中国科学技术大学

博士学位论文

中国科学技术大学

学位论文模板示例文档

作者姓名: 李泽平

学科专业: 数学与应用数学

导师姓名: 华罗庚 教授　钱学森 教授

完成时间: 二〇一八年七月十五日

University of Science and Technology of China

A dissertation for doctor's degree

An Example of USTC Thesis Template for Bachelor, Master and Doctor

Author: Zeping Li

Speciality: Mathematics and Applied Mathematics

Supervisors: Prof. Luogeng Hua, Prof. Xuesen Qian

Finished time: July 15, 2018

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何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贡献均已在论文中作了明确的说明。

作者签名：_____　　签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等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、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

□ 公开　□ 保密（____ 年）

作者签名：_____　　导师签名：_____

签字日期：_____　　签字日期：_____

摘　要

摘要是论文内容的总结概括，应简要说明论文的研究目的、基本研究内容、研究方法或过程、结果和结论，突出论文的创新之处。摘要中不宜使用公式、图表，不引用文献。博士论文中文摘要一般 800 ~ 1000 个汉字，硕士论文中文摘要一般 600 个汉字。英文摘要的篇幅参照中文摘要。

关键词另起一行并隔写在摘要下方，一般 3 ~ 8 个词，中文关键词间空一字或用分号“;”隔开。英文摘要的关键词与中文摘要的关键词应完全一致，中间用逗号“,”或分号“;”隔开。

关键词：中国科学技术大学；学位论文；LaTeX 模板；学士；硕士；博士

ABSTRACT

This is a sample document of USTC thesis LaTeX template for bachelor, master and doctor. The template is created by zepinglee and seisman, which orignate from the template created by ywg. The template meets the equirements of USTC theiss writing standards.

This document will show the usage of basic commands provided by LaTeX and some features provided by the template. For more information, please refer to the template document ustcthesis.pdf.

Key Words: University of Science and Technology of China (USTC); Thesis; LaTeX Template; Bachelor; Master; PhD

目　录

第 1 章　简　　介

1.1　一级节标题

1.1.1　二级节标题

1. 三级节标题

（1） 四级节标题

① 五级节标题

1.2　脚注

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. [①]

[①] This is a long long long long long long long long long long long long long long long long long long long long long long long long footnote.

第 2 章　神经网络简介

2.1　神经网络算法基础

现代神经网络由多种不同类型的层组成，输入数据依次通过各个层被逐层处理，最终被分类, 识别或者检测。在每一层中，神经元接收多个输入进行处理，然后通过连接将输出发送到下一层。神经元之间的连接，即所谓的突触，通常具有独立或者共享的权重。神经网络在图像处理，语音识别，语音合成，自然语言处理等领域有着非常广泛的应用。目前在图像处理应用最广泛的网络是卷积神经网络（convolutional neural networks, 简称 CNNs）和深度神经网络（deep neural networks, 简称 DNNs）, 它们由卷积层，池化层，归一化层，激活层和全连接层等组成。递归神经网络（recurrent neural networks，简称 RNNs）是一类重要的机器学习技术，专门用于处理顺序数据序列和可变长度数据序列。RNNs 在语音识别，自然语言处理，场景语义理解和时间序列分析等方面有着广泛的应用。其中应用最广泛，性能最高的两种类型的 RNNs 是长短时记忆网络（long short term memory, 简称 LSTM）和门控循环单元 (gated recurrent unit, 简称 GRU)。下面将为大家介绍神经网络中集中常见的神经网络层类型。

2.1.1　全连接层

全连接层（fully connected layers）是神经网络算法中常见的一种层类型，主要用来将上一层提取的特征进行组合，综合以及分类，在整个神经网络中起到“分类器” 的作用。全连接层的结构如图 2.1所示，输入层神经元为 m 个，输出层的神经元为 n 个，其中每一个输出神经元与所有输入神经元相连，其结构相当于 n 个 m 输入的感知机，因此全连接层也可以看成是感知机扩展。

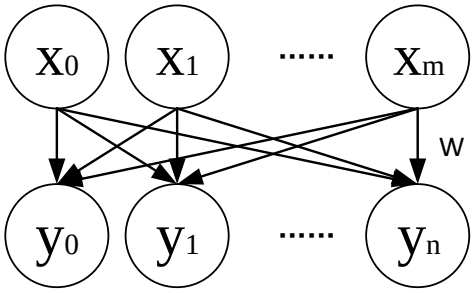


图 2.1　全连接层.

全连接层的核心操作操作是矩阵向量乘积，

$$y = W * x + b \tag{2.1}$$

其中 y 是输出神经元向量，x 是输入神经元向量，W 是权值矩阵，b 是输出神经元的偏置向量。因此全连接层的本质是由一个特征空间线性变换到另一个特征空间，且目标空间的任一维都会收到源空间每一维的影响。由于每个输出神经元都要与输入神经元相连接，这样的话就会造成权值数量巨大，从而造成网络难以训练，并且会出现过拟合的情况。因此在 CNN 中，全连接层常出现在最后几层，用取于对前面设计提的特征做加权和。

2.1.2　卷积层

卷积层（convolutional layers）是卷积神经网络的核心层，主要用于提取特征。卷积层受到生物学上感受野（Receptive Field）的机制而提出的。感受野主要是指听觉系统、本体感觉系统和视觉系统中神经元的一些性质。比如在视觉神经系统中，一个神经元的感受野是指视网膜上的特定区域，只有这个区域内的刺激才能够激活该神经元。在听觉系统中，对于语音则是某一时间戳后的时间段才能激活神经元。卷积层充分借鉴感受野的机制，神经元仅仅能够感受局部特征，而卷积核的大小直接限制感受野的大小，输出神经元只和周围一小块的输入神

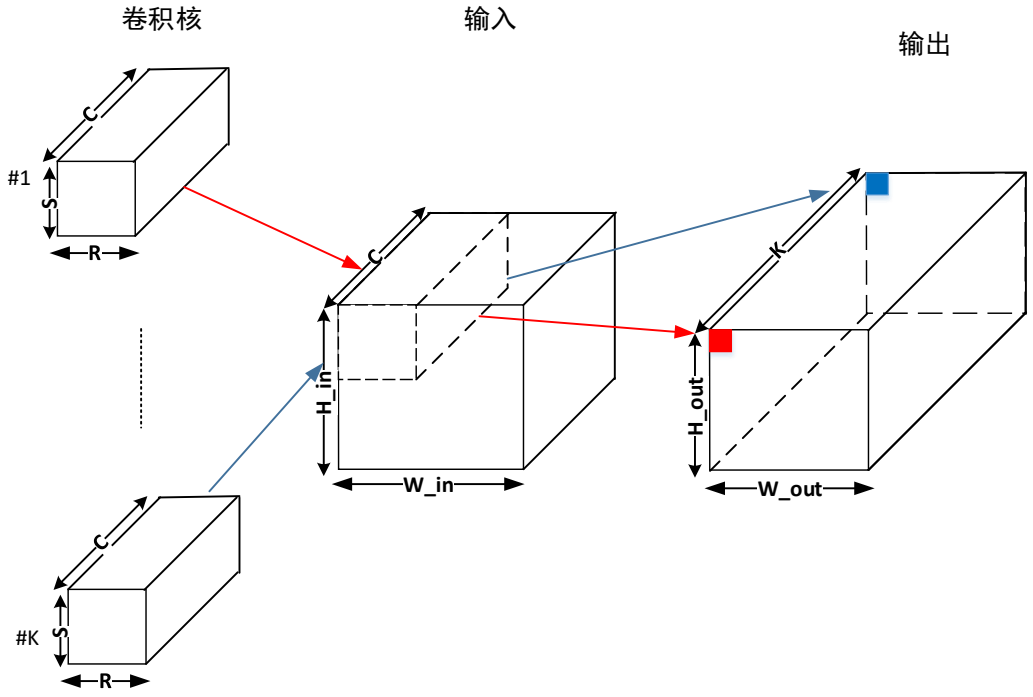


图 2.2　卷积层.

经元存在连接。局部连接的方法大大减少了卷积层中的权值数量。卷积层采用了共享权值的方法进一步减少权值的数量，即同一输出特征图共享一组卷积核。同一个输出特征图像上的所有像素点是同一组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。卷积层采用多组不同的卷积核对输入特征图进行卷积，从而获得多组输出特征图，即多组不同的图像特征。

卷积层的结构如图 2.2所示。卷积层的核心运算是二维滑动窗口卷积运算，即规模为 $R \times S$ 的卷积核在规模为 $W_{in} \times H_{in}$ 的输入特征图上滑动进行二维卷积，最终产生规模为 $W_{out} \times H_{out}$ 的输出特征图。通常情况下，输入特征图不仅只有一个，而是由 C 个组成，即规模为 $C \times W_{in} \times H_{in}$，因此我们对每个输入通道都施加一个卷积核，即使用规模为 $C \times R \times S$ 的卷积核对输入进行卷积，最终获得一个输出特征图。当我们将采 K 组不同的卷积核作用于相同的输入特征图，将获得 K 个不同的输出特征图。最终，卷积层的输入规模为 $C \times W_{in} \times H_{in}$，卷积核的规模为 $K \times C \times R \times S$, 输出规模为 $K \times W_{out} \times W_{out}$。

```
1  for k = 0 to K do
2      for w = 0 to W_out do
3          for h = 0 to H_out do
4              for c = 0 to C do
5                  for s = 0 to S do
6                      for r = 0 to R do
7                          out[k][w][h]+ =
                             in[c][w * sw + r][h * sh + s] * filter[k][c][r][s]
8                      end
9                  end
10             end
11         end
12     end
13 end
```

算法 2.1: 六层卷积循环

卷积层的计算由 K，C，R，S，W 和 H 这六个变量形成嵌套循环完成，而且这六个变量的所有排列都是合法的。算法 2.1展示了其中一种循环嵌套方式，我们可以用 $N->W_{out}->H_{out}->C->S->R$ 来描述这种循环，其中 sw 和 sh 表示卷积操作的步长。不同的循环方式决定了数据的复用形式和数据流的方式，最终将影响神经网络加速器的设计。

2.1.3　池化层

池化层（pooling layer）是神经网络中一个重要的层。池化层一般是在卷积层之后，对输入进行非线性降采样，常用的池化做法是对每个滤波器的输出求最大值，平均值，中位数等。池化层的意义主要体现在两个方面：第一，池化层通过对特征图像进行降维操作，能够在保留显著特征的情况下，有效减少整个神经网络所需要的参数量和计算量。第二，池化层能够保证输入的平移不变性(translation invariant)，这意味着即使图像的像素在邻域发生微小位移时，池化层的输出能够保持不变，从而增强神经网络的鲁棒性，由一定的抗扰动能力。常用的池化层包括最大池化层，平均池化层，ROI (Regions of interest) 池化层.

最大池化层的基本思想是在一个特定的数据区域内选择一个最大值作为输出。其计算公式是

$$out_{x,y} = max(in_{x*sx,y*sy} : in_{x*sx+kx,y*sy+ky}) \tag{2.2}$$

其中 kx 和 ky 是池化窗口的大小，sx 和 sy 是池化的步长。通常情况下池化窗口的大小与池化的步长相同，即池化操作的输入并不会重复。后来也出现了数据复用的池化，相邻池化窗口之间会有重叠区域，此时池化步长小于池化窗口的大小。

平均池化层的基本操作与最大池化层类似，它将一个特定的区域内的数据取算数平均作为输出，其计算公式是

$$out_{x,y} = mean(in_{x*sx,y*sy} : in_{x*sx+kx,y*sy+ky}) \tag{2.3}$$

ROI Pooling 层主要是针对 ROIs 的池化操作，主要应用于物体检测领域的 Fast RCNN 和 Faster RCNN 网络中，它的特点是输入特征图尺寸不固定，但是输出特征图的尺寸固定。ROI Pooling 的输入包含两部分，第一部分是特征图，在 Fast RCNN 中，它位于 ROI Pooling 之前；在 Faster RCNN 中，它是与 RPN 共享的那个特征图。第二部分是 ROIS，在 Fast RCNN 中，指的是 Selective Search 的输出；在 Faster RCNN 中指的是 RPN 的输出，是一系列的矩阵候选框，每一个矩阵候选框用四个坐标和索引来表示。ROI Pooling 的输出则是 batch 个三维向量（$C \times W \times H$），其中 batch 的值为 ROI 的数量。因此 ROI Pooling 的过程可以总结为将大小不同的矩阵框映射为大小固定的矩阵框。

2.1.4　归一化层

随着神经网络的规模不断变大，结构的复杂度增加，神经网络越来越难以训练，同时神经网络越来越容易出现过拟合的现象。归一化层（normalization layer）成为神经网络中不可或缺的一个层，它能够加快神经网络的收敛速度，防止过拟

合,降低神经网络对初始化权重的敏感度,提高神经网络的精度。近几年出现了各种各样的归一化方法, 主要包括 LRN(Local Response Normalization) [1],BN(Batch Normalization) [2]，LN(layer Normalization) [3]，IN(Instance Normalization) [4] 和 GN(Group Normalization) [5] 等。

LRN 是 AlexNet 等网络中使用的归一化方法，它在每一个像素的小领域范围内进行归一化处理。目前主流的归一化方法则更加注重全局范围内的归一化处理，这些全局归一化的方法能够使得我们在训练神经网络时使用较大的学习率，从而加快神经网络训练速度。如图 2.3所示，BN 选择在 batch 维度上进行归一化处理；LN 选择在 channel 维度进行归一化处理；IN 执行类似 BN 的计算, 但是仅仅在单个样本执行归一化；GN 在 IN 的基础上对多个样本进行归一化。下面主要对 BN 的计算方法进行简要介绍。

BN 的计算公式如下所示：

$$\mu_\beta = \frac{1}{N}\sum_{i=1}^{N} a_i \tag{2.4}$$

$$\theta_\beta^2 = \frac{1}{N}\sum_{i=1}^{N}(a_i - \mu_\beta)^2 \tag{2.5}$$

$$\hat{a}_i = \frac{a_i - \mu_\beta}{\sqrt{\theta_\beta^2 + \epsilon}} \tag{2.6}$$

$$b_i = \gamma \hat{a}_i + \beta \tag{2.7}$$

在上述算法中，m 为 batch 数，a_i 是第 i 个 batch 的输入数据，μ_β 和 θ_β^2 分别是 N 个输入的均值和方差，参数 ϵ 是批变化常量，参数 γ 和 β 是训练时需要学习的参数，b_i 是最后归一化后的输出。

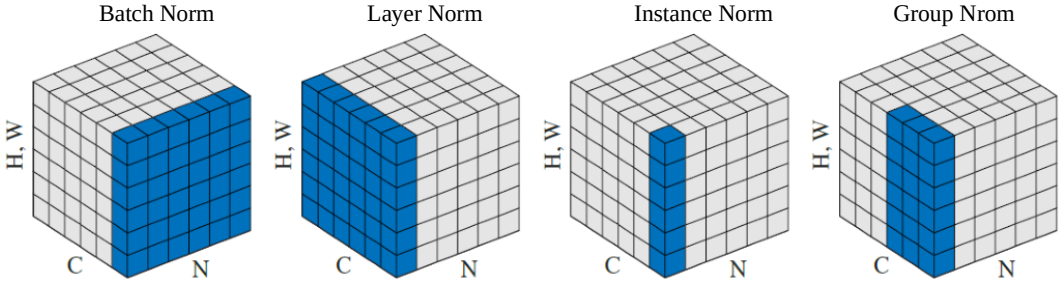


图 2.3　归一化方法。每一个子图显示的是一个 feature map，其中 N 为 batch 轴，C 为 channel 轴，（H,W）为空间轴。蓝色的像素表示归一化的范围。

2.1.5　激活层

激活层（activation layer）是神经网络能够解决非线性问题的关键，它弥补了神经网络中线性模型表达能力不足。目前主流的激活函数包括 Sigmoid, Tanh, ReLU 以及 ReLU 的变种, 如 PReLU 和 RReLU 等。

Sigmoid，Tanh 和 ReLU 的函数图如图 x 所示，公式分别为

$$Sigmoid(x) = \frac{1}{1+\mathrm{e}^{-x}} \tag{2.8}$$

$$Tanh(x) = \frac{1-\mathrm{e}^{-2x}}{1+\mathrm{e}^{-2x}} \tag{2.9}$$

$$ReLU(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leqslant 0 \end{cases} \tag{2.10}$$

Sigmoid 函数主要被应用于早期的神经网络中,Sigmoid 函数拥有良好的性质，Sigmoid 函数的值域范围限制在 (0,1) 之间，这与概率值的范围是相对应的，这样 sigmoid 函数就能与一个概率分布联系起来了; 同时 Sigmoid 函数的导数可以由其本身求得 $Sigmoid(x)' = Sigmoid(x)(1 - Sigmoid(x))$。但是 Sigmoid 函数也有不少缺点，首先 Sigmoid 函数的输出并不是以 0 为中心，这个特性会导致为在后面神经网络的高层处理中收到不是零中心的数据，进而训练时的权值更新产生锯齿晃动。同时 Sigmoid 函数具有饱和性，容易出现梯度消失的现象，使得神经网络难以训练。

Tanh 函数是 Sigmoid 函数的改进版本，它的收敛速度比 Sigmoid 函数更快，相比 Sigmoid 函数，其输出以 0 为中心, 但是仍然存在饱和性而导致梯度消失的问题的。Tanh 函数和 Sigmoid 函数目前主要被用于基于 LSTM 的 RNN 架构或者基于 GRU 的 RNN 架构中。

ReLU 是目前非常流行的激活函数，它是分段线性函数，所有的负值为 0，而正值不变，这种操作被称为单侧抑制。单侧抑制使得神经网络中的神经元也具有了稀疏激活性，尤其体现在深度神经网络模型 (如 CNN) 中，当模型增加 N 层之后，理论上 ReLU 神经元的激活率将降低 2 的 N 次方倍，从而更好地挖掘相关特征，拟合训练数据。同时，ReLU 不存在饱和区，因此不存在梯度消失的问题，使得模型的收敛速度维持在一个稳定的状态。[1] 实验显示 ReLU 单元比 tanh 单元具有 6 倍的收敛速度提升。

2.1.6　LSTM

现代大规模自动语音识别（automatic speech recognition，简称 ASR）系统利用基于 LSTM 的 RNN 作为其声学模型。LSTM 模型由一系列大规模矩阵组成，这是 ASR 的所有步骤中计算量最大的部分。在基于 LSTM 的 RNN 中，时刻 T

的输入取决于时刻 T-1 的输出。一个经典的 LSTM 模型如图 2.4所示。LSTM 模型包含特殊存储单元（图 2.4中的 $cell$）和三个特殊的门（图 2.4中的 i，o，f），其中 $cell$ 用于存储网络的时间状态信息，门用于执行特殊的乘法运算，包括输入门（input gate）、输出门（output gate）和遗忘门（forget gate）。输入门 i 控制输入到存储单元中的输入量；输出门 o 控制输出值；遗忘门 f 自适应地遗忘 $cell$ 中存储的信息，从而控制前一状态对现状态的影响。除了基本的三个众所周知的门和 $cell$ 之外，该 LSTM 模型还引入了窥视孔（peephole）和投影层（projection layer），以便更好地学习。窥视孔将缩放后的 $cell$ 状态添加到三个门，其中缩放尺度由三个对角矩阵决定。投影层线性地将输出转换为低维形式。

LSTM 模型的接收一个输入序列 $X = (x_1; x_2; x_3; ...; x_T)$（其中 x_t 是 t 时刻的输入向量）和上一个状态的输出序列 $Y^{T-1} = (y_0; y_1; y_2; ...; y_{T-1})i$（其中 y_{t-1} 是 t-1 时刻的输出向量）。利用下列公式从 t=1 到 T 计算输出序列 $Y = (y1; y2; y3; ...; y_T)$:

$$i_t = \delta(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i), \tag{2.11}$$

$$i_t = \delta(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i), \tag{2.12}$$

$$f_t = \delta(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f), \tag{2.13}$$

$$g_t = \delta(W_{cx}x_t + W_{cr}y_{t-1} + b_c), \tag{2.14}$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t, \tag{2.15}$$

$$o_t = \delta(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o), \tag{2.16}$$

$$m_t = o_t \odot h(c_t), \tag{2.17}$$

$$y_t = W_{ym}m_t \tag{2.18}$$

其中符号 i、f、o、c、m 和 y 分别是输入门、遗忘门、输出门、$cell$ 状态、$cell$ 输出和投影输出；$\odot$ 表示向量逐元素乘积，W 表示权值矩阵（例如 W_ix 是从输入向量 X_t 到输入门的权值矩阵），b 表示偏置向量。值得注意的是 W_ic、W_fc 和 W_oc 是用于 peephole 连接的对角矩阵，因此它们本质上是向量。δ 是 Sigmoid

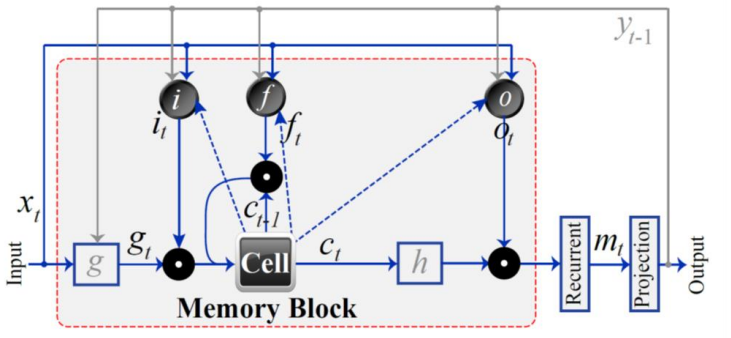


图 2.4　一个基于 LSTM 的 RNN 架构.

