密级：_____

中国科学院大学
University of Chinese Academy of Sciences

博士学位论文

稀疏神经网络加速器研究

作者姓名：张士锦

指导教师：陈云霁　研究员

中国科学院计算技术研究所

学位类别：工学博士

学科专业：计算机系统结构

研 究 所：中国科学院计算技术研究所

二〇一七年五月

Study of Sparse Neural Networks Accelerator

A Dissertation Submitted to

The University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Doctor of Philosophy

in

Computer Architecture

by

Shijin Zhang

Dissertation Supervisor: Professor Yunji Chen

Institute of Computing Technology

Chinese Academy of Sciences

May, 2017

声　　明

我声明本论文是我本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，本论文中不包含其他人已经发表或撰写过的研究成果。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。

作者签名：[signature]　日期：2017.5.19

论文版权使用授权书

本人授权中国科学院计算技术研究所可以保留并向国家有关部门或机构送交本论文的复印件和电子文档，允许本论文被查阅和借阅，可以将本论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编本论文。

（保密论文在解密后适用本授权书。）

作者签名：[signature]　导师签名：[signature]　日期：2017.5.19

摘 要

最近几年，神经网络的应用越来越广泛，在图像、语音、机器翻译等领域都取得了卓越的成绩。为了提升神经网络的效果，神经网络的规模逐年递增，结构日益复杂。这导致神经网络的计算量和访存量急剧增加。为了应对上述挑战，学术界提出了神经网络稀疏化的方法。稀疏后的全连接神经网络权值可以减少 10 倍。然而现有的通用计算平台 CPU、GPU 不能很好利用神经网络的稀疏性，过去的神经网络加速器（如 DianNao 等）甚至不支持稀疏神经网络。

本文从微结构、算法映射和功能验证等方面对稀疏神经网络加速器进行了较为系统的研究，主要取得了以下创新：在微结构方面，我们设计了一种新的稀疏神经网络加速器微结构：Cambricon-X。Cambricon-X 能够充分利用稀疏神经网络的特性，大幅减少神经网络的计算量和访存量。Cambricon-X 的核心是其索引模块。它能根据连接关系处理输入神经元，并将处理后的数据通过胖树发送给不同的计算单元。不同的计算单元存储了不同的权值，计算不同的输出神经元。在索引模块中，我们提出了两种连接关系的表示方法：直接索引和步长索引的方法。经过比较，步长索引比直接索引的代价更小，我们最终采用了步长索引的方法。在 65nm 的工艺下，Cambricon-X 能够达到 544GOP/s 的性能，面积和功耗只有 6.38mm^2 和 954mW。实验结果表明，在一些稀疏神经网络的测试中，我们的加速器能够比 DianNao 神经网络加速器快 7.23 倍，同时能耗降低 6.43 倍。

在算法映射方面，我们提出了各种常见神经网络算法在加速器上的映射方法。在神经网络中，不同的神经网络层具有不同的运算和访存特点。卷积层的计算量很大，卷积层的计算量超过了整个卷积神经网络 80% 以上的计算量，由于共享权值的特点，卷积层的权值数量很少。全连接层与卷积层恰恰相反，全连接层的计算量很小，权值的数量却很多。池化层和各种归一化层都没有权值，计算量都不大。在将不同的神经网络层映射到加速器时，需要考虑每种层的计算和访存特点。对于卷积层，我们采用复用权值的方法，每个权值只会从内存中读一次，直到所有和这个权值的相关的输出都使用过这个权值之后，这个权值才会被丢弃掉；对于全连接层和其他层，我们采用复用输入或者输出神经元的方法，尽量减少神经元在内存中的读写次数。采用我们的算法映射方法，最大化减少了访存量，加速器在很多层上运算效率在 90% 以上。

在验证方面，我们提出了一套针对神经网络加速器的验证方法以及一个通用的参考模型。首先，验证采用层次化方法，不同阶段的验证侧重点不同。在模块级验证阶段，采用白盒验证手段，重点验证模块的功能以及内部信号的状态。在子系统级验证阶段，我们将加速器分为 IO 子系统和运算子系统，不同的子系统分别验证不同的子功能。在系统级验证阶段，我们搭建了两个不同的验证环境。一个是在线比较的验证环境，主要

为了验证加速器在随机指令序列激励下的功能；另一个是离线比较的验证环境，主要为了验证加速器在神经网络配置激励下的运算功能。为此，我们开发了一个基于 C++ 的通用参考模型。该参考模型与加速器结构无关，能够支持各种低精度的神经网络运算，并且支持各种常见的神经网络类型。对于不同的加速器结构，参考模型只需要简单的修改层内的计算顺序即可使用。另外，该参考模型支持多线程，因此运算速度相比 RTL 的仿真速度提高了很多。在验证的不同阶段，我们使用不同的衡量参数来保证验证工作的充分性。

上述工作已部分应用于国际上首个稀疏神经网络处理器芯片上，确保了该芯片的实用性和正确性。

关键词：稀疏；神经网络；加速器；算法；映射；验证

Abstract

Neural networks have recently been used in a great deal of applications. Due to the advances of neural networks, it makes a great step forward in speech recognition, image recognition, machine translation and so on. To get a new progress, new ideas, algorithms and network architectures are developed. Recent evidence reveals that the scale of neural networks is of crucial important. As we all know, neural networks are memory sensitive and computational. Sparse neural networks have emerged as an effective solution to reduce the above overhead. The parameter of sparse neural network could be 10x less than the dense one. Hower, existing accelerators(e.g. DianNao), CPU and GPU can not benefit from sparse neural networks.

On micro architecture design, we propose a novel accelerator micro architecture: Cambricon-X. Cambricon-X is designed for sparse neural networks to reduce memory access and computation. Cambricon-X consists of multiple Processing Elements(PE). An Indexing Module(IM) efficiently selects and transfers needed neurons to connected PEs with reduced bandwidth requirement, while each PE stores irregular and compressed synnapses for local computation in an asynchronous fashion. With 16 PEs, our accelerator is able to achieve at most 544 GOP/s in a small form factor(6.38mm^2 and 954 mW at 65nm). Experimental results over a number of representative sparse networks show that our accelerator achieves on average 7.23x speedup and 6.43x energy saving against the state-of-the-art NN accelerator.

On algorithm mapping, we propose differrent layers mapping methods on Cambricon-X. Different neural network layers vary a lot in memory access and computation. The computation in convolution layers is huge. The total computation of all convolution layers occupies over %80 of all that in a network. Because of parameter sharing scheme used in covolution layers, it drastically reduces the number of free parameters. Contrary to convolution layers, fully connected layers own more free parameters. Howerver, there is less computation in fully connected layers. Pooling layers and normalization layers are simpler. There are no free parameters in these layers. The algorithm mapping on our accelerator is a challenge. For Convolution layers, the key is the reuse of weights. Each weight is loaded into accelerator only once. Weights wouldn't be dropped until they are not needed at all. For other layers, the key is reuse of input and output neurons. Using those mapping methods, our accelerator can achieve over %90 computation efficiency on many situations.

On verification, we propose a verification method on neural networks accelerators and a universal reference model based on C++. First, the verification is divived into different stages

focusing on different functions of accelerator. On module verification stage, each module is verified under a white box test. On subsystem verification stage, the accelerator is divied into several subsystems. Each subsystem is verified alone in terms of the functions of them. On system verification stage, we build two different verification environments. For online checking environment, accelerator is running on random insts and the results are checking on real-time. For offline checking environment, accelerator is running on insts generated on a random configed file of neural networks. Then the results of the neural network on acclerator and reference model are checked after the stimulation. We build a universal reference model for offline checking. The reference model is independent of the architecture of accelerators and can be used in different data type(e.g. float16). For different accelerators, the modification of reference model is just the order of computation. What is more, it is multithreading program which is much faster than RTL stimulation.

Much of above work has been used in the first sparse neural networks accelerator in the world. It makes sure of the function and stability of the accelerator.

Keywords: sparse; neural network; accelerator; algorithm; mapping; verification

目 录

图目录

表目录

第一章　引言

1.1　背景

1947 年出现了第一个晶体管，被视为是电路革新的开始。晶体管相对于真空电子管有体积小，速度快，稳定高效等特点。在构建电路时，所有晶体管都应该被正确的连接。如果存在错误的连接，那么整个电路将不能正确地工作。在早期的电路产业中，所有的工作都是由手工完成。工人们需要将器件摆好在电路板上，然后再用金属线将他们连接起来。工程师们意识到，采用这种方法成功装配一个处理器电路是很困难的。另一个难题是电路的规模，当电路规模很大时，为了保证电路的速度，布局布线的难度增加了很多。1958 年，德州仪器公司的工程师 Kilby 发明了第一块集成电路 [4]。尽管第一块集成电路存在很多问题，但是这个发明具有划时代意义。Jack Kilby 因此成名，并在 2000 年获得了诺贝尔物理学奖。集成电路的发明，大大降低了电路的制造难度和体积，降低了生产成本。

自上世纪 50 年代起，半导体制造工艺不断提高，晶体管的体积和成本不断下降。从而促进了整个半导体产业快速发展，集成电路的规模按照摩尔定律的趋势持续了超过半个世纪。在 1971 年，英特尔第一颗通用处理器 4004[5] 只有 2300 个晶体管，采用 10μm 制程，主频只有 108KHz。到了 2016 年很多处理器的晶体管数量已经超过了 20 亿个，采用 10nm 制程，主频也超过 3GHz。但是，集成电路的发展也面临着重重阻碍。首先，由于布局布线和半导体器件的物理特性，集成电路的主频已经很难继续增长。另一方面考虑到集成电路的功耗，单纯提高主频已不再是一个明智的选择。因此，自 2005 年之后，通用处理器设计者普遍采用增加核的数量来提高性能。其次，由于采用多核的方法来提高处理器的性能，使得芯片封装面积很大。然而芯片散热的能力有限，使得芯片中的各个部分不能同时全速工作。因此，通用处理器的发展已经遇到瓶颈。

专用集成电路（application specific integrated circuit，简称 ASIC）可以在一定程度上解决上述的问题。对于通用处理器，可以得到更多的用户以及灵活性。通用性的硬件设计者实现了很多指令集架构（instruction set architecture，ISA），指令集架构定义了一系列基本操作，也就是指令。上述的不同的指令序列组合形成不同功能的程序。不像通用处理器，专用集成电路是根据特定的用途设计的。现在的专用集成电路通常包含很多种类的电路：所有的内存内存单元（包括只读存储器，动态随机存储器，电可擦除只读存储器，闪存等），协处理器以及其他大的专用电路等。一些专用集成电路也被叫做片上系统（system on chip，SoC）。对于硬件设计者，专用集成电路通常采用硬件描述语言来描述其功能。

浮点运算协处理器就是一个典型的专用集成电路。1976 年英特尔开始开始研究浮点运算协处理器，用来协助处理器做浮点运算。四年后，1980 年，英特尔发布了 8087 浮点运算协处理器，该协处理器用来协助 8086[6] 通用处理器实现浮点运算的硬件加速。8087 协处理器做为可选部件，被广泛用在了当时的个人电脑上。9 年后，英特尔发布了 80486[7] 处理器，该处理器内部集成了浮点运算部件。这种由专用到通用的转换恰恰证明了市场由需求决定。与此类似，当今如火如荼的深度神经网络应用也正是需要加速器的时候。

1.2 神经网络算法加速的发展

最近几年，Yann LeCun，Yoshua Bengio，Geoff Hinton 在神经网络领域上取得了一系列的突破 [8–10]，将神经网络算法研究推向了一个新的热潮。最近，深度卷积神经网络在图像识别领域 Imagenet[11] 数据集上的识别效果开始不断接近人类的识别精度 [12]。研究表明更大的数据集更深的网络能带来更好的效果，这导致现在神经网络的规模越来越大。然而，神经网络巨大的计算量和访存量严重影响了学者的研究步伐。对于神经网络计算的加速成为一个越来越迫切的需求。实际上，包括软件和硬件，对于神经网络算法的加速已经拥有了很长的历史。如表1.1所示列出了神经网络算法加速相关的软件和硬件工作。

1.2.1 神经网络软件库

现在有很多流行的神经网络加速库，几乎涵盖了各种常用的编程语言。每个编程库都有自己的特点，实现的功能也不尽相同。

FANN 是一个采用 C 语言编写的神经网络加速库，它不仅支持普通的全连接神经网络而且还支持稀疏连接的神经网络。考虑到交叉平台，它支持定点和浮点的数据计算格式。它拥有一个自动处理训练模型的友好框架。为了使更多的人能够使用 FANN，作者把 FANN 库封装到了 20 多种语言的接口中。由于这个库在 2003 年就被开发出来，在开发时只做了对 CPU 的支持。对于现在以 GPU 运算为主的运算平台上，这个库显得过时了。

Theano 是一个 python 库，在 2008 年诞生于蒙特利尔理工学院。Theano 可以允许用户自己定义优化和衡量数学表达式，尤其是高维数组（numpy，ndarray 等）。使用 Theano 可以让用户得到高性能的代码，甚至可以和手写的 C 语言代码相匹敌。另外，Theanno 也支持 GPU 的运算，使得运算速度超越 CPU 几十倍。Theano 是一个优秀的科学计算平台，后来衍生出了很多知名的深度学习 python 包，如 Keras[34]，Blocks[35] 等。

不像 FANN 和 Theano，最近又出现了多款专门用于深度学习的加速框架，在这其中包括 Tensorflow，Caffe，MXNet 等，他们都是专注于神经网络的加速框架，并且都针对 GPU 平台做了优化支持，使得神经网络在计算库上都能够充分发挥 GPU 运算平台

表 1.1 一些神经网络加速的软硬件工作

类别	工作
软件库	FANN[13] Caffe[14] Tensorflow[15] MXNet[16] CNTK[17] Theanoi[18] Torch[19] cuDNN[20] Marvin[21] Chainer[22] Neon[23] Brainstorm[24]
FPGA 硬件	Ganglion[25] RRANN[26] A Fast FPGA implementation of General Purpose Neuron[27] FPGA based implementation of deep neural networks using onchip memory only[28]
人工神经网络硬件	RAP[1] NPU[2] DianNao[29] ShiDianNao[30] DaDianNao[31] PuDianNao[32]
脉冲神经网络硬件	TrueNorth[33]

应有的计算能力。在 GPU 编程中，用户需要将写好的代码转成 GPU 能够识别的语言，比如说 openGL。然而在 2007 年，NVIDIA 公司推出了嵌入到 C 语言的 GPU 编程方法：cuda[36]。cuda 的出现大大简化了程序员对 GPU 的编程难度。神经网络内在的并行性和 GPU 的众核结构互相呼应，因此神经网络在 GPU 上的计算速度相比 CPU 有了很大的提高。

在神经网络 GPU 开发的开始阶段，作者都是使用 cuBLAS 的库函数实现神经网络的各种操作，这些 cuBLAS 库函数是通用的线性运算库函数，而神经网络算法中有很多算法，比如卷积操作，不能直接映射成线性运算，而是需要进行数据重新排放之后再映射，这些数据拷贝搬运同样耗费时间。因此，仅仅通过线性运算库实现神经网络算法，

不能够充分发挥 GPU 的计算能力。很快，NVIDIA 公司意识到了这点，在 2014 年，发布了专门针对深度学习优化的 GPU 加速库 cuDNN[20]。

随着神经网络算法规模和复杂度逐年增加，硬件性能捉襟见肘，学者又处于不同的优化目的发布了更多的神经网络运算框架。MXNet 针对内存的使用进行了优化，甚至可以在手机设备上运行各种神经网络。Brainstorm 针对大规模神经网络做了专门的支持，能够运行几百层的神经网络。Neon 是最近刚发布的神经网络加速库，速度比 Caffe 有了质的提升。

1.2.2 硬件神经网络加速器

在生物的神经系统中，信号传输是一个复杂的生物化学过程 [37]。信号传输的目的是改变神经元中神经递质的量，当这个量超过一定的阈值时，这个神经元便被激活，并向周围相连的神经元继续传递神经递质。人工神经网络试图通过不同的层次来模拟这个过程。

在本节我们将介绍两种模拟和数字的神经元实现以及脉冲神经元。采用模拟电路模拟神经元的行为，往往是出于价格和面积的考虑，但是在神经网络精度上往往会打折扣。数字电路正好相反，在牺牲了面积和成本的因素，能够实现更高精度的神经网络。两种实现有不同的目的，往往需要折中考虑。

1.2.2.1 数字神经元

在数字电路神经元中，权值一般存储在寄存器或者 RAM 中。运算部件都是数字电路构成的加法器，乘法器等。非线性的激活操作可以使用查表或者加法器和乘法器来运算。数字神经元的实现简单，组装方便，容易量产，但是数字神经元的速度往往很慢。由于输入输出都有可能是模拟信号，所以也可能会用到模拟信号转数字信号和数字信号转模拟信号等一些外围电路。

Skrbek[38] 提出了一个基于移位加法器的神经元模型，它能够实现很多操作如乘法，开方，指数运算等等。所有的运算都采用近似计算的方式。比如计算 2^x 的线性近似运算方式如下：

$$EXP_2(x) = 2^{int(x)}(1 + frac(x)) \tag{1-1}$$

移位的操作计算 2^n，其中 n 是自然数，在范围 $[2^n, 2^{n+1})$ 范围内，使用线性近似的方法。因为 $n = int(x)$，我们将 x 分成整数部分 int(x) 和小数部分 frac(x)。(1+frac(x)) 表达式近似计算 $2^{frac(x)}$。

1.2.2.2 模拟神经元

在模拟电路神经元中，权值通常存储在以下几种元件中：电阻器 [39]，电容器 [40]，浮栅技术的 EEPROM[41] 里。存储权值的可变电阻可以由两个 MOS 晶体管 [42] 组成，

然而 MOS 管的长宽和距离会影响到电阻的精度。神经网络中向量的内积操作和随后的激活操作可以由带饱和处理的累加放大器实现 [43]。

在模拟电路中，实现激活操作比在数字电路中要简单很多。累加器可以简单的使用一个电流的加和作为输出。模拟电路的元件往往比数字电路的实现面积要小，功能也更简单。另一方面，要想获得高精度的输出，模拟电路在制造过程中，需要更复杂的设计和封装。在模拟模型中，信号常用电压 [44] 或者电流 [41] 的大小来表示，电流和电路中的电阻模拟了乘法的过程。因此电路中的电阻起到了神经网络中权值的作用，这些电阻的值即是神经网络中待学习的权值参数。另外，FET 晶体管的电压电流的非线性曲线关系使得这些晶体管很适合做出激活的电路。

1.2.2.3 脉冲神经元

实际上，在生物的神经系统中，神经元之间的通信是非持续稳定的信号，而是脉冲信号。集成和激活（integrate and fire，简称 I&F）脉冲神经元 [45] 模型是脉冲神经元模型中较为简单的一种，它能够处理连续一段时间内的变化信号，并且能够实现信号的同步。相比于非脉冲神经经元，脉冲神经元具有更为强大的计算能力 [46]。脉冲神经元需要为神经元设定一个激活的阈值，早期 Schultz 和 Jabri[47] 提出了一种为神经元设定精确阈值的方法。由于这些神经元能够实现不同函数的功能，因此这些电路的面积和功耗也相对大一些。后来，又有很多学者做了很多简化脉冲神经元的计算电路并提高脉冲神经网络精度的事情 [48–51]。

1.2.2.4 神经网络加速器

神经网络加速器的研究兴起于上世纪八十年代。最开始，学者还只关注全连接神经网络。在卷积神经网络流行之后，支持卷积操作的神经网络加速器成为主流。考虑到性能，功耗，面积等的因素，在具体实现上又采用了不同的方法 [52]。

由于受精度的影响，模拟电路神经网络加速器的研究相对较少。ETANN 是英特尔提出的一个有着 64 个全连接神经元的模拟电路神经网络加速器。ETANN 不支持在线学习，因此 ETANN 需要在主机上完成训练之后将模型下载到 ETANN。该芯片能够达到 2GCPS 的运算速度，计算精度为 4 比特，总线位宽为 64 比特，拥有 10240 个可编程权值。Mod2[53] 是一个采用了 ETANN 芯片做成的神经运算机器，被用来执行图像处理的相关任务。

脉冲神经网络 [54] 是一种与生物神经元行为更为相似的仿生物算法模型，已经被广泛应用在各种生物感官应用上，如图像处理，语音识别等。他们拥有时间延迟，静态阈值和饱和处理等特点。脉冲神经网络已经被认为是一种相比其他神经网络算法在计算上更为友好的神经网络算法。然而计算大规模的神经网络依旧是一个计算代价昂贵的事情。硬件实现脉冲神经网络是一种有效地加快神经网络算法运算速度的好方法。知名的

脉冲神经网络并不多，最有名的莫过于 IBM 公司于 2012 年发布的 TrueNorth 脉冲神经网络加速器芯片。

1992 年，Cox[25] 等人第一次使用 FPGA 实现了一个神经网络分类器。但是至今，使用 FPGA 实现神经网络算法依旧是一个难处理的问题，因为神经网络算法变化变化太多，很难做到完整支持，另一方面在资源受限的 FPGA 上，实现很多乘法器也拥有不小的难度。由于 FPGA 具有可重配置的特点，因此很多不同的神经网络算法可以在短期内映射到 FPGA 的硬件上去。

在八十年代，大家还只关注全连接神经网络，因此加速器结构和实现简单，可以得到很高性能，这之中的典型代表是 RAP。由于结构运算相对简单，RAP 采用现成的数字信号处理器（digital signal processor，DSP）作为其运算单元。如图1.1所示，RAP 将多个 DSP 通过环形（RING）拓扑结构相连，RING 拓扑结构采用 FPGA 实现，每个 DSP 中的神经元可以通过 RING 互相广播，从而实现神经网络的计算。RAP 的设计思想是尽量集成大量商用的高性能计算单元，整个电路设计只需要考虑设计路由电路和排版。由于不需要自己设计大规模集成电路，RAP 从硬件设计到组装使用只花了八个月时间。

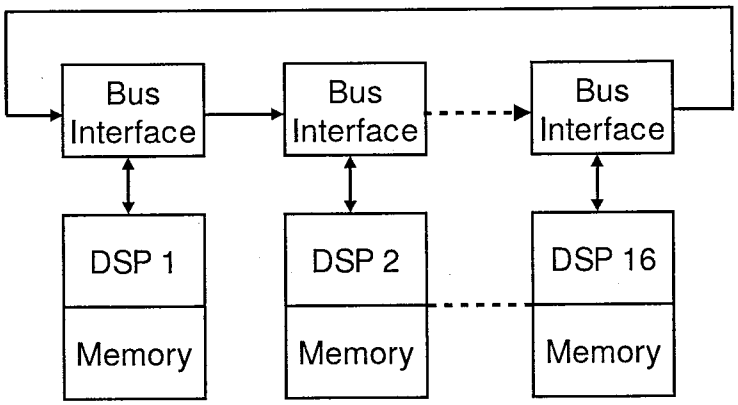


图 1.1　RAP 结构示意图 [1]

到了九十年代，卷积神经网络在 MNIST 数据集 [55] 上取得的巨大成功 [8]，引起了诸多学者的重视。卷积神经网路与全连接神经网络有着截然相反的特征，卷积神经网络的计算量巨大访存相对较小。因此神经网络加速器的重心也向卷积神经网络倾斜。Bernhard 等人 [56] 做了一个采用模拟电路做的卷积神经网络加速器，该加速器外部输出再转换成数字信号，模拟电路加速器具有性能高，功耗低的特点，每秒可以识别 1000 个手写数字。除此之外，该加速器支持可编程的特性，虽然只能支持 3 种指令，但通用性和灵活性大大提高。

在随后的几年里，通用性和高性能是神经网络加速器的发展趋势。Hadi Esmaeilzadeh 等人 [2] 发表了一个可编程的神经网络加速器 NPU，不仅可以用于神经网络，也可用于图像处理等领域。NPU 的运算单元如图1.2所示，主要包括乘，加，激活三个运算部件，这一经典的运算单元设计被之后的很多神经网络加速器所借鉴，如 Diannao 等。为了便

于使用，NPU 还拥有自己的 ISA。但是 NPU 编程复杂，对用户不友好。NPU 相比 CPU，性能平均只提升了 2.3 倍，能耗只降低了 3.0 倍。

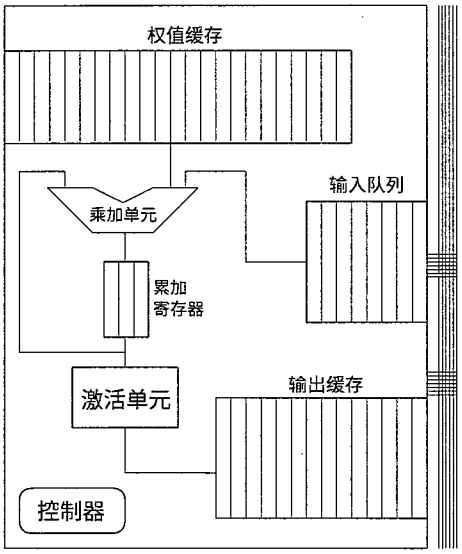


图 1.2 NPU 运算逻辑示意图 [2]

针对神经网络加速器通用性和性能的问题，一些学者提出了神经网络领域专用的 ISA，如 NISA[57] 和 Cambricon[58] 等。Cambricon 是一种 LOAD-STORE 结构的指令集，针对当今的各种神经网络算法，Cambricon 集成了对张量，向量，矩阵，逻辑运算，数据转换和控制等指令。Cambricon 能够更好的指导硬件设计。使用 Cambricon 指令集的硬件，在延迟，能耗和面积等方面都有较大的优势。另外，Cambricon 指令集的代码更加精炼，并且灵活性很高。

神经网络加速器的通用性和能耗是互相矛盾的因素，为了在保持通用性的前提下，而又尽可能的降低神经网络加速器的能耗，有些加速器的设计者采用了低精度的运算单元。DianNao 采用 16 位定点的运算部件，功耗只有 485mW。然而 DianNao 的性能却比高端 CPU 快 117 倍，能耗降低了 21.08 倍。

此外，最近几年还有很多方法和结构各异的神经网络加速器。PuDianNao 不仅支持神经网络的计算，并且支持各种机器学习算法的计算；ShiDianNao 适合集成到嵌入式设备，实现端到端的神经网络应用解决方案；DaDianNao 采用片上 eDRAM 作为存储来降低访存；PRIME[59] 设计了一种专门针对神经网络的 Re-RAM；ISAAC[60] 采用忆阻器作为存储单元，同时采用模拟电路作为运算单元。

在 2016 年，Han[61] 等人发布了支持稀疏全连接层神经网络加速器 EIE。在稀疏的全连接神经网络上，采用 256 个 PE 的 EIE 在性能和能耗上都获得了很大的优势。但由于 EIE 只支持稀疏的全连接神经网络，大大限制了它的使用场景。

总之，神经网络加速器的设计越来越成熟，越来越进步。现在的神经网络加速器不仅具有体积小，性能高，功耗低的特点，而且灵活性也越来越高。不久的将来，也许神

经网络加速器可以像浮点运算单元一样成为通用处理器密不可分的一部分。

1.3 主要研究内容及贡献

最近的研究表明 [62–64]，神经网络中有很多可训练参数是冗余的。在保持神经网络正确率不变的前提下，采用一些神经网络稀疏化的方法对神经网络进行重新训练之后，很多权值都是为零的值。然而之前的神经网络加速器并不能支持这种稀疏的特性，因此不能从稀疏神经网络中获益。CPU 和 GPU 的稀疏矩阵运算库获得的收益也不高，甚至有很多神经网络采用稀疏矩阵运算库之后会变得更慢。本文提出了一种稀疏神经网络加速器微结构实现了对稀疏神经网络的支持，该加速器针对稀疏神经网络的特点，设计了一个专门用来处理稀疏连接关系的模块。该模块使得加速器在处理稀疏神经网络的时候，既减少了计算量，又减少了访存量。因此，稀疏神经网络加速器在性能和能耗上相比其他神经网络加速器以及 CPU、GPU 拥有很大的优势。

