复杂，这种表达方式利用科学计数法来表达实数，即用一个尾数（ManTIssa），一个基数（Base），一个指数（Exponent）以及一个表正负的符号来表达实数。

定点数表示法，小数点位置的固定决定了整数部分和小数部分固定的位数，不利于同时表达特别大的数或者特别小的数。初始数据、中间结果或最后结果有时会在很大的范围内变化，在运算的各个阶段预先引入比例因子，把数据统一放大或缩小。一定长度的定点数据所能表示的数据范围和精度是很有限的。

查找表（Look Up Table)简称 LUT。系统设计中采用查表法来实现 Sigmiod 函数，查表法是最简单直接的方法，用简单的查询操作替换运行时计算的数组或者 associative array 这样的数据结构。LUT 本质上就是一个 RAM，直接将每个自变量所对应的 Sigmiod 函数值预先存储，存储空间的地址线为函数的自变量，通过存储访问读出的数据为函数值，即可实现。适用于任何一个复杂函数，且理论上可做到无限逼近。不消耗乘法器，还能在一个时钟内出结果。由于从内存中提取数值经常要比复杂的计算速度快很多，所以这样得到的速度提升是很显著的。尽管查找表通常情况下效率很高，但是如果所替换的计算相当简单的话就会得不偿失，这不仅仅因为从内存中提取结果需要更多的时间，而且因为它增大了所需的内存并且破坏了高速缓存。如果查找表太大，那么几乎每次访问查找表都会导致高速缓存缺失，这在处理器速度超过内存速度的时候愈发成为一个问题。

目前 FPGA 中多使用 4 输入的 LUT，每一个 LUT 看成一个 4 位地址的 16x1 的 RAM

组合电路实现查找表：描述了一个电路以后，把结果事先写入 ROM，每输入一个信号进行逻辑运算就等于输入一个地址进行查表，找出地址对应的内容，然后输出即可。

采用 Chisel 设计组合电路，就是在写 Scala 程序，它利用了 Scala 的一些高级特性。经过编译，得到针对 FPGA 的 Verilog HDL 代码。在 Chisel 中每个电路都是一些 node（节点）的集合，一个 node 就是一个硬件操作单元，它具有 0 个、1 个或者多个输入，依据输入驱动输出。

硬件描述语言 Verilog HDL 最初设计的目的是用来仿真的，所以有很多不可综合的语法，Verilog HDL 缺少目前高级语言具备的一些特性，比如对象、继承等。所以才有了 Chisel。

Scala 中的 map，zip 和 reduce 函数可以帮助我们更容易处理列表列表的内容并结合 Option 对象工作。

map 函数将一个函数应用于列表的每一个元素并且将其作为一个新的列表返回。与 foreach 函数相似。区别在于 foreach 函数没有返回值，且只针对参数。

zip 函数合并两个列表。但返回的列表长度取决于较短的列表。可使用 zipAll 函数来对较长列表的剩余元素进行处理。使用 zipWithIndex 可增加元素的下标(从 0 开始)。

reduce 函数可处理列表的每个元素并返回一个值，但方向是不被保证的。可使用 reduceLeft 函数和 reduceRight 函数强制处理元素的方向。reduce 返回的值的类型必须和列表的元素类型相关（类型本身或其父类）。

2.4 FPGA 芯片 Zynq-7000

2.4.1 芯片特点及结构

Zynq-7000 系列是全可编程片上系统，该系列 FPGA 继承了两个 1 GHz 的 ARM CortexA9 内核，打破了传统的 FPGA+ARM/DSP 的架构，使用单片 FPGA 就能很好地完成工作。FPGA 和 ARM 通过高达 100 Gbps 的内部高速总线通信，比 FPGA+ARM/DSP 外部通信的架构更加迅速而且更加可靠。带有 DDR 控制器硬核，支持 1 GB 多种数据位的地址宽度，在 ARM（PS）端有 64 位的数据通道，最高支持 1 066 MT/s 的速度,其在速度、稳定性和泛用性上都非常优秀，特别适用于做系统的高速存储。

与传统的 FPGA 相比，Zynq-7000 系列最大的特点是将处理系统 PS 和可编程资源 PL 分离开来，固化了 PS 系统的存在，实现了真正意义上的 SOC（System On Chip）。它包含了完整的 ARM 处理系统，实现差异化、分析和控制功能的创新型 ARM + FPGA 架构处。处理器系统中集成了各种控制器和大量的外设，使 Cortex-A9 在 Zynq-7000 中完全独立于 PL 部分，在可编程逻辑部分紧密地与 ARM 的处理单元相结合。这种软硬件均可编程的全可编程 SoC 集成了 ARM 处理器的软件可编程性与 FPGA 的硬件可编程性，不仅可实现算法的硬件加速，还在单个器件上高度集成 CPU、DSP slice,高速收发器以及模拟信号处理等功能。是单位功耗性价比最高的全面可扩展的 SoC 平台。FPGA 的部分用于扩展子系统，其有丰富的扩展能力，有超过 3000 个内部连接，连接资源非常的丰富，可提供 100Gb/s 以上的内部带宽。

系统的整体，主要包含处理系统（processing system，PS）和可编程逻辑（Programmable Logic，PL）两部分。PS 负责存储，PL 负责识别。

PL 和 PS 的接口类型总共有两种：（1）功能接口：AXI、EMIO、中断、DMA 流控制、时钟调试接口。（2）配置接口：PCAP、SEU、配置状态信号和 Program /Done/Init 信号。这些信号连接到 PL 内配置模块的固定逻辑上，给 PS 提供对 PL 的控制能力。

PS 和 PL 部分之间有多个接口，具体包括：

1、AXI 类数据接口：2 个 32bit 的 AXI 主接口、2 个 32bit 的 AXI 从接口、专用于 PL 访问 DDR 控制器的 32/64bit 的 AXI 从接口、1 个 64bit 的访问 CPU 存储器的从接口

2、其他接口：DMA 通道信号、PS 的中断输入信号、事件信号、触发信号、EMIO、PS 提供给 PL 的时钟信号及复位信号、XADC 接口、JTAG 接口

2.4.2 FPGA 与 ARM 的数据传输

AXI(Advanced eXtensible Interface)是 ARM 公司提出的一种总线协议，是 AMBA 中一个新的高性能协议，满足了超高性能以及复杂的片上系统设计需求。AMBA 总线结构的高性能及 ARM 微处理器的广泛应用，已成为 SOC 设计中使用很广泛的总线标准。

AXI 协议用于 PL 和 PS 之间的通信。FPGA 和 ARM 通过 AXI 互联总线交换数据如图 2.4 所。

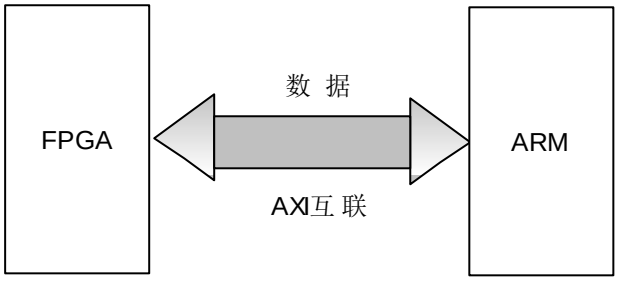

图 2.4 FPGA 与 ARM 数据交换

AXI 协议基于猝发式传输机制，主要描述了主从设备之间的数据传输方式。在地址通道上，每个交易有地址和控制信息，这些信息描述了需要传输的数据性质。主设备和从设备之间通过握手信号建立连接。AXI 协议的总线地址/数据和控制通道是分离的，读/写数据通道分离，这样可以提高吞吐率。一条 AXI 总线有 5 个独立通道：写数据、读数据、写地址、读地址和写响应。各个通道有独立的 AXI 握手信号，互相没有严格的时间要求。这种读/写地址分离的结构可使数据并发传输，为开发者提供灵活的设计方式。传输数据时，分别使用从设备的写数据通道和到主设备的读数据通道。在主设备到从设备的写数据交易中， 有一个额外的写响应通道。从设备通过响应通道向主设备发出信号表示写交易完成。