激活函数，h 是用户自定义的激活函数，通常情况下会采用 $tanh$ 激活函数。

2.1.7　GRU

GRU 是 LSTM 的变体，它把遗忘门和输入门组合成一个更新门（update gate）。它还合并了 $cell$ 状态和隐藏状态，并进行了一些其他更改。GRU 架构如图 2.5所示。类似地，它遵循如下的公式从 t = 1 到 T 进行迭代计算：

$$z_t = \delta(W_{zx}x_t + W_{zc}c_{t-1} + b_z), \tag{2.19}$$

$$r_t = \delta(W_{rx}x_t + W_{rc}c_{t-1} + b_r), \tag{2.20}$$

$$\tilde{c_t} = h(W_{\tilde{c}x}x_t + W_{\tilde{c}c}(r_t \odot c_{t-1}) + b_{\tilde{c}}), \tag{2.21}$$

$$c_t = (1 - z_t) \odot c_{t-1} + z_t \odot \tilde{c_t} \tag{2.22}$$

其中符号 z、r、$\tilde{c}$、c 分别是更新门、复位门、复位状态和 $cell$ 状态；$\odot$ 表示逐元素乘法。W 表示权值矩阵，δ 是 Sigmoid 激活函数，h 是用户定义的激活函数，这里我们使用 $tanh$ 激活函数。值得注意的是 GRU 有两个门（更新门和复位门），而 LSTM 有三个门（输入门、遗忘门、输出门），GRU 中不存在 LSTM 中存在的输出门，而是将 $cell$ 状态作为输出。LSTM 中输入门和遗忘门耦合称为 GRU 中的更新门 z，复位门 r 直接作用于到上一个 $cell$ 的状态。

2.2　神经网络低能耗的技术

随着神经网络算法被应用于更复杂的处理任务和更广泛的场景中，神经网络的规模也越来越大。最新的 DNNs [] 需要数百兆字节甚至前兆字节来存储神经元和权值；同时需要数十亿次的乘加操作完成运算。考虑未来神经网络将朝着

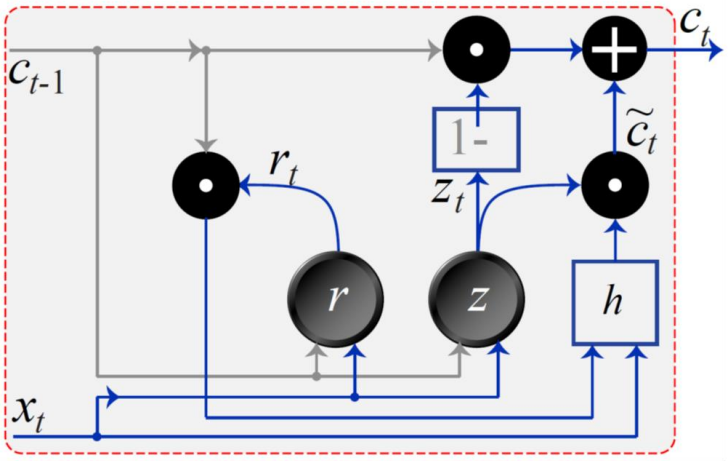


图 2.5　一个基于 GRU 的 RNN 架构.

规模更大，层数更深的趋势发展，未来的大规模网络很难部署到嵌入式系统中。因此很多研究人员致力于减少神经网络运算需求，降低神经网络运算时的能耗。目前已经有了一些有效的算法来解决这个问题，主要方法包括神经网络的低精度计算和神经网络压缩模型。

2.2.1　神经网络的低精度计算

最近研究 [6] 表明，在深度神经网络的训练中，不需要全精度的数据。使用低比特的权值表示，可以显著减少网络存储，特别是当使用极低比特位，如二进制/三元权值时。研究显示，非精确的计算（在神经元和权重中加入噪音）具有正则化的效果，从而减少泛化误差 [7]。 Gupta et al. [6]使用 16 位定点数字表示来训练网络。 Köster et al. [8]提出了动态数据格式（FlexPoint），用来完全替代 32 位浮点格式。Flexpoint 具有一个可动态调整的共享指数，以最小化溢出和最大化可用动态范围。在训练过程中，数据范围会随着训练轮数的增长而连续变化，其中共享指数部分可以根据历史数据进行预测。实验结果显示，采用 16 位尾数和 5 位共享指数的 Flexpoint 格式数据在训练神经网络时，能够获得 32 位浮点类似的精度，并且远远超过 16 位浮点获得的精度。 Dettmers [9]使用 8 比特量化神经网络，在保持性能的情况下，加快神经网络训练速度。在 [10] 和 [11] 中，作者表明，深度神经网络可以通过二进制权值进行训练，在某些情况下，它的精度可能超过使用浮点数进行训练的结果。 Rastegari et al. [12]首先提出了三元权值网络 (Ternary Weight Network, TWN), 并在 ImageNet [13] 数据集上取得良好的效果。三元权值网络在 [14-15] 等工作中被不断深入研究，其中 [15] 提出了学习三元值和三元赋值的方法。 Zhou et al. [16]提出了 INQ (Incremental Network Quantization)，在不降低网络精度的情况下，将神经网络的权值量化为 2^{-n} 的形式。 Wang et al. [17]提出了定点分解网络 (Fixed-point Factorized Network, FFN)，使用定点分解的方式对神经网络权值进行三元化。这些方法可以在 ImageNet 上达到与全精度相当的精度，但是，这些工作只对权值进行量化，而使激活保持浮点格式。

除了对神经网络权值进行量化外，也有许多研究对激活进行量化。通过将权重和激活转换为低比特格式，网络计算需要使用定点操作，从而更有效地节约资源。 Hubara et al. [18]中提出的二值神经网络 (binarized neural network, BNN)，在 CIFAR-10 这样的小数据集上达到了跟浮点数类似的精度。 Rastegari et al. [12]提出了另一个名为 xnorn-net 的网络，该网络将权值和激活都进行了二值化，在像 ImageNet 这样的大数据集上，xnor-net 比 BNN 更精确，但是，它的精度对比浮点数的情况仍然有很大的下降。 Zhou et al. [19]提出了 doref-net，研究了量化时比特位长度对权值、激活和梯度的影响。 Cai et al. [20]提出了 HWGQ (Half-wave

Gaussian Quantization) 的方法对权值和激活同时进行量化。与权值量化相比，激活量化通常会导致更大幅度的精度下降。对 AlexNet 和 VGG16 这样的大型网络进行量化，这些方法会导致精确度下降 5% 10%。因此，如何用低比特位来量化权值和激活仍然是一个具有挑战性的问题。

2.2.2　神经网络压缩模型

大规模的神经网络通常是过拟合的，过量的参数会严重影响神经网络的运算速度。因此，我们可以通过压缩神经网络模型的方式来缓解过拟合的情况，同时减少神经网络的存储需求和运算需求。神经网络压缩的方法可以分为两类，剪枝和矩阵分解。

Han et al. [21]提出了剪枝的策略来减少神经网络中权值的数量，在不影响神经网络准确性的前提下，能够将神经网络所需要的存储量和计算量减少一个数量级。剪枝策略首先对神经网络进行训练，找出那些重要的连接；接下来，修剪那些不必要的连接；最后对神经网络进行冲训练，微调剩余连接的权值。实验显示，剪枝策略能够将 AlexNet 网络的权值数量减少 9 倍，将 VGG16 网络的权值数量减少 13 倍。

在此基础上，Han et al. [22]进一步提出了 Deep Compression 用来深度压缩神经网络。Deep Comression 由三个阶段组成：剪枝（Pruning），量化（quantization）和霍夫曼编码（Huffman Coding），最终在不影响神经网络的精度的前提下将神经网络压缩了 35 倍到 49 倍。Deep Compression 首先通过剪枝阶段学习网络中重要的连接，删除不必要的连接，这个步骤能够减少 9 倍到 13 倍的权值。然后通过聚类算法将权值进行聚类，然后量化权值实现权值的共享，这个步骤能够减少表示每个权值的比特数，从 32 比特减少到到 5 比特。最后采用霍夫曼编码进一步无损压缩神经网络，这个步骤能够节省 20% 30% 的网络存储开销。在 ImageNet 数据集上，Deep Compression 能够将 AlexNet 网络压缩 35 倍，将网络规模从 $240MB$ 压缩到 $6.9MB$。同时，Deep Compression 将 VGG16 网络压缩了 49 倍，将网络规模从 552MB 压缩到 11.3MB。这将允许我们将网络模型存储在加速器的片上 SRAM 缓存中，而不需要在片上和片外之间反复搬运权值，从而减少片外访存开销。

Wang et al. [23]提出了一种新的有效的 CNN 压缩方法 CNNpack，它在频域对神经网络进行剪枝操作，因此它不仅能够关注小的权值，同时关注所有潜在的对计算结果影响小的连接。CNNpack 借助离散余弦变换（Discrete Cosine Transform，DCT）将空间域的权值变换到频域，然后采用聚类的方法将频域中的权重分解为公共部分和私有部分（残差）。在这两部分中，采用剪枝的策略丢弃大量的低能级的频率系数，能够在不显著降低精度的情况下产生高压缩比。在此基础上，

CNNpack 接着采用量化，霍夫曼编码和 CSR 存储的方法进一步压缩神经网络。最终实验实验显示，CNNpack 能够对 AlexNet 和 VGG16 分别压缩 35 倍和 49 倍。

除了以上对权值进行剪枝的策略，还有不少研究工作直接对神经元进行剪枝操作 [24-27]。但是对神经元直接进行剪枝操作会严重降低神经网络的精度，因此不能获得很低的稀疏度。Data-free Parameter Pruning [25] 在 AlexNet 和 LeNet5 上仅仅能够获得 65.11% 和 16.5% 的稀疏度，远远高于权值剪枝策略 [21] 的 11% 和 8%。Network Trimming [26] 能够在 VGG16 和 LeNet5 上获得 37% 和 26% 的系数率，也高于 [21] 中的 7.5% 和 8%。

除了经典的剪枝策略（包括权值剪枝和神经元剪枝），还有一部分的研究工作基于矩阵分解和因式分解 [28-30]，这些方法可以保持原始模型的规则密集计算结构，从而在通用处理器上实现压缩和加速。 Denton et al. [28]是利用神经网路线性结构的特性，对神经网络进行适当的低秩分解（low-rank approximation），在精度损失在 1% 的情况下，在很多模型上都能获得超过 2 倍的加速。 Jaderberg et al. [29]利用将 $k \times k$ 的卷积核分解为 $k \times 1$ 和 $1 \times k$ 的卷积核，从而加速卷积计算。同时 [29] 提出了两种优化方案：一种是利用滤波重构的方法最小化滤波权值误差，另一种是利用数据重构最小化响应误差的。 Lebedev et al. [30]则采用 CP decomposition 的方式对大规模神经网络的卷积核进行分解和加速，实验结果显示，在精度损失低于 1% 的情况下，在 AlexNet 网络能够获得 4 倍的加速效果。

2.3　神经网络加速器

2.3.1　现有神经网络加速器架构

由于严峻的能耗约束和高性能要求，定制加速器成为 CPU 和 GPU 等传统处理平台的替代品。近几年出现了数据流和结构各异的神经网络加速器。神经网络加速器按照数据流的形式可以分为两类，分别是基于向量算子数据流的加速器和和基于乘加算子（multiply-and-accumulate，简称 MAC）空间数据流的加速器。基于向量算子的加速器将神经网络的运算转化为向量运算，主要是将矩阵操作转化为

表 2.1　现有神经网络加速器的数据流形式

数据流	加速器
基于向量算子的数据流	DianNao [31], DaDianNao [32], PuDianNao [33], Cambricon [34], Cambricon-X [35], Cnvlutin [36], EIE [22], ESE [37]
基于乘加算子的空间数据流	ShiDianNao [38], Eyeriss [39], TPU [40], Neuflow [41] SCNN [42]

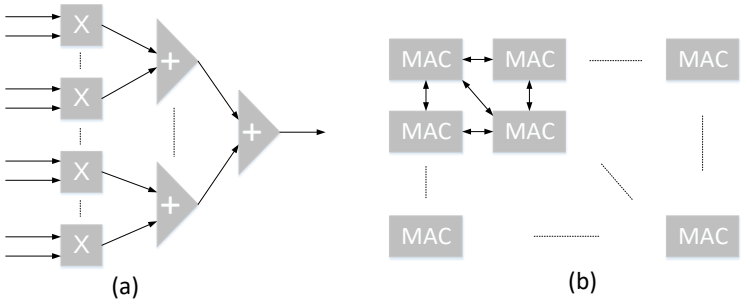


图 2.6　现有的加速器架构.

一系列的向量操作（通常是内积操作）。如图 2.6 (a) 所示的结构中，n 个乘法器和一个 n 输入的加法树就能够完成两个 n 维向量的内积操作。基于乘加算子的加速器（如图 2.6（b））通常具有一系列的规则分布的处理单元，其中每一个处理单元能够完成一次 MAC 操作。处理单元之间采用一种规律的方式进行连接（如二维矩阵），神经元和权值采用某种规律的方式在处理单元之间进行移动，如经典的脉动阵列形式（systolic design），因此这种形式又称为空间数据流形式。表 2.1总结了近年来出现的神经网络加速器和它们的数据流形式。基于向量算子的神经网络加速器包括 DianNao [31],DaDianNao [32],PuDianNao [33],Cambricon [34],Cambricon-X [35],Cnvlutin [36], EIE [22] 和 ESE [37]。基于乘加算子空间数据流的加速器包括 ShiDianNao [38], Eyeriss [39], TPU [40], Neuflow [41] 和 SCNN [42]。

2.3.2　基于向量算子的神经网络处理器

DianNao 是世界上首个深度神经网络处理器，它的结构如图 2.7所示。DianNao 包含以下的主要模块：一个控制器 (Control Processor, 简称 CP)，一个神经功能单元 (Neural Functional Unit, 简称 NFU)，三个片上缓存（NBin，NBout 和 SB）和外部存储接口 (Memory Interface)。CP 采用自定义的指令控制 DianNao 的运行。NBin，NBout 和 SB 分别用来缓存输入神经元，输出神经元和权值。NFU 包含 T_n 个计算单元，每个计算单元包含 T_n 个乘法器和一个 T_n 输入的加法树。

表 2.2　现有支持稀疏的神经网络加速器的比较

	权值稀疏	神经元稀疏	注释
Eyeriss [39]	✗	✔	只能跳过计算，无法带来性能提升
Cambircon-X [35]	✔	✗	
Cnvlutin [36]	✗	✔	
EIE [22]	✔	✔	只适合全连接层
ESE [37]	✔	✗	只适合稀疏的 LSTM 层
SCNN [42]	✔	✔	需要额外计算部分和的坐标

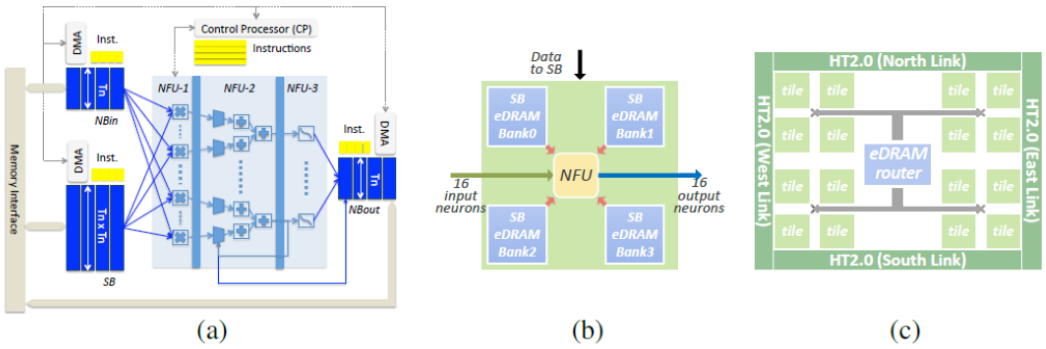


图 2.7　（a）DianNao 加速器结构。（b）DaDianNao 一个 tile 的结构。（c）DaDianNao 一个 node 的结构

在计算过程中每个计算单元共享输入神经元，接收不同的权值，从而计算不同的输出神经元。因此 NFU 最多读取 T_n 个输入神经元，T_nxT_n 个权值，计算 T_n 个输出神经元。在 TSMC65nm 工艺下，采用 T_n=16 的配置，DianNao 的面积，功耗和吞吐率分别为 $3.02mm^2$，$485mW$ 和 $452GOP/s$，对比与一个 2GHz 的 CPU，能够获得 117 倍的性能提升，并且降低 21 倍的能耗。