本文设计了一个支持稀疏神经网络运算的加速器微结构，并提出了各个神经网络层在加速器上的映射方法，最后我们提出了一种针对神经网络加速器的验证方法以及一个通用的神经网络运算参考模型。本文的主要贡献如下：

1. 本文设计并提出了首个支持稀疏卷积神经网络的加速器微结构。该微结构能够灵活高效的支持各种参数的神经网络，在支持普通神经网络的前提下，又进一步实现了对稀疏神经网络的支持。该微结构的提出使得加速器对 CPU 和 GPU 的优势进一步扩大。

2. 本文设计了一些常见的神经网络层算法在神经网络加速器上的映射方法。神经网络算法种类繁多，然而硬件要避免复杂的设计，这样便提高了对硬件设计和算法映射的难度。经过综合考虑，包括硬件设计的复杂度和难度，神经网络算法的特点，以及各种神经网络算法的的计算量比的情况下，我们在硬件简洁的前提下，实现了常见的多种神经网络算法在稀疏神经网络加速器上的映射。对于卷积层和池化层，稀疏加速器的计算效率高达 90% 以上，其他层的计算效率也很高。

3. 本文提出了针对神经网络加速器的验证方法和一个通用的参考模型。首先，验证采用分层的验证方法，不同的验证层次侧重点不同。在模块级验证阶段，采用白盒验证手段，重点验证模块的功能以及内部信号的状态。在子系统级验证阶段，我们将加速器分为 IO 子系统和运算子系统，不同的子系统分别验证不同的子功能。在系统级验证阶段，我们搭建了两个不同的验证环境。一个是在线比较的验证环境，主要为了验证加速器在随机指令序列激励下的功能；另一个是离线比较的验证环境，主要为了验证加速器在运算各种神经网络下的功能。为此，我们开发了一个基于 C++ 的通用参考模型。该参考模型与加速器结构无关，能够支持各种低精度的神经网络运算，并且支持各种常见的神经网络类型。对于不同的加速器结构，参考模型只需要简单的修改层内的计算顺序即可使用。另外，该参考模型支持多线程，因此运算速度相比 RTL 的仿真速度提高了很

多。在验证的不同阶段，我们使用不同的衡量参数来保证验证工作的充分性。

以下是本文的章节安排。第二章将介绍神经网络的基础，神经网络中的各种层算法，以及神经网络未来的两个发展方向：一是以神经网络精度为驱动的发展方向，这个分支是以提高神经网络精度为目的，不断研究新的网络算法，以及层数更多结构更加复杂的神经网络。另一个是以低能耗为发展方向的神经网络算法研究，这个分支主要以降低神经网络能耗，从减小神经网络模型大小以及降低神经网络数据位宽等方向，提高神经网络的适用范围。第三章主要描述了我们设计的稀疏神经网络加速器微结构，该稀疏神经网络加速器微结构能够充分利用稀疏神经网络的特性，进一步提高加速器在神经网络上的计算性能。第四章我们对一些常见的神经网络算法进行分析，并分析它们在 Caffe 代码中使用基本线性算法库的实现，然后提出了各种神经网络算法在稀疏神经网络加速器上的映射方法。第五章我们提出了一套对神经网络加速器的验证方法并开发了一个用于神经网络加速器的通用参考模型。除了完成对神经网络加速器各个模块的验证，我们还实现了加速器在整个神经网络上运算的验证，保证加速器能够在神经网络运算上实现正确的功能。第六章我们对神经网络加速器做了性能和功耗的相关实验。最后，在第七章，我们总结了全文的内容，并设想了未来需要继续的工作。

第二章　神经网络算法

计算机很擅长解决算法和数学问题，但是世界上有很多问题不能被定义成数学算法。人脸识别和语言处理等问题就不能被具体的算法所描述，但是对人脑神经网络来说这些都是很简单的问题。受到生物神经网络的启发，研究人员采用计算机模仿生物的大脑系统，用这种方法可以使计算机能够解决人脸识别等问题。人工神经网络算法便是一种仿生物大脑的一大类算法（以下简称神经网络）。

早在 1943 年，神经生理学家 Warren McCulloch 和数学家 Walter Pitts 根据数学和算法创造了一个神经网络计算模型 [65]。阈值逻辑是一个电路模型，用来描述大脑中神经元的工作原理。

1960 年，斯坦福大学研究人员 Bernard Widrow 和 Marcian Hoff 发明了两个神经网络模型分别叫做 ADALINE 和 MADALINE[66]。ADALINE 被用来识别二进制比特流，当它的输入为来自于电话的二进制流时，它可以预测下一个比特的值。MADALINE 是首个被用来解决实际问题的神经网络，MADALINE 使用一个自适应的滤波器可以过滤掉电话电路中的杂音。MADALINE 是空中交通控制系统的祖先，至今仍在使用。

后来，传统的冯诺依曼架构依旧牢牢占据了计算领域，神经网络的研究开始没落。主要原因是神经网络前期的成功过分夸大了神经网络的潜力，特别是考虑到当时的技术能力，神经网络的效果不能令人满意，很多结果都令人沮丧。

在上世纪 80 年代，学者对神经网络的研究又重新兴起。为了解决多层神经网络的训练问题，三个独立的研究组同时提出了一个解决方案，我们现在称之为反向传播算法 [67]。反向传播算法，可以将神经网络的误差反向逐层传播，因此达到训练权值的目的。因为层数很多，反向传播神经网络被称作慢速学习器，训练完成通常需要成千上万次迭代。

现在，神经网络算法被很多应用程序广泛使用。对于神经网络算法的研究仍旧在继续，未来对神经网络算法的研究严重依赖于硬件的发展。采用神经网络算法的人工智能围棋系统 AlphaGo[68] 在运行时，需要使用 1202 块 CPU 和 176 块 GPU 同时运算。

研究者研究神经网络的进度一直进展缓慢。由于硬件计算平台的性能限制，神经网络的训练周期通常需要几周，甚至几个月的时间。一些公司尝试直接使用电路来做出专用的神经网络芯片。数字电路，模拟电路，光芯片等都曾被尝试过 [69–76]。

随着硬件性能的提升，2006 年开始，深度学习逐渐成为热点。深度学习是机器学习领域的一个全新分支，因为其高维度特征提取和表示能力而知名。深度学习主要分为两种，一种是由多层全连接层神经网络组成的深度神经网络（deep neural networks，简称 DNN），另一种是含有卷积操作的卷积神经网络（convolutional neural networks，简称

CNN)。下面我们将向大家介绍，神经网络中的基本单元神经元以及几种常见的神经网络层类型。

2.1 神经网络算法基础

2.1.1 神经元

神经元是一种能够处理信息的特殊细胞，见图2.1。神经元包含以下几个部分：细胞体和突起，突起又分为轴突和树突两种。神经元通过突触从外部接收或者发送信号。当接收到的信号强度超过一个阈值，神经元会被激活并通过轴突向外部发送信号。发出的信号会被其他突触接收，并可能会激活其他神经元。因此按照用途划分可将神经元划为三类：输入神经元，传出神经元和连体神经元。

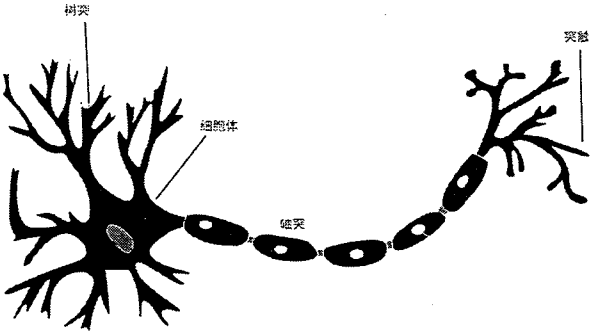


图 2.1 生物神经元

感知机 [77] 便是一个数学化了的神经元模型，它是一个有监督的线性分类学习器。感知机的输入是一些特征向量，这些特征向量和感知机的权值进行乘加操作，产生出预测结果，感知机模型如图2.2所示。感知机最早出现在上世纪 50 年代，是一个由定制电路做成的人工智能产品。

$$f(x)=\begin{cases}1 & w \cdot x+b>0\\ 0 & w \cdot x+b<=0\end{cases} \tag{2-1}$$

如公式2-1所示，感知机算法就是一个由输入向量 x 到输出函数 f(x) 的映射。w 是一个权值向量，w • x 是两个向量的内积，即 $\sum_{i=0}^{n} w_i x_i$，其中 n 表示向量的长度，b 是偏执。偏执是一个独立的常量，可以用来调节输出结果的分界线。对于一个分类问题，f(x) 的输出值 0 或者 1 表示输入是正样本或者负样本。假如 b 是一个负数，那么权值和输入的内积值必须要大于 b 的绝对值，f(x) 的值才会是 1。感知机算法只有当待学习问题是线性可分时才有效，当一个问题不是线性可分时，感知机学习过程不会得到一个合适的权值使问题的输入向量能够合适的分为两类。

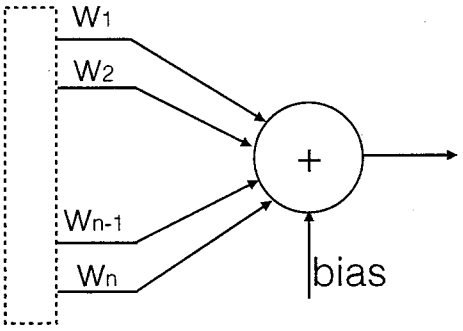


图 2.2 感知机模型

一个感知机的分类能力有限。因此，学者发明了新的算法，将多个感知机互相连接，再增加一些非线性运算，便组成了不同的神经网络。

2.1.2 全连接层 FC

全连接层 (fully connected layer，简称 FC 层)[78] 是神经网络中最经常出现的层，通常用于神经网络最后的特征组合以及分类。全连接层神经网络的结构如图2.3所示，其中层内的神经元之间没有连接，输入层的每个神经元和输出层的每个神经元都存在连接，因此被称为全连接层。全连接层神经网络是感知机由少到多的简单扩展，每个输出神经元的计算方式与感知机的计算方式完全相同。由全连接层的结构可以看出，全连接层的权值数量很多，计算量相对较少。

图2.3中输入神经元为 m 个，输出神经元为 n 个，其结构相当于由 n 个感知机构成。对于每一个输出神经元来说，都会使用所有的输入数据进行计算，计算公式为 $b_i = \sum_{j=0}^{m} w_{ij}a_j + bias_i$。其中，$w_{ij}$ 表示输入第 j 个神经元与输出第 i 个神经元之间的权值，$bias_i$ 表示第 i 个输出神经元的偏执。

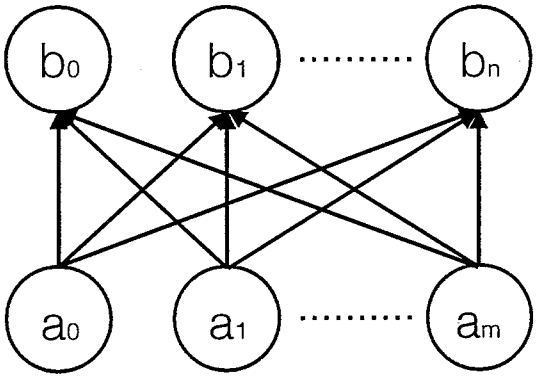


图 2.3 一层全连接神经网络

2.1.3 卷积层 CONV

使用全连接神经网络，在处理小型数据或者图像时（例如 MNIST 数据集中图像的分辨率是 28*28），提取整张图像的特征的计算量还处于可接受的范围。然而，当图像分辨率增大时（例如 Imagenet[11] 数据集图像分辨率为 256*256 左右），从整幅图像中

提取特征的计算量就会变得非常大。输入的神经元数量大约为 20 万个，假如我们要提取 100 个特征，则需要总共 2000 万个可训练参数。计算速度与 28*28 分辨率的输入图像相比要慢 250 倍。

一种直接的解决办法便是限制输入神经元与输出神经元之间的连接数量。具体地，每一个输出神经元只和一定区域内连续的输入神经元存在连接。对于图像是一块空间内的输入数据，对于语音则是某一时间戳后的时间段。生物学里视觉系统中，对外界的认知是从局部到全局的 [79]，在图像的数据中，也是空间相邻的图像像素相关性和连贯性比较高，在空间上距离较远的图像像素相关性则大大降低。研究表明视觉皮层的每个视觉细胞只能接收局部信息的刺激，所有的视觉细胞得到局部图像的特征信息传送给大脑综合处理后，才得到整个图像的最终高维度信息。卷积层（convolution layer，简称 CONV）充分利用了图像的空间关系，卷积核的大小直接限制了输出神经元只和周围一小块的输入神经元存在连接。这种连接关系的原理来自于神经元的局部感受域。局部连接的方法大大减少了卷积层中的权值数量。

卷积层是卷积神经网络的核心层，通常用于图像的预处理和特征提取，最早由 Yann Lecun 在 1998 年提出 [8]。当在处理像图像这种高维数据时，采用全连接神经网络，会拥有上亿的权值数量，同时这种算法也没有考虑图像的空间关系，因此全连接层处理高维数据是不切实际的。在卷积层中，有很多可以有学习能力的滤波器（或者卷积核），这些卷积核通常很小，但是他们却渗透到了输入图像的每一块区域。在正向运算时，每个卷积核在输入图像上滑动，卷积核与对应的输入图像数据做向量内积运算，输出为与这个卷积核相对应的一个二维的特征图像。因此，当卷积核在检测到目标特征时，输出结果便被激活，起到了特征提取的作用。

为了进一步降低权值数量，卷积层采用了共享权值的方法。也就是同一个输出特征图像上的所有像素点都采用同一组卷积核。这组卷积核在输入图像上左右和上下方向平移，与对应的输入神经元做向量内积操作得到输出结果。这就意味着同一个输出特征图像上的所有像素点是卷积核在输入图像不同位置上提取到的图像特征，而这些图像特征具有相同意义。由于不同的输出特征图像采用的卷积核不同，因此不同的输出特征图像上的像素点所表示的输出特征是不同的，这也是为什么输出特征图像有很多组的原因。

2.1.3.1 基本的卷积运算

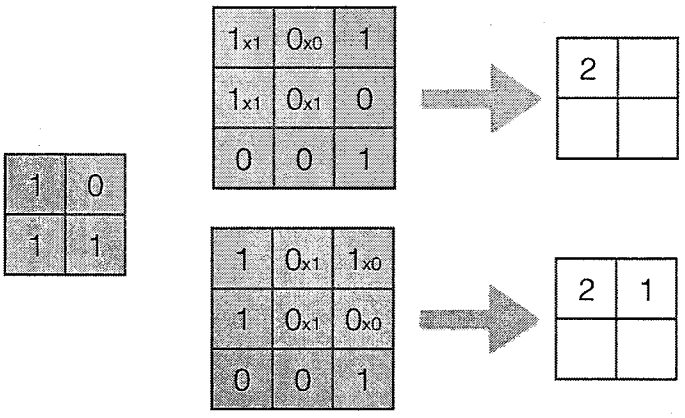


图 2.4 卷积操作示例

在卷积神经网络中，卷积运算与广义的卷积运算不同。为了具体阐明在这里的卷积操作，我们以一个具体的例子描述卷积操作的过程。如图2.4所示，输入数据是一个 3*3 的矩阵，有一个 2*2 大小的卷积核，这个卷积核在输入图像上以步长为 1 的大小从左向右从上向下滑动。每次滑动卷积核都会和对应的输入数据进行向量内积运算，最终得到了 2*2 的输出特征。卷积运算的公式为：

$$b_{x,y} = \sum_{p=0}^{kx} \sum_{q=0}^{ky} w_{pq} a_{x*sx+p,y*sy+q} \tag{2-2}$$

$b_{x,y}$ 表示在坐标（x，y）处的输出，w 为一个卷积核的权值矩阵。kx、ky 分别表示卷积核的长度和宽度，sx、sy 分别表示卷积核在输入图像上滑动时的步长，最小为 1。在图2.4的例子中，ky、kx 都是 2，sx、xy 都是 1。

2.1.3.2 多个特征图像时卷积的计算

上一节讲到的卷积操作输入和输出特征图像均为 1，在多个输入输出特征图时，计算方式也略有改变。每个输入和输出特征图像之间都会有一个卷积核。假设在一个卷积层中，输入特征图像数量是 fi，输出特征图像数量是 fo，那么总共的卷积核数量则是 fi*fo。卷积的运算的公式为：

$$b^{i}_{x,y} = \sum_{j=0}^{fi} \sum_{p=0}^{kx} \sum_{q=0}^{ky} w^{ij}_{pq} a^{j}_{x*sx+p,y*sy+q} \tag{2-3}$$

$b^{i}_{x,y}$ 表示第 i 个输出特征图像在坐标（x，y）处的输出，w^{ij} 表示第 i 个输出特征图像和第 j 个输入特征图像之间的卷积核矩阵。

2.1.4 池化层 POOL

池化层（pooling layer，简称 POOL）也是卷积神经网络中重要的一个操作。池化层仿照生物的视觉系统，对输入数据进行非线性的降采样操作，用更高层的抽象特征来表示数据。池化层主要应用在图像处理中的卷积神经网络中，通常池化层会出现在卷积层之后，对卷积得到的图像特征进行降采样处理。池化层有很多不同的降采样方法，如最大值池化，平均值池化等。池化层操作将输入图像分为几个小的矩形，在最大值池化操作中，对于每一个小的矩形都会输出一个最大值。在平均值池池化操作中，对于每一个小的矩形都会输出矩形中像素的平均值。

池化层操作为卷积神经网络带来了很多好处。首先，池化层操作显著减小了图像的分辨率，从而减小了整个网络的可训练参数数量和计算量。另外，池化层对输入矩形内特征相同对待，而不考虑他们的位置关系，这使得池化层在处理图像时，具有抗平移，旋转，卷曲等特征，因此池化层也能起到防止网络过拟合的作用。

2.1.4.1 最大值池化

最大值池化算法是最常用的一种池化算法。其基本思想是在选定的数据区域内选择出一个最大值作为输出。其计算公式为：

$$b_{x,y}^{i} = MAX_{p=0,q=0}^{p<kx,q<py}(a_{x*sx+p,y*sy+q}^{i}) \tag{2-4}$$

其中 kx、ky 为池化操作的输入范围大小，在本文中称作池化核的长和宽，sx、sy 表示池化操作之间的距离。在最开始的时候池化区域的大小通常和池化操作的距离相等，即池化操作的输入数据不会重复，输入数据会被均等的分成几个独立的区域。后来，出现了数据复用的池化，即 sx 小于 kx 或者 sy 小于 ky。

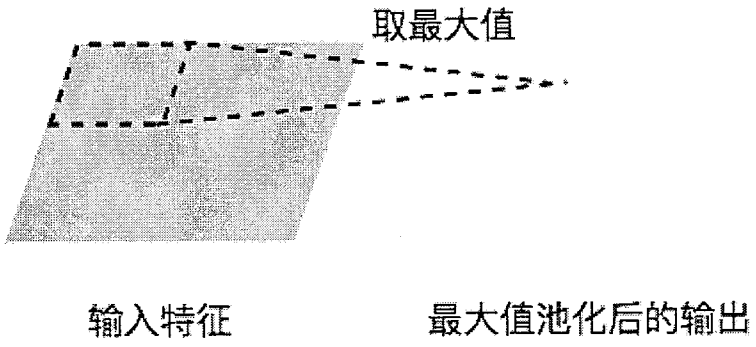


图 2.5 最大值池化

2.1.4.2 平均值池化

平均值池化算法是另一种常用的池化算法，其算法与最大值池化类似，只不过将取最大值的操作替换成求平均数的计算。其公式为：

$$b_{x,y}^{i}=\frac{\sum_{p=0}^{kx}\sum_{q=0}^{ky}a_{x*sx+p,y*sy+q}^{i}}{kx*ky} \tag{2-5}$$

2.1.5 归一化层 NORM

近年来随着，数据量和网络规模和结构的复杂度增加，神经网络越来越容易进入过拟合的状态。近几年出现了各种各样的归一化层 (normalization layer，简称 NORM)，用来避免模型产生非线性决策边界，从而避免网络进入过拟合状态。归一化层有很多种，如局部响应归一化层 (local response normalization layer，简称 LRN)[80]，批归一化（batch normalization layer，简称 BN）[81]，局部中心归一化（local contrast normalization layer，简称 LCN）[82] 等。

2.1.5.1 LRN

局部响应归一化层有两种不同的方法，一种是输出与多个输入特征图像相关，我们称作交叉特征图像的局部响应归一化方法，一种是输出只与和自己对应的输入特征图像相关，我们称作特征图内的局部响应归一化方法。这两种算法虽然都叫局部响应归一化层，但是在算法的具体实现上有着很大的不同。

（1）交叉特征图的 LRN 算法

交叉特征图像的 LRN 算法输出与输入的图像大小相同，每一个输出像素点的值只和输入图像同一位置的周围几个输入特征图像的几个像素点值有关，计算输出点 $b_{x,y}^{i}$ 具体的公式如下：

$$b_{x,y}^{i}=\frac{a_{x,y}^{i}}{(k+\alpha\sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)}{a_{x,y}^{i}}^{2})^{\beta}} \tag{2-6}$$

其中，累加和是在同一空间位置上的周围临近 n 个特征图像上像素点的加和，N 是输入总的特征图像数量。很明显，特征图像的顺序是任意的，在训练之前就已经决定了。这种侧向抑制反应归一化操作是来自生物神经元的启发。局部响应归一化运算使得临近的不同的特征图像之间产生竞争。常量 n、k、α、β 是由验证集得到的超参数。

（2）特征图内的 LRN 算法

与交叉特征图像的 LRN 方法不同，交叉特征图像的 LRN 算法的输出只和当前的特征图像周围的像素点相关，计算输出点 $b_{x,y}^{i}$ 具体的公式如下：

$$b_{x,y}^{i}=\frac{a_{x,y}^{i}}{(k+\alpha\sum_{j=max(0,y-n/2)}^{min(Y,y+n/2)}\sum_{k=max(0,x-n/2)}^{min(X,x+n/2)}{a_{x,y}^{i}}^{2})^{\beta}} \tag{2-7}$$

其中，累加和是在同一特征图像上周围长宽为 n 的正方形内的正方形内的点的加和。X、Y 分别为输入图像的长和宽。这种累加方式和平均值池化的计算很相似。常量 n、k、α、β 是由验证数据集得到的超参数。

2.1.5.2 BN

在神经网络训练中，由于神经网络每一次的输入都是变化的，导致每一次训练时每一层的输出结果也是不固定的。因此在训练神经网络时，需要采用较小的学习率和合适的权值初始化策略。这就使得训练神经网络过程变得漫长而复杂，需要很多经验来调节参数。BN 方法可以有效的解决上面的困难，并且防止过拟合，提高神经网络的精度。BN 方法使得我们可以在训练时使用较大的学习率，减缓权值初始化的限制条件。

BN 的运算比较复杂，分为以下 4 个步骤：

$$\mu_\beta = \frac{1}{m}\sum_{i=1}^{m} a_i \tag{2-8}$$

$$\theta_\beta^2 = \frac{1}{m}\sum_{i=1}^{m}(a_i - \mu\beta)^2 \tag{2-9}$$

$$\hat{a}_i = \frac{a_i - \mu\beta}{\sqrt{\theta_\beta^2 + \epsilon}} \tag{2-10}$$

$$b_i = \gamma\hat{a}_i + \beta \tag{2-11}$$

在上述的算法中，a_i 为第 i 个批数据的输入数据，m 为总的批数据的个数。为了数字稳定，添加了一个批变化常数 ϵ。在最后一步，参数 γ 和 β 是在训练时需要学习的参数。需要注意的是，b_i 的值不仅和自己的输入样本有关也和批数据中其他的训练样本有关，经过缩放和转换运算之后，他们被传到下一层。在第三步 a 的归一化中，虽然只与本层数据有关，但是它的计算相当关键。当我们忽略掉参数 ϵ，并且在一个批数据中所有的样本来自于同一分布的取样，那么 $\hat{a}$ 的期望值为 0，标准差为 1。我们可以由 $\sum_{i=1}^{m}\hat{a} = 0$，$\frac{1}{m}\sum_{i=1}^{m}\hat{a}_i^2$ 得到。

2.1.5.3 LCN

局部差异归一化层采用非线性的方法去归一化一个输入图像。在局部差异归一化算法中不是采用一整张图像的全局信息做为输入去归一化输出，而是每一个输出只采用局部的像素作为输入。这种方法也是受启发于哺乳动物的视觉系统。在哺乳的动物的视觉系统中，每一个神经元也只连接了一些视觉感受细胞，这些视觉感受细胞也只能获得一些局部信息。

局部差异归一化的计算主要采用了局部减法和除法归一化，对于输入点 $a^i_{x,y}$，减法归一化的方法如下：

$$v^i_{x,y} = a^i_{x,y} - \sum_{pq} w_{pq} \cdot a^i_{x+p,y+q} \tag{2-12}$$

其中，$w_{p,q}$ 是一个归一化的高斯权值窗，因此 $\sum_{pq} w_{pq} = 1$。除法归一化的计算方法如下：

$$b^i_{x,y} = \frac{v^i_{x,y}}{max(c, \theta_{x,y})} \tag{2-13}$$

$$\theta_{x,y} = \sqrt{\sum_{pq} w_{pq} \cdot (v^i_{x+p,y+q})^2} \tag{2-14}$$

其中，对于每一个样本，常数 c 被设定为 $\theta_{x,y}$ 的均值。

2.2 激活 ACT

因为线性模型的表达能力不够，神经网络中加入了一系列非线性的操作，激活层(activation layer，简称 ACT) 作为其中的一个大类，在神经网络中起着不可或缺的关键地位。

2.2.1 Sigmoid

在神经网络发展前期，Sigmoid 函数被广泛使用，Sigmoid 的公式为:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2-15}$$

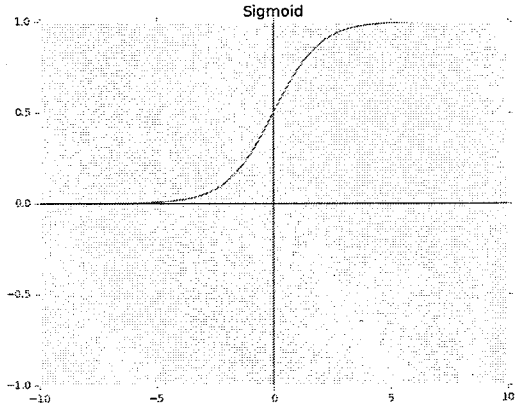


图 2.6 Sigmoid 函数

Sigmoid 函数图如图2.6所示。Sigmoid 函数之所以被广泛使用因为它有很多好处。首先，Sigmoid 可将输入值域转换为 0 到 1 之间的有界限的区间。其次，Sigmoid 是单调递

增的函数，且值一直为正数。最后，Sigmoid 求导方便，导数公式为 $f'(x)=f(x)(1-f(x))$，因此有利于神经网络的反向训练过程。

然而 Sigmoid 函数也有很多缺点，首先一点就是计算复杂，超越函数的运算耗时比较长，影响神经网络运算速度。其次，Sigmoid 函数容易在边缘产生饱和效应，使得梯度的值很小，不利于神经网络的训练。基于以上两种原因 Sigmoid 函数逐渐被冷淡。

