

# AlexNet 调研报告

---

## AlexNet 调研报告

### 一、背景

1. 机器学习与神经网络

2. 卷积神经网络

3. AlexNet 与 ImageNet 图像识别挑战赛

### 二、原理

1. 训练一个神经网络

2. 卷积与池化

    卷积

    池化

3. 全连接层

4. Softmax 归一化

5. 前向传播

6. 反向传播算法

    (1). 计算梯度

    (2). 更新参数

7. AlexNet 结构

    (1). 总体结构

    (2). 激活函数

    (3). Dropout 层

    (4). 双 GPU 并行计算

8. Agilio SmartNIC 在训练 AlexNet 上的优势

### 三、AlexNet 在 Agilio SmartNIC 的实现

1. 在 Agilio SmartNIC 上实现 AlexNet 的难点

2. 实现思路

    (1). 简化结构

    (2). 算法实现

4. C 伪代码示例

### 四、参考文献

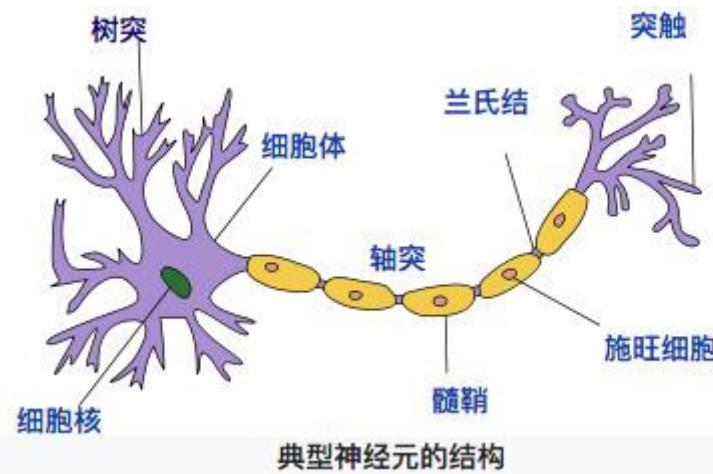
## 一、背景

### 1. 机器学习与神经网络

人工智能是人类迄今为止创造出的最为强大的一种通用性工具。它的通用与强大体现在它可以被应用于一切曾需要人脑智能解决的实际问题中，并且能在某些方面弥补单纯依靠人脑智能所带来的局限性。

作为人工智能领域一个重要的子集，机器学习方法已是家喻户晓，它的目标是通过提出优秀的算法与模型，优化机器对特定领域知识的学习拟合能力与对信息的综合处理决策能力。

神经网络是通过对人脑中数以亿计的神经元细胞处理与传递信息的过程的模仿与高度抽象而得到的一种机器学习算法。

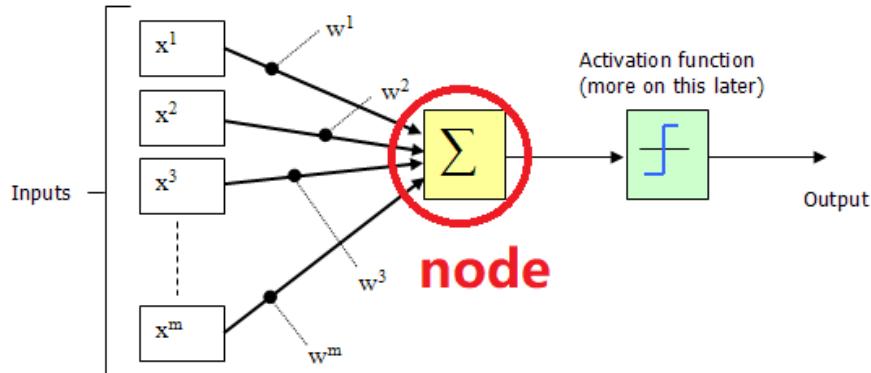


上图：生物体中，信息以电信号形式在神经元之间传播。一个神经元的轴末梢可以多次分支，形成被称为突触的结构，一个神经元通过其数量众多的突触可与数百以计的其它神经元相连，创造出极为复杂的神经网络结构。