DadianNao 是 DianNao 的多核版本，一个 NFU 与外围的存储模块构成一个 tile，多个 tile 通过 fat-tree 进行局部互联组成一个 node，最终多个 node 通过 HyperTransport2.0 互联形成 DaDianNao。DaDianNao 使用 eDRAM 来提供足够的片上缓存用来存储权值和神经元，从而减少片外访存开销，进而有效地处理大规模神经网络。实验显示，在 ST28nm 的工艺下，64 核的 DaDianNao 的面积和功耗分别是 $67.73mm^2$ 和 $15.97W$，对比 Nvidia K20，平均性能提升 21.38 倍，平均能耗降低了 330.56 倍。

PuDianNao 是一个机器学习处理器，它不仅支持神经网络算法，而且支持 K 最近邻算法 (K-Nearest Neighbors，简称 K-NN)，K 均值算法 (K-Means)，线性回归 (Liner Regression, 简称 LR)，支持向量机 (Support Vector Machine，简称 SVM)，朴素贝叶斯 (Naive Bayes，简称 NB) 和决策树 (Classification Tree, 简称 CT) 等多种机器学习算法。PuDianNao 深入分析上述机器学习算法，并且提取出机器学习中的核心运算，其中包括向量内积（LR、SVM 和 DNN), 距离计算（K-NN 和 K-Means），计数（CT 和 NB），排序（K-NN 和 K-Means），非线性函数（如 sigmoid 和 tanh）等。PuDianNao 的核心模块是机器学习单元（Machine Learning Unit，简称 MLU）, 如图 x 所示，MLU 包括六个流水级，分别为 Counter，Adder，Multiplier，Adder Tree，Acc 和 Misc，通过这六个流水级的组合，PuDianNao 能够完成机器学习的核心运算。实验显示，在 65nm 的共一下，PuDianNao 的面积和功耗分别为 $3.51mm^2$ 和 $596mW$ 分为对比 NVIDIA K20 能够提高 1.2 倍的性能并且减少 129.41 倍的能耗。

然而随着神经网络算法的飞速发展，一些新的层类型和新的操作不断涌现，如 ROI Pooling 层，Deconvolution 层等，这又迫使研究者开发新的加速器和指令集来支持这些新的层和操作。为了进一步提高神经网络加速器的通用性，Liu 等人提出了针对神经网络处理器的指令集 Cambricon。Cambricon 的主要思想是为神经网络加速器提供一系列的基本算子（即指令），然后通过这些基本算子逐步搭建完成神经网络的运算。Cambricon 中集成了控制，数据传输，运算和逻辑操作这四种不同的指令，其中运算指令进一步分为矩阵运算，向量运算和标量运算指令，研究者只需要基本指令就能完成神经网络的运算。基于 Cambricon 指令集设计的加速器能够比 DaDianNao 拥有更强的通用性，在 10 个 benchmark 中，DaDianNao 只能支持其中的三种，而基于 Cambricon 的加速器能够全部支持。同时对比与 Intel Xeon E5-2620 和 NVIDIA K40，加速器能够分别获得 91.72 倍和 3.09 倍的性能提升。

2.3.3　基于乘加算子空间数据流的神经网络处理器

ShiDianNao 是面向嵌入式设备的神经网络处理器，实现端到端的神经网络应用。ShiDianNao 的架构如图 x 所示，ShiDianNao 与 DianNao 最大的不同是 NFU 模块。ShiDianNao 的 NFU 是一个大小为 $P_x \times P_y$ 的二维处理单元阵列 (Processing Elements, 简称 PEs)，它采用脉动阵列的形式完成神经网络矩阵向量运算操作。每个 PE 在每个周期能够完成卷积层、全连接层的一次乘法和一次加法，或者平均池化层的一次加法或者最大池化层的比较操作。每个 PE 能够从右方邻居和下方邻居读取神经元，并且能将部分和传播给相邻的 PE，值得注意的是权值通过广播的形式传送给 PE。这种方法能够充分复用神经元和权值，从而提高性能并降低访存能耗。在 $P_x = P_y = 8$ 的配置下使用 65nm 工艺，ShiDianNao 的主频，面积和功耗分别为 $1GHz$,$4.86mm^2$ 和 $320.1mW$，对比 GPU 和 DianNao 能够分别获得 30 倍和 1.87 倍的加速比，减少 4700 倍和 60 倍的能耗。除了 ShiDianNao，Farabet 等人提出的 Neuflow,Chen 等人提出的 Eyeriss 和谷歌的 TPU(tensor processing unit) 均采用二维网格的形式排列处理单元，并且采用脉动阵列的形式完成神经网络运算。其中 Google 的 TPU 主要用来加速其第二代人工智能系统 TensorFolow 的运行，并且相比于 K80 能够有 15 倍的性能提升。

2.3.4　稀疏神经网络处理器

虽然上述加速器能够以低能耗实现高吞吐量，但它们不能利用现代压缩神经网络的稀疏性和不规则性。最近出现了一些能够支持稀疏的神经网络加速器 [22, 35-37, 39, 42]，但它们都有各自的优缺点，如表 2.2所示。

Eyeriss [39] 应用游程压缩方法（run-length-compression,RLC）对稀疏神经元进行编码，具体来说，它在架构中加入 RLC Encoder 对输出神经元进行压缩，同时采用 RLC Decoder 对输入神经元进行解压缩，从而减少访问 DRAM 的数据量，进而减少 DRAM 的带宽需求和访存能耗。同时 Eyeriss 在 PE 单元中加入控制门，当输入神经元为 0 时，PE 单元将被关闭，这样能够跳过不必要的计算，从而减少计算能耗。然而，这两种方法仅仅能够带来能耗的减少，不会带来性能增益。

Cambricon-X [35] 能够充分挖掘稀疏权值带来的收益，包括减少能耗和提升性能。Cambricon-X 使用步长索引的形式压缩静态的权值，从而减少片外和片上访存能耗开销。同时 Cambricon-X 利用索引模块（indexing module，IM）来挖掘稀疏的特性,IM 能够通过稀疏的权值位置信息过滤掉不需要参与计算的神经元。IM 模块的结构如图 x 所示，首先,IM 模块连续地累加权值索引表的值（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，然后采用 MUX 的逻辑就能选出与非零权值对应的神经元（即 n1,n2,n5,n7）。Cambricon-X 对比不支持稀疏特性的 DianNao 提高了 7.23 倍的性能，并减少了 6.43 倍的能耗，但是 Cambricon-X 并不能挖掘稀疏神经元带来的收益。

Cnvlutin [36] 能够利用动态神经元稀疏筛选出需要进行计算的权值，从而获得 1.37 倍的加速比，但是 Cnvlutin 不能利用权值稀疏特性。ESE [37] 是实现在 FPGA 上的加速器，它面向的是稀疏的 LSTM 模型，并不适用于稀疏的 CNN 模型；由于 LSTM 模型中使用 Tanh 作为激活函数，因此不存在神经元稀疏的特性，所以 ESE 仅仅能够挖掘权值稀疏的特性。

以上工作只能挖掘权值稀疏性或者神经元稀疏性，但不能同时从两者中受益。EIE [22] 采用稀疏矩阵的行压缩形式（compressed sparse row，简称 CSR）存储稀疏的权值，并且通过 LZND（leading non-zero detection）筛选出非零的神经元，使得使得 EIE 能够同时利用神经元稀疏性和权值稀疏性，对比与 DaDianNao 能够提高 2.9 倍的性能，减少 19 倍的能耗并且缩小 3 倍的面积。但是这种架构仅仅针对全连接层的计算，并不能针对卷积层进行计算。

SCNN [42] 能够同时利用神经元稀疏性和权值稀疏性。SCNN 中的计算单元构成了一个二维网格的形式，每个计算单元执行非零神经元和非零权值的乘法操作，同时计算非零乘积的坐标。非零乘积通过坐标在二维计算网络进行路由，最终被分配给对应的累加器阵列，每个非零乘积通过读取修改写入操作对包含部分和的本地 RAM 执行累加，最终获得输出神经元。但是这种架构需要不断计算坐标，大大增加了计算成本和存储成本。因此 SCNN 在处理稠密神经网络时的会损失 21% 的性能，同时增加 33% 的能耗；在处理稀疏神经网络时仅仅能够增加 2.7 倍的性能，减少 2.3 倍的能耗。同时 SCNN 对全连接层和规模为 1x1 的卷积层支持并不理想，在这两种类型的层时时只能利用 20% 的乘法器资源。

2.4　脚注

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. [①]

[①] This is a long long long long long long long long long long long long long long long long long long long long long long long long footnote.

第 3 章　一种新的神经网络压缩方法

3.1　背景

现代神经网络由多种类型的层组成，输入在这些层中不断被传递和处理，从而被分类或识别。在每一层中，每一个神经元接收多个输入进行处理，并将输出通过突触送到下一层。目前主流的神经网络包括卷积神经网络神经网络 (CNNs)，深度神经网络 (DNNs) 和循环神经网络（RNNs）。目前深层神经网络已经在图像识别，目标检测，语音识别，自然语言处理，视频处理领域有了越来越广泛的应用。虽然这些神经网络非常强大，但是大规模的神经网络参数（包括神经元和权值）对存储和带宽提出了很高的需求，例如，AlexNet 的网络规模超过 $200MB$，VGG-16 的网络规模超过 $500MB$，这使得这些大规模的深度神经网络难以在嵌入式设备，特别是移动设备上进行部署。

首先，对于百度和 Facebook 等公司，手机端的各种应用程序更新都是通过不同的应用程序商店进行下载，它们是对二进制文件的大小非常敏感。例如，应用程序商店会限制当应用程序大于 $200MB$ 时，必须连接无线网络才能够进行下载。因此，当应用超过 $200MB$ 时，将会收到比 $10MB$ 应用更多的安全审查。尽管在手机端本地运行深度神经网络能够减少网络带宽，进行任务实时处理，获得更好的隐私保护，但是这些大规模的神经网网络需要耗费巨大的存储开销，因此阻碍了深度神经网络被纳入移动应用程序。

第二个问题是能耗消耗。运行大型神经网络需要大量的带宽和计算资源来获得神经网络数据，并进行计算，这个过程需要耗费巨大的能耗。考虑到移动设备电池容量的限制，高能耗的应用，如深层神经网络将难以部署。能源消耗主要是内存访问。在 45 纳米工艺下，一个 32 位浮点加法需要消耗 $0.9pJ$ 的能量，一次 32 位 SRAM 缓存访问耗费 $5pJ$ 的能量，而一次 32 位 DRAM 内存访问需要 $640pJ$ 的能量，这个能耗几乎是浮点加法操作的 3 个数量级。大型神经网络无法缓存在片上的 SRAM 中，因此需要能耗更高的 DRAM 访问。例如我们以 $20fps$ 运行一个拥有 10 亿个连接的神经网络，那么需要为 DRAM 访存提供至少 $P = (20Hz) \times (640pJ) \times (1G) = 12.8W$ 的功率，这个已经完全超过了大多数手机电池能够提供的功率。

3.1.1　神经网络中的稀疏特性

虽然现代神经网络在许多应用中占主导地位，但是庞大的参数给计算、内存容量和内存访问带来了沉重的负担。为了应对这些挑战，研究者从各个方向做出

了各种各样的努力，包括算法级 (例如 Drop out [43]、Sparsity [21]), 架构级 (例如短位宽运算 [44] 甚至是 1 比特运算 [12]、非精确计算 [45]) 和物理级 (例如，动态电压调整技术 [46])。其中，稀疏技术是目前解决这个问题最有效的方法。研究人员在早期工作中已经证明了稀疏性是解决过拟合问题的有效方法。对于现代神经网络，研究者们提出了一系列的训练技术，如稀疏编码 (Sparse Encoder) [47]，自动编解器 (Auto Encoder/Decoder) [48-49]，深度信念网络 (Deep Belief Network, DBN) [50] 等，在步影响精度的前提下修剪冗余的突触和神经元。最近，研究人员提出了一种新的剪枝方法，在 CNNs 上获得了非常理想的稀疏性，在 AlexNet [1] 网络上获得了 88.85% 的稀疏度，在 VGG16 [51] 网络上获得了 92.39% 的稀疏度。在表 3.4，我们报告详细的剪枝结果，这里，我们定义稀疏度 (sparsity) 为被修剪的神经元/突触占总神经元/突触的比例，同时我们定义稠密度（density）为非零神经元/突触占总神经元/突触的比例。

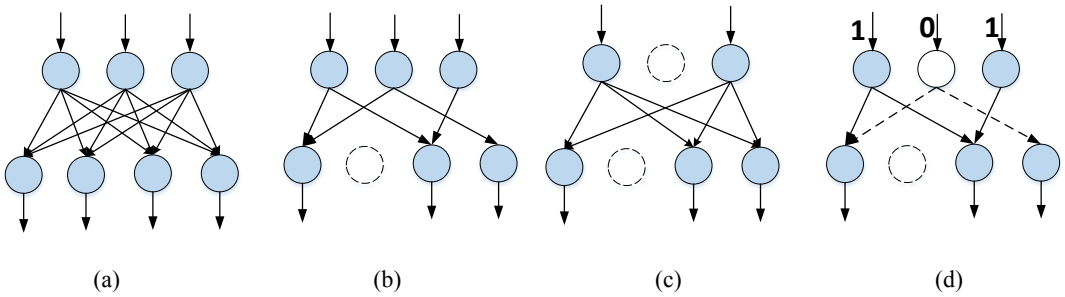


图 3.1　(a) 稠密神经网络 (b) 静态权值稀疏 (c) 静态神经元稀疏 (d) 动态神经元稀疏

如图 3.1所示，我们将神经网络中的稀疏分为两类: *静态稀疏 (static sparsity)* 和*动态稀疏 (dynamic sparsity)*。*静态稀疏*源于剪枝操作，当突触 (图 3.1 (b)) 或者神经元 (图 3.1 (c)) 满足某些条件时，它们将从神经网络的拓扑结构中被永久剪除，因此这种稀疏又被称为*静态稀疏*。当神经元的输出值为 0 时 (图 3.1 (d)) 会发生*动态稀疏*，特别是当使用 ReLU 作为激活函数时，在 ReLU 激活之前结果为负的神经元输出值为 0。形式上，ReLU 用如下公式计算输出神经元

$$ReLU(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leqslant 0 \end{cases} \tag{3.1}$$

其中 x 为 ReLU 激活函数的输入，yR 为 ReLU 的输出。因此，像 ReLU 这样的激活函数可以在网络的神经中中引入许多 0。不同于 *静态稀疏*，*动态稀疏*不改变网络拓扑结构，它与输入密切相关，因为不同的输入将导致不同的计算结果，从而导致不同的动态稀疏结果。值得注意的是，*静态稀疏*又可以分为*静态权值稀疏 (static synapse sparsity, SSS)* 和 *静态神经元稀疏 (static neuron sparstiy, SNS)*，*动态稀疏*仅由*动态神经元稀疏 (dynamic neuron sparsity, DNS)* 组成。

3.1.2　权值编码

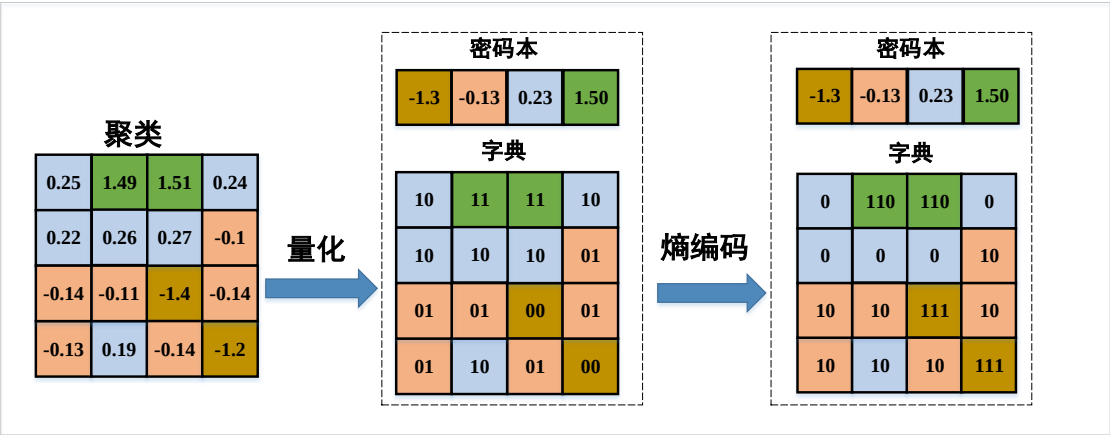


图 3.2　权值编码过程

为了进一步压缩权值数据，研究人员将图像压缩技术，如量化 (quantization) [52]，熵编码 (entropy coding) [53] 等应用到神经网络的领域。Deep Compression [54] 运用细粒度剪枝，全局量化和霍夫曼编码对神经网络进行深度压缩，在基本不损失精度的情况下，在 AlexNet 网络上获得 35 倍的压缩比。另一种有效的压缩方案是 CNNpack [23]，它在频域中对神经网络进行深度压缩，最终在 AlexNet 网络上获得了 40 倍的压缩比。我们在图 3.2中描述了权值编码的过程，它由量化和熵编码两个步骤组成。

1. 量化

首先，我们利用聚类算法 (例如，K-means 聚类) 将分散的权重聚集成 K 簇。在图中，16 个权值被分为 4 个簇，其中 (0.25, 0.24, 0.22, 0.26, 0.27, 0.19) 这六个权值被聚为一个簇。然后我们计算每个簇的中心值，该值与簇中所有权重的总距离最小，每一个权值都用它所在簇的质心值来近似表示，例如在图中，(0.25, 0.24, 0.22, 0.26, 0.27, 0.19) 这六个权值都用 0.23 表示。因此，我们可以用一个密码本和一个字典来表示所有权值，其中密码本中记录了 K 个簇中的中心值，字典中每个元素只需要 $log(K)$ 位来索引密码本从而获得权值。

2. 哈弗曼编码

由于密码本中元素的出现概率是不平衡的，因此我们可以采用熵编码 (如哈弗曼编码) 进一步降低权重的比特数。哈弗曼编码方法完全依赖于码字出现的概率来构造整体平均长度最短的编码方法。进行哈夫曼编码的关键步骤是建立符合哈夫曼编码规则的二叉树，该树又称作哈夫曼树。哈夫曼树是一种特殊的二叉树，其终端节点的个数与待编码的码元的个数等同，而且每个终端节点上都带有

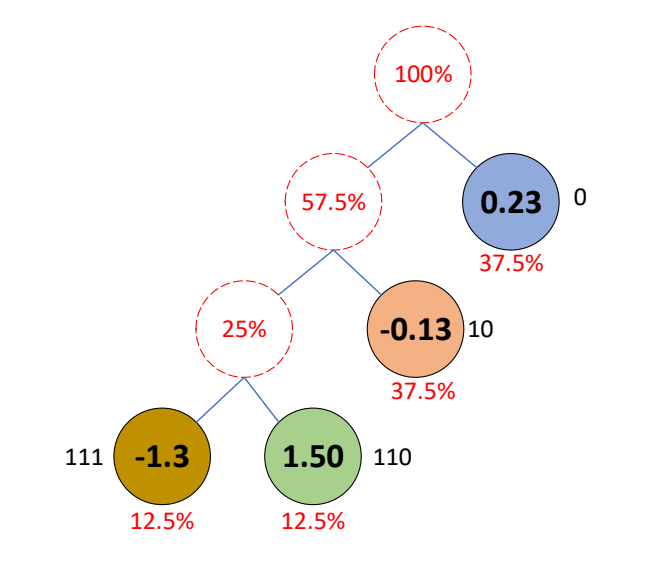


图 3.3　哈弗曼树

各自的权值。每个终端节点的路径长度乘以该节点的权值的总和称为整个二叉树的加权路径长度。在满足条件的各种二叉树中，该路径长度最短的二叉树即为哈夫曼树。在使用哈夫曼编码执行对码元的实际编码过程时，通过递归地合并概率最小的码元来构建哈夫曼树。以图 3.2为例，四个码元-1.3，-0.13，0.23，1.50出现的概率分别是 0.125，0.375，0.375，0.125，最终可以构造出如图 3.3所示的哈弗曼树。在哈夫曼树构建完成后，便可以得到每一个码元的哈夫曼编码的码字。具体方法是：从哈夫曼树的根节点开始遍历，直至每一个终端节点，当访问某个节点的左子树时赋予码字 1，访问右子树时赋予一个码字 0（反之亦然），直到遍历到终端节点时这一路径所代表的 0 和 1 的串便是该码元的哈夫曼编码码字。最终四个码元-1.3, -0.13, 0.23, 1.50 分别被编码成为是 111，10，0，110。