2.2.2 Tanh

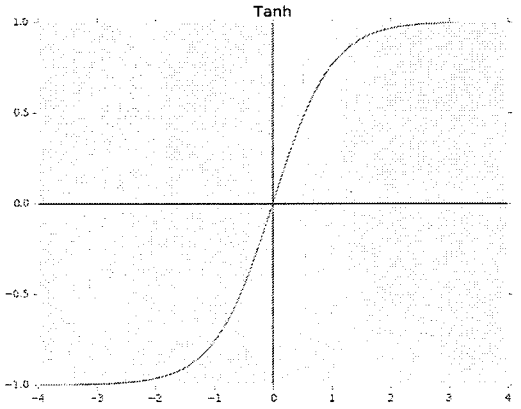


图 2.7 Tanh 函数

Tanh 激活函数与 Sigmoid 函数类似，都是饱和函数。Tanh 激活函数的公式为：

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

Tanh 激活函数的计算量比 Sigmoid 更大，现在也很少被人使用。

2.2.3 ReLU

ReLU[83,84] 激活函数是最近出现的激活函数，ReLU 函数计算相对简单，没有超越函数的运算，公式如下：

$$f(x)=\begin{cases}0 & \text{x} <= 0\\ x & \text{x} > 0\end{cases} \tag{2-17}$$

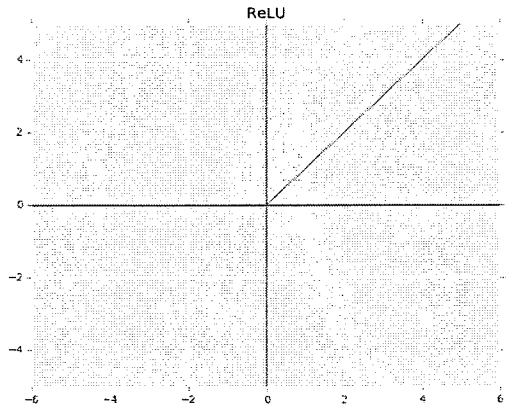


图 2.8 ReLU 函数

ReLU 是非饱和函数并且求导简单，因此相比于使用饱和函数 Sigmoid 等，使用 ReLU 做激活函数，可以大大加快神经网络的训练效率。因此 ReLU 在训练速度和效率上都要比 Sigmoid 好。

2.2.4 PReLU

PReLU[12] 即带参数的 ReLU，与 ReLU 相比，当 x<0 时，PReLU 函数的输出不再为 0。PReLU 的公式如下：

$$f(x)=\begin{cases} ax & \text{x <= 0} \\ x & \text{x > 0} \end{cases} \tag{2-18}$$

PReLU 不仅可以自适应计算得到修正参数，并且可以提高神经网络的精度，而它增加的计算成本可忽略不计。

2.2.5 RReLU

RReLU[85] 在数学形式上与 PReLU 基本相同，RReLU 在 x<0 时斜率是一个随机数。这种随机数不具有确定性，能够在某些程度上起到归一化的作用，从而防止网络进入过拟合的状态，提高神经网络精度。

2.2.6 ELU

ELU[86] 综合了 Sigmoid 和 ReLU 的特征，具有左侧饱和性和右侧开放性的特点，公式如下：

$$f(x)=\begin{cases} \alpha(e^{x}-1) & \text{x <= 0} \\ x & \text{x > 0} \end{cases} \tag{2-19}$$

在 x > 0 时，ELU 向 ReLU 一样，具有 ReLU 的特点，缓解了在神经网络训练时梯度消失的弊端。在 x <= 0 时，ELU 能使得神经网络对输入变化或噪音更鲁棒。ELU 函数的输出均值接近于 0，因此 ELU 使得神经网络在训练时可以更快收敛。

2.3 精确性驱动的神经网络算法发展趋势

神经网络在出现的几十年里，学者都在为提高神经网络解决问题的能力而付出努力。不管是在神经网络的层内结构，还是神经网络的层间结构，甚至是神经网络的规模，都在逐渐的变得复杂。

2.3.1 神经网络层内结构的变化

为了预测时间序列这类问题，学者发明了带有记忆功能的神经元，他们内部相比上面提到的神经网络要复杂的多。在这一类算法中的典型代表是反复神经网络 [87]（recurrent neural networks，简称 RNN）和长短期记忆网络 [88]（long short term memory，简称 LSTM）。

2.3.1.1 RNN

RNN 通常用来预测一个序列的信息。在传统的神经网络中，我们假设所有的输入数据是独立的，这对于很多应用来说是不合理的。假如你想预测语音中的下一个单词，在知道之前的几个单词的情况下，显然会更有助当前的预测。RNN 之所以被叫做反复，因为他们对输入序列的数据做同样的操作，并且每一个当前输出都依赖于之前的输出结果。另一种合理的解释是，可以认为 RNN 拥有记忆功能，它能够记住之前的计算结果，并影响当前的计算结果。

理论上，RNN 可以使用任意长度的序列作为输入，实际上由于计算等限制，只采用一小段片段作为预测的输入。RNN 的结构如图2.9所示。

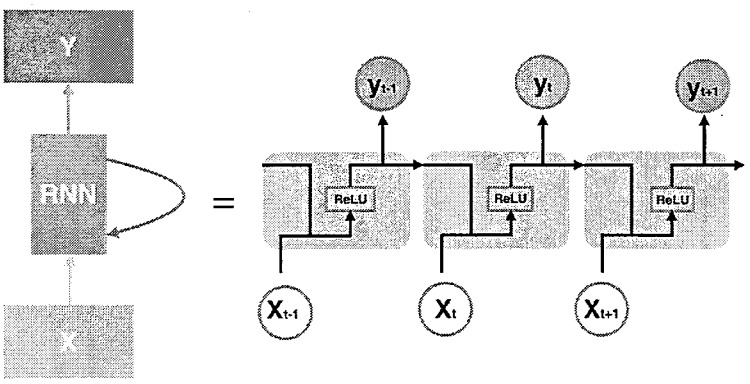


图 2.9 RNN 算法示意图

其计算步骤主要分为两步，第一步是计算中间的隐层变量 h_t，h_t 的值不仅和当前的输入 x_t 有关，也和 h 上一时刻的输出 h_{t-1} 有关。第二步，根据 h_t 的结果计算出最终的输出 y_t。公式如下：

$$h_t = f(w_h h_{t-1} + w_x x_t) \tag{2-20}$$

$$y_t = w_y h_t \tag{2-21}$$

在上面的公式中，总共有三组权值：w_h，w_x，w_y，这三组权值是互相独立的，都是在训练阶段得到的数据。在第一步的公式中，f(x) 函数为激活函数，通常为 Tanh 或者 ReLU 等。

2.3.1.2 LSTM

在 RNN 的隐层中，只有一个激活门，因此它对当前的输入异常敏感，因此不能很好地保存长期输入序列的状态值。因此 RNN 很难处理这种长期依赖问题，Bengio 等人对这个问题做了细致的研究 [89]。

LSTM 的提出成功解决了上面的问题。在结构上，LSTM 是一种更加复杂的 RNN，能够解决长依赖问题，由 Hochreiter 和 Schmidhuber[88] 在 1997 年提出。自诞生之后，LSTM 就受到很多学者的青睐，经过了很多算法的优化，被广泛应用在很多流行的神经网络里。与上面 RNN 的结构类似，LSTM 只不过内部的运算更加复杂:

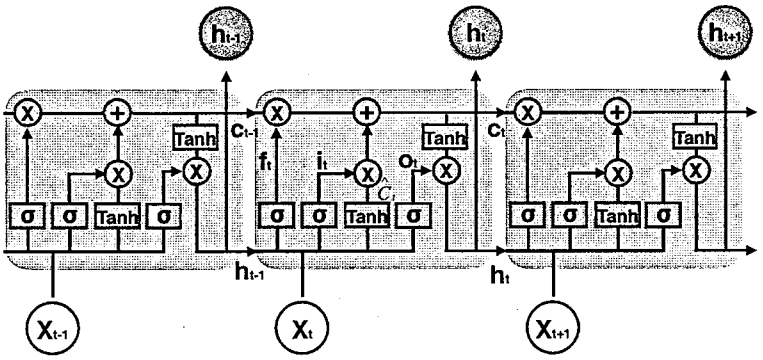


图 2.10 LSTM 算法示意图 [3]

LSTM 中计算分为多步，就是不同的状态，计算过程就像一个传送带一样，从左到右经历了几次线性运算以及激活函数。LSTM 通过门拥有向状态中添加或者删除信息的能力，门即是图中各个激活函数，σ 表示是 Sigmoid 激活函数。在上图中有 4 个门，主要的计算步骤如下：

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{2-22}$$

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{2-23}$$

$$\hat{C}_t = Tanh(W_C[h_{t-1}, x_t] + b_C) \tag{2-24}$$

$$C_t = f_t C_{t-1} + i_t \hat{C}_t \tag{2-25}$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{2-26}$$

$$h_t = o_t Tanh(C_t) \tag{2-27}$$

2.3.2 神经网络层间结构的变化

近年来，研究人员们发现，单纯通过增大模型规模的方法，已经不能达到提高精度的效果。因此，他们开始在模型的结构上进行研究，提出了一系列精巧、高效的模型结构，在训练数据集不变的情况下取得了更好识别效果，同时还降低了权值的数量。神经网络结构，也从原来的线性结构，向着具有更复杂的拓扑结构演变。本小结中，我们将对具有层间结构变化的网络结构进行介绍探讨。我们将首先介绍单纯增大网络规模获得精度提升的劣势，然后介绍一个最近流行的带有复杂拓扑结构的网络模型：GoogLeNet[90]，并说明这种结构的优势以及特点，类似的结构还有 ResNet[91]。

2.3.2.1 增大网络规模

要提高神经网络算法精度，最直接的方法就是直接增大其规模，一方面从深度上进行改善，也就是增加层的数量，另一方面增大每一层的规模，增加层的神经元突触个数。当数据集的数量足够多的情况下，这种方法是最保险可行的，但这种简单的方法也有其缺点。

(1) 过拟合

当神经网络模型的深度和规模增加，就意味着可训练参数数量的大量增加，这可能导致网络过拟合。尤其是当数据集的数量有限，不足够多的情况下，要提供大量的标记数据作为训练数据是一件非常耗费人工和时间的事情。因此，过拟合会导致这种网络的精度达到瓶颈，无法提升。

(2) 计算量

增大网络规模的另一个缺点是，随之增加的计算量。比如说两个相邻的卷积层，任何形式的增加卷积核的数量，都会导致计算量的成倍增长。这些计算中不一定全部能够对结果有较大的影响。比如，经过训练后，一些权值可能非常小接近于 0，这就会导致大量的计算浪费。

2.3.2.2 GoogleNet 网络

Szegedy 等人于 2014 年提出了 GoogLeNet 网络结构，其在 ILSVRC 2014 的图像分类以及目标检测出任务中都取得了第一名的成绩。除了提高识别精度，控制权值的数量也是 GoogLeNet 在设计时的一个重要考虑因素，尤其是考虑到手机和嵌入式平台对深度学习算法计算的需求日益增加。这些平台上计算资源匮乏，如果网络模型的计算规模巨大，可能导致其无法使用。因此，GoogLeNet 在设计时，就将正向预测时乘法和加法的计算量控制在 15 亿次之内，以保证它不仅仅是学术上的研究，同时也能被工业界采纳使用。

(1) Inception 架构

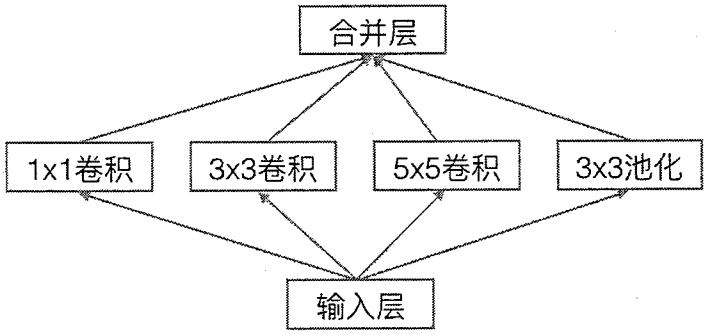


图 2.11 朴素 Inception 结构

Inception 架构的主要思想是：如何估计找出卷积神经网络结构中的最优局部稀疏结构，然后用已经存在的稠密结构代替。在更接近输入图片的层中，相关联的神经元会聚集在一个局部区域内，可以用 1×1 的卷积层提取特征。然而，这个局部区域的大小不是一定的，可能有空间上分布更广的聚类，需要用更大一些的卷积核来覆盖。图2.11中展示的就是一个 Inception 结构，其中包含一个 1x1 的卷积，一个 3x3 的卷积，一个 5x5 的卷积，以及一个 3x3 的池化层。这个结构从一个输入层，通过不同的卷积核计算出不同大小的输出数据，然后通过一个合并层将这些不同大小的拼在一起，作为下一层的输入。

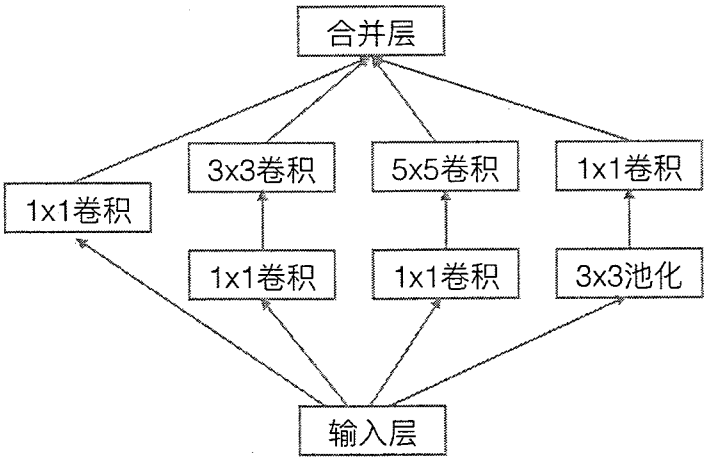


图 2.12　有维度减少的 Inception 结构

这种朴素的 Inception 结构的问题在于，即使是 5x5 的卷积核，如果卷积核的数量过大，也会带来巨大的计算开销。此外，由于池化层的输入输出特征图数量一样，因此一旦将池化层也加进来，这个问题就会变得更加的严重。即使这样的架构能够提取到最优的稀疏结构，也会导致大量的运算浪费，不够高效。由此，在原来的结构上进行变化，加入维度消减，获得如图2.12的结构。

2.4　能耗驱动的神经网络发展方向

很多研究人员致力于提高神经网络的运算速度或者降低神经网络运算时的能耗。有很多不同的研究方向可以做到以上两点，其中主要包括两个大方向：一是神经网络的低精度运算，二是对神经网络的模型进行压缩。

2.4.1　神经网络低精度算法

大型的神经网络往往受限于计算平台，传统的神经网络都采用浮点数据进行运算，浮点运算比定点计算耗时要长，并且能耗开销更大。因此，现在出现了很多使用低精度数据进行神经网络运算的做法。

2.4.1.1　定点数据格式

前期已经有很多神经网络采用了低精度的实现。Hammerstrom[92] 提出了一个在线学习的神经网络框架，权值只需要使用 8 比特或者 16 比特定点数据来表示。Holt 和 Hwang[93] 理论分析了在精度受限的情况下，神经网络在训练时的学习能力。在一些简单的小神经网络上，8-16 位的精度对于神经网络的训练已经是足够的了。

Suyog Gupta[94] 研究了限制精度的情况下，神经网络训练时的数据表示和计算。在使用低精度定点运算格式的情况下，他们观察到数据的进位策略会影响神经网络运算时的行为。他们的研究表明，使用 16 位定点数据格式，运算器中采用随机进位策略，可

以完成深度神经网络的训练。另外，他们还实现了 16 位定点的神经网络加速器，大大降低了能耗。

2.4.1.2 比特数据格式

Matthieu Courbariaux[95] 等人发表了一个训练二进制网络（binary neural networks，简称 BNN）的方法，在 BNN 中，权值和激活都使用二进制数据格式。在神经网络正向运算中，BNN 可以急剧减小内存的占用和访问数量，同时将很多运算用二进制位操作所替代。这样，BNN 可以大幅提升运算速度同时又减少能耗。作者在 Torch 7 和 Theano 编程框架上分别实现了 BNN。在 MNIST、CIFAR10、SVHN[96] 等数据集上，两个采用 BNN 实现的运算平台，计算速度都得到了大幅的提升。最后，作者使用 GPU 代码，编写了二进制运算的矩阵乘法函数。在 MNIST 数据集上，BNN 可以比之前的 GPU 代码速度快 7 倍，同时不损失图像分类的精度。

Mohammad Rastegari[97] 等人提出了一个 XNOR 的低精度神经网络算法。与 BNN 不同，在 XNOR 网络的卷积层中，不仅权值是二进制表示的，输入神经元数据也是二进制表示的。XNOR 网络将所有的卷积运算都使用二进制操作来替代。XNOR 的卷积运算比原始的卷积运算快 58 倍，同时内存占用减少了 32 倍。XNOR 网络使得在 CPU 上运行实时的神经网络识别任务成为可能。XNOR 神经网络使得 BNN 神经网络运算变得简单，准确，高效，并且在真实的视觉识别场景中具有真实的使用价值。在 AlexNet 上，使用二进制权值表示的神经网络和原始的网络相比，精度基本没有损失。和 BNN 相比，XNOR 网络具有更高的识别精度。

表 2.1 XNOR 卷积参数与正常卷积参数的比较

	使用的操作	内存减少	计算加速	精度 (AlexNet)
标准的卷积	加，减，乘	1x	1x	56.7%
二进制权值	加，减	32x	2x	56.8%
二进制权值，二进制输入 (XNOR-Net)	位操作	32x	58x	44.2%

2.4.2 神经网络模型压缩

现在很多神经网络的目标是提高神经网络的识别精度。在一个给定的识别精度的前提下，神经网络的模型存在冗余的参数，参数数量有减少的可能。参数减少可以带来以下几个好处：（1）在神经网络训练时，更少的参数数量可以减少神经网络参数在不同设备上的传递。（2）小的神经网络模型，可以使得移动设备更容易从服务器端导入新的模型。（3）小的神经网络模型适用范围更广，更容易被部署到 FPGA、加速器、手机等内存紧张的设备中。

表 2.2 几种压缩网络的比较

网络结构	压缩方法	数据类型	模型大小的改变	模型减小	top1 精度	top5 精度
AlexNet	无	32bit	240MB	1x	57.2%	80.3%
AlexNet	SVD	32bit	240MB->48MB	5x	56.0%	79.4%
AlexNet	Deep compression	5-8bit	240MB->6.9MB	35x	57.2%	80.3%
SqueezeNet	无	32bit	4.8MB	50x	57.5%	80.3%
SqueezeNet	Deep compression	6bit	4.8MB->0.47MB	510x	57.5%	80.3%

Emily Denton[62] 等人提出了一个神经网络参数压缩的方法，叫做 SVD。SVD 方法使用卷积参数矩阵的金斯表示来减少卷积层的运算，SVD 在很多模型上都能达到超过 2 倍的加速，并且精度损失在 1% 以内。

Song Han[63] 等人提出了一个更好的压缩方法：Deep compression。Deep compression 总共有三个步骤：剪枝，量化，哈夫曼编码。在这个方法中，首先在训练时，每次只保留关键的可训练参数，其次，对权值进行量化，迫使权值之间出现权值共享，最后使用哈夫曼编码，进一步减小数据量。在使用第一步和第二步的方法之后，对网络进行重训练，进一步提高网络量化的结果。量化可以降低权值数据的位宽，可以从 32 比特降低到只有 5 比特。在 Imagenet 数据集上，Deep compression 方法将权值参数降低了 35 倍的存储，从 240MB 减少到了 6.9MB。在 VGG-16 网络上，该方法将权值存储从 552MB 减少到了 11.3MB，减少了 49 倍。这么小的模型可以使得，模型可以存储在偏上 SRAM 中，而不是片外内存中。在 CPU 和 GPU 上，压缩神经网络的计算速度可以提高 3 到 4 倍，能耗也能减少 3 到 7 倍。

Forrest N. Iandola[64] 等人提出了一个减小神经网络参数的方法，叫做 SqueezeNet。使用 SqueezeNet 方法实现的 AlexNet 网络，在不损失精度的情况下，参数的大小可以减少 50 多倍。在使用了模型压缩技术之后，模型大小可以比 AlexNet 模型小 510 倍。

第三章　稀疏神经网络加速器的实现

3.1　背景

由于严峻的能耗问题，加速器成为一个相比于 CPU 和 GPU 更为节能的替代品。一般来说，加速器都专注于某一单一领域，比如浮点运算单元、H.264[98] 解码单元等。然而，最近的研究表明，一系列神经网络算法在众多领域都取得了非凡的成绩，包括：图像语音识别、机器翻译、广告推荐等等 [99–102]。可见神经网络算法具有广阔的应用前景，越来越受到人们的重视。这使得创造一个既高效而又应用广泛的神经网络加速器成为可能并具有现实意义。因此，研究人员提出了很多神经网络加速器设计 [2,29–31,103]。

随着硬件设计能力的提升，硬件的能耗和性能一直都进步飞快，因此设计更大规模的神经网络算法成为可能。最近的研究显示，越大规模的神经网络会带来更好的正确率。2012 年 Krizhevsky 提出的 AlexNet 拥有 605 万可训练参数，2014 年 Karen Simonyan 等人提出的 VGG16 模型 [104]，可训练参数提升到了一千多万。2015 年，Kaiming He 提出的 ResNet 在 Imagenet 数据集上可训练参数的层数高达 152 层，在 CIFAR10[105] 数据集上实现了一个拥有 1202 层可训练参数的网络。数量巨大的权值会带了巨大的计算量和访存量等问题，如何高效地处理这种神经网络仍旧是神经网络加速器需要解决的问题。

面临神经网络参数爆炸式增长的趋势，一些学者另辟蹊径，在保证神经网络精度不变的前提下，提出了一些神经网络稀疏化方法来减少神经网络的权值。这些方法有：训练时的 dropout，稀疏表示，稀疏代价函数等等 [106–110]。图3.1是一个全连接层神经网络在稀疏化前后的对比图。在稀疏化的神经网络中，值为 0 的权值全部被去掉了，因为它们不会影响计算结果，可以在计算中将其跳过。在去掉这些权值之后，没有连接的神经元也可以被去掉。因此相比原始的神经网络，稀疏化后的神经网络权值和神经元都减少了很多。最近，Han[63] 等人提出了一种神经网络稀疏化的方法，这种方法可以在不损失精度的前提下，将神经网络的权值减少近 10 倍。

不幸的是，现有的加速器并不能在稀疏神经网络上获得收益。现有的加速器只能处理规则的矩阵和向量运算，面对不规则连接的稀疏神经网络，它只能按照全连接的矩阵或者向量的方式进行运算。对于 CPU 和 GPU，它们都有对应的稀疏矩阵运算加速库 sparse BLAS[111] 和 cuSPARSE[112]。对于 AlexNet 网络，稀疏化前后的权值数量分别是 5948 万和 699 万，然而在 GPU 上使用 cuSPARSE 获得的速度提升只有 1.78 倍。在 CPU 上采用 sparse BLAS 库，计算速度反而比原始网络更慢。性能卓越的 DianNao 加速器甚至不支持稀疏神经网络，只能将稀疏神经网络当成普通的稠密网络来进行计算，因此也不能从稀疏神经网络中获得收益。

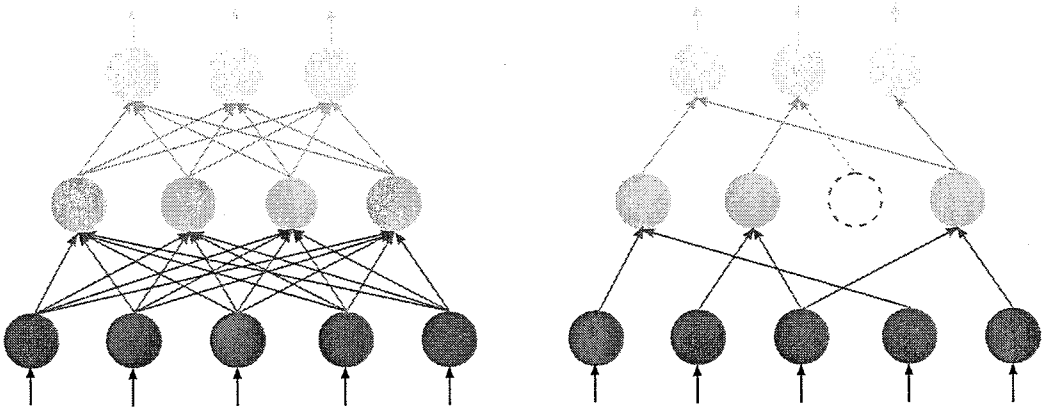


图 3.1　一个全连接层稀疏前后对比图

在本章中，我们提出了一个新异的加速器微结构，稀疏神经网络加速器微结构。该加速器不仅支持原始的稠密神经网络，而且也能更快的执行稀疏的神经网络。稀疏神经网络加速器是基于多运算单元的结构，由众多运算单元、译码器和缓存控制单元等部件协作完成各种神经网络运算。为了实现对稀疏神经网络的加速，在缓存控制单元中集成了高效的连接索引处理单元。连接索引处理单元从神经元缓存中读出输入神经元，根据神经元连接关系，有选择性的将神经元选择出并通过胖树 (Fat-tree)[113] 发送给不同运算单元。采用胖树拓扑结构一方面是为了减少电路的面积，另一方面由于稀疏化的原因，每个 PE 的计算量可能不同，胖树可以令 PE 工作在异步的模式，互不影响。

我们测量了稀疏神经网络加速器在几个流行神经网络 LeNet-5、AlexNet、VGG16 上的性能。这几个网络规模不同，结构相异，并且在稀疏化之后，稀疏程度也不同。和性能优异的加速器 DianNao 相比，稀疏神经网络加速器平均能够达到 7.23 倍的加速比和 6.43 倍的能量减少。与此同时，稀疏神经网络加速器的功耗只有 954mW，面积只有 6.38mm^2。在 GPU 使用稀疏矩阵加速库 cuSPARSE 的前提下，我们的加速器能够达到 10.6 倍的加速比，同时能耗减少 29.43 倍。和使用 sparse BLAS 加速库的 CPU 相比，我们的加速器能够达到 144.41 倍的加速。

3.2　动机

最近神经网络在各个领域都取得了非凡的成绩，神经网络主要包括两大类：卷积神经网络（CNN）和深度神经网络（DNN）。通常，神经网络都会包含以下 4 种类型：卷积层，池化层，正规化层，全连接层。卷积神经网络和深度神经网络的主要区别是，卷积层的权值在前者中是共享的，在后者中是私有的。下图3.2是一个卷积神经网络的典型代表 LeNet-5 卷积神经网络的结构示意图。LeNet-5 含有两个用来提取图像特征的卷积层（C1 和 C3），在 C1 层使用 1*6 个卷积核提取出 6 个特征图，每个卷积核的大小是 5*5，在 C3 层使用 6*16 个卷积核提取出 16 个特征图。S2 和 S4 是两个池化层，采用最大值或者平均值降采样的方法，对图像进行 2*2 像素的降采样，通过降采样层，图像的

分辨率在长度和宽度上都减小了一半。在网络的尾部有三层全连接层 F5、F6 和 F7，全连接层的目的是为了根据提取到的特征对图像进行分类。需要注意的是，LeNet-5 不含有最近新提出的各种正规化层。尽管在 LeNet-5 中权值的数量只有 5 万多，但是最近出现的网络里最大的权值数量已经到了 100 亿的规模 [114]，这急剧增加了神经网络的计算量和访存量。