2.5 本章小结

在本章中，阐述了整体的设计思路。给出了神经网络的整体结构设计图，以及单个神经元的结构设计图，并对设计图做出详细解释。简要阐述了各模块的设计思路，以及实现设计的具体流程。介绍了激活函数 Sigmoid 和 FPGA 芯片 Zynq-7000 的一些特性，分析其优点。说明了如何用查表法实现 Sigmoid 函数，定点数在 FPGA 中的表示方法以及实现查找表的方法。同时还介绍了用于 PL 和 PS 之间的通信的 AXI 协议。

3. 神经元模块设计与实现

3.1 神经元的结构

神经元由函数 Neuron(val numAxons: Int, val num: Int)定义。numAxons 表示神经元的突触的数量，num 表示该神经元在所属层中的编号。

Neuron 的数据结构为：io(神经元输入输出)、sum（计算结果寄存器，所存数据的数据类型为 UInt）、state（当前状态）以及神经元的各种状态 s_idle（空闲状态）、s_busy（状态）以及 s_done（完成状态）。

表示神经元输入输出的 io 由函数 NeuronIO 定义。NeuronIO 数据结构为:in(输入）和 out（输出）。

in 由函数 NeuronDataIn 定义，NeuronDataIn 数据结构为：weight（权值）、axon（突触编号）和 in（输入数据）。其各参数数据类型以及数据位宽如表 3.1 所示。

表 3.1 NeuronDataIn 的参数数据类型以及数据位宽

参数名称	数据类型	数据位宽
weight	UInt	32
axon	UInt	32
in	UInt	32

out 由函数 NeuronDataOut 定义，NeuronDataIn 数据结构为：data(输出数据)。其参数数据类型以及数据位宽如表 3.3 所示。

表 3.3 NeuronDataOut 的参数数据类型以及数据位宽

参数名称	数据类型	数据位宽
data	UInt	32

神经元可以看作一个计算与存储单元。计算是神经元对其的输入进行计算功能,存储是神经元会暂存计算结果，并传递到下一层。

3.2 神经元内部逻辑运算

神经元可以看作一个计算与存储单元。计算是神经元对其的输入进行计算功能,存储是神经元会暂存计算结果，并传递到下一层。

一个神经元将其多个输入及权值经过逻辑运算后的结果作为下层节点的一个输入。若 in 来表示输入，用 w 来表示权重，表示连接的有向箭头这样理解：在初端，传递的信号大小是 in,端中间有加权值 w,经过这个加权后信号变成 in*w,于是在连接的末端，信号大小就变成了 in*w。

用 Z 来表示输出，则得到 Z 所经过的逻辑运算如公式 3.1 所示。

$$Z= w_1 * in_1 + w_2 * in_2 + w_3 * in_3 + \cdots + w_i * in_i \quad (3.1)$$

sum 存储计算结果,每条突触上的输入(io.in.bits.in)与权重(io.in.bits.weight)相乘的结果累加得到最后结果。

例如：如图 3.1 所示的神经元内部计算如公式 3.2 所示。

$$z = in_1 * w_1 + in_2 * w_2. \quad (3.2)$$

若 a_1=1、a_2=2、a_3=3、w_1=2、w_2=1,、w_3=1，那么经过逻辑运算后得到的输出 z 就为 7。

3.3 神经元中的状态机

神经元一般有两种状态：兴奋和抑制。一般情况下，大多数的神经元是处于抑制状态，但是一旦某个神经元受到刺激，导致它的电位超过一个阈值，那么这个神经元就会被激活，处于“兴奋”状态，进而向其他的神经元传递信息。

本论文将神经元的状态细分为如下三个状态：

(1)s_idle 状态。神经元在等待数据、没有在进行计算、计算结果已传给下一层的神经元，可接受输入并计算。

（2）s_busy 状态。神经元已接受到数据、在持续接受数据并进行逻辑运算。

（3）s_done 状态。计算已完成且在等待输出，不可接受数据。

状态迁移如图 3.1。

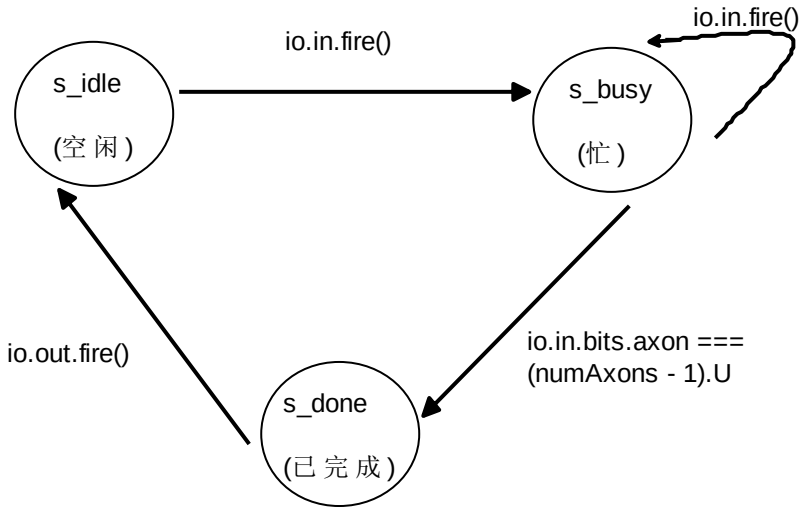

图 3.1 神经元状态转移图

空闲状态下，神经元可接收输入，设该神经元的编号为 0 的突触为当前突触。一旦接受到一个输入(io.in.fire)，就由“空闲”状态转到“忙”状态；“忙”状态下持续顺序接收剩余突触的输入(io.in.fire)并持续进行逻辑运算。直到接收到最后一个突触的输入并完成对该输入的逻辑运算，即当前突触为该神经元的最后一个突触（io.in.

bits.axon === (numAxons - 1).U）时，由“忙”状态转为“完成”状态；“完成”状态下等待输出时机，一旦成功输出（io.out.fire()），则由“完成”状态转为“空闲”状态。