3.1.3　不规则性

尽管稀疏神经网络能够在理论上减少计算量，存储量和数据传输量，但是稀疏会导致原本规则的神经网络计算模式转变为不规则的形式。然而，目前的处理平台由于存在诸多问题，无法有效地处理稀疏性。根据 [35]，CPU 和 GPU 并不能很好地支持稀疏神经网络运算，即使使用稀疏矩阵运算库，如 sparseBLAS 或者 cuBLAS，也不能取得非常理想的加速效果，在某些情况下处理稀疏网络的性能甚至比处理稠密神经网络的性能还要差。尽管目前又不少支持稀疏神经网络的加速器 [22, 35-37, 42]，但是它们仍然不能有效地处理稀疏神经网络带来的不规则性。他们需要显著的稀疏成本, 而且不能完全支持所有稀疏类型，例如 Cambricon-X [35] 中支持稀疏的逻辑占总面积的 31.07%，占功耗的 34.83%，而且只能支持权值稀疏。因此，我们需要一种新的神经网络稀疏方法，能够在保持

高的精度和稀疏度条件下尽可能保持神经网络的规则性，减少稀疏神经网络的不规则性，有利于 CPU,GPU 或者加速器能够充分利用稀疏带来的收益。

3.2　局部收敛 (local convergence)

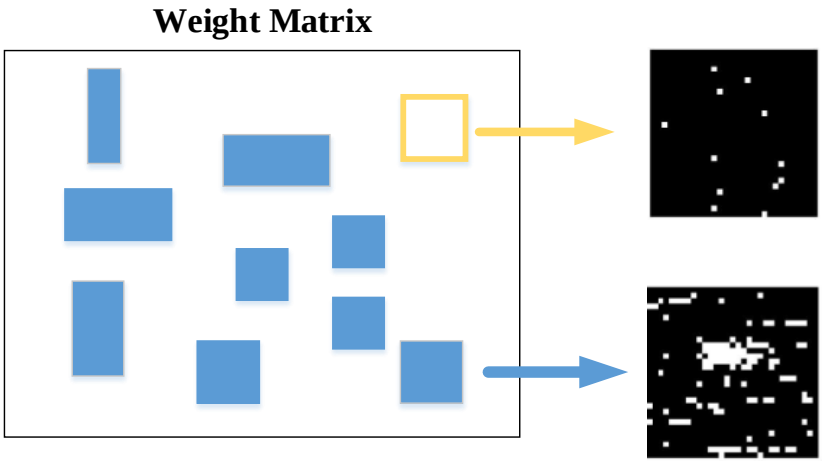


图 3.4　全连接层中的局部收敛现象（白点表示大权值，绝对值大于其余 90% 的权值）

传统的剪枝方法将每一个突触当作独立不相关的元素，当突触符合特定的剪枝条件时（如权值的绝对值小于某一个阈值），该突触将会被剪除，但是这种方式忽略了突触之间的潜在关系。我们充分分析了神经网络中的权重分布情况，观察到了一个有趣的现象。如图 3.4所示，我们使用权值矩阵来表示全连接层的权值，每一行包含与对应输入神经元相连接的所有权值，每一列包含与对应输出神经元相连接的所有权值；我们可以发现，绝对值大的权值往往会聚集成簇，我们将这种现象称为局部收敛（local convergence）。我们利用如下的方式证明局部收敛的存在。首先，我们在权值矩阵上设定一个大小为 k 的滑动窗口，然后将该滑动窗口沿着权值矩阵空间维度上进行滑动（全连接层权值矩阵为两个维度，卷积层权值矩阵为四个维度），同时统计窗口中较大权值的数量，如果某一个权值的绝对值大于剩余 $m\%$ 的权值，我们将其定义为较大的权值。为了方便阐述，我们将一个具有 x 个大权值的窗口标记为 x 窗口。我们选择了 5 个代表性的层: AlexNet 网络的 *fc6* 层, VGG16 网络的 *fc6* 层, MLP 网络的 *ip1* 层, LSTM 网络的 W_{ix} 层, AlexNet 网络的 *conv2* 作为驱动示例。我们将前四层网络的滑动窗口大小设为 $k = 4$, *conv2* 层网络的滑动窗口大小设为 $k = 2$，此外，我们将 $m = 90$ 设置为较大权值的阈值。在图 3.5中，我们描绘了一个随机初始化层与训练后的 5 个层中较大权值的累计分布。我们观察到，在初始化层中，同一个窗口中较大权值的数量不会超过四个。但是，这五个训练过的层存在同一个窗口中较大权值的数量超过七个的情况。因此，较大权值在训练过程中趋向于聚集在一起，也就

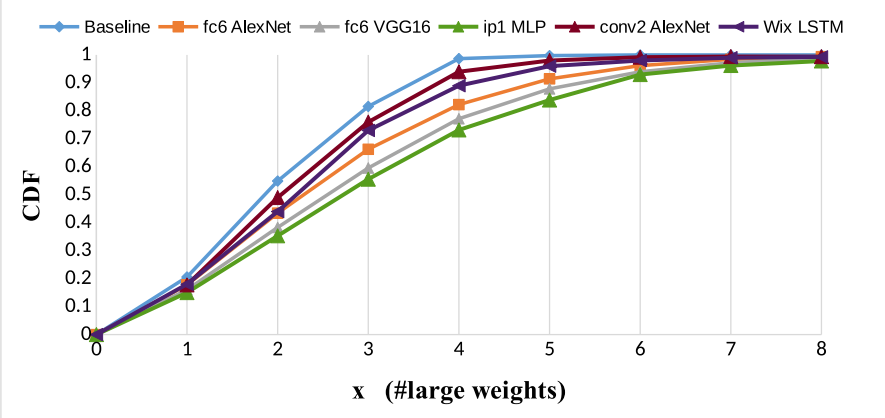


图 3.5　较大权值的累计分布

是局部收敛。

3.3　压缩神经网络

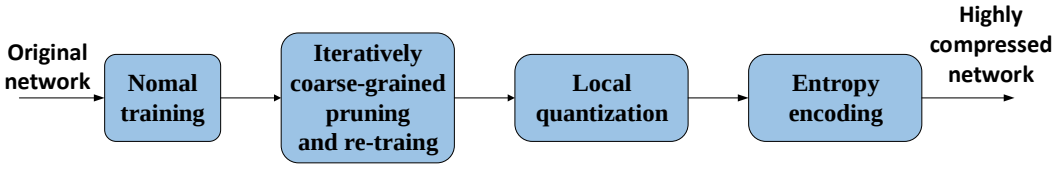


图 3.6　新的压缩神经网络步骤

如图 3.6所示，我们提出了一种新的压缩神经网络的方法，整个流程包括三个步骤：粗粒度剪枝，局部量化和熵编码。下面我们将依次详细阐述这三个步骤。

3.3.1　粗粒度剪枝

我们提出了一种新的剪枝策略：粗粒度剪枝。粗粒度剪枝的核心思想基于神经网络局部收敛的特征，我们将多个突触同时进行剪枝操作，而不是对单个权值进行剪枝。我们首先将权值矩阵分成多个块，如果某一个符合特定的条件，这个权值块将在网络拓扑中被永久剪除，然后我们使用较小的学习率对网络进行重训练，从而保持网络的准确性。值得注意的是，我们在训练中迭代地应用粗粒度剪枝和重训练，以获得更好的稀疏性，同时避免精度损失。为了清晰地解释粗粒度剪枝技术，我们使用全连接层 3.7和卷积层 3.8作为驱动示例。

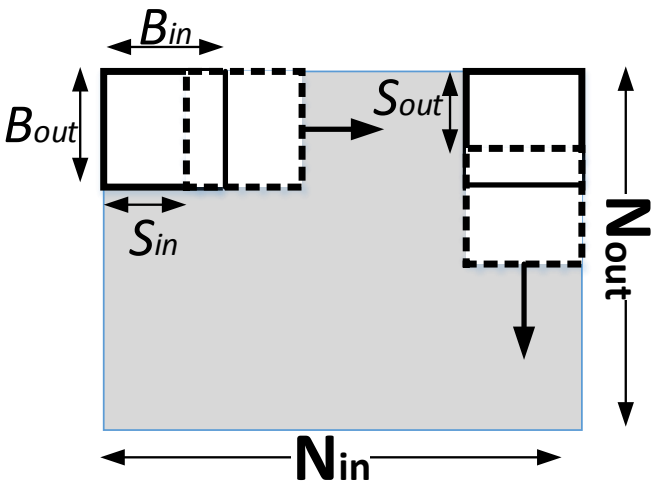


图 3.7　全连接层上的粗粒度剪枝

1. 剪枝策略

在一个全连接层中，(N_{out}) 个输出神经元通过二维权值矩阵 (N_{out}, N_{in}) 与 (N_{in}) 个输入神经元，如图 6 所示。在剪枝过程中，大小为 $B_{in} \times B_{out}$ 的滑动窗口沿着权值矩阵的两个维度分别按照 S_{in} 和 $_{out}$ 的步长进行滑动。一旦滑动窗口中的权值某些条件，窗口内的突触都会同时被剪枝。在迭代剪枝过程中，滑动窗口将跳过修剪过的突触，使所有修剪过的突触块都具有相同的规模大小，从而简化索引过程。

对于卷积层，输出特征图中的输出神经元通过共享的突触连接到输入特征图中的神经元。因此，卷积层中的权值可以表示为一个四维张量，即 $(N_{fin}, N_{fout}, K_x, K_y)$，其中 $Nfin$ 是输入特征图的数量，$Nfout$ 是输出特征图的数量，和 Kx,K_y 是卷积核的大小。如图所示，在剪枝过程中，大小为 $B_{fin} \times B_{fout} \times B_x \times B_y$ 的滑动窗口沿着四维的权值张量分别按照 S_{fin}，S_{fout}，S_x 和 S_y 的步长进行滑动。

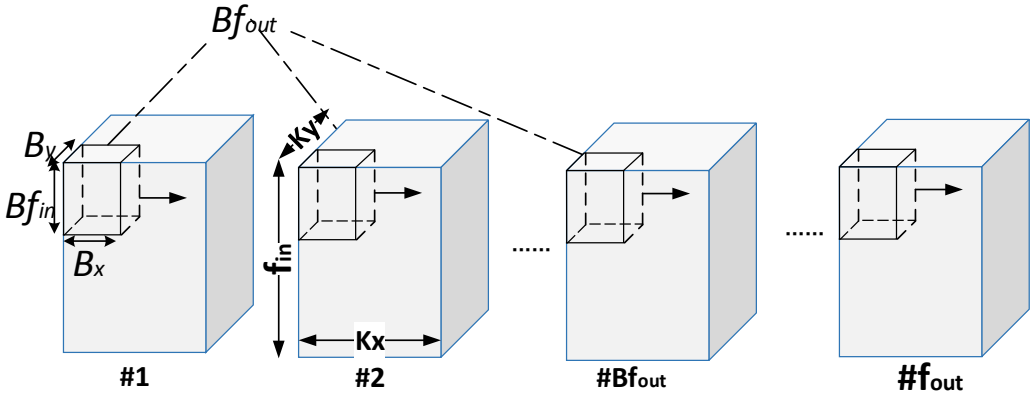


图 3.8　卷积层上的粗粒度剪枝

2. 剪枝块大小

在粗粒度剪枝过程中，寻找最佳的剪枝块大小来权衡压缩比与精度之间的平衡至关重要。使用过大的块进行剪枝将导致突出连接无法完全表达神经网络的特性，从而降低准确性。使用过小的块进行剪枝，并不能充分利用局部权值收敛的特性，从而无法获得很高的压缩比。值得注意的是，如果我们将剪枝块的大小设定为 1，那么粗粒度的剪枝就是 [21] 中的细粒度剪枝。

在本文中，我们仔细权衡神经网络中不同类型层的块大小来修剪不同的网络。理论上，我们应该为不同的层选择剪枝块大小，而不是为不同类型的层选择剪枝块大小，但是考虑到在庞大的设计空间搜索和极长训练时间，我们关注的是为网络中不同类型的层设定剪枝块大小。例如，由于卷积层中的权值比全连接层的权值更敏感，因此我们只在卷积层的某个维度上进行粗粒度剪枝。我们以 AlexNet 为作为驱动实例，在保持网络准确性的同时，阐述不同剪枝块的大小对神经网络稀疏度和压缩效果的影响。如表 3.1所示，我们将 AlexNet 网络的基准精度设定为 top-1 误差 42.8%，同时我们使用 $(1, N, 1, 1)$ 和 (N, N) 的剪枝块分别对卷积层和全连接层进行粗粒度剪枝。在表 2 中，我们将卷积层和全连接层的权值分别用 8 位和 4 位进行量化，然后用 Huffman 编码进行编码。当 N 从 1 增加到 64 时，压缩比 r_c 首先从 40 倍迅速增长到 79 倍，然后迅速下降到 65 倍。当剪枝块从 1 增大到 16 时，压缩比随之增加，主要原因是非零权值索引所需的存储空间不断减少。当剪枝块大小从 16 增加到 64 时，压缩比随之下降，这是因为为了保持相同的精度，神经网络的稀疏度不断降低（卷积层稀疏度从 64.75% 降低到 54.45%），从而阻碍了压缩比。为了更好地平衡压缩比和精度，我们将卷积层中的剪枝块大小设置为 $(1, 16, 1, 1)$，将 fc6,fc7 和 fc8 的剪枝块大小分别设定为 $(32, 32)$,$(32, 32)$ 和 $(16, 16)$。值得注意的是，使用粗粒度剪枝，非零权值索引所需要的存储空间只需要 $29.38KB$，相比于使用细粒度剪枝后 [54] 的索引（$2.95MB$）减少了 102.82 倍。

3. 最大值剪枝 vs. 平均值剪枝

在粗粒度剪枝过程中，我们可以使用两种不同的剪枝策略，第一种是最大值剪枝策略，第二种是平均值剪枝策略。对于最大值剪枝策略，如果窗口中具有最大绝对值的权值小于预定义阈值 (W_{th})，则将窗口中的所有权值剪除。对于平均值剪枝，如果窗口中所有权值绝对值的平均值小于预定义的阈值，则将窗口中的所有权值剪除。这两种剪枝策略的主要特性是完全不同的。最大值剪枝表明只有块内部最大的权重才会影响整个块的重要性。平均值剪枝表明，块内的所有权重将对块的重要性产生影响。我们使用这两种剪枝方法在 Cifar10 快速模型上进行

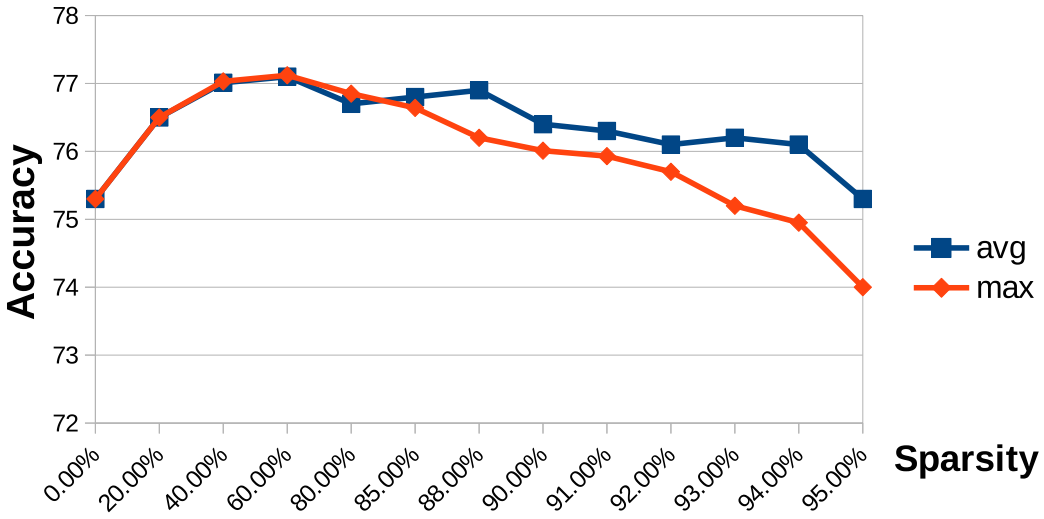


图 3.9　Cifar10 快速模型上的最大值剪枝和平均值剪枝

粗粒度剪枝和训练，在剪枝过程中我们控制这两种剪枝方法使用相同大小的滑动窗口和步长。图 3.9显示，当神经网络的稀疏度高于 85% 时，平均值剪枝比最大值剪枝精度更高。因此，本文选择了平均值剪枝的策略。

4. 不规则性的评估方法

为了更直观地表现粗粒度剪枝的效果，我们提出了一种简单而有效的方法来定量测量粗粒度剪枝减少的不规则性。我们使用如下公式计算

$$R(Irr) = JBIG(I_f)/JBIG(I_c) \tag{3.2}$$

其中 $R(Irr)$ 表示减少的不规则性，I_f 和 Ic 分别表示经过细粒度剪枝和粗粒度剪枝后稀疏神经网络的索引，$JBIG()$ 表示基于 Joint Bi-level Image Experts-Group [55] 的无损二值图像压缩标准。这种方法基于这样一个事实: 常规数据 (特

表 3.1　不同剪枝块大小情况下 AlexNet 网络的稀疏度和压缩比（C: 卷积层；F：全连接层; S: 稀疏度; r_c 压缩比）

N	1	2	4	8	16	32	64
C:S(%)	64.99	64.99	64.89	64.77	64.75	59.95	54.45
F:S(%)	89.99	89.98	89.98	89.97	89.95	89.91	87.78
Weight (MB)	2.86	2.86	2.87	2.87	2.91	3.01	3.59
Index (MB)	2.95	0.76	0.22	0.07	0.03	0.01	0.005
r_c	40×	64×	75×	79×	79×	77×	65×

别是二进制矩阵) 包含了更多的冗余信息，因此可以用更少的数据来表示。因此，我们将突触索引作为二进制图像，用 JBIG 压缩它们。压缩后的数据大小在某种程度上可以衡量数据的不规则性。因此，我们通过粗粒度剪枝与细粒度剪枝后压缩索引大小之比来度量降低的不规则性。

5. 神经元稀疏

在粗粒度的剪枝过程中，我们只修剪突触，并不会对神经元进行直接修剪，仅仅当某个神经元与其他神经元之间没有任何连接时，该神经元会被剪除，因此静态神经元稀疏的比例并不高。然而，在大型网络中，动态神经元稀疏占了很大的比例，即神经元经过激励后的输出值为 0 的情况。在表 x 中显示了在各个网络中，权值稀疏性，静态神经元稀疏性和动态神经元稀疏性。在大规模神经网络中，如 AlexNet、VGG16 和 ResNet-152，静态神经元稀疏性非常低，在卷积层中接近 0%，即基本不会出现静态神经元的情况。然而，在 AlexNet 和 VGG16 中，动态神经元稀疏性却很有前景，在 AlexNet 中为 37.63%，在 VGG16 为 59.48%，动态神经元稀疏为提高计算性能和节省能耗提供了很好的契机。

3.3.2　局部量化

应用权值共享和网络量化的策略可以有效地减少代表权值的比特位 [54]。为了更好地利用局部收敛的特性，我们提出了局部量化的策略，不同于全局量化在整个权值矩阵中进行权值共享，局部量化仅仅在权值矩阵的局部区域中进行权值共享。如图 3.10所示，局部量化策略首先将权值矩阵划分为两个子矩阵，然后对两个子矩阵分别进行聚类。在每个子矩阵中，权值将被编码成一个密码本和一个字典，其中字典中的每个权值只需要 1 比特进行索引 (使用图 3.2中的全局量

表 3.2　神经网络中的稀疏性（C: 卷积层; F：全连接层; SSS：静态权值稀疏; SNS: 静态神经元稀疏; DNS：动态神经元稀疏）.