下图：神经元被抽象为分层的计算节点（也常被称为神经元）。

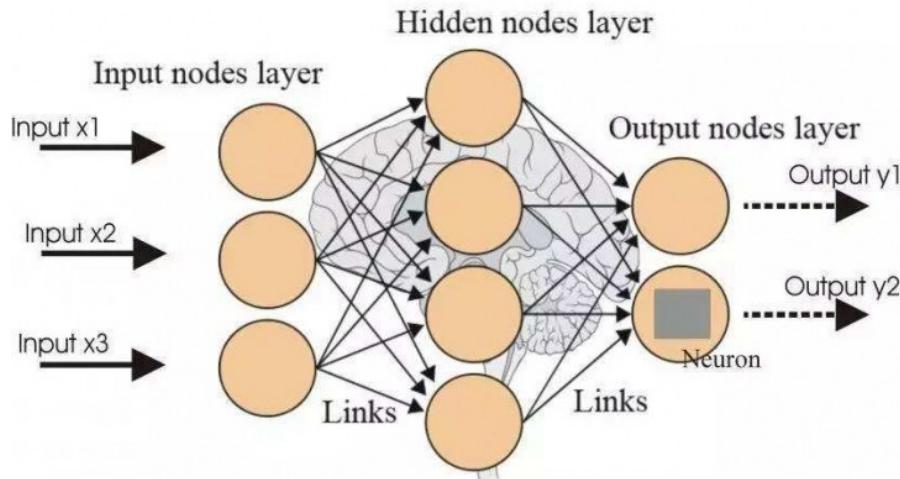
每个神经元的输入数据被乘上权重 (weight)、加上偏置 (bias) 后进行计算，再经激活函数 (Activation function) 进行非线性变换后输出。

$$\text{Output} = \text{Activation}(W \cdot \vec{X} + \vec{b})$$



下图：神经元之间的连接被抽象为层与层之间（节点与节点之间）计算数据的传递，网络的层数被称为深度。输入输出层外的节点层被称为隐藏层。

理论上可以证明，即使是后一层神经元只与前一层神经元相连、后层与前层间没有反馈的简单结构下（前馈神经网络，Feedforward Neural Network），只要节点数量足够，一个两层的神经网络也可以拟合一切数学函数。



## 2. 卷积神经网络

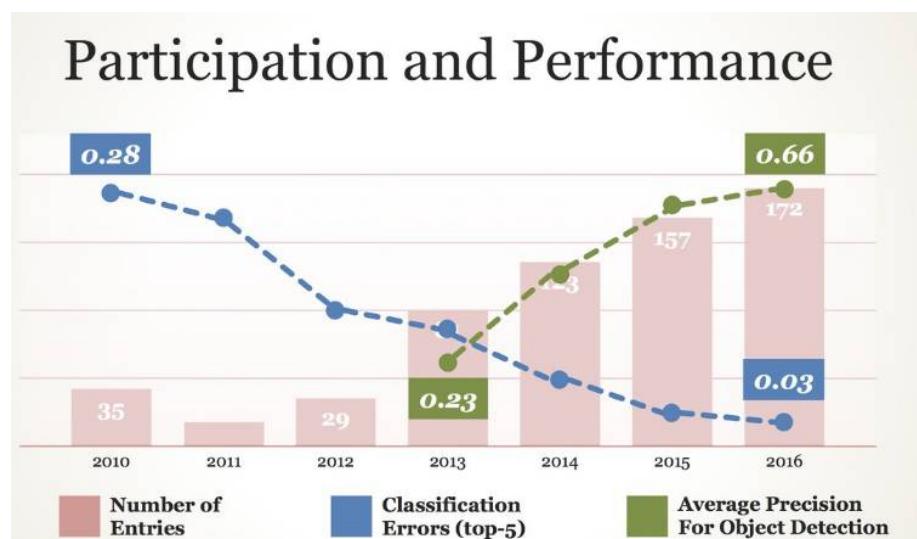
通过仿照与抽象生物的视知觉 (visual perceptual) 结构，研究者们提出了卷积神经网络 (CNN, Convolutional Neural Network)，它是一种包含卷积运算的深度前馈神经网络。

其结构上的独特设计使得它在信息提取与优化计算性能上优势明显，被广泛应用于图像处理等领域。

## 3. AlexNet 与 ImageNet 图像识别挑战赛

ImageNet 是由 Stanford 华人教授李飞飞等人牵头搭建的具有海量分好层次类别的高清晰度、高质量标注的图像数据库。基于 ImageNet 的图像识别挑战赛自 2010 年开始举办，2017 年是最后一届。