3.4 单元测试模块的设计与实现

3.4.1 程序设计流程

单元测试模块 NeuronTester 程序设计流程如图 3.1 所示。

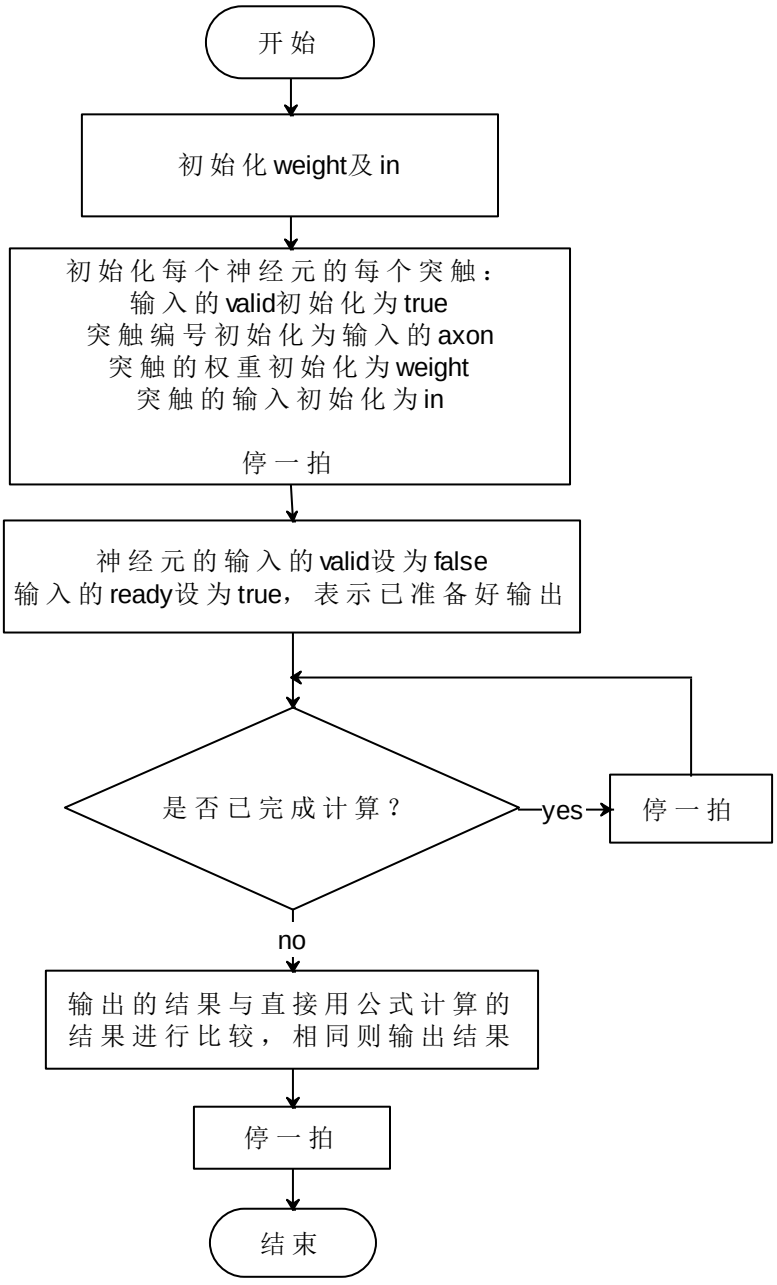

图 3.1 神经元的单元测试程序设计流程图

初始化时设置初值：weight = 2，in = 2，numAxons = 7，神经元编号从 0 开始。

3.4.1 测试结果与分析

执行测试程序得到如图 3.2 所示的结果。

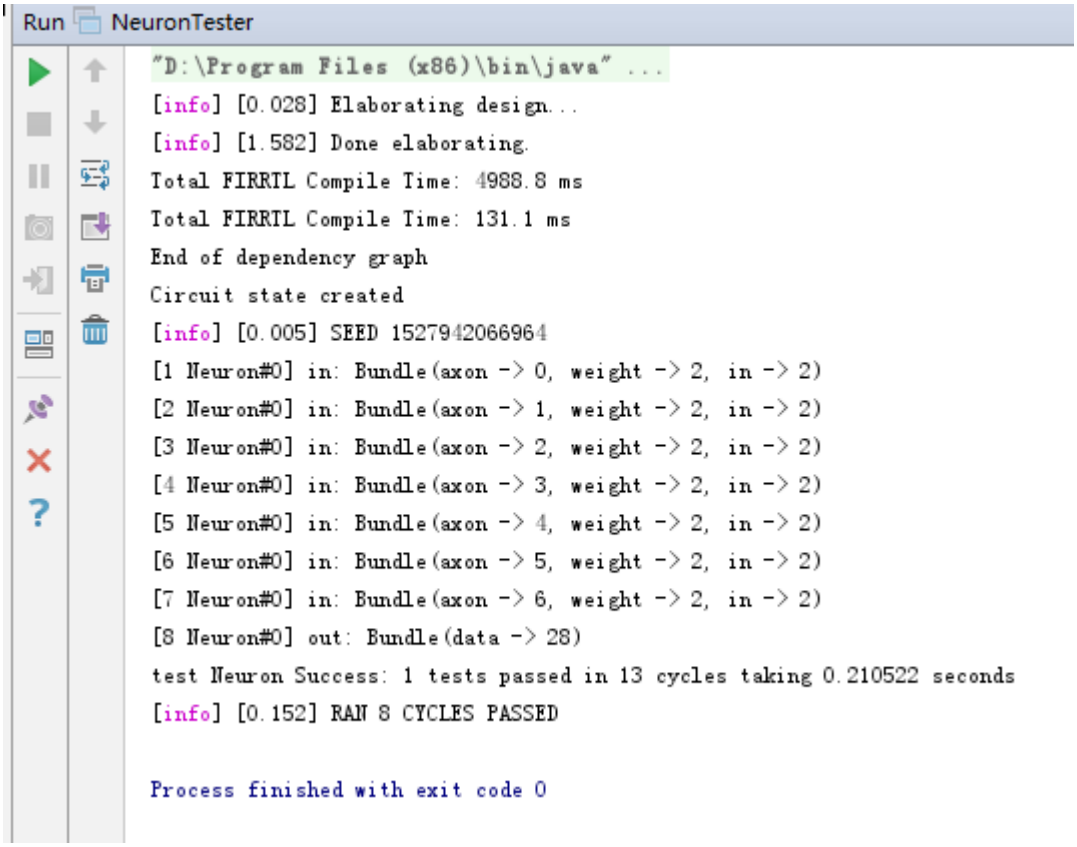

图 3.2 神经元模块测试结果

根据结果，编号为 0 的神经元的 7 个突触都成功的接收到权值和输入数据，最终得到的输出 data 为 28，根据公式 3.1 计算得到的结果为 28。说明测试结果正确，神经元模块设计成功。

3.5 本章小结

在本章中，详细阐述了神经元模块的设计方案。包括神经元的端口设计、数据方向、端口数据描述，神经元里面的状态机和内部的逻辑运算过程。端口设计包括端口数据结构、数据类型和数据位宽。状态机部分详细说明了各状态下的处理模式，状态转移情况

及触发条件。以及单元测试模块的程序设计流程，测试数据及测试结果分析。结果表明神经元的设计成功实现。

4. 处理核心模块设计与实现

4.1 层的设计

将待处理的某张图像的像素值集合当作一个数组，并把像素值当作预测图像中数值的特征。

函数 layer 的数据结构为：io（输入输出）、weights（权重矩阵存储）、neurons（神经元）、counterNeuron（计数器）、state（当前层状态）以及各种层状态 s_idle（空闲）、s_weightBusy（权重忙）、s_weightDone（权重已完成）、s_accumulateBusy、（计算忙）以及 s_accumulateDone（计算已完成）。

层的输入输出 io 由函数 LayerIO 定义，其数据结构为：weights（权重）、in(输入)和 out(输出)。

其中 weight 由函数 Weight 定义，其参数数据类型以及数据位宽如表 4.1：

表 4.1 weight 的参数数据类型以及数据位宽

参数名称	数据类型	数据位宽
weight	UInt	32
axon	UInt	32
neuron	UInt	32

输入 In 和输出 out 由函数 LayerData 定义，参数数据类型以及数据位宽如下表 4.2：

表 4.2 层输入 in 和层输出 out 的参数数据类型以及数据位宽

参数名称	数据类型	数据位宽
index	UInt	32
data	UInt	32

weights 用于存储该层的权重矩阵，需存储 numAxons * numNeurons 个权重值，numNeurons 该层的神经元个数，numAxons 表示该层每个神经元的突触个数。所存储数据的数据类型为 UInt，位宽为 32 位。