–	LeNet5	MLP	Cifar10	AlexNet	VGG16	ResNet152
C: SSS (%)	88.98	–	92.08	64.75	64.83	45.69
SNS (%)	25.39	–	11.49	0.00	0.00	0.00
DNS (%)	0.00	–	30.61	37.63	59.48	50.30
F: SSS (%)	91.47	90.13	93.99	89.95	95.16	0
SNS (%)	47.70	40.25	57.73	15.75	22.18	0
DNS (%)	11.50	66.31	19.93	29.27	43.03	24.10

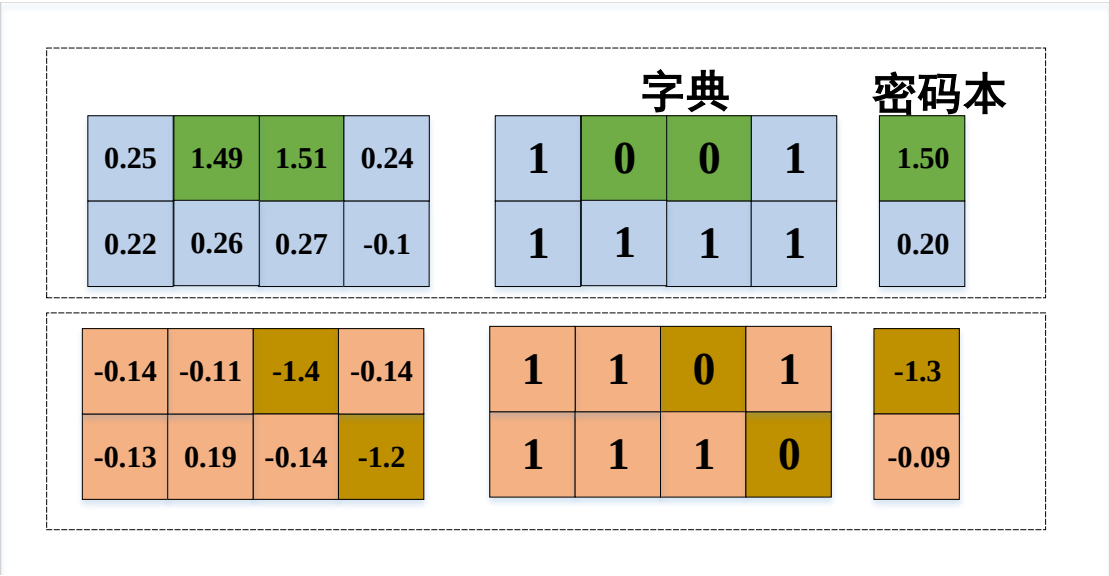


图 3.10　局部量化

子化方法，密码本中每个权值需要 2 比特进行索引)。与全局量化相比，局部量化能够利用局部收敛来进一步减少表示权值的比特数，从而获得更高的压缩比。值得注意的是局部量化的开销很小，以 AlexNet 网络的 fc6 层为例, 全局量化需要一个 $128B$ 的密码本和 $2.0MB$ 的字典，其中字典中每个元素需要 5 比特进行索引; 局部量化过程中我们将权值矩阵分为 64 个小矩阵，因此需要 64 个密码本和 64 个对应的字典，但是字典中每个元素仅仅需要 4 比特进行索引，因此密码本和字典的大小分别为 $4KB$ 和 $1.6MB$，对比与全局量化存储容量减小了 19.81%。

3.3.3　熵编码

熵编码是一种无损的数据压缩策略，它为输入中的每个符号创建并分配唯一的无前缀码。由于每个码字的长度都与该符号出现概率的负对数成比例，所以我们使用更少的比特来编码出现概率更高的符号。目前两种最常见的熵编码技术是 Huffman coding [56] 和算术编码 [57]。实验显示，熵编码能够进一步减少 $20\% - 30\%$ 的网络存储开销。

3.3.4　压缩实验结果

在表 3.3中，我们列出了七个神经网络的详细压缩结果，这七个神经网络包括 LeNet-5 [58], MLP [59] (三层神经网络，包含 300×100 个隐层神经元)，Cifar10 快速模型 [60],AlexNet [1],VGG16 [51],ResNet152 [61] 和 LSTM [62]。注意，我们只关注 LSTM 模型中的 LSTM 层。我们在表中详细地罗列了各个神经网络能够获得的稀疏度（Sparsity），经过三个压缩步骤（即粗粒度剪枝，局部量化，熵编码）后神经网络的规模（分别用 W_p，W_q 和 W_c 表示）和压缩比（分别用 r_p，r_q

表 3.3　压缩后神经网络的稀疏度，压缩比和不规则性减少量 (W_p: 粗粒度剪枝后权值规模; W_q: 粗粒度剪枝，局部量化后权值规模; W_c: 粗粒度剪枝，局部量化，熵编码后权值规模; L: *LSTM* 层; C: 卷积层; F: 全连接层; W: 权值; I: 权值索引; r_p: 粗粒度剪枝后压缩比; r_q: 粗粒度剪枝，局部量化后压缩比; r_c: 粗粒度剪枝，局部量化，熵编码后压缩比; $R(Irr)$: 不规则性减少量)。

Model	Sparsity(%)	$W_p(B)$	r_p	$W_q(B)$	r_q	$W_c(B)$	r_c	$R(Irr)$
Alexnet	C: 64.75	W: 25.65M	9×	W: 3.60M	64×	W: 2.90M	79×	101.65×
	F: 89.93	I: 36.73K		I: 36.73K		I: 29.38K		
VGG16	C: 64.83	W: 42.72M	12×	W: 7.80M	66×	W: 5.25M	98×	28.54×
	F: 95.16	I: 202.80K		I: 202.80K		I: 121.68K		
LeNet-5	C: 88.98	W: 135.96K	12×	W: 23.77K	66×	W: 19.01K	82×	8.87×
	F: 91.47	I: 1.67K		I: 1.67K		I: 1.39K		
MLP	C: –	W: 102.54K	10×	W: 19.23K	51×	W: 12.01K	82×	10.41×
	F: 90.13	I: 1.20K		I: 1.20K		I: 0.61K		
Cifar10	C: 92.08	W: 42.08K	13×	W: 8.68K	62×	W: 7.82K	69×	7.61×
	F: 93.99	I: 0.48K		I: 0.48K		I: 0.42K		
ResNet152	C: 45.69	W: 134.10M	1.7×	W: 33.50M	7×	W: 23.44M	10×	13.02×
	F: 0.00	I: 0.49M		I: 0.49M		I: 0.44M		
LSTM	L: 87.94	W: 1.50M	8.3×	W: 191.28K	60×	W: 152.47K	77×	50.51×
		I: 25.96K		I: 25.96K		I: 17.84K		

和 r_c 表示)，同时我们还罗列了粗粒度剪枝减少的不规则性（$R(Irr)$)。我们的压缩算法在 AlexNet，VGG16，LeNet-5, MLP, Cifar10, ResNet152 和 LSTM 网络上分别获得了 79 倍，98 倍，82 倍，92 倍，69 倍，10 倍和 77 倍的压缩比，造成了精度损失小于 0.2%，平均减少不规则度 20.13 倍。

在表 3.4中，我们将压缩方法与现有的两种最先进的神经网络压缩方法进行了比较，即 Deep Compression [54]（细粒度剪枝）和 CNNpack [23]（频域压缩)。我们的压缩方法获得的压缩比几乎是 Deep Compression 和 CNNpack 的两倍，获得这么高压缩比的主要原因是我们的压缩算法能够显著减少神经网络的不规则度，从而显著减少权值索引信息的存储空间。值得注意的是，我们的压缩算法除了能够获得高压缩比，还能获得与 Depp Compression 类似的稀疏度，同时精度损失几乎可以忽略不计。对于 ResNet 网络, 我们的压缩方法和 Deep Compression 只能获得不到 10 倍的压缩比，远远低于传统的神经网络。出现这种现象的原因是 ResNet 网络中包含多种新的特性，包括快捷连接（shortcut connections）和批

表 3.4　CNNPack, Deep Compression 与我们的压缩方法的对比 (S%: 稀疏度; r_c: 压缩比).

model	Ref	Deep Cmp. [54]			CNNpack [23]			Ours		
	Top1-E(%)	S (%)	r_c	Top1-E(%)	S (%)	r_c	Top1-E(%)	S (%)	r_c	Top1-E(%)
AlexNet	42.78	88.85	35×	42.78	–	39×	41.60 (41.80*)	88.97	79×	42.72
VGG16	31.50	92.39	49×	31.17	–	46×	29.70 (28.50*)	91.93	98×	31.33
LeNet-5	0.80	91.57	39×	0.74	–	–	–	91.40	82×	0.95
MLP	1.64	91.82	40×	1.58	–	–	–	90.13	82×	1.91
Cifar10	24.20	94.98	45×	24.33	–	–	–	92.93	69×	24.22
ResNet152	25.00	45.00	8×	24.40	–	–	–	44.17	10×	25.05
LSTM	20.23	88.47	35×	20.52	–	–	–	87.94	77×	20.72

处理归一化（batch normalization），这些特性从而大大缓和了神经网络的过拟合的现象。因此，我们在 ResNet 中只能获得一个非常有限的压缩比。

与表 3.4，我们将我们压缩算法的准精确与 Deep Compression 和 CNNpack 进行比较。我们可以看出，我们的压缩方法造成的精度损失是可以忽略不计的。与参考准确度相比，我们的压缩方法的精度损失低于 0.2%，类似于 Deep Compression (0.1%)，并且低于 CNNpack(0.7%)。

表 3.3还列出了粗粒度剪枝减少的不规则性。值得注意的是，粗粒度的剪枝可以将不规则性平均减少 20.13 倍，剪枝的块越大，不规则性减少的程度就越高。对于小型网络，如 LeNet-5、MLP 和 Cifar10，减少的不规则性平均是 8.80 倍，因为我们只能用比较小的剪枝块对小的神经网络进行修剪，减少精度损失。相反，大型网络，如 AlexNet、VGG16 和 LSTM，减少的不规则性要大得多，平均是 52.71 倍，因为我们可以选用更大的剪枝快对神经网络进行修剪。另外，对于 ResNet 网络，考虑到新特性，我们只能用比较小的剪枝块来修剪它，能够减少 13.02 倍的不规则性。实验结果进一步证实粗粒度剪枝可以显著降低网络的不规则性。

此外，我们在表 3.5列出了 AlexNet 网络的详细压缩特性。卷积层和 fc6，fc7，fc8 层的块大小分别为 $(1, 16, 1, 1,)$，$(32, 32)$，$(32, 32)$，$(16, 16)$。我们将卷积层和全连接层的权值分别量化为 8 比特和 4 比特。表中的权值和权值索引采用了都用 Huffman 编码。值得注意的是，与权值大小相比，权值索引大小几乎可以忽略不计。而在 Deep Compression 中，权值索引信息几乎是总存储容量的 40%，这严重阻碍了更有效的压缩。此外，使用细粒度的修剪时，权值和权值索引分别是 $25.93MB$ 和 $3.24MB$; 使用粗粒度剪枝时，权值大小和索引大小分别是 $25.65MB$ 和 $36.73KB$(比细粒度剪枝减少 90.32 倍)。局部量化进一步将权值大小减小到 $3.6MB$，而全局只能将权值大小压缩到 $4.95MB$。经过熵编码后，索引大小只有 $29.38KB$，对比与 Deep Compression 的 2.95MB 缩小 102.82 倍。受益于粗粒度稀疏和局部量化，我们新的压缩方法可以将 AlexNet 压缩 79 倍，远远高于 Deep

表 3.5　AlexNet 压缩特征 (W: 权值; S%: 稀疏度 I: 权值索引).

layer	#W	S%	bits per W	W size(KB)	I size(KB)
conv1	35K	24.9	6.5	20.75	0.80
conv2	307K	65.5	5.6	72.45	3.23
conv3	885K	64.1	5.4	209.37	9.69
conv4	663K	66.6	6.2	167.73	6.76
conv5	442K	65.9	5.5	101.28	4.61
fc6	38M	91.1	3.3	1353.37	1.60
fc7	17M	91.10	3.4	626.69	0.72
fc8	4M	74.8	3.3	415.80	1.97
total	61M	88.9	3.7	2967.44	29.38

Compression (35 倍) 和 CNNPack (39 倍)。

第 4 章　稀疏神经网络加速器的架构

我们的压缩方法能够对神经网络进行深度压缩，其中粗粒度稀疏能够充分利用神经网络局部收敛的特征，寻找最能够代表神经网络特征的权值块，从而有效减少稀疏权值的不规则性（平均减少 20.13 倍）; 局部量化进一步利用局部收敛的特性，减少表示每个权值的比特数; 最后熵编码对神经网络进行进一步的无损压缩。深度压缩的神经网络能够减少数据存储需求，减少计算量，减少数据传输量，最终提高计算性能，减少计算能耗。因此我们设计专用的硬件加速器，为了充分挖掘深度压缩神经网络的优秀特性。新型加速器的最主要的特征是两个选数逻辑: 神经元选择模块 (neuron selector module, NSM) 和突触选择模块 (synapse selector module, SSM)，分别过滤不必要的神经元和权值。同时，我们为加速器设计了专用的基于库的编程模型，减轻用户的编程负担。与最先进的稀疏的神经网络加速器 Cambricon-X [35] 相比，我们的加速器能够获得 1.71 倍的加速比，同时减少 1.37 倍的能耗。在 65nm 工艺下，我们的加速器面积和功耗仅仅为为 $6.73mm^2$ 和 $798.55mW$。

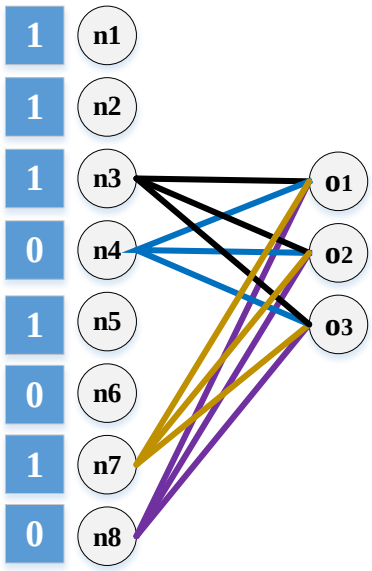
本章中我们首先以一个粗粒度稀疏的全连接神经网络为驱动实例，充分分析神经网络加速器的设计原则。然后我们根据这些设计原则设计对应的神经网络加速器。最后我们为加速器设计了一套基于库的编程框架以减轻用户的编程负担。

4.1　设计原则

我们将使用一个经过粗粒度稀疏的全连接层作为示例 (参见图 4.1)，分析粗粒度稀疏神经网络的特性，进而分析出神经网络加速器的设计原则。在图中，我们使用 2×3 的剪枝块对全连接层进行剪枝，同时我们只关注其中规模为 8×3 的部分连接。由于 ReLU 激励函数，输入神经元 $n4, n6, n8$ 的激活为 “0”。通过观察，我们发现了以下四个粗粒度稀疏神经网络的特性。

首先，多个输出神经元之间共享索引信息。如图 4.1所示，输入神经元 $n1, n2, n5, n6$ 与输出神经元 $o1, o2, o3$ 之间的连接被同时剪除，因此输出神经元之间共享相同的连接拓扑，也就是说它们共享相同的权值索引表示 “*00110011*” (图中的 “Synapse Indexes”)。尽管输出神经元之间不共享相同的权值，但所选权重的位置是相同的，因此是一种部分共享。

第二，多个输出神经元之间共享输入神经元。如图 4.1所示，最终需要参与计算的神经元是 $n3, n7$，它们的值被输出神经元共享。不失一般性，在完全连接



neuron indexes	input neurons	output neurons			synapse indexes
		o1	o2	o3	
1	n1	0	0	0	0
1	n2	0	0	0	0
1	**n3**	**s11**	**s21**	**s31**	**1**
0	n4	s12	s22	s32	1
1	n5	0	0	0	0
0	n6	0	0	0	0
1	**n7**	**s13**	**s23**	**s33**	**1**
0	n8	s14	s24	s34	1

o1 = n3*s11 + n7*s13
o2 = n3*s21 + n7*s23
o3 = n3*s31 + n7*s33

图 4.1　粗粒度稀疏的全连接层

的层中，设定修剪块大小为 (B_{in}, B_{out})，那么 B_{out} 个相邻输出神经元将共享相同的输入神经元。在卷积层中，设定剪枝块的大小为为 $(B_{fin}, B_{fout}, B_x, B_y)$，那么 B_{fout} 个相邻的输出神经元将共享相同的输入神经元。

第三，动态稀疏性能够进一步提高运算效率。在图中，通过挖掘权值稀疏，输入神经元 $n3, n4, n7, n8$ 被筛选出来进行计算，同时由于输入神经元 $n4, n8$ 的激活为零，在运算过程中，它们对输出神经元没有贡献，最后需要进行计算的输入神经元为 $n3, n7$。在图中的实例中，通过挖掘权值稀疏性，共需要进行 12 次乘法，9 次加法完成运算；同时挖掘权值稀疏性和动态神经元的稀疏性，只需要进行 6 次乘法和 3 次加法。对比在稠密情况下 24 次乘法和 21 次加法，分别有 2.14 倍和 5 倍的性能提升。因此，利用动态神经元稀疏性是进一步提高效率的关键 (在上面的示例中，能够进一步提升 2 倍性能)。值得注意的是，即使考虑到动态神经元稀疏，输出神经元仍然共享索引和所选的输入神经元。