下图：总结 2010-2016 年的 ImageNet 挑战赛成果：分类错误率从 0.28 降到了 0.03；物体识别的平均准确率从 0.23 上升到了 0.66。ImageNet 推动图形识别领域发展功不可没。



AlexNet 属于深层卷积神经网络，2015年在 ImageNet 图像识别挑战赛中大放异彩，点燃了研究者们对于深度神经网络算法的热情，在人工智能的发展历程上具有里程碑意义。

区别于此前的神经网络架构，AlexNet 有如下特性：

算法	作用
ReLU & 多个 GPU	提高训练速度
重叠池化	提高精度、不易发生过拟合
局部归一化	提高精度
数据扩充 & Dropout	减少过拟合

- ReLU 作为激活函数。

ReLU 为非饱和函数，论文中验证其效果在较深的网络超过了 Sigmoid，成功解决了 Sigmoid 在网络较深时的梯度弥散问题。

- Dropout 避免模型过拟合。

在训练时使用 Dropout 随机忽略一部分神经元，以避免模型过拟合。在 AlexNet 的最后几个全连接层中使用了 Dropout。

- 重叠的最大池化。

之前的 CNN 中普遍使用平均池化，而 Alexnet 全部使用最大池化，避免平均池化的模糊化效果。并且，池化的步长小于核尺寸，这样使得池化层的输出之间会有重叠和覆盖，提升了特征的丰富性。

- 提出 LRN 层。

提出 LRN 层，对局部神经元的活动创建竞争机制，使得响应较大的值变得相对更大，并抑制其他反馈较小的神经元，增强了模型的泛化能力。

- GPU 加速。

将卷积池化部分分成两组交给两个 GPU 完成，利用 GPU 计算能力增加计算速度。

- 数据增强。

随机从  $256 \times 256$  的原始图像中截取  $224 \times 224$  大小的区域（以及水平翻转的镜像），相当于增强了  $(256 - 224) \times (256 - 224) \times 2 = 2048$  倍的数据量。使用了数据增强后，减轻过拟合，提升泛化能力。避免因为原始数据量的大小使得参数众多的 CNN 陷入过拟合中。

AlexNet 被认为是计算机视觉领域发表的最具影响力的论文之一，它引发了更多的论文采用 CNN 和 GPU 加速深度学习。截至 2019 年，AlexNet 论文已被引用超过 40,000 次。下面将详细解释其原理。

## 二、原理

## 1. 训练一个神经网络

训练神经网络本质上是一个拟合最优化问题。我们的目标是调整神经网络中的参数，使得网络模型根据输入数据得出的输出结果满足我们的预期要求。

为了衡量实际结果与理论预期的偏差，我们引入损失函数 (Cost/Loss function) 的概念。实际中损失函数可以根据数据特点采取均方误差、交叉熵等多种形式。