neurons 为该层的神经元组成的数组，每个神经元由函数 Neuron (numAxons, num) 分别定义，并对其进行编号。

counterNeuron 为一个计数器，负责记录已完成处理的神经元的个数。

state 为寄存器类型，用于存储当前该层的状态，便于状态判断及转移。关于各层状态的细节以及层状态转移的细节将在本章下一节做详细描述。

4.2 层状态机

本论文将层的状态细分为以下 5 个状态：

（1）s_idle（空闲）状态。等待输入、不做计算或计算结果已传给下一层的神经元，可接受输入并计算。

（2）s_weightBusy（权重忙）状态。开始接受到权重数据、在持续接受并存储数据。

（3）s_weightDone（权重已完成）状态。该层各突触已全部接受到相应权值，等待接收输入。

（5）s_accumulateBusy、（计算忙）状态。已接受到该层所有权重数据、在等待或持续接收输入，接收到输入后持续进行计算。

（6）s_accumulateDone（计算已完成）状态。最后一个神经元的最后一个突触的计算已完成，等待输出。

层的状态转移及触发条件如下图 4.3 所示：

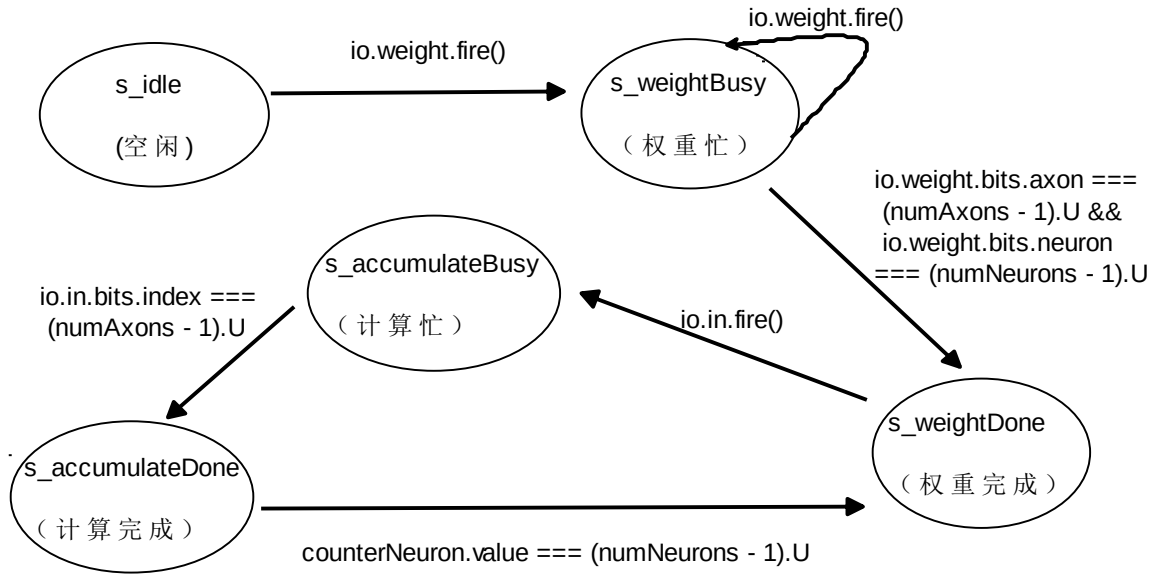

图 4.3 层状态转移图

空闲状态下，层等待接收权重数据输入，一旦接受到一个权重数据输入（io.weight.fire()），就由“空闲”状态转到“权重忙”状态；在“权重忙”状态下持续接收权重数据（io.weight.fire()），直到最后一个神经元（io.weight.bits.neuron === (numNeurons - 1).U）的最后一个突触（io.weight.bits.axon === (numAxons - 1).U）的权重接收完成，由“权重忙”状态转为“权重完成”状态；在“权重已完成”状态下等待输入，一旦接受到输入（io.in.fire()），就由“权重完成”状态转为“计算忙”状态；在“计算忙”状态下持续计算，直到计算完当前神经元的最后一个突触（io.in.bits.index === (numAxons - 1).U），状态由“计算忙”转为“计算已完成”；“完成”状态下等待输出时机，一旦输出（io.out.fire()），当前神经元为该层的最后一个神经元时（(counterNeuron.value === (numNeurons - 1).U)），状态由“计算已完成”转为“权重完成”。

4.3 单元测试模块的设计与实现

4.3.1 程序设计流程

单元测试模块 LayerTester 程序设计流程如图 4.1 所示。

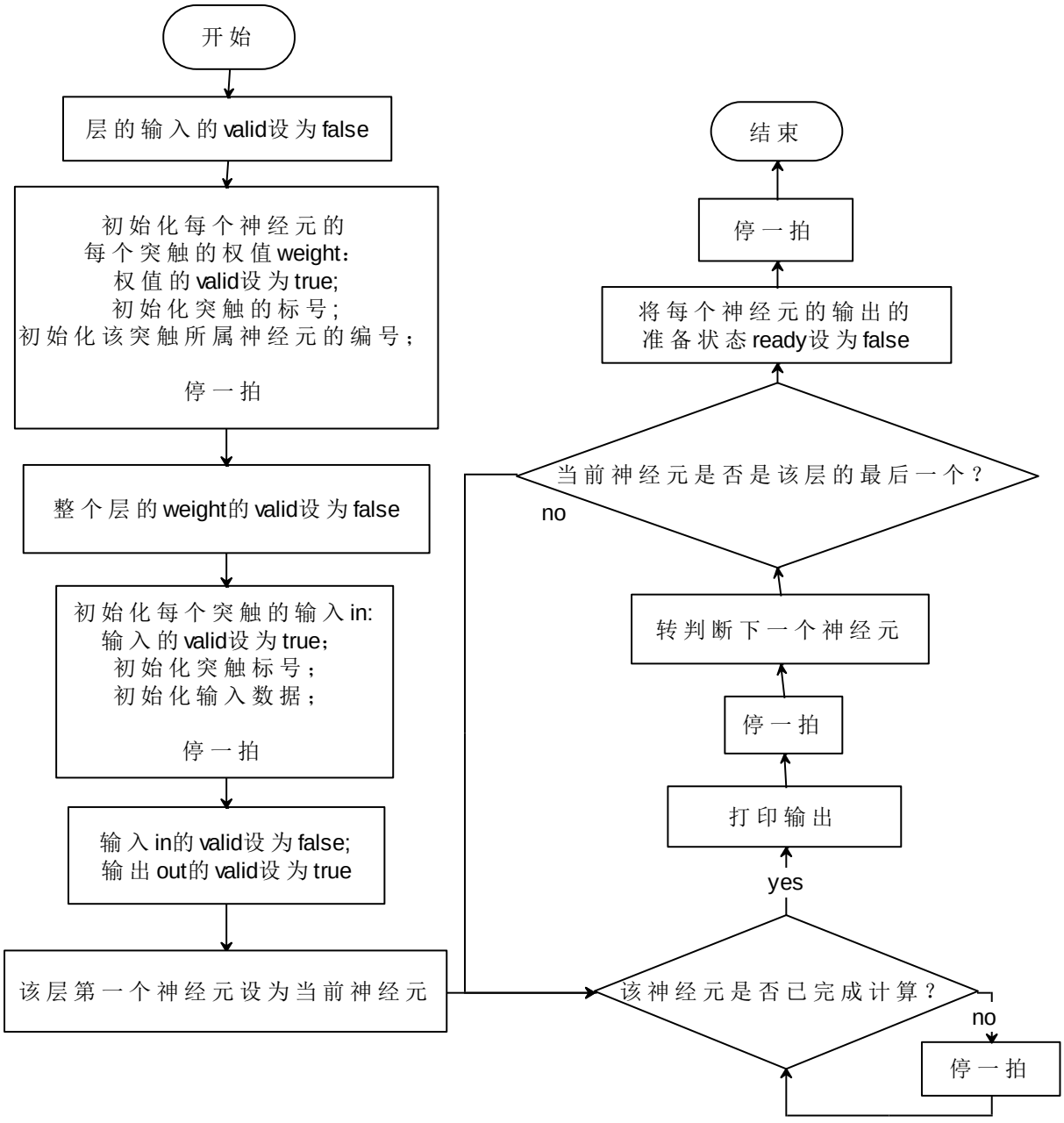

图 4.1　层的单元测试程序设计流程图

初始化：权值 weight = 2，输入 in = 2，突触个数 numAxons = 5，神经元编号从 0 开始，层起始编号 id 为 0，突触个数为 5，神经元个数为 7。