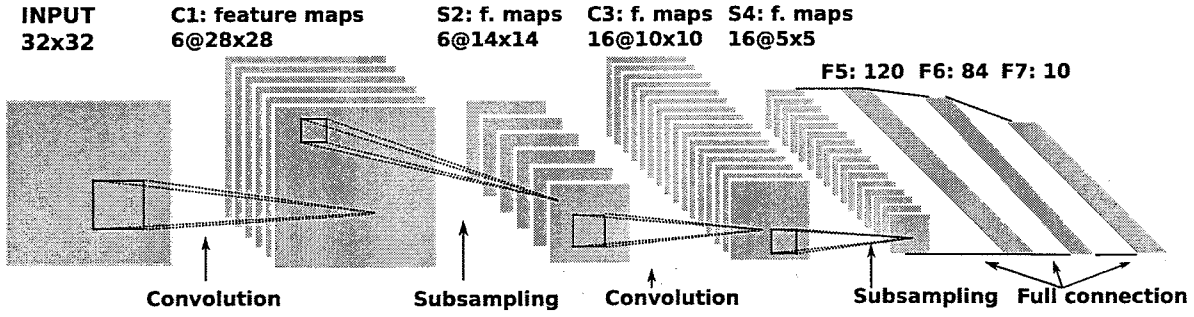


图 3.2 经典的卷积神经网络 LeNet-5 结构图

表 3.1 一些流行神经网络稀疏前后参数比较

NN	稠密网络		稀疏网络			
	神经元	权值	权值	稀疏率 (%)	卷积层权值	全连接层权值
LeNet-5 [8]	8.90K	430.62K	36.30K	8.43	3.33K	32.97K
AlexNet [80]	1.28M	60.95M	6.80M	11.15	864.86K	5.93M
VGG16 [104]	14.53M	138.34M	10.53M	7.61	4.81M	5.72M

由于大规模的神经网络使得处理神经网络性能下降，研究人员提出了一系列的训练技巧使神经网络稀疏化并且不损失原始的正确率，如稀疏编码 [115]，自动编码/解码机，深信度网络 [108] 等。2015 年，Han 等人提出了一个新的神经网络稀疏化方法。Han 的方法主要分 3 步，在第一步中，采用普通的神经网络训练方法将神经网络训练好，然后去掉权值的绝对值小于某个固定阈值的权值，最后，再用一个很小的学习率，重新训练这个网络。为了尽量的增大神经网络的稀疏率，上述过程的后两步可能要被重复进行多次，直到权值不能再被减少为止。表3.1是几个流行神经网络在稀疏化前后的参数数量情况，平均来看，在不损失神经网络正确率的情况下，这几个网络的总体稀疏率能够达到 12%。尽管稀疏神经网络中的运算和访存都会比原始网络小，但是目前的计算平台（CPU、GPU、FPGA 和各种加速器）由于缺少专用的处理不规则的稀疏连接的矩阵的模块，因此从稀疏连接的神经网络中获益微乎其微，在稀疏度不高的情况下，性能反而会下降。

在通用的计算平台 CPU 和 GPU 上，稀疏神经网络的性能提升微乎其微。图3.3表示

了稀疏神经网络与稠密神经网络分别在 CPU 和 GPU 上的性能比较，稀疏神经网络的执行在 CPU 和 GPU 上分别使用了 Sparse BLAS 对比 Caffe 和 cuSPARSE 对比 Caffe。在表格中，我们也列出了每个网络的稀疏度。在 CPU 平台上，除了 LeNet-5 网络，稀疏神经网络的性能都比稠密网络要差，平均下来，性能降低了 2.11 倍。在 GPU 平台上，在所有被测量的神经网络平均稀疏度 9.06% 的前提下，性能只提升了 23.34%。尽管研究人员已经想出了各种性能优化的方法，由于稀疏神经网络自带的带宽问题和非计算部分的开销，稀疏神经网络的运算性能还是要远低于峰值计算能力，在很多矩阵向量的运算中，实际运算效率只有峰值计算能力的 1%。在 FPGA 平台上，稀疏矩阵通常采用 CSR、CSC、COO 以及他们的变种等表示形式。由于这几种表示硬件实现起来相对复杂，硬件开销较大。因此 FPGA 平台上的峰值性能相对较低，大部分都在 100GOPS 以下，另外 FPGA 在稀疏矩阵上的运算效率比 CPU 和 GPU 上高很多，但是在神经网络的稀疏度并不高的情况下，FPGA 的计算效率也只有 10% 左右 [116,117]。

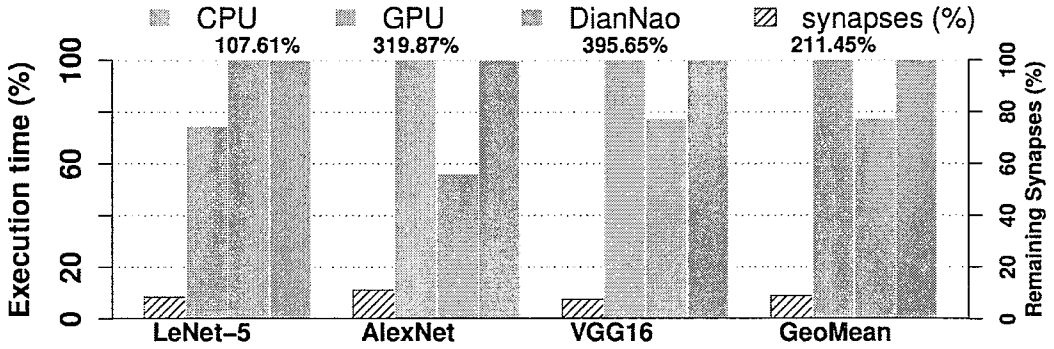


图 3.3　稀疏神经网络与稠密神经网络在 CPU、GPU 平台上的性能比较

稀疏神经网络在各种神经网络加速器，比如 DianNao，DaDianNao 上，更是毫无优势。由于现有的加速器不能直接处理稀疏的各种表示格式，稀疏神经网络只能被当做是普通的稠密网络进行处理。因此，现有的加速器在处理稀疏神经网络时，既不能减少计算量也不能减少访存，从而也得不到性能的提升。对于 DianNao 来说，一个直接的解决办法是在 DianNao 中增加稀疏编解码的模块，从而减少片上的访存带宽。然而，这种办法并不高效，主要原因如下面几个方面：（1）采用这种做法之后，并没有减少计算量，被去掉的权值的位置都被 0 所替代，这样就浪费了很多计算资源。（2）DianNao 的中心结构和 DaDianNao 的同步多计算单元的结构都不适合不规则的稀疏计算。

因此，通过上面的观察，我们充分利用稀疏神经网络的特点，克服不规则连接的困难，做出了一个高效的稀疏神经网络加速器。

3.3　稀疏神经网络加速器设计

在本节，我们将向大家介绍一下稀疏神经网络加速器的细节。图3.4是我们提出的加速器结构示意图。主要包括以下几个模块：控制处理器（control processor，简称 CP），

缓存控制单元 (buffer controler，简称 BC)，两个神经缓存单元 (neuron buffer，简称 NB)，一个直接存储访问单元 (direct memory access，简称 DMA) 和一个含有多个处理单元阵列（process elements，简称 PE）的计算单元（computation unit，简称 CU）组成。所有的 PE 通过胖树（fat-tree）互相连接以避免电路的绕线问题，CP 读入指令将译码出的不同的控制信号发送给不同的模块，缓存控制单元根据连接关系可以将选择出的神经元输出给不同的运算单元。缓存控制中有一个特殊的模块，叫做索引单元，该单元的作用是在处理稀疏神经网络连接关系时，为每个 PE 选择他们所需要的神经元。

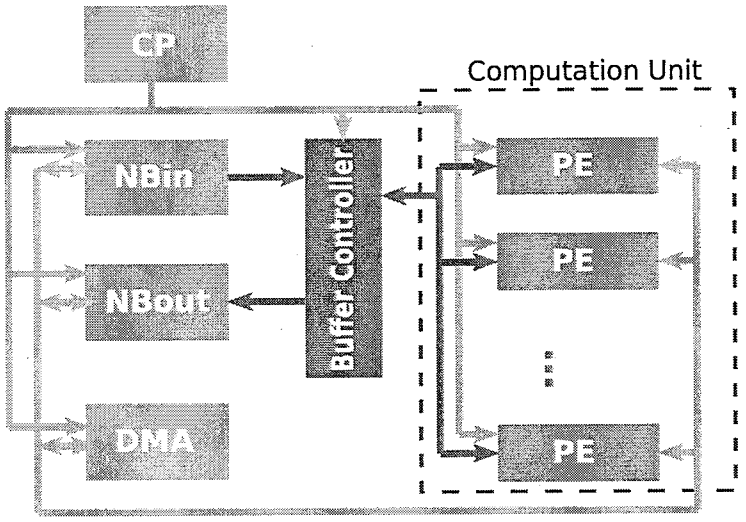


图 3.4 加速器结构示意图

在这个设计中，我们采用了 16 位定点的运算部件而不是传统的 32 位浮点运算部件。采用 16 位定点运算部件的原因是，在几乎不损失神经网络精度的前提下，16 位定点运算部件的硬件开销要比 32 位浮点的部件的运算开销要小很多。在 TSMC 65nm 的工艺下，16 位定点的运算部件的面积比 32 位浮点运算部件的面积小 6.1 倍，能耗低 7.33 倍 [29,118,119]。

3.3.1 PE 运算单元

运算单元根据神经网络的运算特点而设计，具有高效简洁的特点。运算单元能够高效处理矩阵向量的运算同时加入了对一些特殊操作的支持，如取最大值，激活等。图3.5是 PE 的结构示意图，每个 PE 包括一个权值缓存（synapse buffer，简称 SB）和一个神经网络功能单元（process element function unit，简称 PEFU）。PEFU 使用由 SB 中读到的权值和由 BC 发送过来的神经元进行运算，并将计算结果重新发送回 BC。

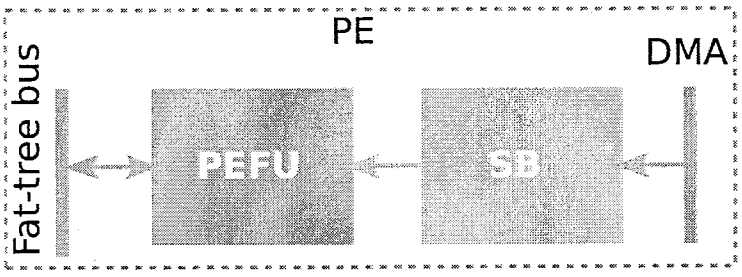


图 3.5 PE 结构示意图

3.3.1.1 PEFU 神经运算单元

PEFU 主要负责神经网络中一些乘加操作，一个 PEFU 中含有多个乘法器，我们把乘法器的数量定义为 Tm，同样的，也会拥有一个输入数量为 Tm 的加法树。图3.6显示了 PEFU 中的结构细节。这样，如果有 Tn 个 PE，那么便可以同时进行 Tn 个 Tm×Tm 的向量乘加操作。为了提高整个运算器的主频，我们把 PEFU 中的运算器分为了两步：第一步对位乘法，第二步进行加法树。以 Tm 个神经元作为输入，所有 Tn 个 PE 能够产生 Tn 个输出。

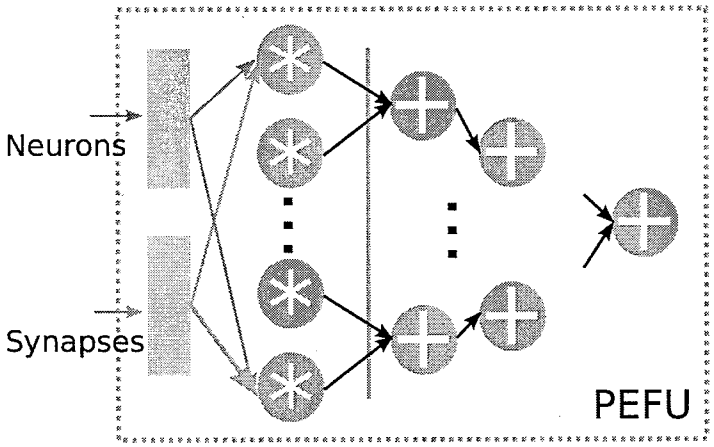


图 3.6 PEFU 流水级示意图

3.3.1.2 权值缓存

权值缓存被用来存放分布式的权值。在设计权值缓存时需要考虑两个问题，一个是权值缓存的大小，另一个是权值的存储方式。

为了减少片外的内存访问，之前的工作已经提出设计合适大小的缓存用来存储神经网络中所有的权值 [30,31]。然而在我们的加速器设计中，权值缓存比较小，存不下所有的权值。原因有两个方面，首先在稀疏的前提下，很多网络的权值也都处在 MB 的量级，AlexNet 的权值有 7MB，VGG 的权值高达 10MB，具体的权值数量如表3.1所示。其次，我们的加速器设计支持具有不同稀疏度的神经网路，此外还支持原始的稠密神经网络。

因此在计算不同的神经网络时，权值数量的变化巨大，设计太大的权值缓存单元会增加很多延迟，面积和能耗开销。实际上，权值缓存的最佳大小是能够掩盖访存延迟，在这个前提下，PENFU 则能够始终有足够的权值进行运算。在我们现在的设计中，我们在每个 PE 中，给 SB 分配了 2KB 的大小，因此总的权值缓存大小是 2×Tn KB。每个 SB 每一拍会为 PEFU 提供 Tm 个数据，也就是 SRAM 的位宽是 Tm×16 比特。

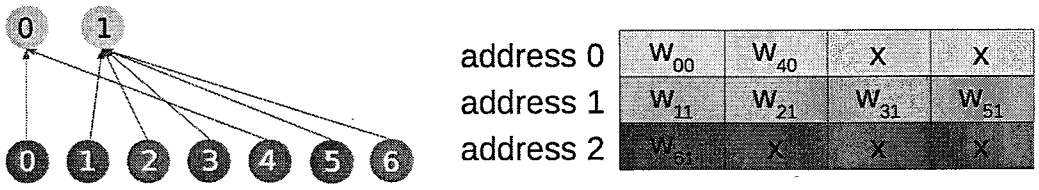


图 3.7 （a）一个全连接神经网络例子（b）权值在 SB 中的存储方式

为了阐明权值在 SB 中的存储方式，我们使用了一个稀疏的神经网络的例子，在这个神经网络中，输入有 7 个神经元，输出有 2 个神经元，如图3.7所示。我们假设这个网络在只有一个 PE 上的加速器上运算，其中 Tm 为 4。我们使用 $w_{i,j}$ 来表示输入神经元是 i，输出神经元是 j 之间的权值。输出神经元相连接的权值按照以每隔 Tm 个输出被紧凑的分配给不同的 PE。如图3.7b 所示，输出神经元的两个权值存储在地址 0，输出神经元 1 的 5 个权值存储在地址 1 和 2。从图3.7中可以看到每个地址中的权值不都是有效数字，这是为了数据对齐的需要。在计算时，输出神经元 0 只需要读一次权值，计算一拍便可得到结果，输出神经元 2 需要读两次权值，计算两拍得到结果。由于不同的神经元连接的权值数量不尽相同，导致计算速度也略有差异，因此 PE 之间的计算是异步的，SB 中 LOAD 权值数据也是异步进行的。

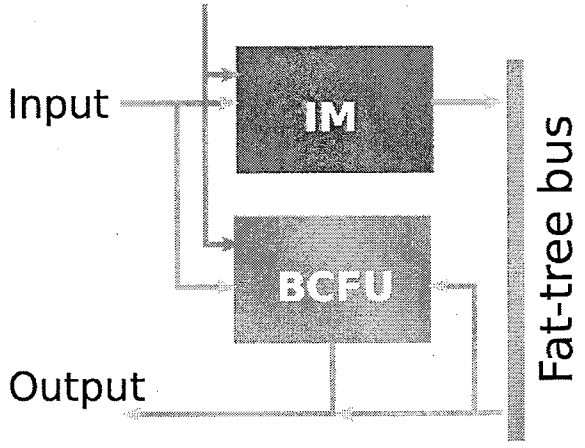


图 3.8 缓存控制单元结构

3.3.2 缓存控制

缓存控制模块的作用是从神经元缓存中读取神经元处理之后传送给不同的 PE，另外缓存控制模块也做少量的计算。图3.8表示了缓存控制模块的结构图，主要包括两个模块：一个根据连接关系处理数据的索引模块（indexing module，简称 IM）和一个运算单元（buffer control function unit，简称 BCFU）。在计算时，根据控制模块发出的控制信号，BC 首先从神经元缓存中读出计算所需要的神经元；然后，索引模块根据读入的连接关系，选择出每个 PE 计算所需要的神经元，并将他们通过胖树发送给每一个 PE，或者直接发送给 BCFU；最后，做完相应的运算之后，结果会写回神经元缓存。需要说明的是，如果在处理稠密神经网络时，数据会直接跳过 IM 模块，以减少延迟和能耗。

3.3.2.1 缓存控制计算单元

缓存控制计算单元主要对 PE 计算完的神经元进行一些后续的处理，如激活操作等，另外还用来存储索引模块选择出来的神经元。总共有 Tm 这种模块，因此可以同时存储 Tm 组神经元。图3.9是缓存控制计算单元的运算器示意图。BCFU 能够进行非线性运算，主要实现各种激活函数或者其他指数、对数、除法等运算。使得加速器对数学运算的支持更为全面，能够支持更多的神经网络算法。

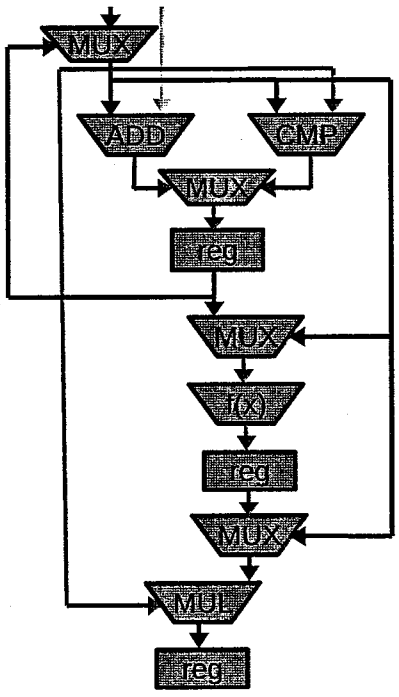


图 3.9 BCFU 运算结构示意图

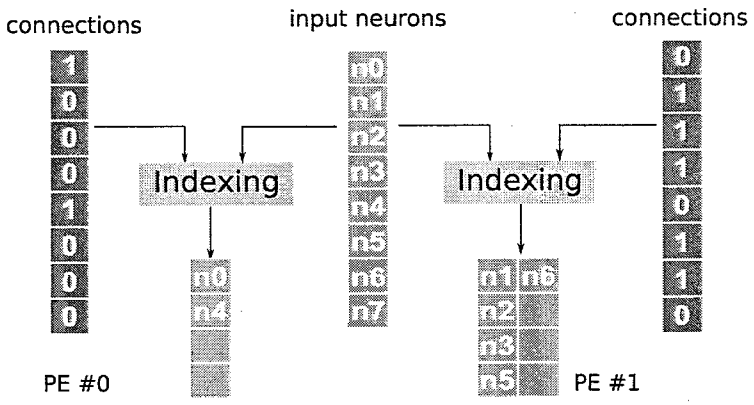


图 3.10 索引模块功能

3.3.2.2 索引模块

索引模块是稀疏神经网络加速器的核心部分。索引模块能够处理各种不同稀疏度的神经网络，如图3.10。我们没有选择采用将索引模块分布到不同的 PE 中的设计，而是将索引模块集中放在 BC，只将选择后的神经元传送给 PE。这是因为选择之后的神经元数据量更小，可以减小数据带宽，从而减少硬件的开销。对于 PE0 来说，只有 n_0 和 n_4 两个神经元从 8 个输入神经元选择出来。

在实现索引模块时，我们调研了两种普遍的索引方式：直接索引和步长索引。直接索引方式采用一个比特串来表示连接关系，每个连接用 1 比特数据表示有无连接，也就是 1 表示有连接，0 表示没有连接。步长索引方法记录了两个连接之间的距离，也就是每个在索引表里的数字表示到下一个连接的距离。

尽管已经存在了很多索引方法，比如压缩稀疏行方法（compressed sparse row，CSR），坐标列表方法（coordinate list，COO）和压缩稀疏列（compressed sparse column，CSC），但是直接索引方法和步长索引方法在硬件上更容易实现。例如，如果使用 CSR/CSC 的方法，需要使用两个数组去存储稀疏矩阵的索引，然而对于神经网络的权值，稀疏度通常都大于 5%，采用这种方式会有很大的存储开销，并且索引模块的设计会更加复杂。在本文中，我们只考虑直接索引和步长索引这两种方式，并会具体比较以上两种实现方法。

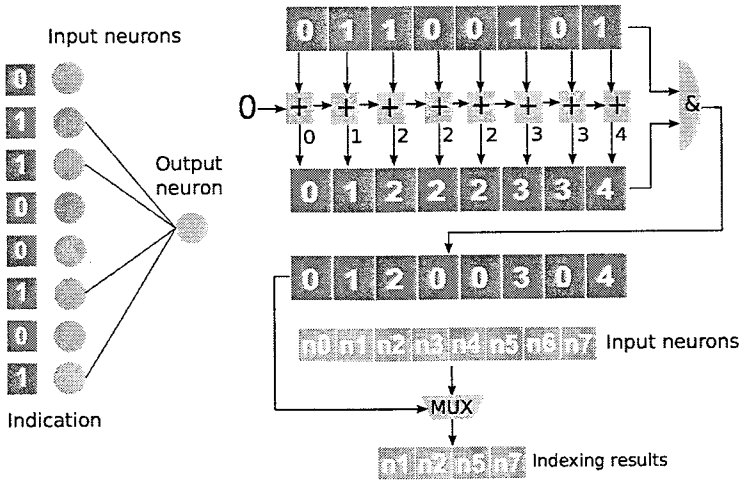


图 3.11　直接索引方法逻辑图

在直接索引方法中，神经元根据存在的连接关系计算出每个输入神经元的输出位置。映射模块通过处理连接关系的比特字符串选择出有效数据。稀疏神经网络连接及直接索引表示比特串如图3.11左图所示，我们也实现了直接索引的硬件，原理图如图3.11右图所示。直接索引的方法的处理过程如下：首先，将这个比特串逐位累加，得到一个累加的比特串，这个比特串表示相关连接的输出位置。然后，累加的比特串与原始的比特串做逻辑与操作之后，便可得到有效输入数据的输出位置。最后，使用选择器将有效数据输出。

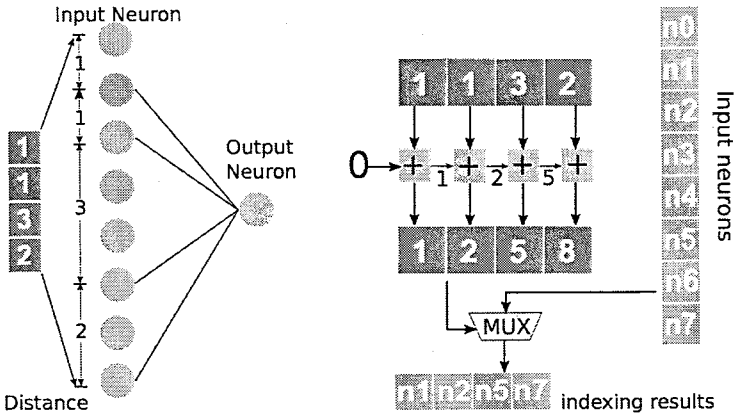


图 3.12　步长索引方法逻辑图

在步长索引的方法中，根据有连接的权值之间的距离选择出有效的神经元，在图3.12中是一个具体的例子。步长索引的具体方法如下：首先，连续地累加索引表中的值（也就是图中的 1132），这些累加后的值就是每个连接相对起始点的位置，然后使用这些位置便可得到对应的输入神经元。与直接索引方式不同的是，距离索引采用数字来表示距离，具体每个数字需要用多少位来表示取决于神经网络的稀疏程度。

我们在 RTL 层上实现了上述两种方法的具体电路并且比较了两者在面积和功耗上

的开销，两种方法电路的综合结果如图3.13。需要注意的是，索引的计算是在并行进行的。在每一拍选出 16 个数据，稀疏度的变化范围为 50% 到 3.12% 的前提下，映射模块输入的数组长度的变化范围从 32 到 512 范围内变化。从图中我们看出随着稀疏度的增长，硬件的开销也逐渐增大。另外，在所有的情况下，步长索引的方式总是好于直接索引。当数组长度是 256 时（这个数字是为我们硬件最终采用的参数），步长索引方式的硬件面积和功耗分别要比直接索引方式低 10% 和 40%。

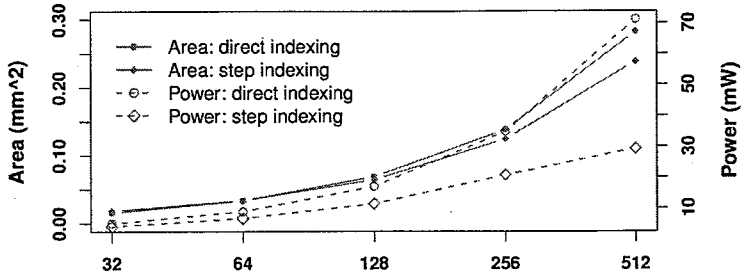


图 3.13 两种索引方式的比较

基于上面的观察，我们最终选择了步长索引的方式去实现我们的索引映射模块。在我们的设计中，IM 每一拍也是读取 Tm×Tm 数据，然后将选择出的数据作为输出送给每个 PE。

3.3.3 控制处理器

控制处理器是整个加速器的大脑，它具有高效又灵活的特点。控制处理器读取超长指令字指令作为输入，译码出对不同模块的控制信号，这些控制信号包括了数据组织，运算操作，内存访问等等。指令被存储在一个很小的缓存中。为了便于用户的使用，减轻用户的编程复杂度，我们提供了一个 C++ 的指令生成器，能够根据神经网络配置，快速并且简单地生成指令。

对各种常用的神经网络算法做了分析之后，我们精心设计了一个高效紧凑的控制处理器，该控制处理器是一个复杂的状态机，是一个两层的分层控制有限状态机（Hierarchicl Finite State Machine，简称 HFSM）。在 HFSM 中，第一层的各个状态与神经网络算法宏观的操作对应，比如卷积操作，内积操作等等。第二层状态则与每一个操作的细节相关，每个状态表示了每种操作的不同状态。

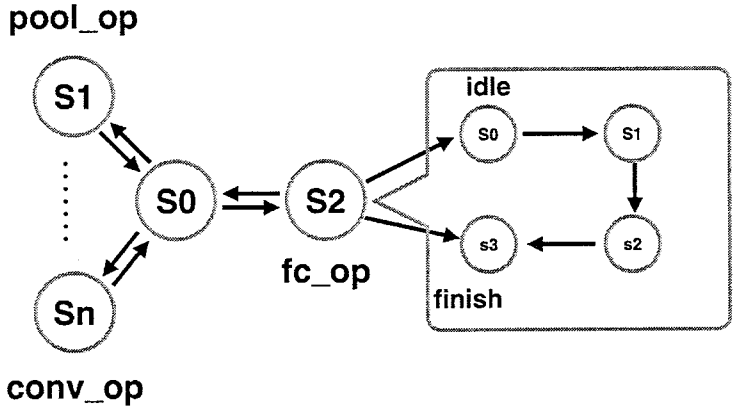


图 3.14　控制处理器的状态机

3.3.4　神经元缓存

神经元缓存包括输入神经元缓存（NBin）和输出神经元缓存（NBout）。输入神经元缓存存储了神经网络计算相关的输入数据，输出神经元缓存存储了神经网络计算相关的中间结果部分和或者最终结果。所有的神经元都被有序的存储，不需要考虑神经网络的连接关系。在我们现在的设计中，NBin 和 IM 之间的位宽设定为 Tm×Tm×16 比特。这样，每一拍最多有 Tm 组数据可以被选择出。

NBin 和 NBout 的大小直接决定了整体的性能和能耗。在分析了不同大小的 NBin 大小之后，我们发现 8KB 大小是一个最佳的容量，功耗和性能都在令人满意的范围内。因此，在我们的加速器实现中，我们设定输入输出神经元缓存的大小为 8KB。神经元缓存的大小比权值缓存 2KB 的大小大 4 倍，原因是，在计算稀疏神经网络时，加速器要通过索引选择出有效的数据进行计算，因此神经元缓存需要存储更多的数据，才能满足计算的需求。