第四，多个输出神经元之间的负载是平衡的，因为它们共享相同的输入神经元。对于图 4.1中的示例，如果考虑静态稀疏，那么每个输出神经元共享相同的输入神经元 $n3, n4, n7, n8$，同时需要四个对应的权值 $S_{Ti1}, S_{Ti2}, S_{Ti3}, S_{Ti4}$ 进行计算，因此每个输出神经元需要进行 4 次乘法和 3 次加法完成运算。如果同时考虑静态稀疏和动态稀疏，那么每次输出神经元共享相同的输入神经元 $n3, n7$，同时需要两个对应的权值 S_{Ti1}, S_{Ti3} 进行计算，因此每个输出神经元需要进行 2 次惩罚和 1 次加法完成运算。因此粗粒度稀疏的神经网络可以避免负载不平衡而造

成的性能损失 [37]。

因此，在设计加速器时要考虑以下原则，以最大限度地提高加速器的效率：加速器 (1) 能够利用共享的索引信息和共享的输入神经元信息，简化加速器的设计；（2）能够利用动态稀疏性进一步提高效率;(3) 利用相邻输出神经元之间的负载均衡。

4.2　加速器架构

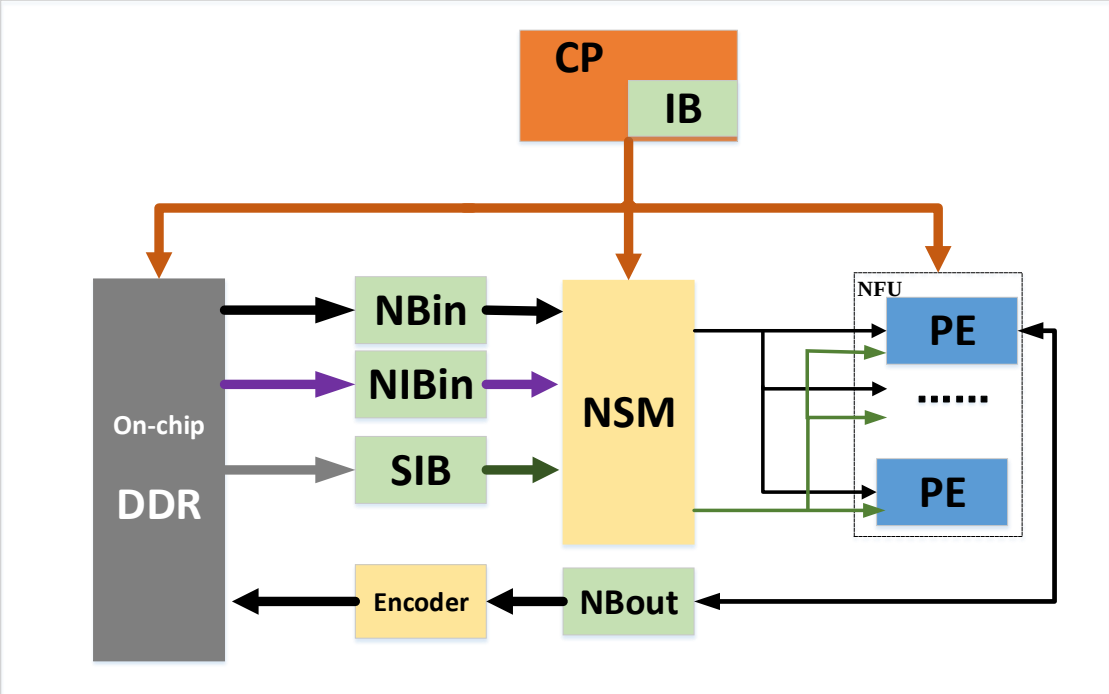


图 4.2　加速器整体架构

在本节中，我们将介绍我们的加速器的详细架构，它能够充分利用深度压缩神经网络带来的收益，有效地解决粗粒度修剪稀疏网络的剩余不规则性。

如图 4.2所示，我们展示了加速器架构。按照 4.1的设计原则，我们在加速器中设计了一个关键模块-神经元选择器模块 (neural selector module, NSM)，用于处理静态稀疏和共享信息（包括共享索引信息和共享神经元信息）。同时我们设计了一个神经功能单元 (neural functional unit，NFU) 用于完成神经网络中的核心计算。NFU 中具有多个处理单元 (processing elements, PEs) 用来并行计算不同的输出神经元。每个 PE 都包含一个本地的突触选择器模块 (synapse selector module, SSM) 来处理动态稀疏性。动态神经元压缩模块 (Encoder) 用于动态地将输出神经元压缩成为非零元素/非零元素索引的模式，从而减少 Load/Store 神经元的开销。存储模块需要存储输入神经元，权值和输出神经元，因此我们需要输入神经元缓存（input neuron buffer, NBin), 输出神经元缓存 (output neuron buffer, NBout)

和突触缓存（synapse buffer, SB），考虑到神经元和权值的稀疏性，我们需要两个额外的缓存，即输入神经元索引缓存（input neuron index buffer, NIBin) 和突触索引缓存（synapse index buffer, SIB) 分别存储输入神经元索引和突出索引信息。注意，其中 SB 被内置与 NFU 的每一个 PE 中，没有表现在图中。控制模块由控制处理器 (control processor, CP) 和指令缓存 (instruction buffer, IB) 组成，CP 有效地将 IB 中存储的各种指令解码为所有其他模块的详细控制信号，这里我们为加速器定义一个 VLIW（very long instruction word）风格的指令集。

下面我们将从稀疏处理，存储，控制和片上互联四个方面介绍加速器。

4.2.1　稀疏处理单元

该加速器旨在利用 (1) 静态稀疏性和 (2) 动态稀疏性，以及 (3) 压缩数据，从而减少数据存储量，减少数据传输量，减少计算量，从而提升性能，并减少能耗。在加速器中，稀疏性由 NSM，NFU 和 Encoder 这三个模块共同处理。NSM 接收来自 NBin 的输入神经元，来自 NIBin 的输入神经元索引和来自 SIB 的权值索引，筛选出需要进行计算的神经元（静态稀疏），同时生成筛选权值的 synapse flags，然后将筛选出的神经元和 synapse flags 通过 NFU 广播给 PE。每个 PE 中的 SSM 通过 synapse flags 筛选出需要进行计算的权值（动态稀疏），从而避免无用的计算和数据传输，提高加速器的性能。最后 Encoder 动态压缩输出神经元，从而减少片外访存能耗。

1. Indexing

在详细说明 NSM，NFU，Encoder 之前，我们将详细说明如何在加速器中存储和索引稀疏数据。表示稀疏数据的方法主要有两类，分别是二进制 binary mask 和 numerical indexing，这两类方法都仅存储非零元素，使用不同的方式索引非零元素。Binary mask 的方式主要包括直接索引（direct indexing），我们使用比特串对非零元素进行索引，其中”0” 表示对应的位置的元素为零，”1” 表示对应位置元素为非零。Numerical Indexing 包括步长索引（step indexing），压缩稀疏行（compressed sparse row, CSR），压缩稀疏列（compressed sparse column，CSC），坐标列表查找（coordinate list，COO）等，它们主要使用非零元素之间相对位置（步长）或者绝对位置信息对非零元素进行索引。由于目前主流的神经网络稠密度大小 5%，采用 CSR，CSC，COO 等方式的存储开销远远大于 direct indexing 或者 step indexing 的方式；而且考虑到神经元和权值同时稀疏，step indexing 这种方式很难索引到两者同时非零的情况，因此我们采用直接索引的方式存储稀疏神经元和稀疏权值。如图 4.1，我们仅需要存储 $n1, n2, n3, n5, n7$ 这五个神经

元和 “*11101010*” 比特串就能够表示稀疏的输入神经元；同时我们仅需要存储 $s11, s12, s13, s14$ 这四个权值和 “*00110011*” 比特串就能够表示与 $o1$ 相连的稀疏突触。值得注意的是，由于稀疏权值是一种静态稀疏，这种压缩过程可以离线完成；但是稀疏神经元是一种动态稀疏，这个过程需要在线实时完成，因此我们需要 Encoder 模块对稀疏神经元进行动态压缩。

2. NSM

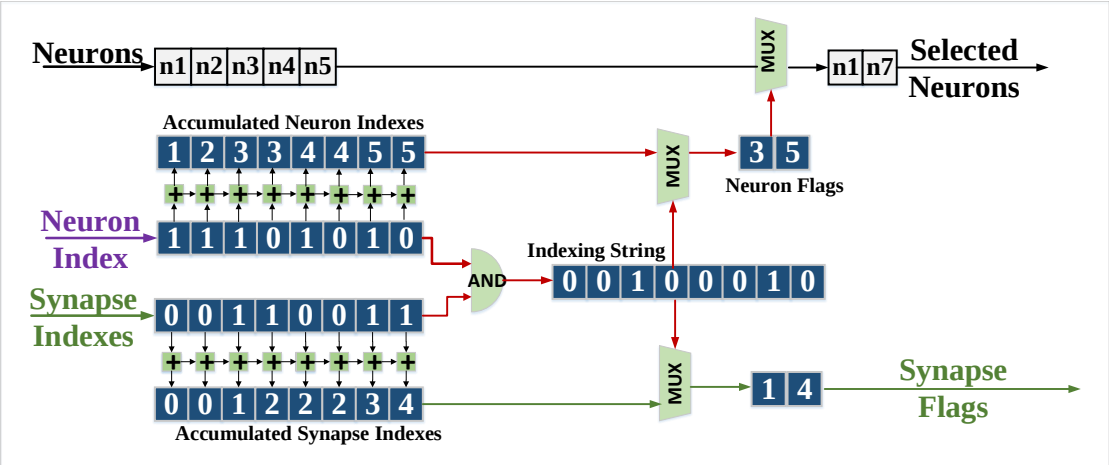


图 4.3　NSM 结构

如图 4.3所示，NSM 模块的输入为输入神经元，神经元索引和权值索引，NSM 模块通过神经元索引和权值索引筛选需要进行计算的输入神经元来处理静态稀疏。为了更有效地利用粗粒度稀疏特性，即多个相邻的输出神经元共享索引和输入神经元，我们设计了一个中心 NSM，它能够广播筛选出的神经元信息，从而被多个 PE 共享。我们利用图 4.1作为实例来描述 NSM 筛选输入神经元的流程。在图 4.1中的实例中，NSM 需要筛从 8 个输入神经元中筛选出需要进行计算的神经元 $n3, n7$ 并且生成筛选权值需要的 synapse flags。具体来说，首先 NSM 对 neuron indexes (“*11101010*”) 和 synapse indexes (“*00110011*”) 执行 “AND” 操作，从而生成 indexing string (“*00100010*”)。同时 neuron indexes 和 synapse indexes 分别累计自身，获得 accumulated neuron indexes (“*12334455*”) 和 accumulated synapse indexes (“*00122234*”) , 然后分别与 indexing string 执行 “MUX” 操作，生成 neuron flags (“*35*”) 和 synapse flags (“*14*”)。neuron flags 和 synapse flags 中的数字就表示了需要筛选的神经元和权值的最终位置信息。最后 neuron flags 与 neurons 执行 “MUX” 操作筛选出最终需要进行计算的神经元 $n3, n7$。值得注意的是，筛选出的神经元与 synapse flags 被多个输出神经元共享，因此这些信息最后被广播到 NFU 的各个 PE 中。最终 synapse flags 在 SSM 中被用于筛选出需要进行计算的权值 (见图 4.5(a))。

3. NFU

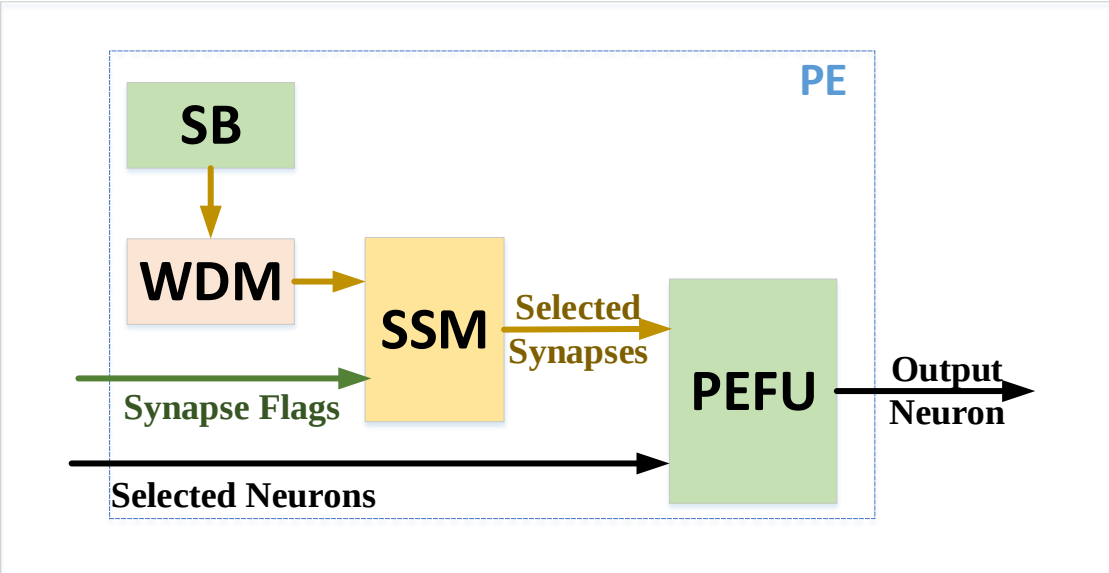


图 4.4　PE 结构

NFU 可以高效地处理神经网络中的所有运算，NFU 包含 T_n 个同构的 PE 来处理动态稀疏性，同时完成神经网络的运算。如图 4.4所示，PE 由一个局部突触缓冲 (synapse buffer, SB)，权码解码模块 (weight decoding module, WDM)，突触选择模块 (synapse select module, SSM) 和处理功能单元 (processing element functional unit, PEFU) 组成。由于每个 PE 处理共享相同的输入神经元，但是使用独立的权值计算不同的输出神经元，因此我们在每个 PE 中集成了一个本地的 SB，使得每个 PE 能够从本地 SB 上加载权值，从而缓减数据传输中的网络拥塞 (network congestion)。同时 PE 内部集成了一个带有查找表的 (look-up table, LUT) 的 WDM，用于处理经过局部量化的权值，提取出实际的权值。SSM 接收权值和 synapse flags，最终筛选出需要进行计算的权值 (如图 4.5(a))。每个 PEFU 由 T_m 乘法单元，一个 T_m 输入的加法器树和一个非线性函数模块组成 (如图 4.5(b))。我们使用分时复用的方法将神经网络映射到 PE，即每个 PE 计算一个输出神经元。理想情况下，如果一个输出神经元 PE 需要 M 个乘法操作，那么 PE 需要 $\lceil M/T_m \rceil$ 个 cycle 完成计算元，因为 PEFU 可以在一个 cycle 内完成 T_m 次乘法。最后，NFU 将从 T_n 个 PE 中收集 T_n 个输出神经元的部分和，传输到 NBout 中。

SSM 用于处理动态稀疏，筛选出需要进行计算的权值。我们采用紧致的形式存储静态稀疏的权值。如图 4.1所示，对于每个输出神经元，第一个和第四个突触 (S_{Ti1}，S_{Ti4}) 是计算所需的两个突触。SSM 根据 NSM 产生的 synapse flags 包含所需突触位置索引 (“14”) 筛选所需的突触。如图 4.5 (a) 所示，SSM 在权值与 synapse flags 之间执行 “MUX” 操作，最终筛选出需要进行计算的权值。由于每个 PE 接收不同的权值，并且处理不同的输出神经元，我们在每个 PE 内部

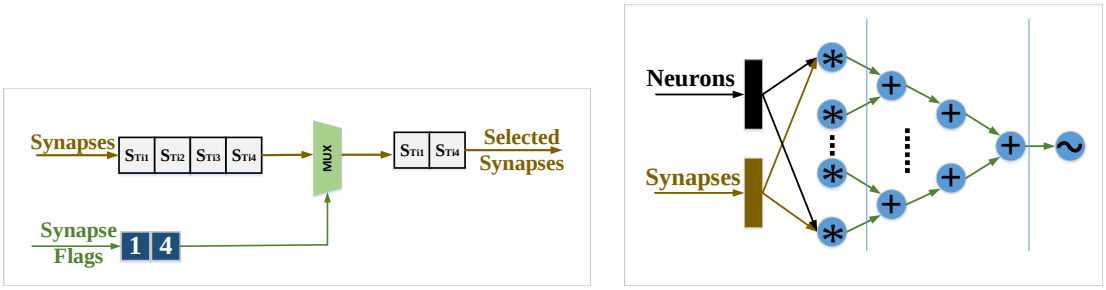


图 4.5　(a) SSM 结构 (b) PEFU 结构

局部集成 SSM 和 SB，从而避免高带宽和长延迟。

4. Encoder

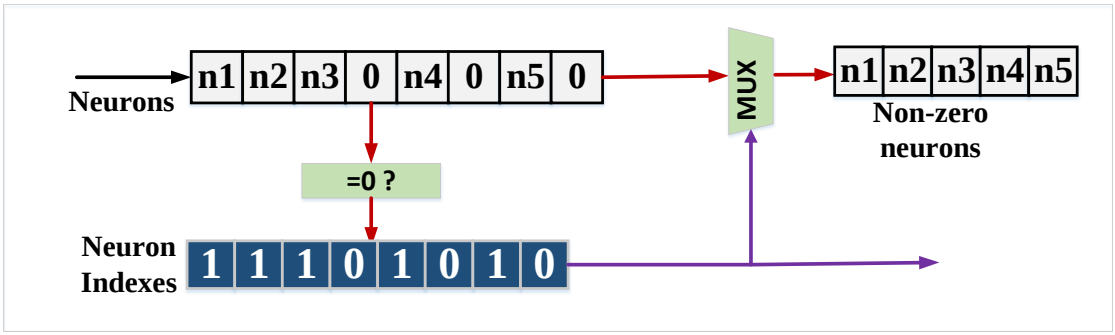


图 4.6　Encoder 架构

图 4.6展示了神经元压缩模块 Encoder 的架构。Encoder 模块与 Nbout 直接相连，用于消除输出神经元中的零元素，最终稀疏的神经元使用非零神经元和比特串进行表示。具体来说，Encoder 根据神经元的激活值，生成 synapse indexes (“11101010”)，每个索引指示相应的神经元激活是否为零。然后 Encoder 在神经元与 synapse indexes 之间执行“MUX”操作筛选出非零的神经元。最终非零神经元与对应的索引 synapse indexes 被传输到片外，然后作为下一层网络的输入被重新加载到 NBin。通过动态压缩神经元，加速器能够显著减少 DRAM 访问神经元的开销，从而减少 DRAM 的访存能耗。

4.2.2　存储模块

考虑到不同数据在加速器中的不同行为模式，我们将偏上缓存分为三个部分：一个输入神经元缓存（NBin），一个输出神经元缓存（NBout）和 T_m 个权值缓存（SB），考虑到神经元稀疏和权值稀疏的情况，我们增加了两个索引缓存，分别是输入神经元索引缓存（NIBin）和权值索引缓存（SIB）。为了实现数据的高速流水，我们使用乒乓（ping-pong）的模式管理所有的缓存。