4.3.2 测试结果与分析

执行测试程序得到如图 4.2 至图 4.9 所示的结果。

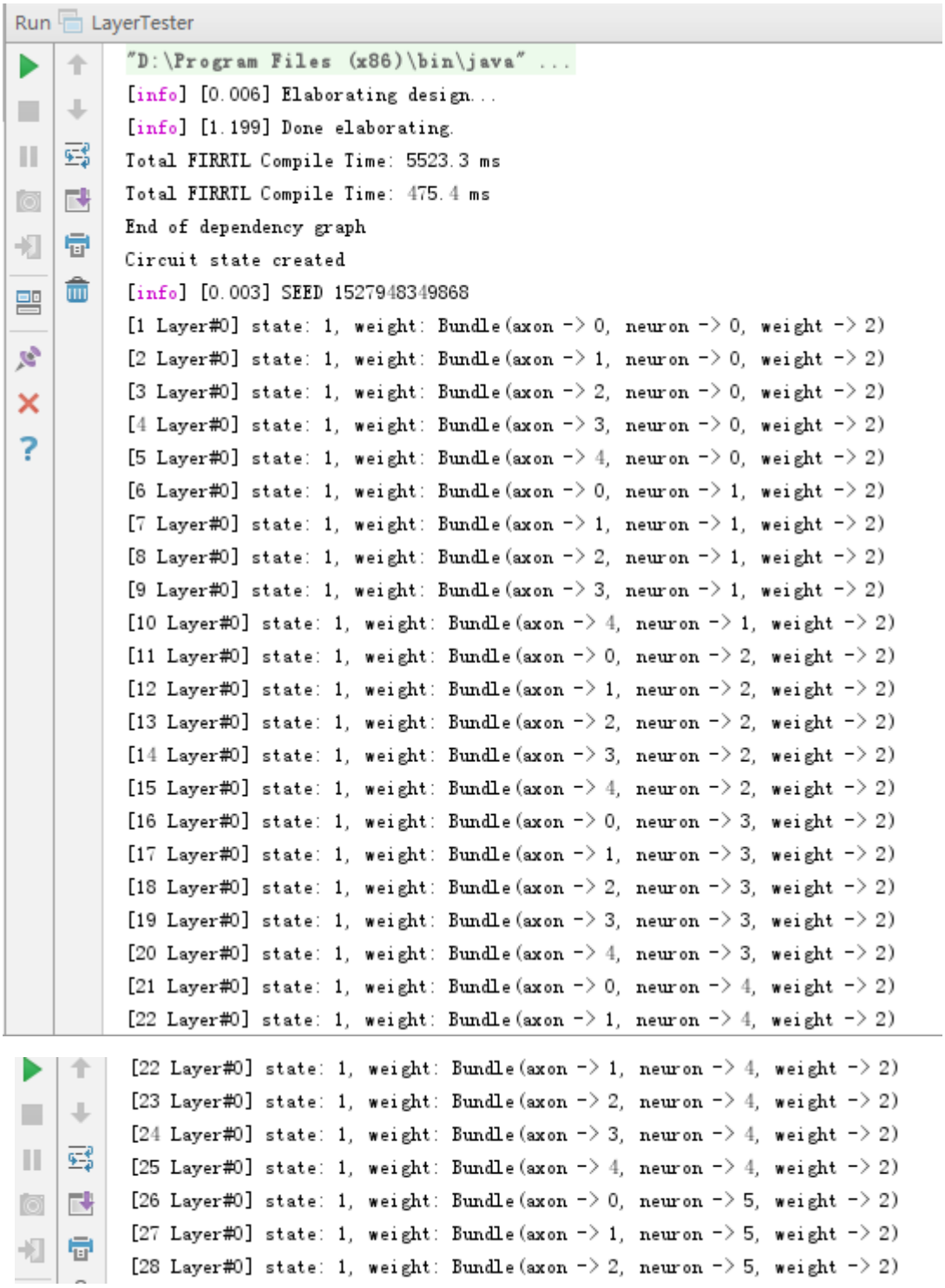

图 4.2　权重信息

```
[22 Layer#0] state: 1, weight: Bundle(axon -> 1, neuron -> 4, weight -> 2)
[23 Layer#0] state: 1, weight: Bundle(axon -> 2, neuron -> 4, weight -> 2)
[24 Layer#0] state: 1, weight: Bundle(axon -> 3, neuron -> 4, weight -> 2)
[25 Layer#0] state: 1, weight: Bundle(axon -> 4, neuron -> 4, weight -> 2)
[26 Layer#0] state: 1, weight: Bundle(axon -> 0, neuron -> 5, weight -> 2)
[27 Layer#0] state: 1, weight: Bundle(axon -> 1, neuron -> 5, weight -> 2)
[28 Layer#0] state: 1, weight: Bundle(axon -> 2, neuron -> 5, weight -> 2)
[29 Layer#0] state: 1, weight: Bundle(axon -> 3, neuron -> 5, weight -> 2)
[30 Layer#0] state: 1, weight: Bundle(axon -> 4, neuron -> 5, weight -> 2)
[31 Layer#0] state: 1, weight: Bundle(axon -> 0, neuron -> 6, weight -> 2)
[32 Layer#0] state: 1, weight: Bundle(axon -> 1, neuron -> 6, weight -> 2)
[33 Layer#0] state: 1, weight: Bundle(axon -> 2, neuron -> 6, weight -> 2)
[34 Layer#0] state: 1, weight: Bundle(axon -> 3, neuron -> 6, weight -> 2)
[35 Layer#0] state: 2, weight: Bundle(axon -> 4, neuron -> 6, weight -> 2)
```

图 4.3　权重信息

```
[36 Neuron#0] in: Bundle(axon -> 0, weight -> 2, in -> 2)
[36 Neuron#1] in: Bundle(axon -> 0, weight -> 2, in -> 2)
[36 Neuron#2] in: Bundle(axon -> 0, weight -> 2, in -> 2)
[36 Neuron#3] in: Bundle(axon -> 0, weight -> 2, in -> 2)
[36 Neuron#4] in: Bundle(axon -> 0, weight -> 2, in -> 2)
[36 Neuron#5] in: Bundle(axon -> 0, weight -> 2, in -> 2)
[36 Neuron#6] in: Bundle(axon -> 0, weight -> 2, in -> 2)
[36 Layer#0] state: 3, in: Bundle(index -> 0, data -> 2)
```

图 4.4　每个神经元的编号为 0 的突触的数据信息

```
[37 Neuron#0] in: Bundle(axon -> 1, weight -> 2, in -> 2)
[37 Neuron#1] in: Bundle(axon -> 1, weight -> 2, in -> 2)
[37 Neuron#2] in: Bundle(axon -> 1, weight -> 2, in -> 2)
[37 Neuron#3] in: Bundle(axon -> 1, weight -> 2, in -> 2)
[37 Neuron#4] in: Bundle(axon -> 1, weight -> 2, in -> 2)
[37 Neuron#5] in: Bundle(axon -> 1, weight -> 2, in -> 2)
[37 Neuron#6] in: Bundle(axon -> 1, weight -> 2, in -> 2)
[37 Layer#0] state: 3, in: Bundle(index -> 1, data -> 2)
```

图 4.5　每个神经元的编号为 1 的突触的数据信息

```
[38 Neuron#0] in: Bundle(axon -> 2, weight -> 2, in -> 2)
[38 Neuron#1] in: Bundle(axon -> 2, weight -> 2, in -> 2)
[38 Neuron#2] in: Bundle(axon -> 2, weight -> 2, in -> 2)
[38 Neuron#3] in: Bundle(axon -> 2, weight -> 2, in -> 2)
[38 Neuron#4] in: Bundle(axon -> 2, weight -> 2, in -> 2)
[38 Neuron#5] in: Bundle(axon -> 2, weight -> 2, in -> 2)
[38 Neuron#6] in: Bundle(axon -> 2, weight -> 2, in -> 2)
[38 Layer#0] state: 3, in: Bundle(index -> 2, data -> 2)
```