定义了损失函数之后，我们将神经网络的优化问题转化成了寻找损失函数的最小值点问题。

## 2. 卷积与池化

AlexNet 采用重叠卷积池化的方法，步长小于卷积核的尺寸。

两个卷积层移动步长是 4 个像素，分成两组在两个 GPU 上计算。

ReLU 后的像素层再经过池化运算，池化运算的尺寸为  $3 \times 3$ 。

池化后的像素层再进行归一化处理，归一化运算的尺寸为  $5 \times 5$ ，归一化后的像素规模不变，同样分成两组在两个 GPU 上计算。

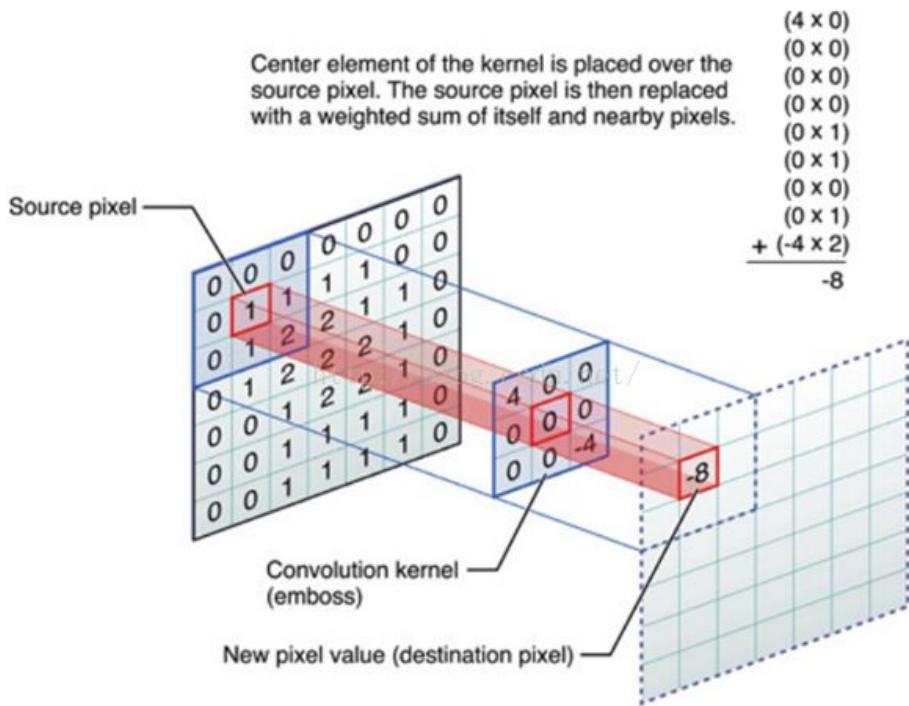
### 卷积

在高等数学中，我们学过，函数  $f(x), g(x)$  的卷积运算为

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(x-t)g(t)dt$$

其中  $g(x)$  可以称为该卷积运算的卷积核 (kernel)。

由于图像在计算机内部以矩阵形式存储，下面我们考虑卷积运算的矩阵形式。以下图为例，直观表示矩阵卷积的过程： $k \times k$  大小的卷积核矩阵与  $m \times n$  大小的输入矩阵进行对应位相乘并求和，得到的结果作为新矩阵中的一个元素。



卷积运算的功能是对图像进行信息的提取。

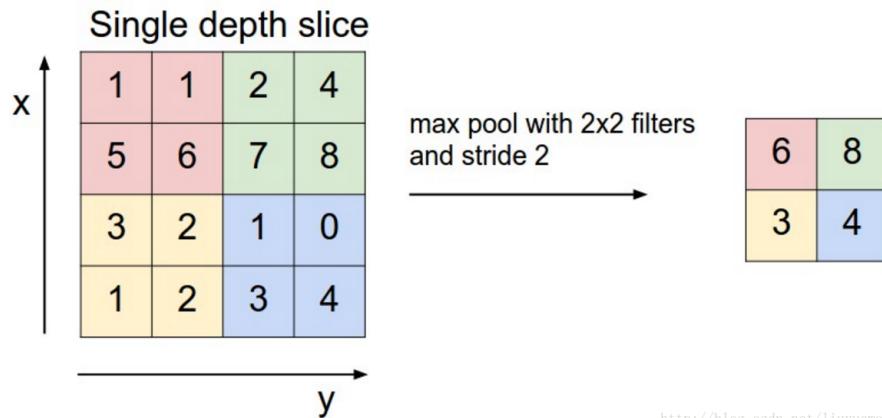
我们可以看到，卷积核每次作用于输入图像上的一个局部区域（被称为感受野）进行运算，可以理解为将该局部位置的特征积累起来得到一个特征值。显然，不同大小、数值的卷积核，提取到的特征也是不同的。通过调整卷积核的大小、数值等参数，我们可以控制对图像特征提取的偏好，达到筛选特征进行分类的目标。