对于 NBin，我们设置带宽为 $16 \times T_m \times 16$ 比特，设置这个带宽主要是考虑

到 PE 的利用率。虽然 PE 共享相同的输入神经元，但是每个 PE 在一个 cycle 需要 T_m 个输入神经元来避免 PEFU 中计算单元的空转。考虑到现有大部分神经网络的稠密度都高于 $1/16$（即稀疏度低于 $15/16$），特别是卷积层，一般稠密度为 30% 左右，因此，我们在一个 cycle 发送给 NSM$16 \times T_m$ 个输入神经元时，就能够保证 NSM 在一个 cycle 能够筛选出 T_m 个输入神经元，从而保证 PEFU 的运行效率。

对于 NBout，我们设置其带宽为 $4 \times T_n \times 16$ 比特，这部分主要是考虑到动态神经元稀疏。如表 3.2考虑到现有大部分神经网络的神经元稠密度高于 25%（即稀疏度低于 75%），当我们在每一个 cycle 发送给 Encoder$4 \times T_n$ 个输出神经元时，就能够保证 Encoder 在一个 cycle 筛选出 T_n 个神经元进行压缩，从而保证流水顺利进行。

对于每一个 PE 中的 SB，我们设置带宽 $4 \times T_m \times 16$ 比特，保证能够在一个 cycle 读取 $4 \times T_m$ 个权值。考虑到现有大部分神经网络的动态稀疏度低于 75%，$4 \times T_m \times 16$ 比特的带宽能够保证 SSM 在一个 cycle 筛选出 T_m 个权值，从而保证 PEFU 的运行效率。由于权值是以压缩形式（通过剪枝和局部量化）存储在 SB 中，局部量化会导致权值在不同的神经网络和不同层中的位宽不同，例如，在 AlexNet 网络的卷积层中权值被量化为 8 比特，在 MLP 网络的全连接层中权值被量化为 6 比特，在 AlexNet 网络的全连接层中权值被量化为 4 比特。不同的量化比特会导致 WDM 模块庞大的面积/功耗开销，因此我们将权值按照 4 比特进行对齐操作。例如，假设 SRAM 的一行为 128 比特，那么当权值被量化为 16 比特，8 比特和 4 比特时，SRAM 的一行分别能够存储 8 个，16 个和 32 个权值。因此，$T_m \times 64$ 比特的权值能够被 WDM 解码成为 $T_m \, times16$ 个权值（当权值被编码成小于或者等于 4 比特），或者 $T_m \times 8$ 个权值（当权值被编码成大于 4 比特而小于等于 8 比特），或者 T_m 个权值（当权值被编码成大于 8 比特，小于等于 16 比特）。对比于支持所有位宽编码形式的解码模块，我们的 WDM 能够减少 5.14 倍的面积和 4.27 的功耗。

对于 NIBin 和 SIB, 我们设定其宽度为 $16 \times T_m \times 1$ 比特，因为我们使用直接索引的方式存储稀疏的神经元和权值，索引中的每一比特用于表示对应的元素是否为 “0” 表明是否存在相应的突触。因此,NIBin 和 SIB 则需要 $16 \times T_m \times 1$ 比特的带宽与 Nbin 提供给 NSM 的 $16 \times T_m$ 个输入神经元一一对应。

五个缓存的大小决定了整个架构的整体性能和能耗。尽管有部分研究 [22, 32] 提出，加速器需要提供足够大的缓存来保存神经网络所有的权值和神经元从而避免昂贵的片外访存开销，但是这种设计会大大增加延迟，面积，功耗；同时这种设计方案缺少可扩展性，并不能很好地支持新兴的规模更大，层数更深的神经网络。因此我们使用小的缓存以及适当的数据替换策略，从而权衡加速器的可

扩展性，性能和能耗。在探索了不同大小的缓存区后，我们设定 NBin，NBout，SB，SINin 和 SIB 的大小分别是 $8KB$，$8KB$，$32KB$，$1KB$ 和 $1KB$。

4.2.3　控制

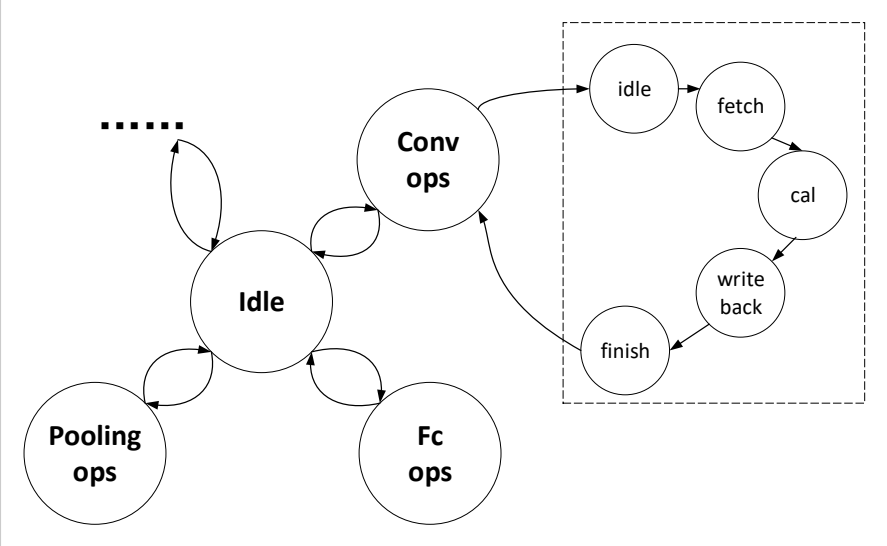


图 4.7　CP 的有限状态机

CP 用于控制加速器的整个执行流程。它从内部指令缓存中读取指令然后进行解码，从而有效地协调数据组织，数据访问和数据计算的过程。CP 通过监控各个模块对应的控制寄存器来监控每个模块的状态。同时我们利用基于库的编译器生成高效的超长指令字（Very Long Instruction Word，VLIW）。同时，我们为 CP 设计了高效的有限状态机（finite state machine, FSM），如图 reffig:FSM 所示。FSM 分为两层控制状态，第一层状态表示神经网络中的宏观操作, 如卷积，内积，池化，激活等操作; 第二层状态则是每一个宏观操作中的细节，如对于卷积操作还会有数据预取，计算，写回等不同的状态。

4.2.4　片上互联

如图 4.8所示，我们采用 H 树的拓扑结构连接 NSM 与 NFU 中的 T_n 个 PE。由于 NSM 产生的 selected neurons 和 synapse flags 被所有 PE 共享，而且 PE 之间的负载是一种均衡的状态，因此 NSM 通过 H 树将这些信息通过广播的形式传送给 PE，从而避免 NSM 和 PE 之间由于距离差异引起的不平衡延迟的问题；同时 H 树拥有非常良好的可扩展性，能够非常方便地对 PE 进行扩展，在不改变整体拓扑结构的情况下通过增加 PE 的数量增加加速器的计算能力，从而应对更大规模的神经网络。

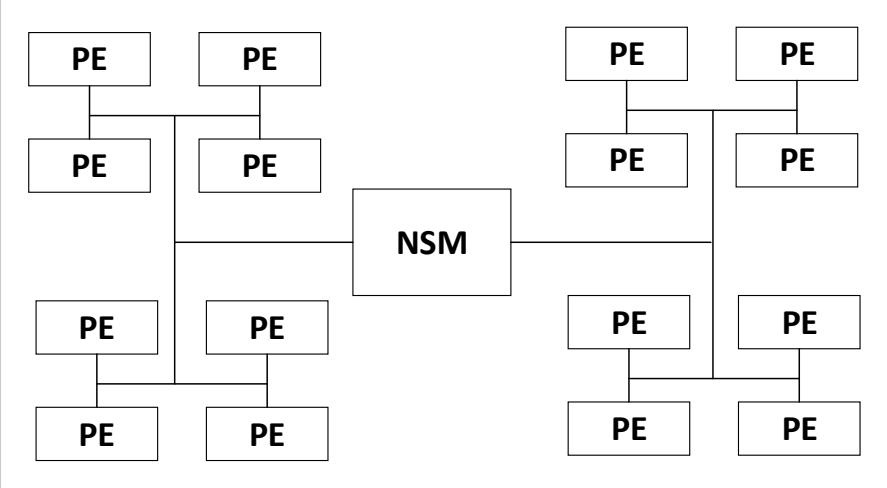


图 4.8　H 树互联结构

4.3　编程模型

4.3.1　基于库的编程模型

考虑到实际应用场景和开发效率, 高级编程模型, 如 Caffe [**?**], Tensorflow [**?**], MXNet [**?**] 等为用户提供了一系列 C++ 的库函数接口，使得用户能够方便地调用这些接口，从而控制 CPU 或者 GPU 完成神经网络的运算。高级编程模型能够大幅度增加用户或者开发人员的编程效率，特别是算法和应用程序开发人员。因此，专用的加速器必须为用户提供类似于高级编程框架的接口，以减轻用户的编程负担。为此，我们提供了基于库的编程模型，其基本思想是为用户提供一系列 C++ 的高级库函数接口，每一个函数对应了神经网络中一个基本操作，如卷积运算，内积，池化等操作。代码 4.1就是卷积层库函数的接口，用户首先通过 TensorDescriptor 和 FilterDescriptor 定以卷积操作涉及输入/输出和权值; 同时分别使用 setTensorDescriptor 和 setFilterDescriptor 分别描述输入/输出和权值的特性, 如稀疏, 大小, 地址等。然后用户通过 ConvDescriptor 和 setConvDescriptor 分别定义卷积操作描述符和卷积操作的特性, 如输入/输入/权值规模, 步长, padding 等。最后用户调用 convForward 执行卷积操作。用户通过调用这个库函数接口能够非常方便地最终驱动加速器完成卷积操作。

```
1  // Data Declaration
2  TensorDescriptor input, output;
3  FilterDescriptor weight;
4  setTensorDescriptor (input, ...);
5  setTensorDescriptor (output, ...);
6  setFilterDescriptor (weight, ...);
7  // Operations
8  ConvDescriptor conv;
9  setConvDescriptor (...);
```

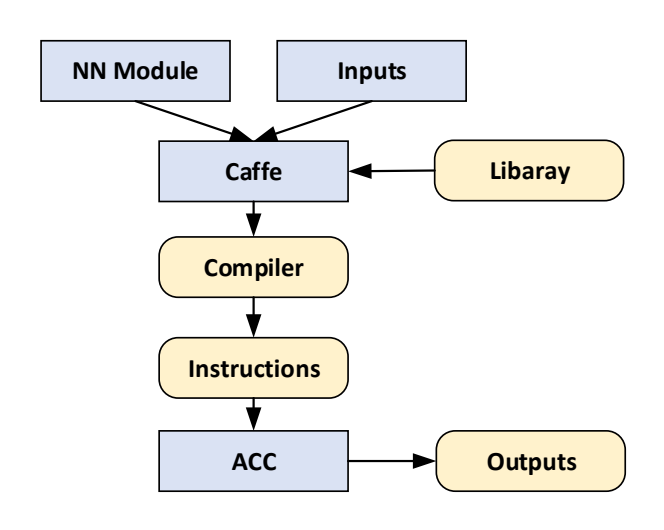


图 4.9　编程框架

```
10  ConvForward(conv, input，weight，output );
```

代码 4.1　卷积层库函数接口

我们将库嵌入到深度学习框架 Caffe 中，如图 4.9所示，这一套编程框架对用户完全透明，使得用户能够无缝使用我们的加速器。值得注意的是我们需要为我们的加速器设计专用的编译器（compiler），使其能够将 C++ 源码编译成为加速器中的可执行指令。

4.3.2　编译器

深度学习框架（如 caffe）与我们的加速器之间存在巨大的鸿沟，如何为加速器建立一套完整的编程生态系统是一个巨大的挑战。为此我们需要为加速器设计专用的基于库的编译器，从而将基于库的源文件编译成为加速器中可执行指令。编译的主要挑战来源于加速器中复杂的片上数据管理。

具体来说，第一，神经网络中的数据多样性和特征多样性增加了数据分配和数据搬运的复杂度。由于神经网络中有不同类型的参数，如输入/输出神经元，权值等；同时不同的数据拥有不同的特征，如神经元有动态稀疏的特性，权值有静态稀疏和量化的特性。如何根据数据的特征为不同的数据种类分配数据空间，如何搬运不同种类的数据成为一个巨大挑战，因此, 我们为不同的数据使用适当的数据分配和数据搬运策略来提升效率。

第二，有限的片上缓存（4.2.2）增大了数据调度的复杂性。考虑到神经网络中庞大的神经元/突触的和加速器中有限的片上内存之间的矛盾，我们需要使用 loop tiling 对数据进行切分，同时使用合适的数据重用策略提升加速器的性能。而不同的 loop tiling 策略和数据重用 (data reuse) 策略会进一步影响片外数据访

问，片上数据访问，计算调度，从而影响加速器的效率和性能。因此我们需要在编译期间探索不同的 loop tiling 策略和数据复用策略，选择最优的 loop tiling 方式和最佳的数据服用策略，最大程度挖掘加速器的性能。

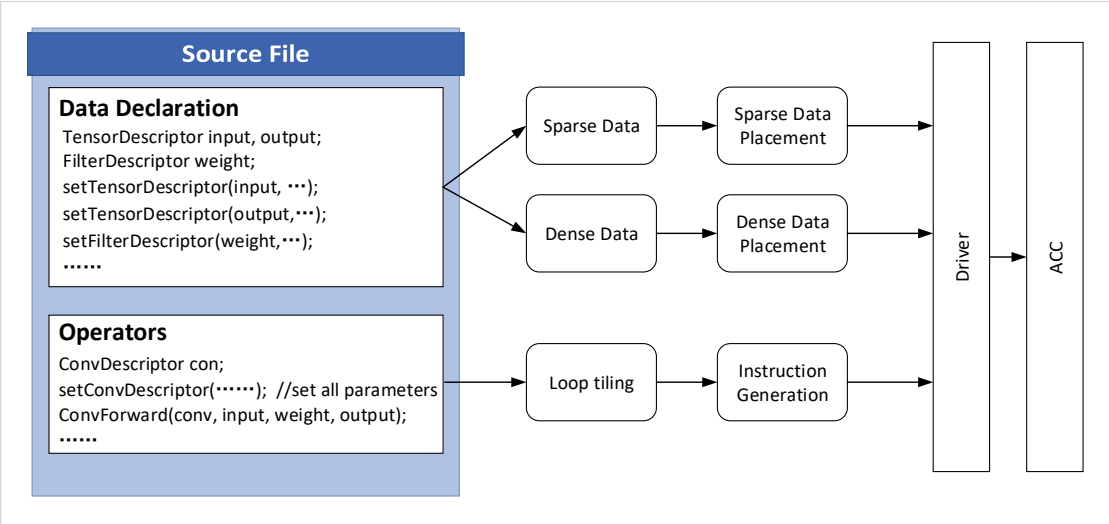


图 4.10　源文件编译成为加速器可执行代码的过程

我们设计为加速器设计了一个专用的编译器来弥合深度学习框架与加速器之间的差距。如图 4.10所示，我们显示了如何将基于库的源文件编译成为加速器的可执行代码。编译过程分为两个阶段：数据摆放（data placement）和指令生成（instruction generation）。

在数据摆放阶段，源代码中的数据结构被合理地分配到加速器的片上缓存中。我们使用张量（tensor）对输入和输出神经元进行封装，同时我们用过滤器（filter）对权值进行封装，值得注意的是，tensor 和 filter 都是四维的张亮。我们使用不同的数据结构封装不同数据类型，是因为不同类型的数据会被分配到加速器不同的片上缓存中（如输入神经元，输出神经元和权值会分别被分配到 NBin，NBout 和 SB 中）。同时考虑到数据稀疏的特性，我们使用 sparse 这个枚举类型作为描述符添加到 tensor（动态稀疏）和 filter（静态稀疏）的属性中。

在指令生成阶段，我们使用加速器专用运算操作（如卷积操作，池化操作等）对数据进行运算操作，同时我们使用 loop tiling 将这些操作分解为子操作，以确保子操作对应的数据能够（即输入/输出神经元，权值）能够填充到有限的片上缓存中。值得注意的是，在指令生成阶段，数据搬运操作将与数据运算操作同时生成。最后，代码生成器将这些操作转换成为可执行的指令，使得其能够在加速器上解码和执行。

4.3.3　loop tiling

Loop tiling 能够对计算进行合理划分，以确保子计算所涉及的数据能够被完全填充到片上缓存中。由于片外访存会花费巨大的能耗（占用超过 90% 的整体能耗），因此 loop tiling 策略优先考虑如何提高片上数据重用性，从而减少片外数据访存，从而减少加速器的访存能耗。Eyeriss [39] 提出了四种不同的数据重用策略，分别是卷积重用（*convolutional reuse*），卷积核重用（*filter reuse*），输入特征图重用（*input feature map reuse*）和部分和重用（*partial sums reuse*）。考虑到我们的加速器与 Eyeriss 架构上的差异，我们将数据重用策略分为三种类型，即输出重用（*output-reuse*，与 Eyeriss 中的 *partial sums reuse* 对应），突触重用（*synapse-reuse*，与 Eyeriss 中的 *convolutional reuse* 和 *filter reuse* 对应）和输入重用（*input-reuse*，与 Eyeriss 中 *input feature map resue* 对应）；这三种策略分别表示最大程度重用输出部分和数据，突触数据和输入数据。对于 *output-reuse* 策略，在运算过程中，一段输入神经元被加载到 NBin 中，一段权值被加载到 SB 中，然后计算出一段输出神经元的部分和，然后将其存储到 NBout 中；在下一个阶段，第二段的输入神经元和权值被加载到片上缓存持续计算 NBout 中输出神经元部分和，这个过程一直循环直至 NBout 中的这一段输出神经元被完全计算完成，最后 NBout 中的输出神经元被存储到片外；因此在 *output-reuse* 策略中，输入神经元和权值会多次从片外加载到片上缓存，而输出神经元会持续存储在片上缓存中直至被完全计算完成。同理，在 *synapse-resue* 策略中，输入神经元和输出神经元会被多次从片外加载到片上，直至片上的权值被完全重用。在 *input-reuse* 策略中，权值和输出神经元会被多次从片外加载到片上，直至片上的输入神经元被完全重用。因此我们在编译阶段需要探索最佳的数据重用策略，减少片外访存的能耗。

不失一般性的，我们以 VGG16 网络的 *conv4_2* 卷积层为例，来计算不同的数据重用策略对片外访存的需求。具体来说，*conv4_2* 的输入规模为 $30\times30\times512$（考虑到输入需要进行 padding），权值的规模为 $32\times32\times512\times512$，输出的规模为 $28\times28\times512$，其中输入神经元和权值的稠密度分别是 40.52%（稀疏度为 59.48%）和 34.20%（稀疏度为 65.80%）（如表 3.2所示)。加速器中 NBin，SB 和 NBout 的大小分别是 $8KB$，$8KB$ 和 $32KB$。在 loop tiling 过程中，我们将数据且分为 $SegN$ 块，其中每一块数据量为 $SegSize$。为了尽可能提高片上缓存的利用率，我们将输出的 channel 维度按照 4×128（$SegN\times SegSize$）的方式进行切分，在 height 维度按照 28×1 的方式进行切分；同时我们将输入的 channel 方向按照 16×32 的方式进行切分。经过 loop tiling 后，输入，权值，输出每一个数据块的大小分别为 $SegSize_{in}=30\times3\times32\times40.52\%$，$SegSize_{syn}=3\times3\times32\times128\times34.20\%$