明显地，8KB 大小的神经元也不能全部放下整个神经网络的神经元数据，因此需要采用一个合适的数据替换策略以减小片外存储访问的开销。仅仅当所有的输出神经元都已经不再依赖 NBin 中的数据或者 NBout 空间满了时，才会加载新的输入神经元或者存储出输出神经元。

为了提升数据的读写效率，每个神经元缓存都有两个端口：一个读端口，一个写端口。两个神经元缓存总共有 4 个端口，为了让四个端口可以同时访问同一组 SRAM，我们设计了一个采用交叉开关互联的电路，该电路的优势是既可以实现四个端口并行访问不同的 SRAM bank，又可以做到数据存储在同一组 SRAM 中。电路原理图如图3.15。

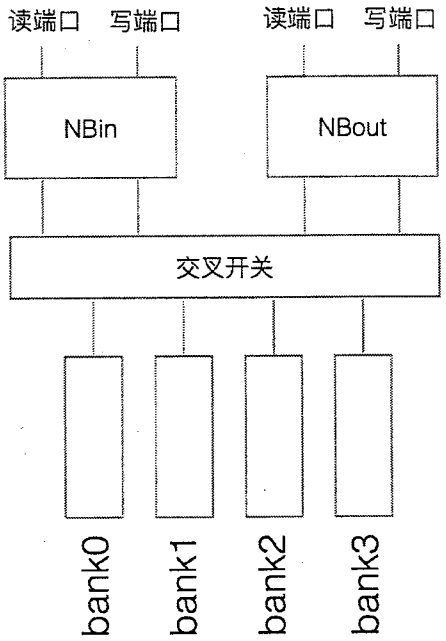


图 3.15 神经元缓存示意图

3.3.5 互联和通信

（1）互联

我们采用胖树的拓扑结构。胖树的特点是根部数据位宽比较宽，越靠近叶节点数据位宽越窄。通过胖树将 BC 与中 PE 相连，可以尽可能的提高 PE 的数据带宽需求。我们采用胖树的拓扑结构，主要有以下两个原因：首先，相比其他的非树的互联拓扑结构，使用胖树可以避免 BC 和 PE 之间由于距离原因引起的不平衡的延迟问题。其次，相比其他树型的互联拓扑结构，胖树能够为每个节点提供私有连接，可以避免数据总线过宽引起的绕线问题，因为每个 PE 在计算时，所需求的神经元数据是不同的。

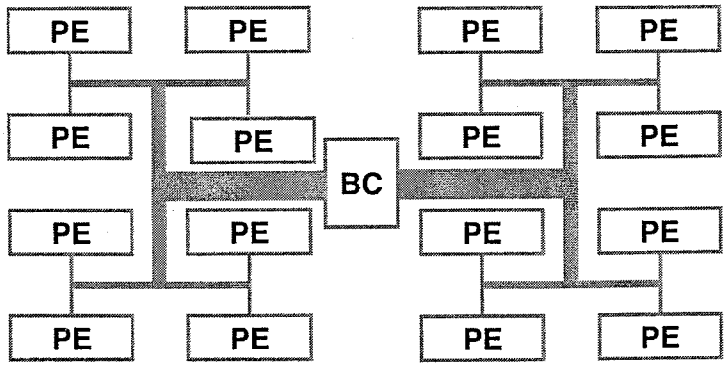


图 3.16 胖树互联结构

（2）通信

加速器片上缓存与片外的数据通信采用 DMA 方式。为了平衡不同 PE 的内存访问

避免出现访存拥堵，我们先把所有的权值分成若干个大块，然后内存访问端口在一个很短的时间段内只服务一个 PE，这样一个 PE 在那个时间段内只能加载一些块的数据。在这种情况下，在不同的时刻，每一个 PE 都有机会获得自己计算所需要的权值。每个 PE 的计算互相独立，这种异步的计算方式能够减少访存拥堵。

第四章　神经网络算法分析及在加速器上的映射方法

4.1　简介

在大数据时代，从众多多媒体信息（声音、文字、视频等）提取到有用的信息是一个相当困难的事情。在前几年的时间，多媒体数据的特征提取方法都是采用人工的方法，特征提取效果趋于稳定。然而，最近几年流行的神经网络算法将特征提取效果带到了一个新的高度。在很多领域，神经网络模型效果超越了人类的成绩，并且在很多领域也使得以前解决不了的问题变得可能。

神经网络算法作为科学计算中的一个大类，由于其卓越的性能变得越来越流行。神经网络算法的规模和复杂性也随着时间的增长而增长。在 Imagenet 挑战赛中，几乎所有的赢家都使用了深度很深的神经网络，这种大规模的神经网络在处理一张图片时就需要进行超过 10 亿次计算。这种高计算量和高复杂性的应用引起了很多学者对神经网络算法系统设计和实现的兴趣。

为了降低用户使用神经网络算法的开销，以及提升神经网络算法性能及可移植性等，很多学者提出不同的神经网络学习系统，如 Caffe，MXNet，tensorflow 等等。很多机器学习系统都将自己的领域专用语言嵌入到一个主语言中（比如 C++，python 等），在底层一般都调用一些线性运算加速库，如 BLAS 等。Caffe 作为一个被广泛使用的深度学习框架，采用了 C++ 语言编写，底层调用了 BLAS 和 cuBLAS 或者 cuDNN 等高性能线性代数子程序库。

基础线性代数子程序库（basic linear algebra subprograms，简称 BLAS）被广泛应用在各种通用和专用的科学计算程序中。在大多数的软件中，有很多都是因为基本线性运算阻碍了程序的更大规模或者更加精确的计算。更大规模或者更加精度的计算或者仿真更接近现实，因此人们迫不及待的想提高程序的运算速度。有两种方法可以提高程序的运算速度。一种方法是优化软件的性能，一方面优化代码，降低算法复杂度，减少无用的操作；另一方面，根据硬件结构和特点，编写高速的运算函数，使计算能力能够达到或者接近硬件的峰值性能。另一种方法是提升硬件的计算能力，制造各种计算能力更强的硬件，比如 GPU，FPGA，加速器等，从根本上解决运算能力不够的问题。

在接下来，本章将分析神经网络算法在 Caffe 上的代码，并描述神经网络算法在加速器上的映射方法，尽可能提高加速器在各个层上的运算性能。

4.2 在本章用到的一些函数

接下来在对神经网络算法进行解释说明时，会用到一些 BLAS 函数和其他非线性运算的函数。在此预先对要用到的函数做一下简单说明。BLAS 函数库总共有三个级别，级别 1 的函数为向量的运算的函数，级别 2 的函数为矩阵与向量的运算函数，级别 3 的函数为矩阵与矩阵运算之间的运算函数。

4.2.1 BLAS 级别 1 的函数

4.2.1.1 AXPY

AXPY 函数实现了向量的乘加操作，公式如下：

$$Y = \alpha X + Y \tag{4-1}$$

其中 X、Y 为向量，α 为一个常数。

4.2.2 BLAS 级别 2 的函数

4.2.2.1 GEMV

GEMV 函数实现了矩阵与向量的乘法操作，公式如下：

$$Y = \alpha AX + \beta Y \tag{4-2}$$

其中 A 为一个矩阵，X 和 Y 都是一个向量，α、β 都是常量。

4.2.3 BLAS 级别 3 的函数

4.2.3.1 GEMM

GEMM 函数实现了矩阵与矩阵的乘法操作，公式如下：

$$C = \alpha AB + \beta C \tag{4-3}$$

其中 A、B、C 为矩阵，α、β 为常量。

4.2.4 非线性函数

4.2.4.1 POW

POW 函数实现对于输入的幂运算，输入可以是标量或者向量。

4.2.4.2 VMUL

VMUL 实现了两个向量的对位乘法运算，公式如下：

$$y = a * b \tag{4-4}$$

其中 a、b、y 都是长度相同的向量，y 中的每个元素等于 a、b 中对应元素的乘积，即 $y_i = a_i * b_i$。

4.3 神经网络算法在 Caffe 上的实现

Caffe 提供了一套从训练，测试到优化等操作一整套完备的方案。Caffe 以模块化为指导思想，可扩展性强。神经网络算法在 Caffe 是以层为单位进行运算的，Caffe 提供了一套完全的神经网络操作，包括卷积层、全连接层、池化层、非线性层如 ReLU、归一化层、损失层等。这些层都是一些应用经常需要的层。Caffe 支持在 CPU 和 GPU 平台上的计算，两个平台上的代码都调用了各自的线性代数加速库，如 BLAS 和 cuBLAS。两个库在接口上的参数基本一致。

4.3.1 FC

全连接层 (fully connection layer，简称 FC) 的计算比卷积层的计算要简单很多。在全连接层中，每一个输出点都会使用全部的输入与对应的权值进行点积运算，因此全连接层的计算可以直接采用 GEMM 进行计算。输入神经元可以被看做是矩阵 A，其中 M=1，K=fi，权值可以被看做是 fi*fo 的矩阵 B，其中 K=fi，N=fo。这样计算就可以得到维度为 1*fo 的矩阵 C。

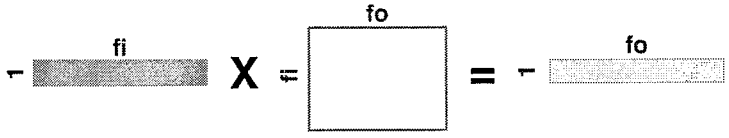


图 4.1 全连接层使用 GEMM 计算方法

在进行完权值相关的运算后，如果该层网络有偏执，还要进行加偏执的运算，Caffe 中加偏执的运算也使用 GEMM 函数完成。

4.3.2 CONV

在卷积层 (Convolution layer，简称 CONV) 的运算中，Caffe 使用矩阵乘法加速库函数 GEMM。GEMM 函数可以实现两个矩阵的乘法运算 C=A*B，其中假设矩阵 A 的行数为 M，列数为 K，矩阵 B 的行数为 K，列数为 N，由上面参数可得矩阵 C 的维度为 M*N。

卷积层是整个卷积神经网络的核心，运算量也占据了整个卷积神经网络 80% 以上。卷积层中有很多小的卷积核，这小小的卷积核在输入图像上滑动，便得到了特征输出。对于每个输出点来说，是由一个 3 维的神经元和一个 3 维的权值经过向量点积得到的。如果在代码中采用循环计算的话，计算效率会很低。因此，Caffe 中没有采用这种方法，而是采用了矩阵运算的加速库函数。因为数据依赖的关系，卷积层操作不能直接采用矩阵运算的加速库函数，因此在计算前要对输入数据做处理。

因为在卷积层中，权值是共享的，所以对于每一个输出特征图来说，权值可以被存放成一个列向量，每一个输出点所依赖的输入神经元被存放成一个行向量，需要注意的

是，不同输出特征图所依赖的输入神经元是相同的，所以只需要 OX*OY 行输入神经元。

图4.2是一个小的卷积层的例子，为了简化问题，这个卷积层没有 group 和 pad 的参数。这个卷积层的输入特征图像 fi=2，输入图像长宽，iy、ix 都是 3，输出特征图 fo=3，在卷积核 ky、kx 都为 2，滑动步长 stride 为 1 的情况下，输出特征图的 oy、ox 都为 2。为了能够实现调用 GEMM 计算卷积层，我们将卷积层的权值当做为矩阵 B，其中 K=fi*ky*kx，N=fo。我们把输入数据重新摆放成矩阵 A，其中 M=oy*ox。这样矩阵 A 的每一行与矩阵 B 的每一列相乘，得到的结果便是最终的输出点结果。采用加速库的矩阵乘法运算可以得到比采用循环计算的代码更高的性能。

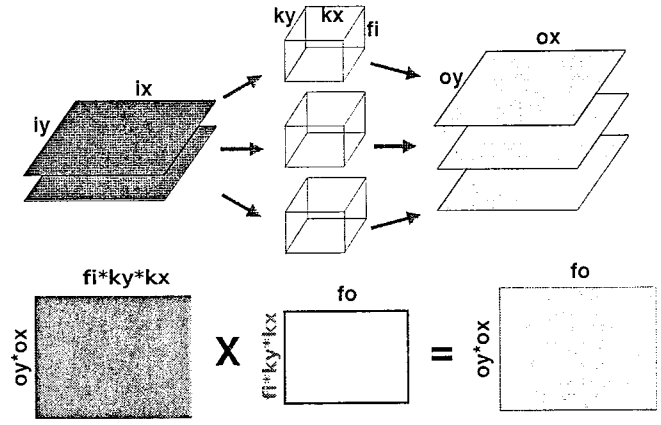


图 4.2　卷积层使用 GEMM 计算方法

卷积层也需要进行加偏执的操作，同全连接层一样也使用 GEMM 进行加偏执的运算。

虽然使用 GEMM 计算性能很高，但是不容忽视的是，在卷积计算之前，将卷积层的输入数据转换成输入矩阵 A 的过程中，会有很多访存操作。访存操作的时间影响了整体的计算效率，因此卷积层在 CPU，GPU 上的计算效率并不高。

4.3.3　POOL

常用的池化层 (pooling layer，简称 POOL) 算法有两种，一种是最大值池化，另一种是平均值池化。理论上在平均值池化层的计算中，也可以将输入数据进行重新存放，然后调用 GEMM 运算。然而由于池化层的计算量非常小，对数据重新摆放的代价又非常高，因此在 Caffe 中直接采用循环的代码进行池化层的计算。最大值池化的运算如下，平均值池化的运算相似。

Listing 4.1　池化层的运算

```
for(int f = 0; f < fi; ++f){
  for(int h = 0; h < oy; ++h){
    for(int w = 0; w < ox; ++w){
      int hstart = h*sy;
```

```
        int wstart = w*sx;
        int pool_index = h * ox + w;
        for(int kh = 0; kh < ky; ++kh){
          for(int kw = 0; kw < kx; ++kx){
            int index = kh * inx + kw;
            if(top_data[pool_index] < bottom_data[index]){
              top_data[pool_index] = bottom_data[index];
            }
          }
        }
      }
    }
}
```

4.3.4 POW

幂运算层（POW）是 Caffe 中的辅助运算层，一般用作辅助其他层内的计算。POW 层运算公式如下：

$$y = (shift + scale \times x)^{power} \tag{4-5}$$

其中 shift、scale 和 power 是输入的常量，x 为输入数据，y 为输出数据。

4.3.5 Split

Split 层也没有具体运算，作用是让多个输出数据共享同一个输入数据地址，主要是为了辅助实现拥有分支的神经网络。

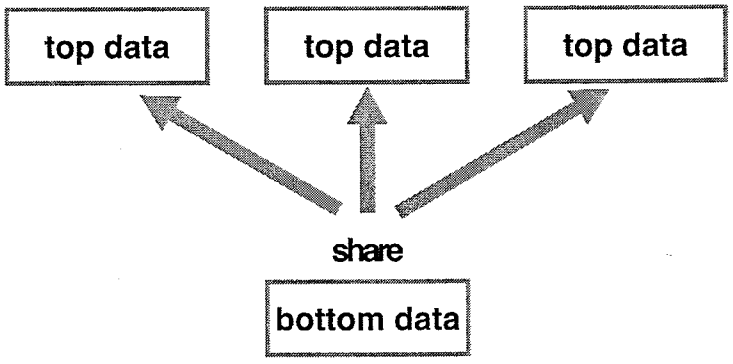


图 4.3 Split 层实现分支数据共享

4.3.6 Eltwise

Eltwise 的运算主要是将多个输入 blob 数据按照指定的运算合并成一个输出 blob。Eltwise 层中目前的操作主要有三个：累乘、累加、取最大值。

（1）累乘

使用 VMUL 函数首先对两个输入 blob 进行运算，作为输出 blob。如果输入 blob 的数量大于 2，则继续使用 VMUL 函数将剩下的每个输入 blob 与输出 blob 进行运算。

（2）累加

使用 AXPY 函数首先将两个输入 blob 进行运算，作为输出 blob。如果输入 blob 的数量大于 2，则继续使用 AXPY 函数将剩下的每个输入 blob 与输出 blob 进行运算。

（3）取最大值

和池化层操作类似，取最大值运算也在代码中使用循环进行计算。

4.3.7 NORM

在 Caffe 上实现了批归一化和局部响应归一化的算法。

4.3.7.1 BN

批归一化 (batch normalization layer，简称 BN) 的运算都很适合采用线性代数运算加速库。首先使用 GEMV 函数计算图像的均值，第二步使用 GEMM 函数将输入图像减掉均值，第三步使用 GEMV 函数计算输入图像方差，第四步使用 GEMM 函数计算得到输出结果。

4.3.7.2 LRN

局部响应归一化（local response normalization，简称 LRN）分为两种，实现方法也大有不同。

（1）交叉特征图像的 LRN 算法

这层的算法相对简单，首先先计算出所有神经元的平方，然后再使用 AXPY 函数累加不同特征图上的值。这样就得到了相邻几个特征图神经元的平方和。

（2）特征图像内的 LRN 算法

这一层的计算在 Caffe 中被分成了几个子层，所有子层的操作共同实现了本层的算法。首先使用平方层计算神经元的平方，其次使用平均值池化层计算局部区域的神经元平方和，然后使用幂操作层计算平方和的幂次运算，最后使用 Eltwise 层计算最终的输出。

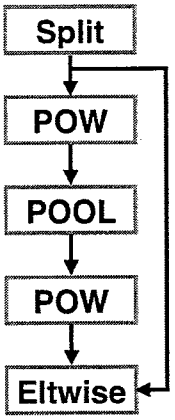


图 4.4 特征图像内的 LRN 计算流程

4.3.8 ACT

多种激活（activation，简称 ACT）层的操作虽然各不相同，但都是一输入神经元对一输出神经元的操作，因此在 Caffe 中都是用了简单的循环来进行运算。如下面代码所示，其中 ACT 表示不同的激活函数。

Listing 4.2 激活层运算

```
for(int f = 0; f < fi; ++f){
  top_data[f] = ACT(bottom_data[f]);
}
```

4.3.9 RNN

循环神经网络（recurrent neural network，简称 RNN）的计算比较复杂，在 Caffe 中被拆分成多个子层进行运算。如图所示，RNN 的计算主要是三个全连接层的计算，其他操作都是计算量和访存量较小的运算。

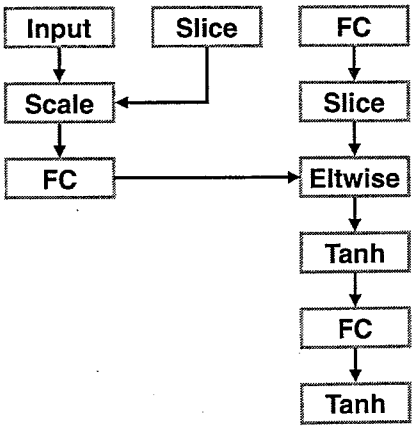


图 4.5 RNN 运算分解示意图

4.3.10 LSTM

LSTM 与 RNN 相似，也是由多个子层合作实现 LSTM 的运算。但是由于 LSTM 更加复杂，为了使 LSTM 实现更加简单，结构更清晰，Caffe 代码先实现了一个 LSTMUnit 层，去做一些 LSTM 算法中专有的操作，包括门的激活操作等。

4.3.10.1 LSTMUnit

LSTMUnit 实现了 LSTM 计算过程中的最后一步，在所有的全连接计算完成之后，所有门的激活以及最终的输出都在 LSTMUnit 中完成。如下的代码中，i、f、o 分别是输入门，忘记门，输出门的结果，C 和 H 两个数组分别是细胞和隐层的输出。

Listing 4.3 LSTMUnit 中的运算

```
for (int d = 0; d < hidden_dim_; ++d) {
    const Dtype i = sigmoid(X[d]);
    const Dtype f = (*cont == 0) ? 0 :
        (*cont * sigmoid(X[1 * hidden_dim_ + d]));
    const Dtype o = sigmoid(X[2 * hidden_dim_ + d]);
    const Dtype g = tanh(X[3 * hidden_dim_ + d]);
    const Dtype c_prev = C_prev[d];
    const Dtype c = f * c_prev + i * g;
    C[d] = c;
    const Dtype tanh_c = tanh(c);
    H[d] = o * tanh_c;
}
```

4.3.10.2 LSTM 运算分解

在使用 LSTMUnit 层的基础上，LSTM 的计算结构大大简化了，如下图是 LSTM 的运算分解图。

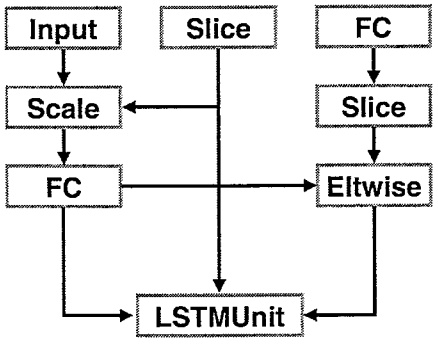


图 4.6 LSTM 运算分解示意图

4.4 加速器编程模型

4.4.1 基于库的编程方式

为了减少用户编程负担，我们为此加速器开发了一个基于库的编程模型。基本思想是为用户提供一系列的高级 C++ 库函数接口，每个函数对应着神经网络中的一个基本操作，比如向量内积、激活等操作，因此用户可以使用高级语言直接调用我们的加速器。下面代码是使用我们的库进行卷积操作的一个例子。除了对神经网络操作的支持，我们也支持一些低级的运算操作，像矩阵向量的乘加等。这样用户可以使用高级语言实现带有分支跳转等的高级运算。最终，原始的 C++ 代码会被我们内嵌的编译器编译成高效的二进制指令。

Listing 4.4 卷积层的库函数接口

```
ConvolutionForward(
    TensorDescriptor_t inputDesc,   //input descriptor
    void* input,                    //input data
    TensorDescriptor_t filterDesc,  //filter descriptor
    void* filter,                   //filter data
    TensorDescriptor_t outputDesc,  //output descriptor
    void* output);                  //output data
```

4.4.2 编程框架

为了使高性能编程框架具有更广阔的使用空间，我们将库嵌入到了使用广泛的深度学习框架 Caffe 中。因此算法研究人员可以直接使用 Caffe 的接口去调用我们的加速器，而不需要修改任何代码。图4.7是我们加速器的编程框架示意图。最开始，我们在训练阶段得到稀疏神经网络模型，通过分析稀疏神经网络模型得到稀疏表示的信息。然后上述的神经网络配置，神经网络模型，稀疏表示信息作为 Caffe 的输入。在 Caffe 中，我们的库函数调用加速器运算得到输出。对于普通的稠密神经网络来说，就像普通的 Caffe 一样，不需要任何修改。总之，我们的编程框架对于用户是透明，用户在使用我们的框架时，不需要花费高额的学习成本。

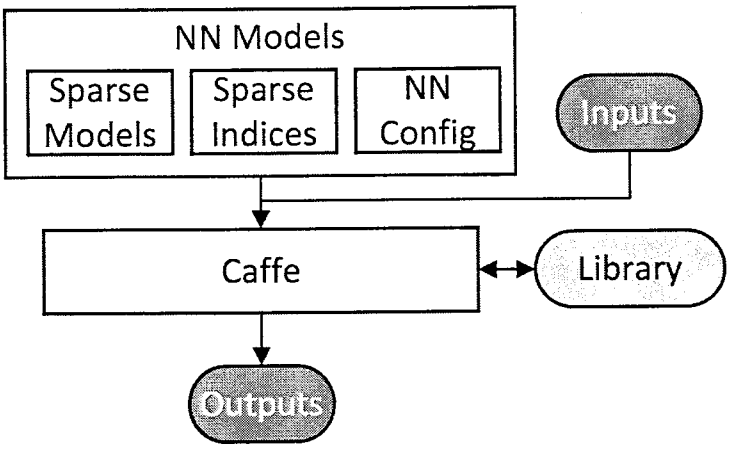


图 4.7　编程框架

4.5　神经网络算法在加速器上的映射

现在很多的神经网络计算框架中，都将神经网络的各种操作，努力转换成能够使用加速库的运算。然而，在对数据的转换过程中，也消耗了大量的时间。因此，在通用运算平台 CPU、GPU 上，神经网络运算的效率并不高。神经网络中的运算都是访存密集型的运算，在这一节我们详细分析第二章中提到的神经网络算法，挖掘每层神经网络的运算和访存特点，然后提出了各种神经网络算法在神经网络加速器上的映射方法。

4.5.1　FC

全连接层的算法很简单，对于每一个输出神经元来说，计算只是一个向量的内积。但是每一个输出神经元会复用相同的输入神经元。另外全连接层的权值没有复用，每个权值只会参加一次运算。因此，对于全连接层的运算，我们采用了复用输入输出神经元的运算方法。

4.5.1.1　按照 PE 分配输出

每一个输出的计算都使用相同的输入神经元，因此在每一拍计算中，所有的 PE 都使用相同的输入神经元，分别乘上自己局部存储的权值去计算不同的输出神经元。如图4.8所示 PE_0 计算第 0 个输出神经元 O_0，在 PE_0 的权值缓存中存储了第 0 个输出神经元计算所需要的权值，其中 w01 表示第 0 个输出神经元与第 1 个输入神经元之间的权值。在每一拍的计算中，输入神经元都被复用了 Tn 次。

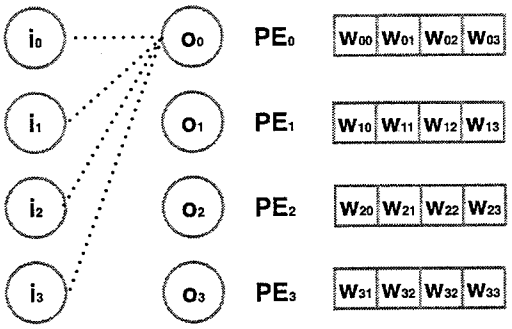


图 4.8 全连接层在加速器上的映射

4.5.1.2 分段处理

由于加速器的 SRAM 大小有限，有时候不能将一层神经网络的数据一次性全部放到片内，因此我们采用了分段计算的方法。我们将输入输出神经元划分成若干个段，每个段的输入输出大小按照 SRAM 的大小划分，使数据能够尽量填满片内的 SRAM。

在划分了数据段之后，数据就要不停的被反复换入换出，IO 量不可避免的增加了，段之间计算顺序也要仔细考虑。假设有一个神经网络，输入被划分为 3 段，输出被划分为 2 段，简单起见，我们认为每一段的大小都相同。不管怎样的计算方式都不会影响权值的 IO，因此在下面的讨论中不考虑权值数据。

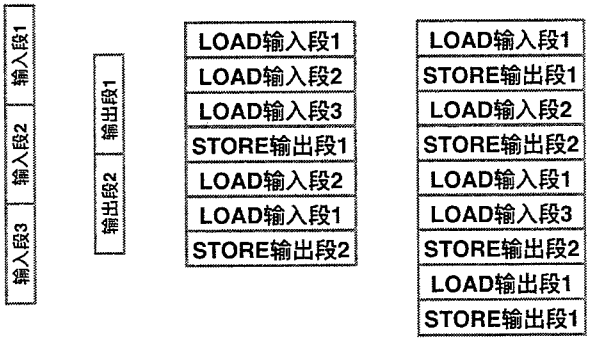


图 4.9 （a）全连接层分段处理（b）输出段优先的访存顺序（c）输入段优先的访存顺序

图4.9（a）是一个全连接神经网络的分段情况，图4.9（b）是在输出段优先计算的情况下，输入输出数据的访存行为，图4.9（c）是在输入段优先计算的情况下，输入输出数据的访存行为。在图4.9（b）中，首先分别 LOAD 三个输入段的数据计算第一个输出段，因为一个输出段的数据可以一直放在加速器中，因此输出段数据此时不需要访存，直到计算出第一个输出段的结果之后，才把第一个输出段的结果数据 STORE 出去。接着由于加速器内有第三个输入段的数据，便可以接着计算第二个输出段的数据，然后再依次 LOAD 第二个段和第一个段的输入数据，直到计算完第二个输出段，最后将第二个输出段的数据 STORE 出去。图4.9（c）中，首先 LOAD 第一个输入段的数据，分别计算不同的输出段，然后将输出段的部分和 STORE 出去，接着 LOAD 下几个输入段的数