图 4.6　每个神经元的编号为 2 的突触的数据信息

```
[39 Neuron#0] in: Bundle(axon -> 3, weight -> 2, in -> 2)
[39 Neuron#1] in: Bundle(axon -> 3, weight -> 2, in -> 2)
[39 Neuron#2] in: Bundle(axon -> 3, weight -> 2, in -> 2)
[39 Neuron#3] in: Bundle(axon -> 3, weight -> 2, in -> 2)
[39 Neuron#4] in: Bundle(axon -> 3, weight -> 2, in -> 2)
[39 Neuron#5] in: Bundle(axon -> 3, weight -> 2, in -> 2)
[39 Neuron#6] in: Bundle(axon -> 3, weight -> 2, in -> 2)
[39 Layer#0] state: 3, in: Bundle(index -> 3, data -> 2)
```

图 4.7　每个神经元的编号为 3 的突触的数据信息

```
[40 Neuron#0] in: Bundle(axon -> 4, weight -> 2, in -> 2)
[40 Neuron#1] in: Bundle(axon -> 4, weight -> 2, in -> 2)
[40 Neuron#2] in: Bundle(axon -> 4, weight -> 2, in -> 2)
[40 Neuron#3] in: Bundle(axon -> 4, weight -> 2, in -> 2)
[40 Neuron#4] in: Bundle(axon -> 4, weight -> 2, in -> 2)
[40 Neuron#5] in: Bundle(axon -> 4, weight -> 2, in -> 2)
[40 Neuron#6] in: Bundle(axon -> 4, weight -> 2, in -> 2)
[40 Layer#0] state: 4, in: Bundle(index -> 4, data -> 2)
```

图 4.8　每个神经元的编号为 4 的突触的数据信息

```
[41 Neuron#1] out: Bundle(data -> 20)
[41 Layer#0] state: 4, out: Bundle(index -> 1, data -> 20)
[41 LayerTester] neuron: 1, out: 20
[42 Neuron#2] out: Bundle(data -> 20)
[42 Layer#0] state: 4, out: Bundle(index -> 2, data -> 20)
[42 LayerTester] neuron: 2, out: 20
[43 Neuron#3] out: Bundle(data -> 20)
[43 Layer#0] state: 4, out: Bundle(index -> 3, data -> 20)
[43 LayerTester] neuron: 3, out: 20
[44 Neuron#4] out: Bundle(data -> 20)
[44 Layer#0] state: 4, out: Bundle(index -> 4, data -> 20)
[44 LayerTester] neuron: 4, out: 20
[45 Neuron#5] out: Bundle(data -> 20)
[45 Layer#0] state: 4, out: Bundle(index -> 5, data -> 20)
[45 LayerTester] neuron: 5, out: 20
[46 Neuron#6] out: Bundle(data -> 20)
[46 Layer#0] state: 4, out: Bundle(index -> 6, data -> 20)
[46 LayerTester] neuron: 6, out: 20
[47 Layer#0] state: 2, out: Bundle(index -> 0, data -> 20)
test Layer Success: 0 tests passed in 53 cycles taking 1.409884 seconds
[info] [1.217] RAN 48 CYCLES PASSED
```

4.9 各神经元的输出结果

测试结果显示：各神经元的各个突触的权值和输入都是 2，与所设置的初始值相同，表明成功接收。最终得到的每个神经元的输出 data 为 20，与根据公式 3.1 计算得到的结果一致。说明测试结果正确，层的模块设计成功实现。

4.5 本章小结

在本章中，详细阐述了层的模块设计方案。包含端口设计、层的数据结构、层模块中的状态机以及权重矩阵的存储。端口设计包括端口数据结构、数据类型和数据位宽。状态机部分详细说明了各状态下的处理模式，状态转移情况及触发条件。以及单元测试模块的程序设计流程，测试数据及测试结果分析。结果表明层的设计成功实现。

5.神经网络模块设计与实现

5.1 神经网络结构

按照第 2 章中所描述的神经网络的结构，在已实现的神经元模块和层模块的基础上实现整个神经网络的设计。

在函数 NeuralNetwork 中设计神经网络，其数据结构为：io（输入）、layer0（第 1 层）和 layer1（第二层）。io 由函数 NeuralNetworkIO 生成，layer0 和 layer1 由函数 layer 生成。

定义整个神经网络的输入输出的函数 NeuralNetworkIO 的数据结构为权重 weight、输入 in 以及输出 out，数据类型都是都是 Decoupled。

其中，weight 的结构包含表示突触序号的 axon、表示神经元序号的 neuron 和表示权重值的 weight,具体数据类型以及位宽如下表 5.1：

表 5.1 weight 的参数数据类型以及数据位宽

参数名称	数据类型	数据位宽
weight	UInt	32
axon	UInt	32
neuron	UInt	32

函数 NeuralNetworkIO 中的输入 in 和输出 out 由函数 LayerData 定义。其参数为指数 index 和数据 data，数据类型以及位宽如下表 2.2。

表 5.2 输入 in 和输出 out 的参数数据类型以及数据位宽

参数名称	数据类型	数据位宽
index	UInt	32
data	UInt	32

layer0 和 layer1 由函数 Layer 定义，关于层的详细设计在第 4 章中已描述。将整个神经网络的输入 io.in 作为 layer0 的输入 layer0.io.in；layer0 经过逻辑运算得到的输出 layer0.io.out 作为 1 层的输入 layer1.io.in；1 层经过逻辑运算得到的输出 layer1.io.out 作为整个神经网络的输出 io.out。如此便将整个神经网络连接起来。具体神经元内部逻辑运算见第 3 章。