和 $SegSize_{out} = 28 \times 1 \times 128$。值得注意的是，我们对输入数据块和权值数据块分别乘以网络的稠密度来进行近似。因此，使用 *input-reuse*，*output-reuse* 和 *synapse-reuse* 策略，所需要的片外访存的开销可以按照如下公式进行计算

$$MEM_i = 16 \times 1 \times 28 \times (SegSize_{in} + SegSize_{syn} \times 4 + SegSize_{out} \times 4 \times 2) \times 2 = 68.59MB$$
$$MEM_o = 4 \times 28 \times 1 \times (SegSize_{out} + SegSize_{in} \times 16 + SegSize_{syn} \times 16) \times 2 = 47.85MB$$
$$MEM_s = 16 \times 4 \times (SegSize_{syn} + SegSize_{in} \times 28 \times 1 + SegSize_{out} \times 28 \times 2) \times 2 = 30.03MB$$

因此我们为 *conv4_2* 卷积层选择 *synapse-reuse* 的策略。

4.4 Discussion

第 5 章　加速器的验证和实验结果

5.1　加速器验证方法

5.1.1

5.1.2　加速器功能验证

5.1.3　加速器性能验证

5.2　实验方法

5.3　实验结果

5.3.1　硬件属性

5.3.2　性能

5.3.3　能耗

5.3.4　讨论

1. 熵编码和熵解码模块

目前我们的加速器中并没有加入熵解码模块（entropy decoding module）来使得加速器支持权值熵编码，主要是考虑到熵解码模块需要耗费非常大的面积和能耗，但是仅仅能够获得非常有限的性能提升。一个熵解码模块（entropy decoding module) 的面积为 $6.781 * 10^{-3} mm^2$，它能够在一个 cycle 解码出一个码字。由于熵编码是一种变长编码，因此对应的解码模块必须串行进行解码。即使我们可以将数据划分为许多并行的数据流，然后对为每个数据流提供一个熵解码模块，这种方法将会引入巨大的面积和能耗开销。考虑到每一个 SB 需要在一个 cycle 提供 $T_m \times 4$ 个数据，为了避免性能损失，我们必须为一个 SB 提供 $T_m \times 4$ 个熵解码模块，因此我们在加速其中总共需要集成 $T_n \times T_m \times 4$ 个熵解码模块。在 $T_m = T_n = 16$ 的配置下，我们总共需要 1024 个熵解码模块，这将引入额外 $6.94 mm^2$ 的面积和 $971.37 mW$ 的功率，因此加速器的总面积和功率分别是 $13.67 mm^2$ 和 $1769.92 mW$, 分别是原始设计的 2.03 倍和 2.22 倍。然而，新增熵解码的加速器在卷积层上几乎没有性能提升，在全连接层也只有 1.18 倍的性能提升，对比与额外的面积和功率开销，这种性能提升是非常有限的。因此，我们在

加速器中不加入熵解码模块。

2. 稀疏性与性能

3. 减少的不规则度与性能

4. 类似粗粒度稀疏的方法

5. 其他稀疏神经网络加速器

第 6 章　数　　学

6.1　数学符号

模板定义了一些正体（upright）的数学符号：

符号	命令
常数 e	`\eu`
复数单位 i	`\iu`
微分符号 d	`\diff`
$\arg\max$	`\argmax`
$\arg\min$	`\argmin`

更多的例子：

$$\mathrm{e}^{\mathrm{i}\pi} + 1 = 0 \tag{6.1}$$

$$\frac{\mathrm{d}^2 u}{\mathrm{d}t^2} = \int f(x)\,\mathrm{d}x \tag{6.2}$$

$$\arg\min_{x} f(x) \tag{6.3}$$

6.2　定理、引理和证明

定义 6.1　If the integral of function f is measurable and non-negative, we define its (extended) **Lebesgue integral** by

$$\int f = \sup_{g} \int g, \tag{6.4}$$

where the supremum is taken over all measurable functions g such that $0 \leqslant g \leqslant f$, and where g is bounded and supported on a set of finite measure.

例 6.1　Simple examples of functions on $\mathbb{R}^d$ that are integrable (or non-integrable) are given by

$$f_a(x) = \begin{cases} |x|^{-a} & \text{if } |x| \leqslant 1, \\ 0 & \text{if } x > 1. \end{cases} \tag{6.5}$$

$$F_a(x) = \frac{1}{1+|x|^a}, \qquad \text{all } x \in \mathbb{R}^d. \tag{6.6}$$

Then f_a is integrable exactly when $a < d$, while F_a is integrable exactly when $a > d$.

引理 6.1 (Fatou)　Suppose $\{f_n\}$ is a sequence of measurable functions with $f_n \geqslant 0$. If $\lim_{n\to\infty} f_n(x) = f(x)$ for a.e. x, then

$$\int f \leqslant \liminf_{n\to\infty} \int f_n. \tag{6.7}$$

注　We do not exclude the cases $\int f = \infty$, or $\liminf_{n\to\infty} f_n = \infty$.

推论 6.2　Suppose f is a non-negative measurable function, and $\{f_n\}$ a sequence of non-negative measurable functions with $f_n(x) \leqslant f(x)$ and $f_n(x) \to f(x)$ for almost every x. Then

$$\lim_{n\to\infty} \int f_n = \int f. \tag{6.8}$$

命题 6.3　Suppose f is integrable on $\mathbb{R}^d$. Then for every $\epsilon > 0$:

i. There exists a set of finite measure B (a ball, for example) such that

$$\int_{B^c} |f| < \epsilon. \tag{6.9}$$

ii. There is a $\delta > 0$ such that

$$\int_E |f| < \epsilon \qquad \text{whenever } m(E) < \delta. \tag{6.10}$$

定理 6.4　Suppose $\{f_n\}$ is a sequence of measurable functions such that $f_n(x) \to f(x)$ a.e. x, as n tends to infinity. If $|f_n(x)| \leqslant g(x)$, where g is integrable, then

$$\int |f_n - f| \to 0 \qquad \text{as } n \to \infty, \tag{6.11}$$

and consequently

$$\int f_n \to \int f \qquad \text{as } n \to \infty. \tag{6.12}$$

证明　Trivial. □

6.3　自定义

Axiom of choice　Suppose E is a set and E_α is a collection of non-empty subsets of E. Then there is a function $\alpha \mapsto x_\alpha$ (a "choice function") such that

$$x_\alpha \in E_\alpha, \qquad \text{for all } \alpha. \tag{6.13}$$

Observation 1　Suppose a partially ordered set P has the property that every chain has an upper bound in P. Then the set P contains at least one maximal element.

A concise proof　Obvious. □

第 7 章　浮　动　体

7.1　三线表

三线表是《撰写手册》推荐使用的方式，如表 7.1。

表 7.1　这里是表的标题

操作系统	TeX 发行版
所有	TeX Live
macOS	MacTeX
Windows	MikTeX

注：一个很长长长的表注。

7.2　长表格

超过一页的表格要使用专门的 `longtable` 环境（表 7.2）。

表 7.2　长表格演示

名称	说明	备注
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC

续下页

表 7.2　长表格演示（续）

名称	说明	备注
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC
AAAAAAAAAAAA	BBBBBBBBBBB	CCCCCCCCCCCCCC

7.3　插图

有的同学可能习惯了“下图”、“上表”这样的相对位置引述方式，希望浮动体放在固定位置。事实上，这是不合理的，因为这很容易导致大片的空白。在科技论文中，标准的方式是“图7.1”、“表 7.1”这样的因数方式。

图 7.1　测试图片

关于更多的插图方式，arXiv 上的大部分文献会提供 TEX 源码，大家可以参考学习。

7.4　算法环境

模板中使用 `algorithm2e` 宏包实现算法环境。关于该宏包的具体用法，请阅读宏包的官方文档。

注意，我们可以在论文中插入算法，但是插入大段的代码是愚蠢的。然而这并不妨碍有的同学选择这么做，对于这些同学，建议用 listings 宏包。

Data: this text
Result: how to write algorithm with LaTeX2e
initialization;
while *not at end of this document* **do**
    read current;
    if *understand* **then**
        go to next section;
        current section becomes this one;
    else
        go back to the beginning of current section;
    end
end

算法 7.1: 算法示例 1

第 8 章　引用文献标注方法

8.1　顺序编码制

8.1.1　角标数字标注法

8.2　其他形式的标注

参 考 文 献

[1] KRIZHEVSKY A, SUTSKEVER I, HINTON G E. Imagenet classification with deep convolutional neural networks[C]//Advances in neural information processing systems. 2012: 1097-1105.

[2] IOFFE S, SZEGEDY C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[J]. arXiv preprint arXiv:1502.03167, 2015.

[3] BA J L, KIROS J R, HINTON G E. Layer normalization[J]. arXiv preprint arXiv:1607.06450, 2016.

[4] DMITRY U, ANDREA V, VICTOR L. Instance normalization: The missing ingredient for fast stylization[J]. arXiv preprint arXiv:1607.08022, 2016.

[5] WU Y, HE K. Group normalization[J]. arXiv preprint arXiv:1803.08494, 2018.

[6] GUPTA S, AGRAWAL A, GOPALAKRISHNAN K, et al. Deep learning with limited numerical precision[C]//International Conference on Machine Learning. 2015: 1737-1746.

[7] GOODFELLOW I, BENGIO Y, COURVILLE A, et al. Deep learning: volume 1[M]. MIT press Cambridge, 2016.

[8] KÖSTER U, WEBB T, WANG X, et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks[C]//Advances in Neural Information Processing Systems. 2017: 1742-1752.

[9] DETTMERS T. 8-bit approximations for parallelism in deep learning[J]. arXiv preprint arXiv:1511.04561, 2015.

[10] COURBARIAUX M, BENGIO Y, DAVID J P. Binaryconnect: Training deep neural networks with binary weights during propagations[C]//Advances in neural information processing systems. 2015: 3123-3131.

[11] HU Q, WANG P, CHENG J. From hashing to cnns: Training binaryweight networks via hashing[J]. arXiv preprint arXiv:1802.02733, 2018.

[12] RASTEGARI M, ORDONEZ V, REDMON J, et al. Xnor-net: Imagenet classification using binary convolutional neural networks[C]//European Conference on Computer Vision. Springer, 2016: 525-542.

[13] RUSSAKOVSKY O, DENG J, SU H, et al. Imagenet large scale visual recognition challenge [J]. International Journal of Computer Vision, 2015, 115(3): 211-252.

[14] LI F, LIU B. Ternary weight networks.(2016)[J]. arXiv preprint arXiv:1605.04711, 2016.

[15] ZHU C, HAN S, MAO H, et al. Trained ternary quantization[J]. arXiv preprint arXiv:1612.01064, 2016.

[16] ZHOU A, YAO A, GUO Y, et al. Incremental network quantization: Towards lossless cnns with low-precision weights[J]. arXiv preprint arXiv:1702.03044, 2017.

[17] WANG P, CHENG J. Fixed-point factorized networks[C]//Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on. IEEE, 2017: 3966-3974.

[18] HUBARA I, COURBARIAUX M, SOUDRY D, et al. Binarized neural networks[C]// Advances in neural information processing systems. 2016: 4107-4115.

[19] ZHOU S, WU Y, NI Z, et al. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients[J]. arXiv preprint arXiv:1606.06160, 2016.

[20] CAI Z, HE X, SUN J, et al. Deep learning with low precision by half-wave gaussian quantization[J]. arXiv preprint arXiv:1702.00953, 2017.

[21] HAN S, POOL J, TRAN J, et al. Learning both weights and connections for efficient neural network[C]//Advances in Neural Information Processing Systems. 2015: 1135-1143.

[22] HAN S, LIU X, MAO H, et al. Eie: efficient inference engine on compressed deep neural network[C]//Proceedings of the 43rd International Symposium on Computer Architecture. IEEE Press, 2016: 243-254.

[23] WANG Y, XU C, YOU S, et al. Cnnpack: Packing convolutional neural networks in the frequency domain[C]//Advances In Neural Information Processing Systems. 2016: 253-261.

[24] HE T, FAN Y, QIAN Y, et al. Reshaping deep neural network for fast decoding by node-pruning[C]//Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on. IEEE, 2014: 245-249.

[25] SRINIVAS S, BABU R V. Data-free parameter pruning for deep neural networks[J]. arXiv preprint arXiv:1507.06149, 2015.

[26] HU H, PENG R, TAI Y W, et al. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures[J]. arXiv preprint arXiv:1607.03250, 2016.

[27] MARIET Z, SRA S. Diversity networks[J]. arXiv preprint arXiv:1511.05077, 2015.

[28] DENTON E L, ZAREMBA W, BRUNA J, et al. Exploiting linear structure within convolutional networks for efficient evaluation[C]//Advances in Neural Information Processing Systems. 2014: 1269-1277.

[29] JADERBERG M, VEDALDI A, ZISSERMAN A. Speeding up convolutional neural networks with low rank expansions[J]. arXiv preprint arXiv:1405.3866, 2014.

[30] LEBEDEV V, GANIN Y, RAKHUBA M, et al. Speeding-up convolutional neural networks using fine-tuned cp-decomposition[J]. arXiv preprint arXiv:1412.6553, 2014.

[31] CHEN T, DU Z, SUN N, et al. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning[C/OL]//Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS). 2014: 269-

284. http://dl.acm.org/citation.cfm?id=2541967.

[32] CHEN Y, LUO T, LIU S, et al. Dadiannao: A machine-learning supercomputer[C]// Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2014: 609-622.

[33] LIU D, CHEN T, LIU S, et al. Pudiannao: A polyvalent machine learning accelerator[C]// ACM SIGARCH Computer Architecture News: volume 43. ACM, 2015: 369-381.

[34] LIU S, DU Z, TAO J, et al. Cambricon: An instruction set architecture for neural networks [C]//ACM SIGARCH Computer Architecture News: volume 44. IEEE Press, 2016: 393-405.

[35] ZHANG S, DU Z, ZHANG L, et al. Cambricon-x: An accelerator for sparse neural networks [C]//Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on. IEEE, 2016: 1-12.

[36] ALBERICIO J, JUDD P, HETHERINGTON T, et al. Cnvlutin: Ineffectual-neuron-free deep neural network computing[C]//Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on. IEEE, 2016: 1-13.

[37] HAN S, KANG J, MAO H, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga[C]//Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2017: 75-84.

[38] DU Z, FASTHUBER R, CHEN T, et al. Shidiannao: Shifting vision processing closer to the sensor[C]//ACM SIGARCH Computer Architecture News: volume 43. ACM, 2015: 92-104.

[39] CHEN Y H, KRISHNA T, EMER J S, et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks[J]. IEEE Journal of Solid-State Circuits, 2017, 52(1): 127-138.

[40] JOUPPI N P, YOUNG C, PATIL N, et al. In-datacenter performance analysis of a tensor processing unit[C]//Proceedings of the 44th Annual International Symposium on Computer Architecture. ACM, 2017: 1-12.

[41] FARABET C, MARTINI B, CORDA B, et al. Neuflow: A runtime reconfigurable dataflow processor for vision[C]//Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on. IEEE, 2011: 109-116.

[42] ANGSHUMAN P, MINSOO R, ANURAG M, et al. Scnn: An accelerator for compressed-sparse convolutional neural networks[J]. In 44th International Symposium on Computer Architecture, 2017.

[43] SRIVASTAVA N, HINTON G E, KRIZHEVSKY A, et al. Dropout: a simple way to prevent neural networks from overfitting.[J]. Journal of Machine Learning Research, 2014, 15(1): 1929-1958.

[44] HOLI J L, HWANG J N. Finite precision error analysis of neural network hardware imple-

mentations[J]. IEEE Transactions on Computers, 1993, 42(3): 281-290.

[45] VENKATARAMANI S, RANJAN A, ROY K, et al. Axnn: energy-efficient neuromorphic systems using approximate computing[C]//Proceedings of the 2014 international symposium on Low power electronics and design. ACM, 2014: 27-32.

[46] PILLAI P, SHIN K G. Real-time dynamic voltage scaling for low-power embedded operating systems[C]//ACM SIGOPS Operating Systems Review: volume 35. ACM, 2001: 89-102.

[47] OLSHAUSEN B A, FIELD D J. Emergence of simple-cell receptive field properties by learning a sparse code for natural images[J]. Nature, 1996, 381(6583): 607.

[48] BOUREAU Y L, CUN Y L, et al. Sparse feature learning for deep belief networks[C]// Advances in neural information processing systems. 2008: 1185-1192.

[49] LEE H, EKANADHAM C, NG A Y. Sparse deep belief net model for visual area v2[C]// Advances in neural information processing systems. 2008: 873-880.

[50] LEE H, BATTLE A, RAINA R, et al. Efficient sparse coding algorithms[J]. Advances in neural information processing systems, 2007, 19: 801.

[51] SIMONYAN K, ZISSERMAN A. Very deep convolutional networks for large-scale image recognition[J]. arXiv preprint arXiv:1409.1556, 2014.

[52] HENNEAUX M, TEITELBOIM C. Quantization of gauge systems[M]. Princeton university press, 1992.

[53] MACKAY D J. Information theory, inference and learning algorithms[M]. Cambridge university press, 2003.

[54] HAN S, MAO H, DALLY W J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding[J]. arXiv preprint arXiv:1510.00149, 2015.

[55] Coded representation of picture and audio information progressive bi-level image compression [C]//ISO/IEC International Standard 11544:ITU-T Rec.T.82. 1993.

[56] HUFFMAN D A. A method for the construction of minimum-redundancy codes[J]. Proceedings of the IRE, 1952, 40(9): 1098-1101.

[57] WITTEN I H, NEAL R M, CLEARY J G. Arithmetic coding for data compression[J]. Communications of the ACM, 1987, 30(6): 520-540.

[58] LECUN Y, BOTTOU L, BENGIO Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11): 2278-2324.

[59] SRIVASTAVA N, HINTON G E, KRIZHEVSKY A, et al. Dropout : A Simple Way to Prevent Neural Networks from Overfitting[J]. Journal of Machine Learning Research (JMLR), 2014, 15: 1929-1958.

[60] KRIZHEVSKY A. cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks[Z]. 2012.

[61] HE K, ZHANG X, REN S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016: 770-778.

[62] SAK H, SENIOR A W, BEAUFAYS F. Long short-term memory recurrent neural network architectures for large scale acoustic modeling.[C]//Interspeech. 2014: 338-342.

附录A 论 文 规 范

致　谢

在研究学习期间，我有幸得到了三位老师的教导，他们是：我的导师，中国科大 XXX 研究员，中科院 X 昆明动物所马老师以及美国犹他大学的 XXX 老师。三位深厚的学术功底，严谨的工作态度和敏锐的科学洞察力使我受益良多。衷心感谢他们多年来给予我的悉心教导和热情帮助。

感谢 XXX 老师在实验方面的指导以及教授的帮助。科大的 XXX 同学和 XXX 同学参与了部分试验工作，在此深表谢意。

在读期间发表的学术论文与取得的研究成果

已发表论文

1. A A A A A A A A A
2. A A A A A A A A A
3. A A A A A A A A A

待发表论文

1. A A A A A A A A A
2. A A A A A A A A A
3. A A A A A A A A A

研究报告

1. A A A A A A A A A
2. A A A A A A A A A
3. A A A A A A A A A