据完成上面的循环。由图4.9可以看到，在全连接神经网络规模大到要分段的时候，采用优先输出段的计算方式可以减少整个运算过程中的访存量，是一种更高效的计算方式。

4.5.1.3 稀疏的全连接层

稀疏的全连接层的算法映射与稠密的映射方法完全相同。需要注意的是稀疏的全连接层存在连接关系的相关数据，这些连接关系数据因为也不存在复用，因此也和计算方式没关系，不存在减少访存的优化空间。

4.5.2 CONV

卷积层的运算比较复杂，但是对于每个输出点来说，都可以认为是一个向量内积运算。卷积层与全连接层的一个显著不同是卷积层的权值是共享的，因此参数数量大大减少。每一个特征图像上的所有输出都共享同一组权值，不同的特征图像上的输出所使用的权值不同。卷积运算过程中，卷积核在输入图像上滑动，所以输入图像对于同一个特征图的输出点也会被复用多次，一个输入点总的参加运算次数最多为：$\frac{fi*kx*ky}{sx*sy}$。由上可知，在卷积层中既存在输入数据的复用，又存在权值数据的复用。

4.5.2.1 按照 PE 分配输出特征图像

既考虑到权值的复用，又考虑到加速器的结构的情况下，将不同的输出特征图像，分配给不同 PE 进行运算是一个高效合理的做法。因为不同的 PE 拥有自己的局部权值缓存，按照输出特征图像去分配运算任务可以使得不同的 PE 存储不同的权值，充分的利用了 PE 中的存储空间。

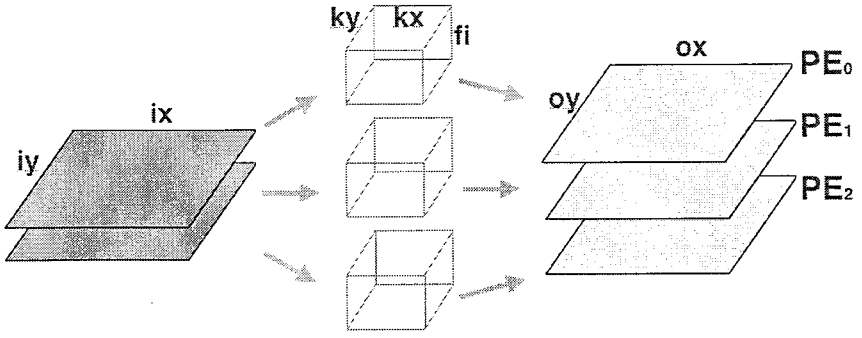


图 4.10 卷积层运算在加速器上的映射

4.5.2.2 输入数据

在卷积层的运算中，输入数据有两个维度的复用。首先是由于卷积核在一个特征图像上的滑动带来的数据复用，这种复用可以使得每个输入神经元最多可以被复用 $\frac{kx*ky}{sx*sy}$ 次。其次是由于多个输出特征图像带来的数据复用，每个输入神经元都会被复用 fo 次。当一个卷积层的所有输入数据都能在加速器中放的下时，是最理想的情况。反之则需要

反复输入神经元数据，待前面的几行输入数据计算完毕之后，加速器可以继续加载下面的 sy 行数据，继续进行下面的运算。

4.5.2.3 分段

限于 PE 的局部权值缓存大小有限，在遇到不能将整个卷积层的所有权值完全放的下的情况，也需要像全连接层那样对神经网络进行分段。同全连接层相似，我们在输入输出特征图像这个维度划分数据段。假设一个卷积层的卷积核大小为 1*1，输入图像长宽为 1*1，那么这个卷积层就等于是全连接层，对于段的划分方式也完全相同。

4.5.2.4 私有卷积核

在私有卷积核的卷积神经网络 [120] 中，每个输出神经元计算所需要的卷积核都与别的输出神经元不同，因此同全连接层相似，权值不存在复用。因此在加速器的运算时，只能考虑输入神经元数据的复用，这种算法对加速器的带宽需求很高。私有卷积核和共享卷积核是两个各有特点算法，都不能被互相取代。因此，加速器在设计时，考虑了两者的算法特点，并都做了支持。

4.5.3 POOL

池化层的计算量相对较小，输入数据的复用也比卷积层要少。不同于卷积层，池化层没有权值，并且输入输出特征图像相同。池化层的算法属于特征内的计算，输出神经元只和与自己对应的输入特征图像相关。同卷积层一样，我们也是将不同的输出特征图像分配给不同的 PE。在计算时，每个 PE 都得到相同的输入，每个 PE 分别取自己对应特征图像的输入数据，并做累加操作或者取最大值操作，两种操作分别对应平均值池化和最大值池化。输入数据按照卷积核的顺序传送给 PE，直到算出最终的输出，再开始下一个输出点的计算。对于输入数据的复用，只存在于特征图内，每个输入神经元可以被复用 $\frac{kx*ky}{sx*sy}$ 次。

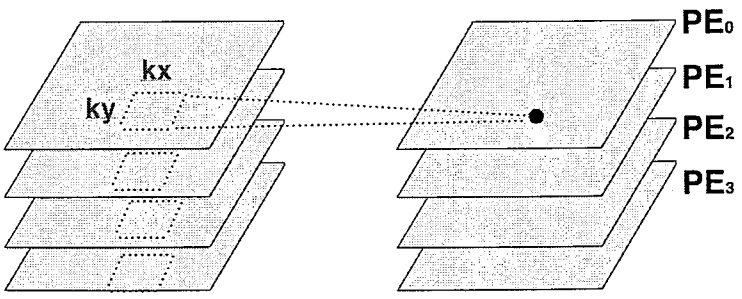


图 4.11 池化层运算在加速器上的映射

4.5.4 LRN

4.5.4.1 交叉特征图像的 LRN 算法

交叉特征图 LRN 层和池化层相反，输出神经元数据只和相同坐标点上的不同输入特征图像上的点相关。如下图4.12所示是 LRN 层一个坐标点处的输入神经元与输出神经元的连接关系，其中局部连接大小为 3。LRN 层也是按照 PE 划分不同的输出特征图像进行计算。LRN 的输入输出之间的连接不是全连接的，因此不利于在 PE 中进行内积计算，我们设计了一个巧妙的方法。对于同一输出特征图像的输出神经元，他们与输入神经元数据的连接关系都是固定的，我们将这个连接关系分别记录在不同的 PE 中。在计算时，每个 PE 根据联系关系先将输入神经元做一下过滤处理，凡是没有连接的输入神经元都被设为 0。做了这个处理之后，输出神经元就可以认为是和输入神经元是全连接的。计算输入神经元的平方和就相当于是计算两个向量的内积。再这一步之后的运算便是可以在 ALU 里完成的标量运算，算法的整个运算过程如图4.13。

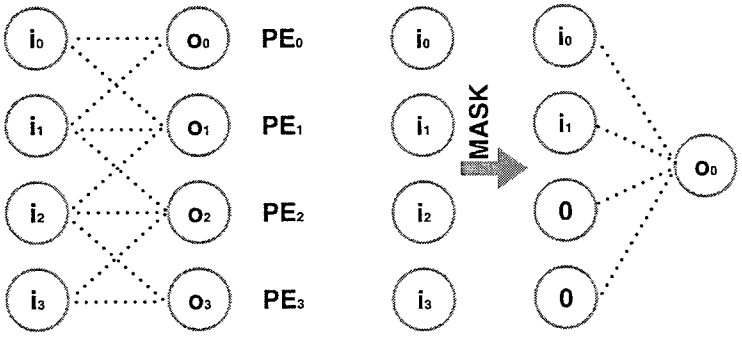


图 4.12 （a）LRN 层的连接关系及在加速器上的映射（b）LRN 层在 PE 中的部分计算

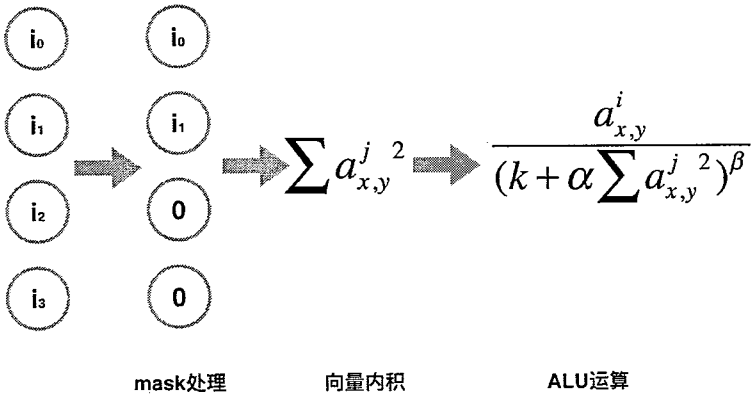
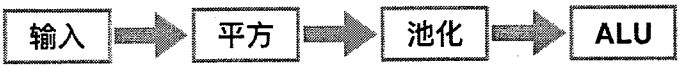


图 4.13 交叉特征图层在加速器上的运算过程

4.5.4.2 特征图内的 LRN 算法

特征图内的运算与交叉特征图的 LRN 算法实现完全不同。首先，我们先计算所有神经元的平方，然后累加局部区域的神经元采用池化操作，然后剩下的步骤也是采用

ALU 计算。



$$a_{x,y}^{j\ 2} \qquad \sum a_{x,y}^{j\ 2} \qquad \frac{a_{x,y}^{i}}{(k+\alpha\sum a_{p,q}^{i\ 2})^{\beta}}$$

图 4.14 交叉特征图层在加速器上的运算过程

4.5.5 ACT

由于激活层的运算复杂，并且输入与输出数据通常是一一对应的关系，不存在数据的复用，因此运算都在加速器的 ALU 中完成。

4.5.6 RNN

RNN 的运算主要是向量内积，在加速器上也是将 RNN 拆成几个全连接层计算，其他部分的运算简单且计算量小，都可以由 ALU 完成。

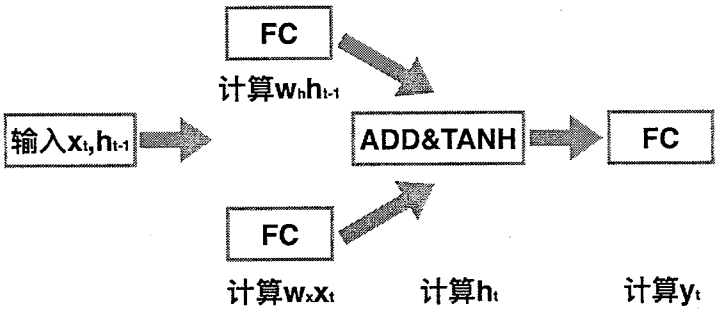


图 4.15 RNN 在加速器上的运算过程

4.5.7 LSTM

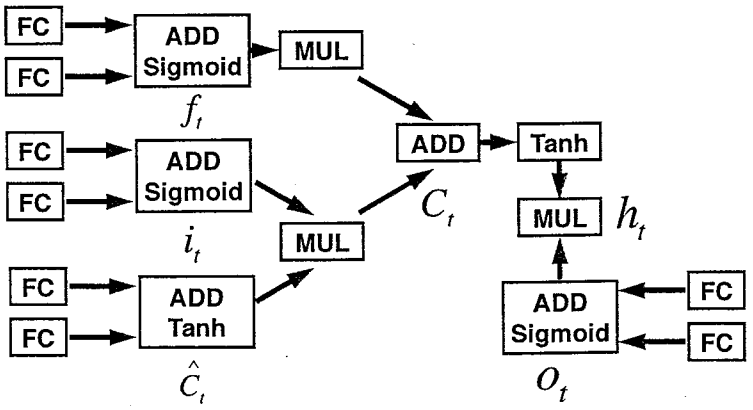


图 4.16 LSTM 在加速器上的运算过程

LSTM 的运算可以参照第二章的公式进行运算过程的拆解，主要过程如下图4.16。首先通过六个全连接层和加法，激活运算求出四个门的值 i_t、$\hat{C}_t$、f_t、o_t。然后通过乘法和加法运算计算出输出 C_t，最后通过 Tanh 运算和乘法求出 h_t。

第五章　神经网络加速器的验证

5.1　简介

在数字集成电路的制造过程中，从最初的需求细节到最后的产品，要经历几个不同阶段。每个不同的阶段都采用不同的形式来描述这个系统。寄存器传输级（registers transfer level，简称 RTL）仿真和验证是集成电路生产工程中的一个初始但又至关重要的一步。这一步确保设计是逻辑正确的并且没有太大的时序问题。RTL 的验证越早进行越好，最好在刚做完逻辑设计便开始验证工作。因为在验证中尽早发现问题，可以减少后面综合和布局布线的工作量。RTL 验证也可以尽量减少后续步骤出现的错误。

5.1.1　集成电路设计流程

下图5.1表示了集成电路从需求到最后产品的设计流程图。图中的设计流程从上到下描述了设计的几个主要步骤，当然实际的设计流程可能比图中要复杂的多，每一个步骤都可能需要很多次的迭代，直到设计的各个指标达到预先的设计需求，包括功能、面积、功耗、时序、开销等。设计需求是采用文字描述的一系列功能列表，最终的产品要实现上述的所有功能。功能设计是设计的第一步，通过对设计需求的分析，将需求拆分成可以实现的子模块或者子功能。这个步骤也可以认为是对需求的一个建模的过程，通常包括对软硬件的折中考虑和微结构的设计。

考虑到设计的复杂性，功能的开发通常采用分层的设计方法，使得开发者可以在一段时间内，可以只专注于一个小的子集。模块化的设计使得每个结构只能实现部分需求的功能，需要多个模块组合起来实现所有的需求。每个模块定义好他们的输入输出和通信协议，使得模块之间可以互相交互。这个阶段的设计通常采用高级函数描述，通常采用 C 语言或者类似的语言来描述模块的功能，能够实现设计在时钟精度级别的行为模拟。它可以被用来做性能评估的模拟器，也可以作为参考模型用于后面的验证。

功能设计阶段之后，硬件设计团队开始进行 RTL 的设计。在这个阶段，对集成电路的结构定义会添加更多的细节，所有模块的功能都使用硬件描述语言来设计。这个阶段不仅要实现目标功能，另外还要关注实现硬件的代价、主频、功耗等参数 [121]。

在上述步骤都完成之后，便进入了验证阶段。RTL 验证的目的是为了证明在没有制造错误的情况下，集成电路能够正常实现预想的功能。避免在进入花费高昂的流片阶段之后设计还存在设计错误。每一次发现功能错误之后，产品都要被重新设计以实现正确的功能。在 RTL 验证阶段，验证人员使用多种验证技术和测试工具去检查产品的各个功能是不是和预期功能相符。一旦发现有功能与预期不相符，RTL 设计将会被更新。

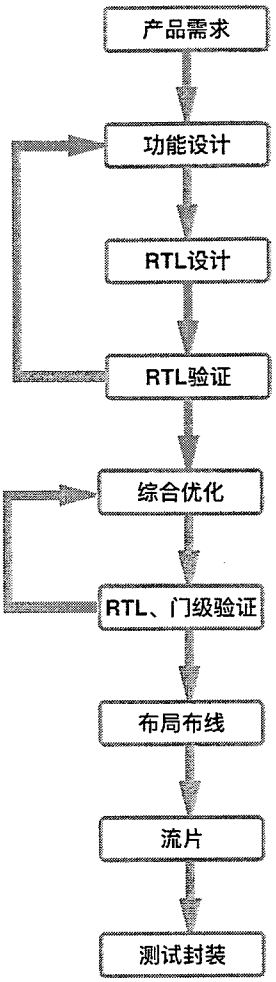


图 5.1　集成电路生产流程

在图5.1中，我们看到 RTL 验证的阶段似乎是独立于其他阶段。但实际上并不是这样，RTL 验证模块和其他设计工作几乎是同时开始，并且可能一直持续到流片。

在 RTL 设计完之后，便进入了 RTL 综合和优化阶段。这一步骤是在预设的一些电路限制参数之内，生成真实的逻辑电路。例如我们可以为设计指定最优的功耗生成结果或者是面积最优的生成结果。这个阶段生成的逻辑电路都由一些基本的逻辑电路元件组成，比如与门、或门、非门、异或门等等。在受限于电路的面积和功耗等参数的前提下，最优化网表成为越来越难的一个挑战。综合往往需要迭代好几次才能得到一个在各个方面令人满意的结果。在最优化的过程中，也可能会引起新的错误，从而需要进行额外的验证工作。

综合后的网表同样需要验证，以保证网表的行为和期望的行为或者 RTL 的行为一致。验证网表的行为和 RTL 的行为是否一致可以由 EDA 工作自动进行。

此时，我们已经可以进行物理电路的映射和布局布线了。这一步骤采用真实的库器件去映射电路中的逻辑门元件，得到真实尺寸的几何电路，这个电路会交给芯片流片厂

商。流片之后，经过测试和封装，最终的芯片便被生产出来。

5.1.2 数字电路模型

数字电路主要分为组合逻辑电路和时序逻辑电路两种。任意时刻，电路输出结果只和当前的电路输入信号有关，这种电路被称为组合逻辑电路。电路的输出结果不仅和当前的输入信号有关还和电路的当前状态或者说电路之前的输入有关，这种电路被称为时序逻辑电路。组合逻辑电路相对简单，只由一些逻辑运算部件和线组成。时序逻辑电路拥有记忆功能，因此需要触发器等部件来存储电路的状态。

在理想情况下，数字电路可以被看做是只含有一些逻辑门和触发器组成的电路，这些基本元件通过金属线互相连接实现复杂的逻辑。最常用的几种逻辑门元件是：与门、或门、非门和异或门。

5.1.3 状态机

有限状态机可以用来描述一个数字电路系统的功能。有限状态机可以由状态转换图来表示。状态转换图是一个有向图，每一个节点表示数字电路一个可能的状态，有向边表示数字电路可以由一个状态转换到另一个状态，状态的触发条件是输入信号，并且状态的转换是在一个时钟周期完成。状态转换图只描述了设计的功能转换图，而不涉及具体的实现细节。采用什么方式实现代码并不重要，只要状态机的跳转能够与状态转换图相符即可。

下图5.2是个 3 位的移位电路状态转换的示意图，初始状态是 001，初始节点的图形是两个圆圈，用以区分和其他状态的不同。每一个状态的三位数都被存放在寄存器中，用来记录当前所处的状态。所有的边都从当前状态指向了要跳转到的状态，在边上的文字即是状态跳转的触发条件，这些触发条件也是系统的输入信号。对于图中的状态机只能取到 3 个可能的状态：001、010、100。其他寄存器的取值：000、011、101、110、111都是到不了的状态。因为在正常的工作条件下，状态机不会跳到这些值。

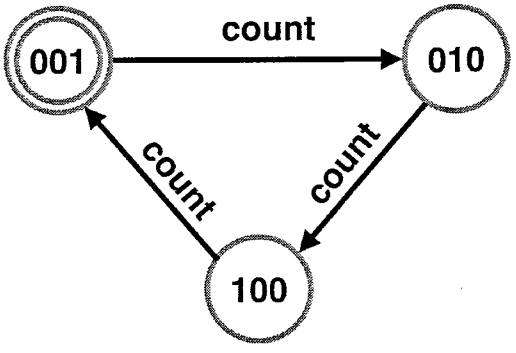


图 5.2 简单移位电路状态机

5.1.4 有限状态机的数学表达

有限状态机的另一种描述方法是采用数学表达式。在数学形式上，有限状态机是一个六元组，定义如下:

$$M = (\Sigma, \Gamma, S, s_0, \sigma, w) \tag{5-1}$$

其中的每个变量含义如下：

- σ 是输入向量。
- Γ 是输出向量。
- S 是一个非空的状态集合。
- s_0 是状态机的初始状态，属于 S 中的一个元素。
- σ 是状态转换函数：$\sigma : S \times \Sigma \rightarrow S$。

我们可以简单分析一下图5.2中状态机的各个数学参数，其中的各个参数如下：

- $\sigma = \{count\}$。
- $S = \{001, 010, 100\}$。
- $s_0 = 001$。
- σ 函数功能表如下：

count	001	010	100
0	001	010	100
1	010	100	001

5.2 RTL 验证

正如上文所说，RTL 验证是整个集成电路生产阶段最至关重要的一个步骤。由于流片的成本非常高，所以在流片之后才发现设计缺陷无疑会增加生产成本。同时，RTL 验证也是一个越来越具有挑战性的工作。对于一些通用或者功能相似的模块，验证人员可以复用或者简单修改之前的验证环境就可以了，比如说 AXI 接口的模块等。但是目前 RTL 大部分的验证工作都需要由设计和验证人员针对待测试模块做特别的验证环境和验证例子。另外，RTL 验证目前没有统一的标准或是方法。每个验证团队都根据自己的实践经验对待测模块进行验证，当面对一个全新的待测模块时，很多团队都会显得经验不足，不能对验证工作具有足够的信心。鉴于验证工作的困难程度，很多 IC 厂商都表示在一个产品的研发周期中，验证工作占了总工作的 70% 以上 [122]。

最常用的 RTL 验证方法是功能验证。对待测试模块进行仿真，输入是由脚本或者程序生成的期望的输入数据集，输出是待测试模块的仿真结果。将待测试模块的结果与期望的结果进行对比，确定待测试模块的行为与预期是否一致。期望的结果通常使用参考模型仿真得到或者使用软件计算得到。

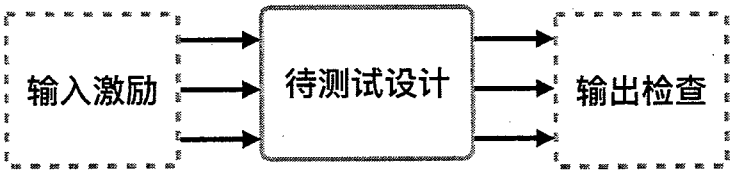


图 5.3 功能验证示意图

RTL 验证的层次通常分为两个层次：模块级和系统级。模块级验证是对每个模块分别去验证，确保每个模块的输出信号行为与预期一致。模块级验证要分别对每个模块搭建验证环境，编写验证激励和参考模型等。模块与模块之间的工作基本不能复用，对于每个模块的验证又要分别进行，因此模块级验证是一个很细致的工作，同时又是一个工作量很大，又很耗时的工作。幸运的是，现在有很多技术的进步减缓了验证人员的工作[123]，尤其是一些高层次的验证语言的出现如 SystemC[124]、SystemVerilog[125] 等面向对象的语言，以及一些验证方法学 UVM[126]、VMM[127] 等使得验证环境的搭建变得简单了很多。

系统级验证是将所有的模块搭建出整个系统之后，对这个系统进行验证。系统级验证的层次更高，不会去关心每个模块的输出结果，而只专注去验证整个系统的行为。通常，系统级验证需要在模块级验证的基础上进行，这样才能有足够的信心去忽略模块级的输出。系统级的仿真计算量很大，因此验证的周期也很长。输入激励往往是根据设计手册中的约束条件随机生成。在系统级验证中，RTL 的仿真和参考模型或者软件系统同时进行仿真，然后比对两者的计算结果以及两者存储空间中的数据是不是一一对应。

衡量验证工作的质量也是一个很难的问题，目前通常使用各种覆盖率的结果来衡量验证工作的充分性。覆盖率主要分为功能覆盖率和代码覆盖率，代码覆盖率又可细分为：行覆盖率，分支覆盖率，条件覆盖率，翻转覆盖率，状态机翻转率。功能覆盖率的质量完全取决于对于待测试模块功能点的编写，功能覆盖率的目标是尽量编写各种情况下的功能点，并尽量覆盖到这些功能点。行覆盖率统计在仿真时被激活过的 RTL 代码行，分支覆盖率统计 RTL 代码中所有的分支覆盖情况，条件覆盖率统计 RTL 代码中所有的条件的覆盖情况，状态机覆盖率统计 RTL 代码中的所有的状态机的状态覆盖情况。代码覆盖的统计可以由 EDA 工具自动完成，验证人员需要做的是设计各种情况的输入激励，使 RTL 代码覆盖率能够完成上述几个方面的覆盖。

由于数字集成电路的设计越来越复杂，对验证工作的挑战也越来越大。复杂的设计不仅使仿真速度减慢，更是增加了统计覆盖率的难度。验证人员需要更仔细的设计输入激励，使尽量少的输入激励覆盖到尽量多的情况。

5.2.1 UVM 验证方法学

随着集成电路规模逐年增长，验证工作在整个产品周期中所占的比例已经超过了70%。因此提高集成电路的验证效率变得至关重要，一个高效、灵活、可扩展性强的验证平台成为验证工作的关键因素。验证方法学便是一套可以帮助验证以及搭建验证环境的方法学。

通用验证方法学（universal verification methodology，简称 UVM）是一个用于验证集成电路的标准化的方法学。UVM 最初起源于开放验证方法学 [128]（open verification methodology，简称 OVM）。OVM 又起源于早期的 e 语言重用方法学（e reuse methoflology，简称 eRM）。eRM 是 Verisity Design 公司在 2001 年，采用 e 验证语言编写的一个验证方法学。

5.2.2 覆盖率

覆盖率是对验证工作的总结，作为一个验证结果，能够在一定程度上说明验证工作的充分性。下面将分别介绍各个覆盖率的测量方法以及意义。

5.2.2.1 功能覆盖率

功能覆盖率需要开发人员根据设计手册手动编写功能覆盖点 [129]。功能覆盖点的编写质量，直接决定了功能覆盖验证工作的效果。由于此工作的重要性，在实际工作中，往往需要开发人员和验证人员协作一起编写功能覆盖点。功能覆盖点包括了很多种情况，主要概括为以下几种：重要信号的某些取值，几个信号的取值组合，信号的取值跳变，即一个信号从一个值跳变到另一个值的情况，多个状态机的取值组合，地址非对齐的情况，倍压的情况等等。

5.2.2.2 行覆盖率

行覆盖率是在众多覆盖率中，最容易达到要求的一个指标。故名思意，代码在仿真中被激活的行即认为被覆盖到。

5.2.2.3 分支覆盖率

分支覆盖率是指在分支代码中，所有的分支被覆盖的情况统计。在出现分支的情况下，往往会有判断条件，因此分支覆盖率常和条件覆盖率结伴出现。但分支覆盖率又和条件覆盖率有着本质不同。所谓分支即代码的语句有不同的分支，比如说 switch 语句的不同 case 子语句即为分支，if 语句和与其配对的 else 语句也是代码中不同的分支。

5.2.2.4 条件覆盖率

代码覆盖率是指在有判断的代码中，所有的子条件被覆盖到的情况统计。比如，在一个 if 语句中，if 语句的判断条件是几个不同信号的逻辑运算，那么这几个不同的信号则都是 if 语句的一个条件，这些信号的所有取值构成了该 if 语句的条件覆盖率。

5.2.2.5 翻转覆盖率

翻转覆盖率表示信号在仿真过程中有没有经历过翻转，即信号的值有没有从 0 跳变到 1 或者从 1 跳变到 0。

5.2.2.6 状态机覆盖率

状态机覆盖率表示在有状态机的代码中，状态机有没有经历过它所能取到所有可能状态。

5.3 待验证的神经网络加速器的功能

图5.4是神经网络加速器的模型，加速器与外部的通信通过两组 AXI 信号进行，CP 是加速器的控制处理器，通过解析超长指令字向加速器各个模块发送控制指令，完成各种运算功能。在加速器中有很多 PE 核，PE 负责各种矩阵和向量运算。另外，还有一个 ALU，负责各种标量运算。因此，待验证的功能如下：

- AXI master 的 IO 功能。
- AXI slave 的 IO 功能。
- PE 的运算功能。
- ALU 的运算功能。
- CP 的控制功能。
- 整个加速器的功能。
- 神经网络的运算。