5.2 单元测试模块的设计与实现

5.2.1 程序设计流程

单元测试模块 NeuralNetworkTester 程序设计流程如图 5.1 所示。

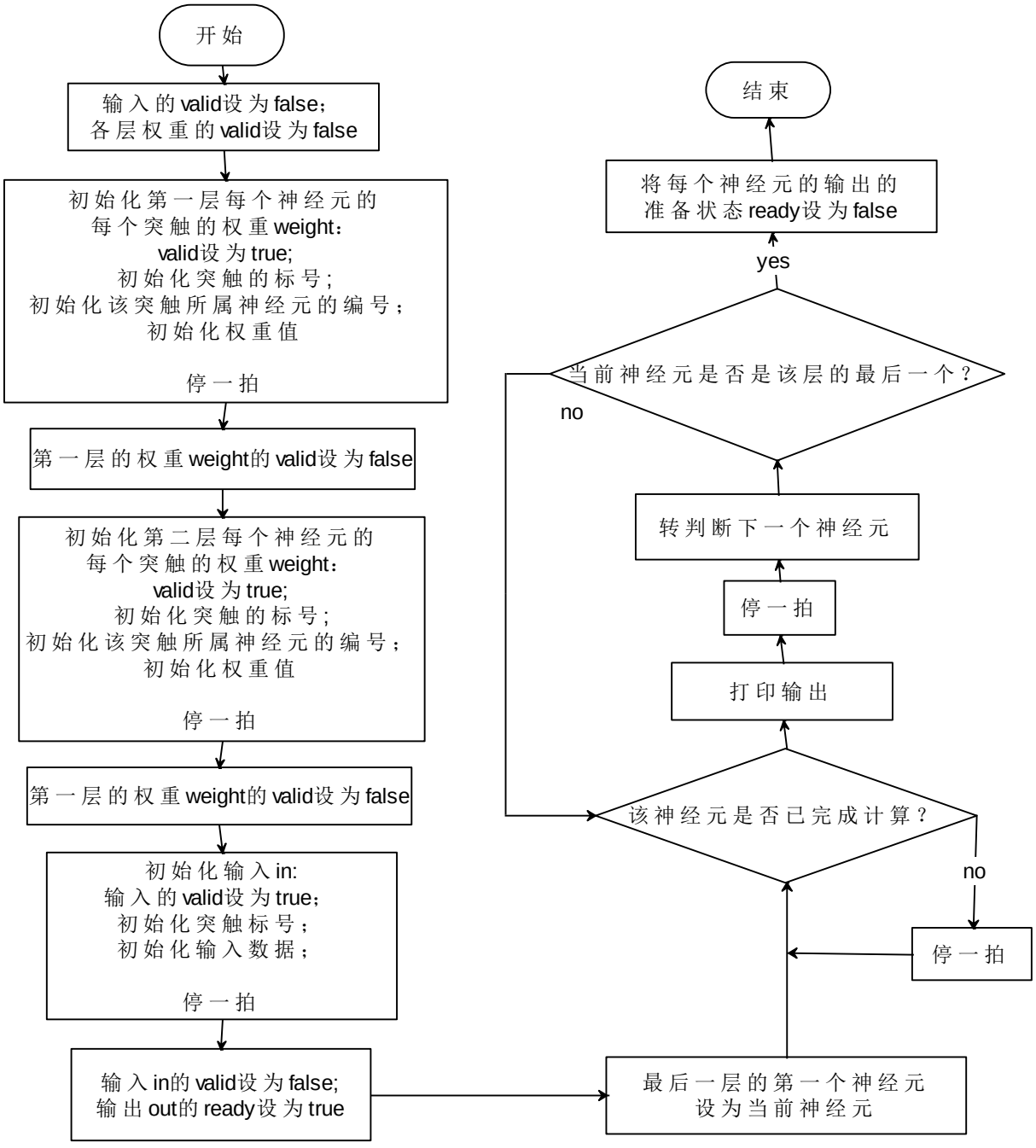

图 5.1 整个神经网络的单元测试程序设计流程图

初始化：权值 weight = 3，输入 in = 3，有 2 个输入；神经元编号从 0 开始，层起始编号 id 为 0；layer0 和 layer1 的神经元个数均为 2。

5.2.2 测试结果与分析

执行测试程序 NeuralNetworkTester 得到如图 5.2 所示的结果。

测试结果显示：各神经元的各个突触的权值和输入都是 3，与所设置的初始值相同，表明成功接收。最终得到的每个神经元的输出 data 为 108，与根据公式 3.1 计算得到的结果一致。说明测试结果正确，神经网络的模块设计成功实现。

```
NeuralNetworkTester
"D:\Program Files (x86)\bin\java" ...
[info] [0.015] Elaborating design...
[info] [2.294] Done elaborating.
Total FIRRTL Compile Time: 5339.1 ms
Total FIRRTL Compile Time: 484.9 ms
End of dependency graph
Circuit state created
[info] [0.005] SEED 1527991629857
[1 Layer#0] state: 1, weight: Bundle(axon -> 0, neuron -> 0, weight -> 3)
[2 Layer#0] state: 1, weight: Bundle(axon -> 1, neuron -> 0, weight -> 3)
[3 Layer#0] state: 1, weight: Bundle(axon -> 0, neuron -> 1, weight -> 3)
[5 Layer#1] state: 1, weight: Bundle(axon -> 0, neuron -> 0, weight -> 3)
[6 Layer#1] state: 1, weight: Bundle(axon -> 1, neuron -> 0, weight -> 3)
[7 Layer#1] state: 1, weight: Bundle(axon -> 0, neuron -> 1, weight -> 3)
[9 Neuron#0] in: Bundle(axon -> 0, weight -> 3, in -> 3)
[9 Neuron#1] in: Bundle(axon -> 0, weight -> 3, in -> 3)
[9 Layer#0] state: 3, in: Bundle(index -> 0, data -> 3)
[10 Neuron#0] in: Bundle(axon -> 1, weight -> 3, in -> 3)
[10 Neuron#0] out: Bundle(data -> 18)
[10 Neuron#1] in: Bundle(axon -> 1, weight -> 3, in -> 3)
[10 Layer#0] state: 4, in: Bundle(index -> 1, data -> 3)
[10 Layer#0] state: 4, out: Bundle(index -> 0, data -> 18)
[10 Neuron#0] in: Bundle(axon -> 0, weight -> 3, in -> 18)
[10 Neuron#1] in: Bundle(axon -> 0, weight -> 3, in -> 18)
[10 Layer#1] state: 2, in: Bundle(index -> 0, data -> 18)
[11 Neuron#1] out: Bundle(data -> 18)
[11 Layer#0] state: 4, out: Bundle(index -> 1, data -> 18)
[11 Neuron#0] in: Bundle(axon -> 1, weight -> 3, in -> 18)
[11 Neuron#1] in: Bundle(axon -> 1, weight -> 3, in -> 18)
```

图 a

```
[11 Layer#1] state: 3, in: Bundle(index -> 1, data -> 18)
[12 Neuron#0] out: Bundle(data -> 108)
[12 Layer#1] state: 4, out: Bundle(index -> 0, data -> 108)
[12 NeuralNetworkTester] neuron: 0, out: 108
[13 Neuron#1] out: Bundle(data -> 108)
[13 Layer#1] state: 4, out: Bundle(index -> 1, data -> 108)
[13 NeuralNetworkTester] neuron: 1, out: 108
test NeuralNetwork Success: 0 tests passed in 20 cycles taking 0.446662 seconds
[info] [0.329] RAN 15 CYCLES PASSED

Process finished with exit code 0
```

图 b

图 5.3 神经网络模块的单元测试结果

5.3 本章小结

在本章中，详细阐述了整个神经网络的模块设计方案。包含端口设计端口数据结构、数据类型和数据位宽。以及单元测试模块的程序设计流程，测试数据及测试结果分析。结果表明神经网络的设计成功实现。

结论

近年来随着计算机硬件和理论的不断发展，深度学习应用成为一个热点，越来越受到大众的关注。神经网络是完全按照生物学上的神经网络来设计实现的。FPGA 是常用的加速手段，具有高性能、低功耗和可编程的优点。通过分析深度学习应用的基本计算和数据访问特征；分析深度神经网络的预测过程和训练过程的算法共性和特性，在此基础上设计多层神经网络。

总结本文完成的神经网络技术和神经网络加速器设计的主要工作如下：

（1）简单介绍了神经网络和深度学习的概念起源及发展，以及 FPGA 的一些特性。对国内外研究现状进行了简单系统的分析。

（2）介绍了本文设计所涉及到的感知机网络、MNIST 数据集并阐述了数据集的具体使用方法，同时简要说明了在 FPGA 对神经网络的加速所起到的作用。

（3）给出了所设计的神经网络的整体结构设计图，以及单个神经元的结构设计图。阐述设计思路及所设计的主要数据的相关信息。

（4）在 MATLAB 中实现一个神经网络，用 MNIST 的训练集 (training set)将隐藏层和输出层的最优权重矩阵训练出来，并将其分别存储到 csv 文件中。

（5）在 IntelliJ IDEA Community Edition 中 Scala 依次分别设计实现了神经元模块、神经元单元测试模块、层模块、层的单元测试模块、一个两层的神经网络以及该神经网络的单元测试模块。同时利用一些数据进行测试，分析得到的测试结果说明所设计的各模块逻辑结构正确。

本文的不足之处：

（1）考虑到设计过程中时间的需求，在设计神经网络时没有遵循经典的神经网络模型的结构设计。没有设计实现输入层模块，而是直接将输入传递给隐藏层。

（2）在测试各模块时为了缩短程序执行所耗费时间和资源，避免等待过久，将测试数据都设置得比较小，以方便快速的得出结果，没有将边界情况考虑全面。

参考文献

[1] K.Jarrett,K.Kavukcuoglu,M.Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In Proceedings of the 12th IEEE International Conference on Computer Vision, 2009

[2] Shaoli Liu , Zidong Du , Jinhua Tao, Dong Han, Tao Luo, Yuan Xie†, Yunji Chen and Tianshi Chen，《Cambricon: An Instruction Set Architecture for Neural Networks》， ACM/IEEE 第 43 届国际计算机体系结构研讨会，2016

[3] Alex Krizhevsky, Sutskever Ilya, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems 25. 2012.