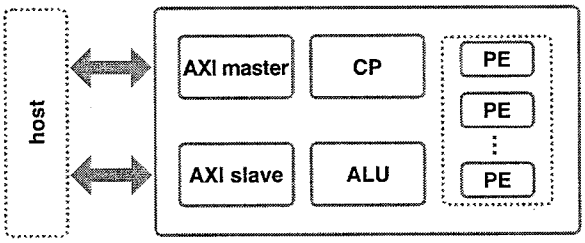


图 5.4 神经网络加速器模型

5.4 神经网络加速器验证方法

神经网络加速器作为一个新兴硬件，刚出现在学术界不久。验证人员面对神经网络加速器的验证都表现得经验不足。本文针对神经网络加速器的特点，结合 UVM 验证方法学，建立了一个针对神经网络加速器的验证平台。另外本文开发了一个和加速器硬件结构无关的 C++ 参考模型。该参考模型支持各种低精度数据结构的运算，并能支持各种神经网络算法。在验证不同的加速器时，参考模型只需要修改运算顺序与硬件一致，便可以投入使用。

5.4.1 验证层次

在芯片的设计阶段，设计人员从最初的需求开始将芯片划分为几个子系统，进一步又会将子系统划分成多个不同的模块。在验证阶段，则会自底向上，逐层次采用不同的策略对整个芯片进行验证。对于神经网络加速器，我们采用三个层次的验证：模块级，子系统级，芯片级。

5.4.1.1 模块级

在神经网络加速器芯片中，有很多功能不同的模块，对于这些模块，我们采用 UVM 验证框架一一搭建了模块级的仿真验证环境。具体地，我们使用黑盒和白盒混合的验证方法。在白盒验证中，采用监视器和断言检查设计中的重要逻辑，确保模块内部的重要逻辑没有异常。在黑盒验证中，采用监视器检查模块的输出信号，并使用 scoreboard 将输出结果与 reference model 的结果进行对比，确保模块的外部输出没有异常。

5.4.1.2 子系统级

在神经网络加速器芯片中，对于很多模块共同协作完成一个功能的多个模块被我们人工划成几个不同的子系统，子系统之间的模块可能存在交叠。比如负责向量运算的多个 PE 和它的控制模块神经运算控制单元共同组成了一个可以完成简单向量运算的一个子系统。在子系统级的验证中，每个模块都被当做黑盒来对待，我们的验证重点是，模块之间的信号以及子系统的输出信号。

5.4.1.3 系统级

我们对整个神经网络加速器做了系统级的验证，采用黑盒验证的方法，采用监视器检查子系统之间的通信有无异常。在仿真结束时，检查芯片的输出结果与参考模型的输出结果是否一致。另外还要对比芯片中所有的存储区域的数据与参考模型中的存储区域的数据，防止芯片中有异常的读写操作。

5.4.2 验证流程

在验证流程上，除了采用了上述分层的几种验证方法，还进行了在 FPGA 上的验证。验证流程图如图5.5所示。

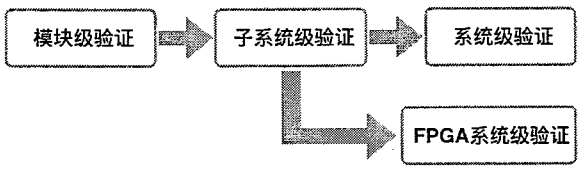


图 5.5 验证流程图

通过以上不同层次和不同的验证方法，可以确保验证的完备性，具体每个验证层次和验证步骤侧重点各不相同，他们优势互补，使验证结果更加可靠可信。

表 5.1 不同验证步骤的特点

验证步骤	验证侧重点	验证方法	验证透明度	激励
模块级	模块内部功能	仿真验证	白盒	随机激励
子系统级	模块间交互	仿真验证	白盒，黑盒	随机激励
系统级	模块间交互	仿真验证	黑盒	随机激励，直接激励
FPGA 验证	系统交互，参数遍历	仿真验证	黑盒	随机激励，直接激励

5.5 验证环境方案及架构

在不同的验证阶段，我们采用了不同的验证环境架构，具体的有以下两种：在模块级验证和子系统级验证中，我们采用了在线对比结果的验证环境，在系统级和 FPGA 的验证环境中，我们还采用了离线对比结果的验证环境。

5.5.1 模块级验证方案

对于加速器内部的各个模块，我们搭建了在线比较结果的验证环境。验证环境如图5.6所示。sequencer 负责控制随机激励的生成，driver 将随机激励转换成模块的具体信号，reference model 是一个使用 SystemVerilog 编写的与设计文档相一致的模块，assertion 用来检测 DUT 内部的逻辑，scoreboard 比较 reference model 和 DUT 的结果，并记录。

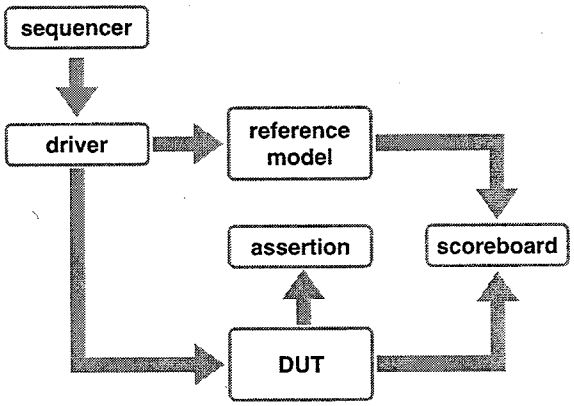


图 5.6 模块验证架构图

在模块级的验证中，为了使覆盖率达到预期，需要对 sequencer 反复进行修改，使得 sequencer 能生成各种不同的随机激励。模块级验证的覆盖率是相对容易达到目标的，因此模块级验证通常需要达到行，分支覆盖率和功能覆盖率都要达到 100%。模块级验证是其他阶段验证的基础，模块级验证工作做的越充分，对后来的验证工作帮助越大。

5.5.2 子系统级验证方案

为了验证方便，我们把设计中不同的模块按照完成的功能不同分成了几个小的子系统：AXI master 子系统，AXI slave 子系统，向量运算子系统，ALU 运算子系统。这几个子系统分别实现了不同的功能，在验证环境上也有差异。

5.5.2.1 AXI master 子系统的验证

在神经网络加速器芯片中，有一个 DMA 模块，DMA 模块作为 master 使用 AXI 3.0 协议与外部通信。为此我们将包括 DMA 相关的部分作为 AXI master 子系统进行验证。AXI master 子系统的验证架构图如下图5.7所示，主要包括以下几个模块：sequencer 负责控制随机激励的生成，具体地讲主要是配置 DMA，给 DMA 发送一些 IO 请求命令。reference model 是一个根据需求采用 SystemVerilog 语言编写的参考模型，参考模型和 AXI master 分别接受来自于 sequencer 的激励，分别和自己所连接的 AXI slave 互动，完成各种 IO 操作。

在验证的过程中，scoreboard 会对 AXI 总线上读写数据进行对比，AXI master 的有效数据应该与参考模型上的有效数据完全相同。另外，在所有事务的仿真都结束之后，scoreboard 还会对 AXI master 端的 SRAM 数据与 reference model 中的 SRAM 做对比，同时也会对比两个 AXI slave 中的数据。理论上，两者所有的存储空间数据都应该相同，否则就是发现了错误的情况。

外部 sequencer 去配置 DMA，并启动 DMA 进行读写，从而使 AXI master 得到激励。DMA 发出的只是一些数据的读写信号，但是这些读写信号仍需要精巧的设计才能使覆盖率完全，结果更可信。对于 DMA 的激励，我们考虑了以下因素：

- 片外地址的对齐情况。
- DMA 一次工作的数据总长度。
- 片内地址的对齐情况。
- 源地址和目的地址都为片内。
- 源地址和目的地址都为片内，且源地址与目的地址相同。
- 源地址和目的地址都为片内，且源地址数据空间与目的地址空间有交叠。

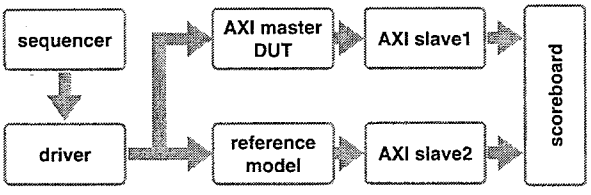


图 5.7 AXI master 子系统验证架构图

5.5.2.2 AXI slave 子系统的验证

除了 DMA 模块，神经网络加速器芯片还有一组作为 slave 的 AXI 接口，用于外部去访问神经网络加速器的寄存器和 SRAM。因此我们将包括 AXI slave 的模块和与其相关的模块作为另一个子系统进行验证。AXI slave 子系统的验证架构图如下图5.8所示，因为 AXI slave 是一个被动的模块，所以 AXI slave 的验证比 AXI master 的验证要简单的多。在 AXI slave 的系统中主要包括以下几个模块：sequencer 负责控制随机激励的生成，主要是生成各种 AXI slave 能够支持的 AXI 读写请求。AXI slave SRAM 作为一个参考模型，同 DUT 一同响应 sequencer 发过来的请求。scoreboard 会对两者 AXI 总线上的有效数据进行对比，另外，在所有事务的仿真结束之后，scoreboard 也是要会对比 DUT 和 AXI slave SRAM 中的数据存储空间，两者的存储空间内的数据应该完全相同。

对于 AXI slave 的激励的约束取决于 AXI slave 支持的 AXI 请求范围，如 xlen，xsize 参数等。在对 AXI slave 做了必要的随机约束情况下，对于 AXI slave 的激励随机生成，我们考虑了一下因素：

- 地址对齐的情况。
- xlen 和 xsize 等 AXI 参数的随机。
- 发出请求的频率。
- 接收数据的频率。

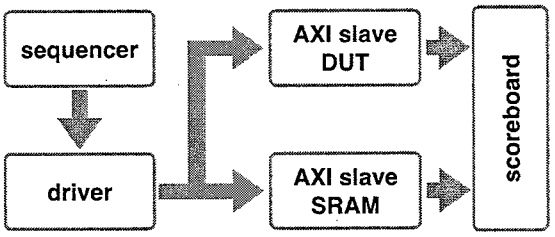


图 5.8 AXI slave 子系统验证架构图

5.5.2.3 向量运算子系统的验证

向量运算单元是神经网络加速器的核心运算模块，在整个神经网络加速器内存在数量众多的向量运算单元。为了便于验证，我们将多组向量运算单元和它们的控制单元组成一个向量运算子系统来作为一个验证的整体，这样便能验证到加速器的核心运算功能是否正确。

该子系统的验证架构图和模块的验证架构图相似，它的特别之处在于只采用黑盒验证，验证环境的随机激励也根据自己系统的特点进行了设计。向量运算子系统的主要验证目标是验证该系统对于向量运算的支持情况。向量子系统验证的项目如下：

- 向量的加，减，内积，外积操作。
- 矩阵与向量的乘操作。

- 矩阵与矩阵的加，减，乘操作。

5.5.2.4 ALU 子系统的验证

ALU 是神经网络加速器的辅助运算模块，负责神经网络中的一些标量计算，如加减乘除，取最大值，非线性函数等。ALU 子系统的验证架构也和模块验证架构图类似。ALU 子系统验证的项目如下:

- 标量的四则运算。
- 激活函数运算。
- 取最大值运算。

5.5.3 系统级验证方案

系统级验证是验证工作的最后一道屏障，重要性不容忽视。因此，在系统级验证中，我们搭建了两个不同的验证环境：一个是在线比较的验证环境，另一个是离线比较的验证环境。在线比较的环境用来进行指令级的随机验证，离线比较的验证环境用来进行神经网络运算的验证。

5.5.3.1 在线比较的系统级验证

在线比较的验证环境结构也与模块级验证结构相似。在线比较的验证环境的激励是一些随机的指令序列，用来验证系统执行指令的结果是否正确。

5.5.3.2 离线比较的系统级验证

离线比较的验证环境主要分为四个部分：激励生成、硬件仿真、软件计算、结果比较，如图5.11所示。随机激励模块由 C++ 语言编写，首先会随机生成不同的神经网络配置，随后根据神经网络配置由 code generator 生成加速器的指令和数据，这些指令和数据传递给硬件，作为硬件的输入进行仿真。同时，软件参考模型也根据神经网络配置和数据进行神经网络的计算。最后，结果比较模块对两者的神经网络计算结果进行比较，另外除了比较结果也要比较两者存储空间中的数据是否完全一致。

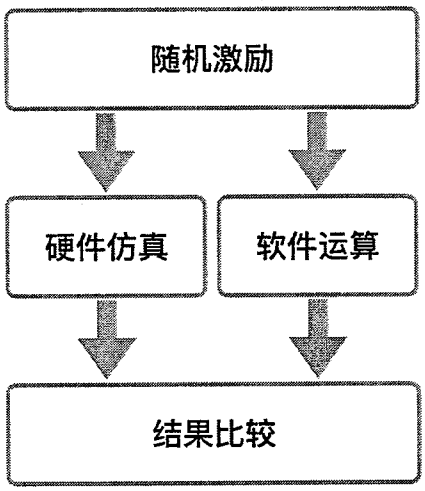


图 5.9 离线比较的系统级验证

离线比较的系统级验证主要为了验证神经网络加速器芯片的各个功能，比如实现卷积运算，激活运算等。另外，离线比较的系统级验证的目标是提升待测试模块的各种覆盖率，离线比较的系统级验证工作的充分性也使用覆盖率来衡量。

5.5.4 FPGA 验证方案

FPGA 验证方案和离线比较的系统验证类似，但 FPGA 验证环境比仿真环境更加真实，FPGA 验证环境如下图5.10所示，在激励和 FPGA 之间多了一层驱动层，FPGA 作为系统的一个设备由驱动负责和操作系统交互。

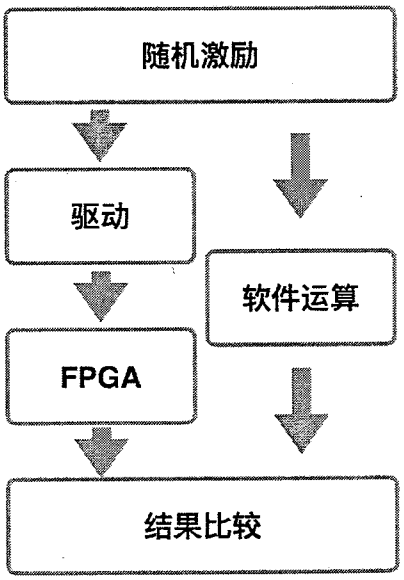


图 5.10 FPGA 验证环境

由于 FPGA 的运算速度远大于系统仿真，FPGA 可以从事更大量的验证工作。FPGA 验证的重点主要包括以下三个方面：

- 大规模的随机神经网络运算。
- 常用的神经网络运算。
- 神经网络的参数遍历：如卷积核参数，步长参数，特征图参数等。

5.6 参考模型

神经网络算法的规模和复杂性正在逐步增长。神经网络算法的复杂和巨大的计算量给神经网络编程框架的设计和实现带来了巨大的挑战。神经网络算法库也层出不穷，支持的功能也越来越多。看起来这些神经网络编程框架可以直接用来作为验证工作的参考模型。实际上并不是这样，有两个主要的原因使得我们不得不自己开发一个神经网络编程框架：（1）在神经网络加速器有很多自己定义的低精度数据模型，现有的神经网络编程框架并不支持这些低精度数据的运算。（2）在验证的工作的，最后比较参考模型和加速器的运算结果要完全一样，然而数据的运算顺序不同会导致运算结果会有偏差，因此我们的神经网络运算框架必须要满足在神经网络运算的各个过程中，运算顺序都要始终保持和我们神经网络加速器运算顺序一致。

为了验证加速器在神经网络上的运算结果，我们采用 C++ 语言写了一个参考模型，用来和加速器比较神经网络的运算结果。该参考模型支持各种低精度数据格式，运算器参照硬件设计，并且使用了运算符重载的代码编写方式，便于修改支持不同的加速器。另外，对于每一个输出神经元来说，运算顺序也采用了分层的设计方法，总体的计算顺序也同加速器运算顺序相同，并且易于修改。

5.6.1 参考模型框架

很多神经网络编程框架都使用了“先定义后运行”的策略。具体地，首先先定义一个网络结果，然后用户周期性的将不同的输入送到这个网络里计算。由于这个网络是在正向和反向运行之前就定义好了的，整个网络的数据流必须也要定义好。这些定义的过程就是一个具体的声明过程。我们的神经网络编程框架就采取了这种策略。

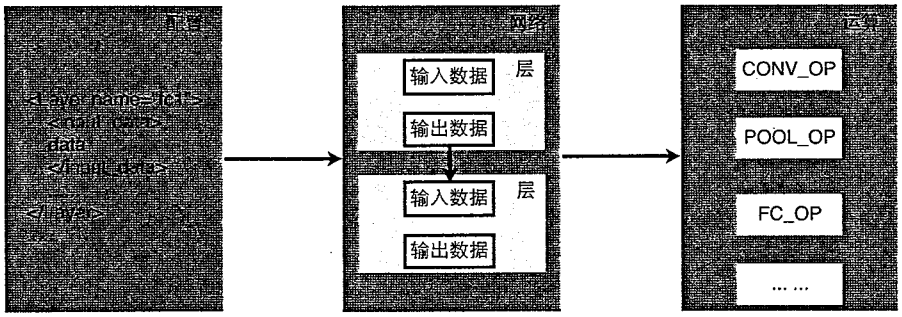


图 5.11 从输入配置到计算

首先我们定义了一个神经网络的配置格式作为输入，参考模型根据输入配置，解析参数并构建出一个以层为基本单位的神经网络结构。所有的输入输出数据存储在网络

中，而可训练连参数存储在层中。在网络构建完毕后，神经网络框架根据网络结构逐层进行运算，并将结果在网络中逐层传递。

5.6.2 输入配置

我们使用 xml 数据文件作为神经网络的输入配置。在 xml 中主要定义了神经网络的结构，以及每一层神经网络的具体参数，一个 xml 的配置部分参数如下面代码所示。在这个配置中定义了一个叫做 example 的神经网络网络，其中有一层叫做 fc 的全连接层。在 fc 全连接层中，定义了输入数据和输出数据的来源和数据格式。因为加速器支持多种低精度数据格式，所以也要同时指定各个数据的数据格式。

Listing 5.1 xml 格式的神经网络配置

```
<Net name="example">
    <Layer name="fc">
        <layer_type>FC</layer_type>
        <input_data>data</input_data>
        <input_data_type>float16</input_data_type>
        <weight_data>weight</weight_data>
        <weight_data_type>fix8</weight_data_type>
        <bias_data>bias</bias_data>
        <bias_data_type>float32</bias_data_type>
        <output_data>output<output_data>
        <output_data_type>float16<output_data_type>
    </Layer>
</Net>
```

5.6.3 数据存储

在我们的参考模型编程框架里，所有的数据都被分别存储在被称作张量的结构体里。每一个张量里存储了大小和维度不同的 N 维数组。在这个结构体里，也包括一些描述符，这些描述符用于描述张量中数据的格式以及数据的存储方式。另外，结构体内也实现了数据的重新摆放以及数据精度转换等方法。这些都是为了满足验证加速器的功能而添加的。

一般情况下，一个多个 batch 的输入图像在张量结构体重的存储顺序是：batch 数量 N* 图像特征图数量 C* 图像高度 H* 图像宽度 W。张量的存储顺序是以行优先存储的。例如，在一个 4 维的张量中，索引 (n，c，h，w) 数据在物理存储中的位置为 ((n*C+c)*H+h)*W+w。需要注意的是，张量在存储图像数据时，通常是 4 维的数组。但

是张量也可以用来存储其他非图像数据。比如，在存储全连接层的神经元时，数据只是一个一维数组。

神经网络权值参数的维度会根据配置的不同而不同。在一个卷积层中，有 256 个 3*3 大小的滤波器，而输入有 256 个特征图像。那么，这个卷积层的权值参数是一个四维数组：256*256*3*3。对于一个输入为 4096 神经元，输出为 1000 个神经元的全连接层来说，数据只有 2 个维度：4096*1000。

一般来说，数据在张量中的存储格式是 float32，为了验证加速器的低精度运算功能，张量中的数据可能会按照描述符中的格式转换成低精度，再进行计算。另外，在计算完成之后，可能需要与加速器对比结果，因此张量中的数据也需要转换成与加速器中对应的顺序。

一个层的输出张量可能会是下一个层的输入张量，张量之间的数据通过共享指针解决数据共享的问题。

5.6.4 层结构和连接

层是神经网络结构的基础组成部分也是计算的基本单元。层的类型包括：卷积、池化、激活等。不像传统的神经网络编程框架，我们添加了一些更基础的操作，作为基本运算层，比如加法层、减法层。这样做可以更加细致地验证神经网络加速器的各种功能。每个层以若干个张量作为输入，通过运算得到若干张量作为输出。每个层定义了两个基本的函数：create 和 forward。create 函数在构建网络时，初始化本层的参数及连接等。forward 函数顾名思义，是完成当层计算的函数。

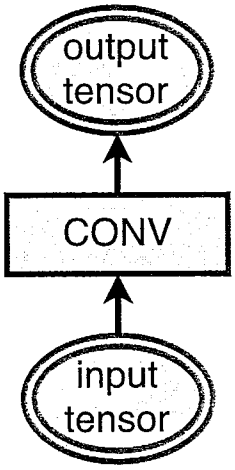


图 5.12 层结构

层之间的连接关系由他们的输入输出张量决定。

5.6.5 网络

网络是一系列的计算层组成的有向无环图。我们的计算框架能够保证层间的计算顺序，确保计算正确。在一个用于分类的网络结构中，一个网络通常以数据层开始，以全

连接层结束，得到分类结果。在网络构建时，也需要进行网络初始化函数，将每个层初始化，并且保存层间的张量结构体。

5.6.6 计算顺序

在数字的计算过程中，由于有精度截断溢出等原因，导致计算结果会略有不同。假设有一个 8 位有符号定点带饱和处理的运算器，在计算 3，127，-2 这三个数相加时，计算顺序不同会导致不同的结果：

- 3+127+（-2）的计算结果为 125。
- （3-2）+127 的计算结果为 127。

因此作为一个验证用的参考模型，每一个输出的计算顺序必须与加速器完全一致。为此，我们采用了分层的计算顺序划分方法来保证和加速器运算顺序。

作为一个参考模型，我们编写的神经网络框架要和待验证的加速器计算出一模一样的结果。因此需要保证计算顺序与神经网络加速器保持一致。我们将神经网络的计算顺序划分为不同模式，每种计算模式在我们的参考模型上都有对应的实现。首先，加速器每一拍都可以处理一个向量，这便是参考模型中最细粒度的计算函数；其次，加速器每一次访存到 SRAM 中的运算称作一个段运算，参考模型中也需要有对应的数据计算，称作段计算函数。最后，参考模型中也需要有段之间的运算函数与加速器对应。

5.6.6.1 多线程

为了加快参考模型的执行速度，我们采用了多线程的代码。由于要确保每一个输出计算过程的顺序，因此我们将不同的输出分给不同的线程去计算。为了提高参考模型的适用性，总的线程数作为一个输入参数，可以进行动态调节。

5.7 验证随机策略

出于不同的验证目标，我们在不同的验证阶段使用不同的验证策略。

5.7.1 模块级验证随机策略

模块级验证随机策略主要为了满足以下覆盖：代码覆盖率，功能覆盖率和条件组合。为此，模块级验证随机激励发生器综合考虑以上的情况，分门别类地产生随机激励，保证模块得到充分验证。

5.7.2 子系统级验证随机策略

子系统级随机验证随机策略主要为了满足功能覆盖率的要求。为此，子系统级随机激励发生器根据功能列表，随机产生出各种不同的功能组合。

5.7.3 系统级验证随机策略

5.7.3.1 指令随机

在在线比较的系统级验证中，主要目标是为了验证各个指令的功能以及指令序列的组合功能是否正确。在我们验证的神经网络加速器结构中大概有几十条指令。我们的指令随机激励不仅要将这几条指令全部随机出来，而且要随机出所有的可能的指令组合序列。

例如对于一款指令级具有 26 条指令的神经网络加速器，指令的名字分别为从 A 到 Z 的大写字母。随机激励经过几次随机很容就能将 A-Z 这几条指令都随机到，然而指令之间的随机顺序却很难保证。在实际的随机激励中，我们统计长度为 3 的指令序列的随机情况，在拥有 26 条指令的加速器上，指令序列组合总共具有 26*26*26 种情况。选用长度为 3 的指令序列的原因是，译码器的流水级只有 3 级，因此统计三条指令的组合足够确保指令的组合对于硬件行为的影响。指令随机系统会在随机过程中记录已经随机出的指令序列，并有更大概率随机出未出现过的指令组合。

5.7.3.2 神经网络参数随机

在离线比较的系统级验证中，主要的目标是为了验证加速器运算各种神经网络的正确性。在验证中我们只随机出一些基本的层，具体随机的层有:CONV、FC、POOL、LRN、BN、SCALE、ACT。其他像 LSTM 这样复杂的神经网络层，可以由上面简单的层组合得到，因此在验证中不需要考虑。神经网络参数随机需要考虑两个方面：一是层间的随机，而是层内参数的随机。

表 5.2 层间随机组合情况

上一层类型	下一层类型
DATA, CONV, POOL, LRN, SCALE, BN	CONV, POOL, LRN, SCALE, BN, FC
FC	FC

如表5.2所示，层间的参数组合很随意，CONV、POOL、LRN、SCALE、BN 层可以随意进行组合，而 FC 层只能出现在网络的尾部，ACT 层没有出现在表中，ACT 作为一个可选的操作，可以出现在上面任意层的后面。

层内可随机的参数更多，每一个变量都有很多取值，因此我们只考虑了每一个参数的取值随机，没有考虑多个参数之间的取值组合，一些层内随机的参数如下：

- 数据类型：每一层的输入、输出、权值、偏执等参数的数据类型都互相独立，可以取得值包括：8 位定点、16 位定点、16 位浮点、32 位浮点。
- 卷积核 kx、ky 的大小，步长 sx、sy 的大小，pad px、py 的大小，上述三组参数之间互有依赖，为了让三组参数都能尽可能的覆盖到更多的值，我们的随机策略

采用先随机出其中两个参数，然后第三个参数通过计算得到，这种策略能尽可能地覆盖更多的参数。

- 特征图数量。
- 一些层内常数。
- 是否稀疏。

第六章　实验和结果

6.1　实验方法

在这一节，我们将介绍稀疏神经网络加速器相关的实验方法。

6.1.1　Benchmarks

表 6.1　实验用到的 Benchmarks 和它们的权值和稀疏度明细

benchmark	Lenet-5		AlexNet		VGG16	
	权值	稀疏度	权值	稀疏度	权值	稀疏度
总和	36.30K	8.43%	6.80M	11.15%	10.53M	7.61%
卷积层	3.33K	13.06%	864.86K	37.08%	4.81M	32.69%
全连接层	32.97K	8.14%	5.93M	10.12%	5.72M	4.63%
benchmark	Drop NN1		Drop NN2		CIFAR10	
	权值	稀疏度	权值	稀疏度	权值	稀疏度
总和	44.38K	6.99%	5.89M	8.00%	6.15K	5.02%
卷积层	–	–	–	–	4.62K	5.84%
全连接层	44.38K	6.99%	5.89M	8.00%	1.53K	4.07%

我们使用了 6 个流行的神经网络作为 benchmarks，他们分别是：LeNet-5、AlexNet、VGG、DropoutNN1（2 层的全连接网络，800 个隐层神经元）、DropoutNN2（3 层全连接神经网络，8192*8192 个隐层神经元）和 CIFAR10 快速模型。表6.1列出了这些神经网络的一些特征，包括每个网络的权值数量，不同层的稀疏度。这些特征足够说明我们加速器的通用性。

（1）衡量方法

我们采用 Verilog 语言编写了加速器的 RTL 代码，在 TSMC 65nm 的工艺下，对设计进行了综合。我们使用 CACTI 6.0 对 DRAM 的访问进行能耗估计。

（2）衡量基准

我们的加速器与 CPU、GPU 和 DianNao 进行了对比。

CPU：我们在 CPU 上使用了流行的深度学习框架 Caffe 去衡量 benchmarks 与我们加速器的性能。CPU 的型号是英特尔志强 E5-2620 v2（简称为 CPU-Caffe）。为了测量 CPU 在稀疏神经网络上的性能，我们使用 sparse BLAS 的运算加速库实现了能够在 CPU 上运行的稀疏神经网路算法，稀疏表示方法使用的是 CSR（简称为 CPU-Sparse）。

GPU：我们在 GPU 上也是用 Caffe 去衡量 benchmarks 与我们加速器的性能。GPU 的型号是英伟达 K20M，拥有 5GB GDDR5 显存，单精度浮点的峰值运算能力为 3.52Tflops（简称为 GPU-Caffe）。另外，为了公平比较，我们使用 cuBLAS 也实现了一个 GPU 版本的神经网络运算程序（简称为 GPU-cuBLAS）。对于稀疏的版本，我们使用 CSR 的稀疏表示方法，在 cuSPARSE 加速库的基础上，实现了一个 GPU 版本的稀疏神经网络计算程序（简称为 GPU-cuSPARSE）。

加速器：我们也将我们的加速器与当下流行的加速器 DianNao 进行了对比。在 DianNao 加速器作者的帮助下，我们按照 DianNao 论文中的细节重新实现了 DianNao。为了公平比较，我们实现的 DianNao 计算能力与本文中加速器相当，乘法器是 16*16 个，16 个 16 输入的加法树和 16 个非线性计算模块。

6.2 硬件属性

在我们的设计中，Tm 和 Tn 的值我们都设定为 16。因此，加速器总共有 16 个 PE，每个 PE 有 16 个乘法器和一个 16 输入的加法树。在下表6.2中，我们列出我们加速器和我们实现的 DianNao 加速器的参数。其中，DianNao 中的 ALU 指的是 DianNao 的运算单元最后一级的操作：非线性运算单元，其功能和 BCFU 相同。在这样的设计下，我们的加速器可以达到和 DianNao 相同的峰值运算能力，即每一拍 528 个定点操作。与 DianNao 相同，加速器也是采用了 16 比特定点数据格式。

我们在表6.3中列出了加速器的面积和能耗的版图特征。此加速器总共含有 56KB 的片上 SRAM 和 528 运算器，比 DianNao 的面积大 2.11 倍。加速器总共的功耗是 954mW，比 DianNao 的功耗高 485mW。另外，此加速器的主频高达 1GHz，比 DianNao 的 0.98GHz 略高。

表 6.2 加速器与 DianNao 硬件参数对比

	Cambricon-X	DianNao
# BC	1	–
# PE	16	–
# 乘法器	256	256
# 16 输入加法树	16	16
# ALU	16	16
# 数据类型	16 位定点	16 位定点

表 6.3 加速器详细属性

accelerator	Area(mm^2)	%	Power(mW)	%
Total	6.38	100	954	100
BC				
NB	1.1	17.32	186.64	19.56
BCFU	0.11	1.72	31.63	13.31
IM	1.98	31.07	332.62	34.83
CP	0.16	2.54	75.06	7.86
PEs				
PEFU	1.78	27.94	153.01	16.02
SB	1.05	16.51	151.91	15.91

6.3 性能

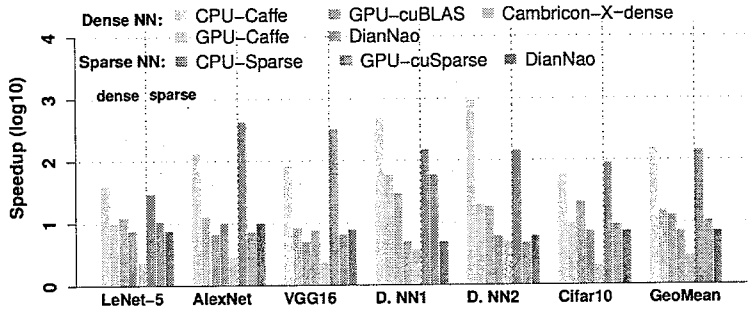


图 6.1 加速器和 CPU、GPU、DianNao 在 benchmarks 上的性能比较

我们在表6.1中的 benchmarks 上，比较了我们的加速器与 CPU，GPU 和 DianNao 之间的性能。在 CPU 和 GPU 上，除了使用了稠密表示的库（CPU-Caffe，GPU-Caffe 和 GPU-cuBLAS），我们也使用了稀疏矩阵加速库（CPU-Sparse 和 GPU-cuSparse）。为了公平的比较，我们也加入了我们加速器在稠密网络上的性能结果（Cambricon-X-dense）。在图6.1中，我们将所有的性能数据归一化到了我们加速器在稀疏神经网络上的性能。在稠密的神经网络 benchmarks 上，我们的加速器平均比 CPU-Caffe，GPU-Caffe 和 GPU-cuBLAS 分别快了 51.55 倍，5.20 倍和 4.94 倍。在稀疏神经网络 benchmark 上，我们的加速器分别比 CPU-Sparse 和 GPU-Sparse 快 144.41 倍和 10.60 倍。相比于 DianNao，我们的加速器在稀疏神经网络上的性能要好 7.23 倍。这一点说明我们的加速器比 DianNao 拥有更高的计算效率。因此，在计算稠密神经网络时，我们的加速器比 DianNao 要快 2.46 倍。

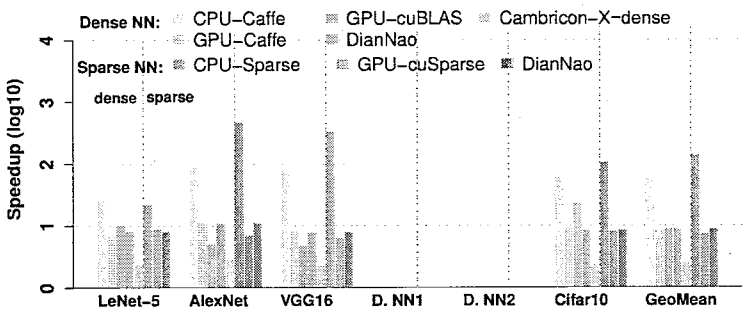


图 6.2　加速器和 CPU，GPU，DianNao 在卷积层上的性能比较

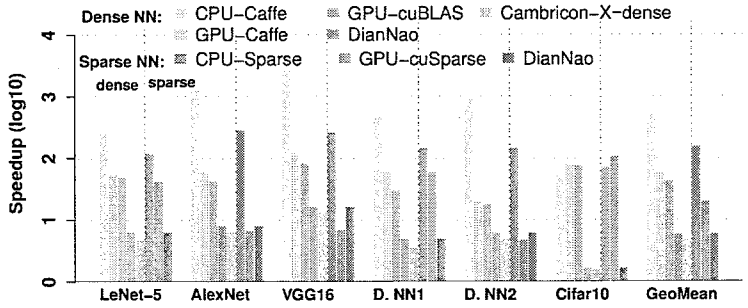


图 6.3　加速器和 CPU，GPU，DianNao 在全连接层上的性能比较

为了更具体的分析我们的加速器与其他计算平台的性能差异，我们单独统计了卷积层和全连接层的性能数据。如图6.2、6.3所示，所有的性能结果都归一化到我们的加速器在稀疏神经网络上的性能。对于卷积层来说，我们的加速器分别比 GPU-cuBLAS，GPU-cuSparse 和 DianNao 快 8.90 倍，7.67 倍和 8.89 倍。对于全连接层来说，我们的加速器分别比 GPU-cuBLAS，GPU-cuSparse 和 DianNao 快 44.28 倍，20.07 倍和 5.99 倍。在我们的加速器上，全连接层性能普遍比卷积层性能要好，这是因为全连接层比卷积层要稀疏的多（5.23% 比 22.65%）。另外，在我们的加速器上，稀疏的卷积层和全连接层分别要比稠密的卷积层和全连接层性能要好 2.51 倍和 4.84 倍。

6.4　能耗

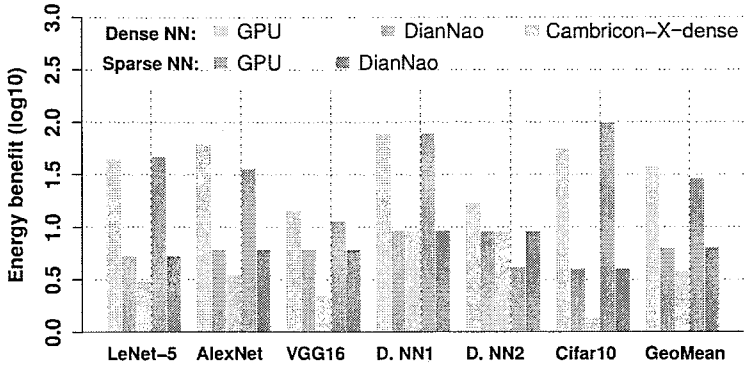


图 6.4　加速器相比 GPU 和 DianNao 在能耗上的收益

在图6.4中，我们报告了我们的加速器在所有 benchmarks 上与 CPU，GPU 和 DianNao 的在能耗数据上的对比结果，能耗的结果也都包含了片外访存的能耗数据。与 GPU 相比，我们的加速器平均可以在稀疏神经网络和稠密神经网络的能耗分别要低 37.79 倍和 29.43 倍。与 DianNao 相比，我们的加速器平均可以在稀疏神经网络和稠密神经网络的能耗全部都低 6.43 倍。我们发现与 GPU 相比，在全连接网络和 CIFAR10 两个 benchmarks 上，我们加速器的能耗表现最好。原因主要有以下两个方面，一方面是全连接层的稀疏度比较高，因此加速器的片外访存数据量要小的多，计算量也会相应的减少。另一方面，GPU 在 CIFAR10 这种小规模的网络上并行度不够高，计算效率低，因此能耗比较高。与 DianNao 相比，我们的加速器在全连接层的 benchmarks 上能耗表现最好，原因也是因为全连接层的稀疏导致我们的加速器在访存和计算上相比 DianNao 都获得了优势。

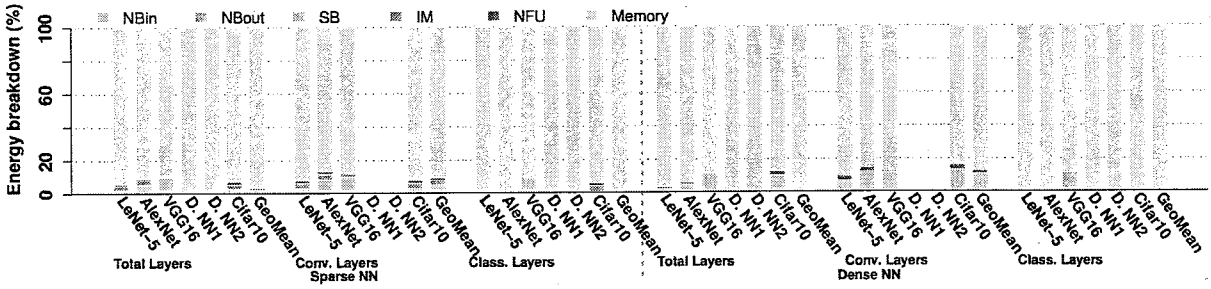


图 6.5 加速器在 benchmarks 上的能耗比重分布

另外，我们还统计了加速器在不同的层上的能耗比重，结果包括稀疏和稠密的神经网络。如图6.5所示，我们可以看到内存访问消耗的能量占据了总能量的 80% 以上，这个结果和陈等人 [29] 结果一致。由于全连接层的权值数量众多，并且不存在复用，因此在全连接层中，访存能耗比例比卷积层的访存能耗比例要高（平均数据是 98.39% 比 90.63%）。同样地，我们也对比了稀疏和稠密网络的能耗比重。我们发现，在稀疏神经网络上，访存能耗比例是要高于稠密神经网络的。换句话说，稀疏神经网络的访存问题变得更加严峻，因为稀疏神经网络的计算量更小。

6.5 计算效率

这一节，我们使用上面介绍的各种层映射方法，选取了一些神经网络层作为 benchmarks，衡量了加速器在这些 benchmarks 上的计算效率，这些层的参数见表6.4所示。

对于上表各个神经网络层，我们都在加速器上进行了仿真，得到了真实的性能数据，我们根据加速器上的运行时间和各个神经网络的计算量，经过计算得到了每一层神经网络在加速器上的实际的计算效率，结果如图6.6所示。

表 6.4 benchmarks 中每层的参数

名称	来源	fi	iny	inx	ky	kx	sy	sx	fo
CONV1	AlexNet	3	227	227	11	11	4	4	96
CONV2	VGG	512	16	16	3	3	1	1	512
POOL1	AlexNet	256	13	13	3	3	2	2	256
POOL2	VGG	512	14	14	2	2	2	2	512
LRN1	AlexNet	96	55	55	-	-	-	-	96
LRN2	GoogLeNet	192	56	56	-	-	-	-	192
FC1	GoogLeNet	1024	1	1	-	-	-	-	1000
FC2	VGG	4096	1	1	-	-	-	-	4096

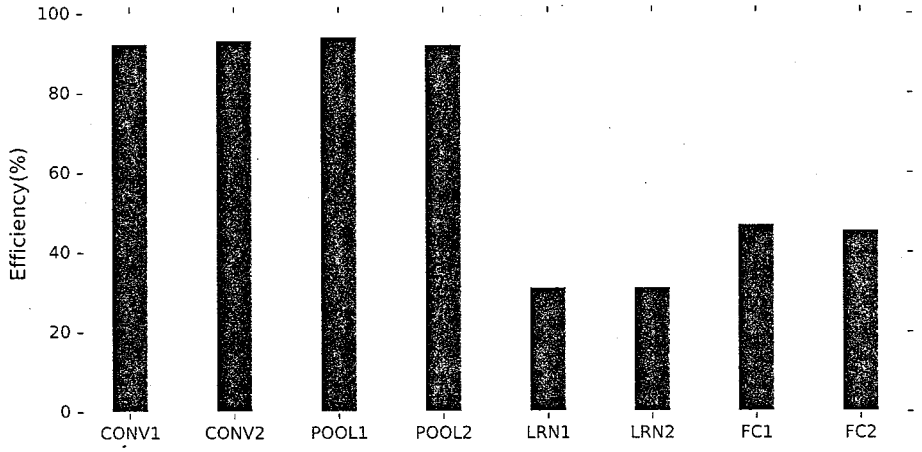


图 6.6 加速器在 benchmarks 上的计算效率

由图6.6可以看到，加速器在 CONV 和 POOL 上的计算效率都能达到 90% 以上。在全连接层上，加速器的计算效率不高，只是由于全连接层权值的数量太多，没有复用，IO 的速度是瓶颈引起的。

6.6 验证结果

验证工作很难被量化，对于神经网络加速器芯片的验证，我们采用三个指标来衡量验证工作的可靠程度：覆盖率、指令组合、神经网络参数。

6.6.0.1 覆盖率

对于几款神经网络加速器芯片，最终验证工作得到的覆盖率都达到了预定的目标，各项覆盖率指标如下表所示：

行和分支覆盖率在经过修改激励之后最终都到达了 100%，覆盖完全。因为代码风格的问题，条件覆盖率很难达到 100%。在对覆盖率的分析中，我们把所有未覆盖到的条件全都进行了分析，验证人员和设计人员一同确认，这些条件都属于正常状态不会触

表 6.5 覆盖率指标

项目	值
行覆盖率	100%
分支覆盖率	100%
条件覆盖率	95%
翻转覆盖率	95%
功能覆盖率	100%

发的条件。翻转覆盖率的提高主要靠数据的随机，控制信号范围的遍历，通过以上两种手段，翻转覆盖率也能达到 95% 以上。

6.6.0.2 指令组合

在系统级的验证中，我们对随机生成的所有指令进行了统计，确保所有的指令都被生成，并且所有的的指令序列都被遍历过。

6.6.0.3 神经网络参数

同样地，我们也统计了所有随机出的神经网络，并对神经网络层内的参数进行了参数遍历，确保神经网络的每个参数在一定取值范围内被覆盖完全。

表 6.6 在验证中设置的神经网络参数范围

参数	值
输入、输出、权值、偏执的数据格式	8 位定点，16 位定点，16 位浮点，32 位浮点
输入图像的宽度或者高度	1–2048
输入、输出图像的特征图像数量	1–4096
卷积核 (池化核) 的长或者宽	1–256
步长的长或者宽	1–256
pad 的长或者宽	1–256
LRN 的局部范围	1–15
神经网络的稀疏度	0%–100%

经过调查，上述验证工作所采用的神经网络参数范围远远超过了现在所有的神经网络参数的范围，并且所有的参数在范围内的取值全部都遍历过一遍。验证结果证明对于以上几种神经网络层，神经网络加速器全部都能正确地进行运算。

综上所述，我们对于神经网络加速器芯片的验证工作充分并且完善。神经网络加速器的各个功能都经过了可靠的验证，完成了预期所有功能的验证工作。

6.7 讨论

6.7.1 稀疏度与性能

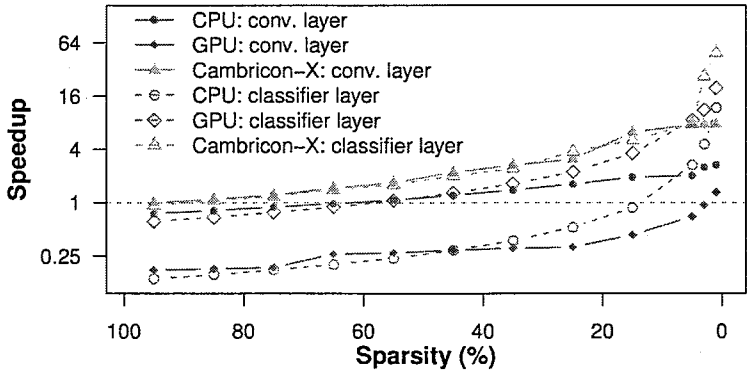


图 6.7 不同平台在不同稀疏度情况下的加速比

我们研究了不同稀疏度的神经网络在不同计算平台上的性能变化情况。图6.7显示了不同层的神经网络在不同稀疏度的情况下加速比的变化。其中稀疏度的变化范围从1% 变化到 95%，另外也包括了稠密的网络在这几个计算平台上的加速比。从图中我们得到了几个有趣的结论：当稀疏度比较高的时候 (高于 70%)，因为处理稀疏连接的代价比较高，同时运算量并没有减少很多，CPU 和 GPU 不能从稀疏神经网络中获得加速。例如，当稀疏度为 85% 时，CPU 和 GPU 在稀疏神经网络上的性能分别比自己在稠密神经网络上的性能差 85% 和 31%。在 GPU 平台上，全连接层和卷积层有着不同的结果。当稀疏度小于 55% 时，稀疏神经网络全连接层在 GPU 上的性能是好于稠密神经网路的。随着稀疏度进一步下降，稀疏神经网络的性能不断提升。然而在卷积层上，只有当稀疏度小于 3% 时，GPU 在稀疏神经网络上的性能才能好过稠密的神经网络。内在的原因是，全连接层计算所调用的稀疏向量乘法库函数（SpMV）对于稀疏度的减小更敏感，而卷积层计算所调用的稀疏矩阵乘法库函数（SpMM）对于稀疏度的减小比较迟钝。在 CPU 平台上，结果与 GPU 完全不同。在 CPU 上，稀疏神经网络卷积层的加速比比全连接层要好的多。然而，随着稀疏度下降，全连接层的性能会比卷积层以更快的速度极具提升。这些结果与 GPU 的结果一致，由于带宽受限，SpMV 操作对减小的稀疏度更为敏感。然而我们的加速器在稀疏度为 95% 时，就会得到收益。当稀疏度下降时，我们的加速器的加速比也会极具增长。对于全连接层，当稀疏度为 1% 时，我们的加速器能得到 48.53 倍的加速比，然而 CPU 和 GPU 只能分别得到 19.42 倍和 11.72 倍的加速比。随着稀疏度降低，不管是卷积层还是全连接层，加速器在稀疏的网络上的性能都比稠密的网络要好。不仅如此，上面的结果也说明，我们的加速器对于稀疏度特别低的网络有着更好的适应性。

6.7.2 裁剪神经元

尽管当一个神经元没有连接时，这个神经元是可以被去掉的，但是我们的加速器在卷积层的时候并不能从中受益，原因如下。简单起见，我们假设在一个卷积层中，只有一个神经元可以被去掉。由于这个被去掉的神经元不会在 PE 中消耗计算资源，为了重复使用输入神经元，因此另一个来自于不同的特征图像的神经元可能会分配给这个 PE 去做运算。这样，每个 PE 中分配的输出神经元就不再是以 Tn 为间隔的平均分配。这种不对齐的分配会大大增加硬件在卷积层的开销。因为现在神经元的存储不再是按照顺序存储的。不同的是，全连接层的神经元仅仅是一个一维向量，不需要考虑对齐问题。因此，裁剪神经元的方法只能使全连接层获益。

6.7.3 DaDianNao

我们也分析了 DaDianNao 的结构，DaDianNao 是一个更大规模的神经网络加速器，里面包含了 16 个运算核，其中每一个的计算能力都相同并且等于我们加速器的计算能力。DaDianNao 了中集成了 36MB 大小的片上 eDRAM。eDRAM 和所有的运算核通过共享的数据总线相连，因此所有的运算核都从 eDRAM 中得到相同的输入。有两个办法可以在 DaDianNao 的结构中增加对稀疏神经网络的支持。第一个方法是在每个运算核中都集成一个索引模块，每一个索引模块都为自己的运算核选择出计算所需要的输入。然而，这种设计要求 eDRAM 和所有计算核之间的数据总线需要提供很大的带宽。因为稀疏神经网络对输入神经元的需求量更大。例如，在处理一个稀疏度为 10% 的网络时，每一个运算核需要 10 拍才能读到 160 个 16 比特的数据，由于只有 10 个数据是有效数据，因此运算核只需要计算一拍。也就是说，在计算稀疏神经网络时，DaDianNao 的数据总线的带宽需求是之前的至少 10 倍。第二个方法是在中央 eDRAM 中添加一个索引模块然后该模块将选择出的数据顺序的发送给不同的运算核。这种方法对于共享的数据总线来说同样不高效。总之，DaDianNao 不能在保持高效计算的前提下，增加对稀疏神经网络的支持。

第七章　总结和展望

7.1　总结

本文针对稀疏神经网络，研究了微体系结构、算法映射与编程模型以及功能验证三个方面的问题，研究工作具有系统性。

首先，我们提出了一个世界上首个稀疏卷积神经网络加速器微结构。该加速器微结构不仅能处理稀疏神经网络，也能处理稠密神经网络。该加速器微结构基于多 PE 的结构，主要由一个缓存控制单元和众多 PE 实现各种神经网络的运算。在处理稀疏神经网络时，缓存控制单元将选择出的神经元通过胖树传送给 PE。每个 PE 各自存储了自己计算所需要的压缩后的权值，并且 PE 之间的计算是异步的，互相不受对方的影响。我们的加速器面积只有 6.38mm^2，功耗只有 954mW。我们的加速器每一拍能同时输出 16 个输出神经元，最高能够达到 544GOPs 的计算能力。和高性能的加速器 DianNao 相比，在处理稀疏神经网络时，平均来讲，我们的加速器性能会好 7.23 倍，能耗会低 6.43 倍。

我们研究了两种稀疏索引方式: 直接索引和步长索引。我们在 RTL 层上实现了上述两种方法的具体电路并且比较了两者在面积和功耗上的开销。随着稀疏度的增长，硬件的开销也逐渐增大。另外，在所有的情况下，步长索引的方式总好于直接索引。当数组长度是 256 时，步长索引方式的硬件面积和功耗分别要比直接索引方式低 10% 和 40%。

另外，我们还分析了各种神经网络算法，详细阐述了他们在 Caffe 中的运算过程。受此启发并结合我们的加速器特征，我们想出了各个层在加速器上映射的方法。在神经网络中，不同的神经网络层具有不同的运算和访存特点。卷积层的计算量很大，卷积层的计算量超过了整个卷积神经网络 80% 以上的计算量，由于共享权值的特点，卷积层的权值数量很少。全连接层与卷积层恰恰相反，全连接层的计算量很小，权值的数量却很多。池化层和各种正规化层都没有权值，计算量都不大。在将不同的神经网络层映射到加速器时，需要考虑每种层的计算和访存特点。对于卷积层，我们采用复用权值的方法，每个权值只会从内存中读一次，直到所有和这个权值的相关的输出都使用过这个权值之后，这个权值才会被丢弃掉；对于全连接层和其他层，我们采用复用输入或者输出神经元的方法，尽量减少神经元在内存中的读写次数。采用我们的算法映射方法，最大化减少了访存量，加速器在很多层上运算效率在 90% 以上。

最后，我们提出了一套针对神经网络加速器的验证方法以及一个通用的参考模型。首先，验证采用层次化方法，不同阶段的验证侧重点不同。在模块级验证阶段，采用白盒验证手段，重点验证模块的功能以及内部信号的状态。在子系统级验证阶段，我们将加速器分为 IO 子系统和运算子系统，不同的子系统分别验证不同的子功能。在系统级

验证阶段，我们搭建了两个不同的验证环境。一个是在线比较的验证环境，主要为了验证加速器在随机指令序列激励下的功能；另一个是离线比较的验证环境，主要为了验证加速器在运算各种神经网络下的功能。为此，我们开发了一个基于 C++ 的通用参考模型。该参考模型与加速器结构无关，能够支持各种低精度的神经网络运算，并且支持各种常见的神经网络类型。对于不同的加速器结构，参考模型只需要简单的修改层内的计算顺序即可使用。另外，该参考模型支持多线程，因此运算速度相比 RTL 的仿真速度提高了很多。在验证的不同阶段，我们使用不同的衡量参数来保证验证工作的充分性。

本文以稀疏神经网络加速器为例，提出并实践了一套“设计-实现-映射-验证”的芯片开发工程方法，对芯片创新和研发具有参考价值。上述工作已部分应用于国际上首个稀疏神经网络处理器芯片上，确保了该芯片的实用性和正确性。

7.2 展望

本文提出了一个稀疏神经网络加速器微结构，相比以前的加速器增加了对稀疏神经网络的支持，进一步提升了能耗和性能。但是，未来围绕着这一稀疏神经网络加速器，还有很多工作要做。

硬件的发展和用户的支持是分不开的，想要得到用户的支持，良好的编程环境是不可或缺的。首先，在底层需要有灵活、高效的汇编语言；其次，在上层要有简单、方便的高级语言。

7.2.1 加速器指令集的汇编语言

稀疏神经网络加速器采用超长指令字指令集，指令格式复杂。目前，我们做了编程库的 API 开放给用户使用，以减小用户在使用时的学习代价和开发周期。但是编程 API 也有很多限制。首先，编程库实现的函数数量和功能有限，不够灵活。另外，编程库无法保证各种神经网络规模的高效计算，用户没法优化性能。因此，在未来的工作中，开发一个稀疏神经网络加速器汇编语言的工作显得很有必要。汇编语言可以在保证加速器处理的效率前提下，降低用户的编程难度，并且给予用户足够的灵活度。

7.2.2 神经网络的高级编程语言

为了让代码具有广泛的适用性以及开发效率，未来我们将针对稀疏神经网络处理器开发一种高级的编程语言。这种语言的语法形式要针对神经网络算法进行特殊定制，比如神经网络特有的数据类型（神经元，突触），连接方式等，这些元素的设计不仅需要考虑到神经网络算法实现的便利性，还要考虑底层的编译过程能够生成出高效的机器指令，即兼顾开发效率和运行效率。