[4] HugoLarochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An Empirical Evaluation of Deep Architectures on Problems with Many Factors of Variation. In Proceedings of the 24th International Conference on Machine Learning, 2007.

[5] Q.V. Le. Building high-level features using large scale unsupervised learning. In Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 2013.

[6] Clément Farabet, Berin Martini, Polina Akselrod, Selc ,uk Ta-lay, Yann LeCun, and Eugenio Culurciello, “Hardware accelerated convolutional neural networks for synthetic vision systems,” in Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on. IEEE, 2010, pp. 257－260.

[7] G.E. Dahl, T.N. Sainath, and G.E. Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. In Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 2013.

[8] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning Deep Structured Semantic Models for Web Search Using Clickthrough Data. In Proceedings of the 22Nd ACM International Conference on Conference on Information; Knowledge Management, 2013.

[9] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh, “A fast learning algorithm for deep belief nets,” Neural computation, vol. 18, no. 7, pp. 1527－1554, 2006.

[10] Geoffrey E Hinton and Ruslan R Salakhutdinov, “Reducing the dimensionality of data with neural networks,” Science, vol.313, no. 5786, pp. 504－507, 2006.

[11] Zhilu Chen, Haibo He, Jing Wang, and Xinming Huang, “A fast deep learning system using GPU,” in Circuits and Systems(ISCAS), 2014 IEEE International Symposium on. IEEE, 2014,pp. 1552－1555.

[12] Ying Zhang and Saizheng Zhang, “Optimized deep learning architectures with fast matrix operation kernels on parallel platform,” in Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on. IEEE, 2013, pp. 71－78.

[13] Shaoqing Ren Jian Sun Kaiming He, Xiangyu Zhang. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In arXiv:1502.01852, 2015.

[14] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung, “Fixed point optimization of deep convolutional neural networks for object recognition,” in Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on. IEEE, 2015, pp. 1131－1135.

[15] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A Machine-Learning Supercomputer. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014.

[16] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 2014.

[17] Kyuyeon Hwang and Wonyong Sung,“Fixed-point feedforward deep neural network design using weights +1, 0, and -1,” in Signal Processing Systems (SiPS), 2014 IEEE Workshop on.IEEE, 2014, pp. 1－6.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. In Nature, 2015.

[19] Janardan Misra and Indranil Saha, “Artificial neural networks in hardware: A survey of two decades of progress,” Neurocomputing, vol. 74, no. 1, pp. 239－255, 2010.

[20] Song Han, Jeff Pool, John Tran, and William Dally, “Learning both weights and connections for efficient neural network,” in Advances in Neural Information Processing Systems, 2015, pp.
1135－1143.

[21] Dong Yu, Frank Seide, Gang Li, and Li Deng, “Exploiting sparseness in deep neural networks for large vocabulary speech recognition,” in Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on. IEEE, 2012, pp. 4409－4412.

[22] Jonghong Kim, Kyuyeon Hwang, and Wonyong Sung, “X1000 real-time phoneme recognition VLSI using feed-forward deep neural networks,” in Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on. IEEE, 2014, pp. 7510－7514.

[23] Kyuyeon Hwang and Wonyong Sung, “Fixed-point feedforward deep neural network design using weights +1, 0, and -1,” in Signal Processing Systems (SiPS), 2014 IEEE Workshop on. IEEE, 2014, pp. 1－6.

[24] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung, “Fixed point optimization of deep convolutional neural networks for object recognition,” in Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on. IEEE, 2015, pp. 1131－1135.

[25] C. Farabet, C. Poulet, J.Y. Han, and Y. LeCun. CNP: An FPGA-based processor for Convolutional Networks. In Proceedings of the 2009 International Conference on Field Programmable Logic and Applications, 2009.

[26] M. Peemen, A.A.A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for Convolutional Neural Networks. In Proceedings of the 31st IEEE International Conference on Computer Design, 2013.

[27] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. NeuFlow: A runtime reconfigurable dataflow processor for vision. In Proceedings of the 2011 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2011.

[28] P. Sermanet and Y. LeCun. Traffic sign recognition with multi-scale Convolutional Networks. In Proceedings of the 2011 International Joint Conference on Neural Networks, 2011.

[29] V. Gokhale, Jonghoon Jin, A. Dundar, B. Martini, and E. Culurciello.A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks.In IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2014.

[30]O. Temam. A defect-tolerant accelerator for emerging high-Performance applications. In Proceedings of the 39th Annual International Symposium on Computer Architecture, 2012.

[31] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen.PuDianNao: A Polyvalent Machine Learning Accelerator. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, 2015.

[32] R. Sarikaya, G.E. Hinton, and A. Deoras. Application of Deep Belief Networks for Natural Language Understanding. Audio, Speech, and Language Processing, IEEE/ACM Transactions on, 2014.

[33] R Salakhutdinov and G Hinton. An Efficient Learning Procedure for Deep Boltzmann Machines. Neural Computation, 2012.

[34] M.A. Motter. Control of the NASA Langley 16-foot transonic tunnel with the self-organizing map. In Proceedings of the 1999 American Control Conference, 1999.

[35] Zhengyou Zhang, M. Lyons, M. Schuster, and S. Akamatsu. Comparison between geometry-based and Gabor-wavelets-based facial expression recognition using multi-layer perceptron. In Proceedings of the Third IEEE International Conference on Automatic Face and Gesture Recognition, 1998.

[36] 王昆，周骅.深度学习中的卷积神经网络系统设计及硬件实现[J].电子技术应用，2018，44(5)：56-59.

[37] Yakun Sophia Shao and David Brooks, “Research Infrastructures for Hardware Accelerators,” in Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2015.

[38] Yu-Ting Chen, Jason Cong, Michael Gill, Glenn Reinman, and Bingjun Xiao, “Customizable Computing,” in Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2015.

致谢

大学四年的时间转瞬即逝，这期间里我系统地学习了计算机各方面知识，还学到不少做人做事的道理。培养了自己独立思考、解决问题的能力，同时还提升了动手能力。感谢学部各位老师在我成长过程中的无微不至的帮助，尤其是各位任课老师的认真负责，使我能够很好的掌握和运用专业知识。

特别感谢导师蔡旻对我的细心指导，严谨细致、一丝不苟的的作风是我学习的榜样，当我遇到困难一筹莫展的时候，导师的循循善诱，让我能及时找出问题关键点，不断进步。他的言传身教将使我终生受益。

感谢家人一直以来无微不至的关心和爱护，让我能无忧无虑的成长，全心投入学习和工作之中，顺利完成学业。

感谢选择了蔡旻老师毕设题目的另外两位同学，在我做实验遇到问题的时候热情帮助和支持；在我的论文写作过程当中，提供为各方面的信息支持。

同时感谢我的室友们，在整个大学生涯中的相互扶持、相互鼓励、共同成长。在做毕设期间的解惑答疑以及互相监督，让我能满怀信心的顺利完成毕设。

毕设后期一直在参加 iwill（上海）企业发展有限公司北京分公司入职前的培训，总是在公司和学校两边跑。公司所有职员以及同期培训生给了我无条件的理解和包容，对此不胜感激。