## 池化

池化常是卷积的下一步，也是一种矩阵运算。其目的是通过只保留主要特征、忽略次要特征减少数据量，优化计算复杂度。

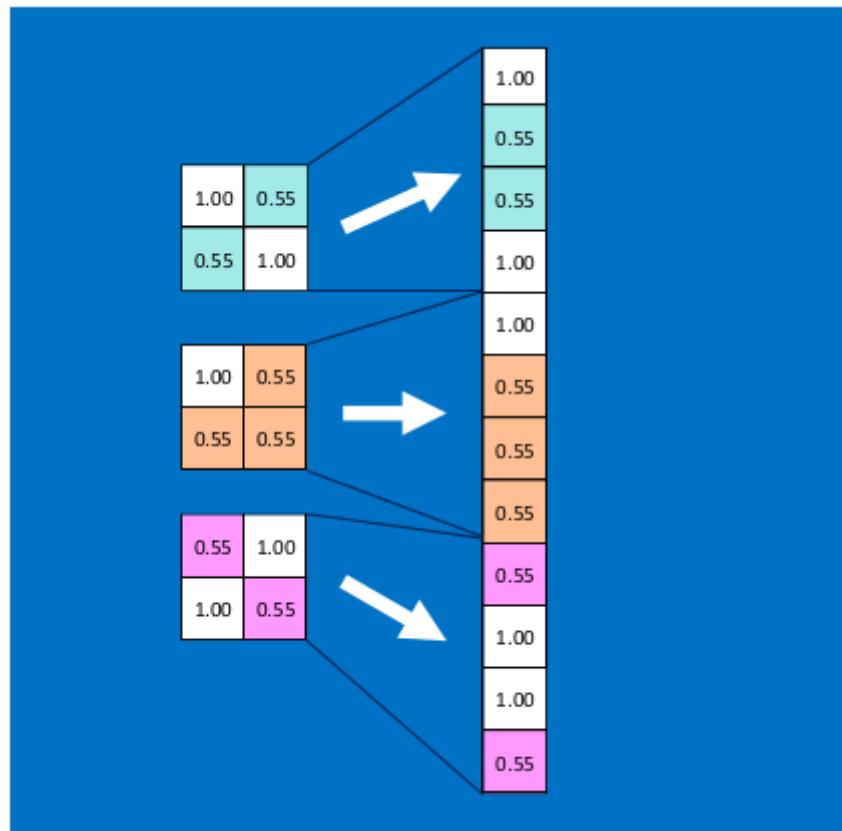
池化有重叠池化 (overlapping pooling)、最大值池化 (max pooling) 等方式。

以“最大值池化”方式为例，如下图，将一个  $4 \times 4$  大小的中间结果矩阵，通过对每个子矩阵取元素最大值，压缩为一个  $2 \times 2$  大小的矩阵进行后续运算。

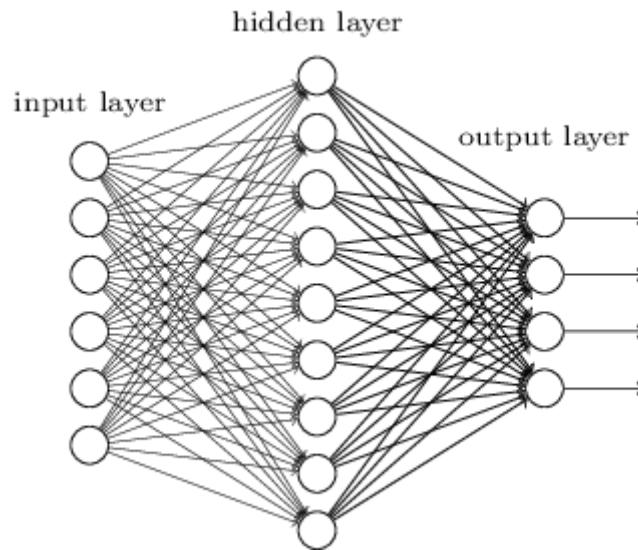


## 3. 全连接层

第一个全连接层 (FC, fully connected layer) 把前层得到的特征全部整合起来，如下图。



全连接层与全连接层之间神经元两两相互连接，形成一个密集的数据传输网络，参数量很大。如下图。



全连接层的存在可以排除特征所在空间位置对特征识别结果的干扰，提高模型的鲁棒性。(实际应用中，也有其他替代全连接层以减少参数量的方法)。

#### 4. Softmax 归一化

Softmax 被用于接收来自全连接层的输入，产生最后的结果（以图像分类问题为例，最终的结果是各个可能类别的概率）。

公式如下：

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=0}^n e^{x_j}} \in [0, 1]$$

Softmax 函数的值域是  $[0, 1]$ , 很容易想到, 这正是输出结果  $\text{label} = x_i$  的概率  $P(x_i)$ 。

在我们小组的实现方案中, 为了方便进行数据处理, 我们将公式中的  $e$  替换为 2。这样幂次可以直接通过移位实现。

$$\text{Softmax}^*(x_i) = \frac{2^{x_i}}{\sum_{j=0}^n 2^{x_j}} \in [0, 1]$$

## 5. 前向传播

在两个全连接层, 计算该层输出结果使用如下公式:

$$L_{i+1} = W_i L_i + b_i$$

对于 ReLU 和 Softmax 层, 函数作用在矩阵上的方式为作用在矩阵的每个元素上。

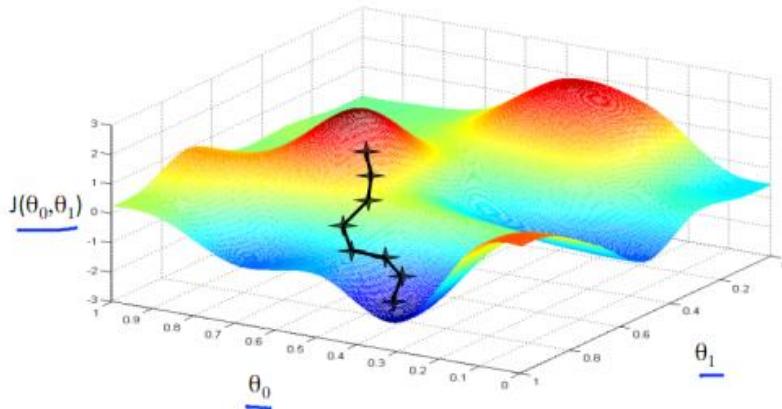
## 6. 反向传播算法

反向传播算法基于简单的梯度下降法。

根据微积分知识我们易知函数梯度的逆方向是函数值下降最快的方向。因此, 对需要调整的参数  $W$ , 若我们能够求出损失函数关于当前  $W$  的偏导数值, 并人为设定基于该偏导数的梯度下降步长  $\eta$  (称为学习率), 可由下公式得到更新后的  $W$ :

$$W' = W - \eta \times \frac{\partial f_{\text{Loss}}}{\partial W}$$

梯度下降直观过程如下图。类似一步步走下山坡知道最低点 (存在的问题是得到的目标点有可能是极小值点而非最小值点)。



### (1). 计算梯度

输入数据以及在 L1、L2 两层的权重和偏置, 用矩阵表示如下:

$$\left\{ \begin{array}{l} W(\text{weight}) = \begin{pmatrix} w_{11} & w_{12} & \cdots w_{1n} \\ w_{21} & w_{22} & \cdots w_{2n} \\ \vdots & & \vdots \\ w_{m1} & w_{m2} & \cdots w_{mn} \end{pmatrix} \\ x(\text{input}) = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \quad b(\text{bias}) = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \end{array} \right.$$

输出层结果可以表示为：

$$\left\{ \begin{array}{l} S(\text{softmax layer}) = \text{Softmax}(W_2 \cdot (\text{ReLU}(W_1 \cdot x + b_1) + b_2)) \\ \text{Loss} = -\ln\left(\frac{e^{S_i-t}}{\sum_{i=0}^n e^{S_i-t}} | i = \text{Label}\right), \quad N = 1 \\ t = \max\{S_i\} \end{array} \right.$$

计算每层输出对于输入的梯度：

$$\left\{ \begin{array}{l} \nabla_S \text{Loss} = \left( \frac{\partial \text{Loss}}{\partial S_0} \quad \frac{\partial \text{Loss}}{\partial S_1} \right) = \left( -\frac{1}{S_0} \quad -\frac{1}{S_1} \right) \\ \nabla_{L_2} S = \left( \frac{\partial S_0}{\partial L_{21}} \quad \frac{\partial S_0}{\partial L_{22}} \\ \frac{\partial S_1}{\partial L_{21}} \quad \frac{\partial S_1}{\partial L_{22}} \right) = (\dots) \\ \nabla_{\text{ReLU}} L2 = \left( \frac{\partial L_{21}}{\partial \text{ReLU}_1} \quad \frac{\partial L_{21}}{\partial \text{ReLU}_2} \quad \frac{\partial L_{21}}{\partial \text{ReLU}_3} \\ \frac{\partial L_{22}}{\partial \text{ReLU}_1} \quad \frac{\partial L_{22}}{\partial \text{ReLU}_2} \quad \frac{\partial L_{22}}{\partial \text{ReLU}_3} \right) \\ \nabla_{L_1} \text{ReLU} = (1 \ 1 \ 1) \end{array} \right.$$

由链式法则：

$$\frac{\partial \text{Loss}}{\partial L_{P_{ij}}} = \frac{\partial \text{Loss}}{\partial L_i} \cdots \frac{\partial L_j}{\partial L_{P_{ij}}}$$

这样可以得到偏差对于每层输入的梯度表达式。

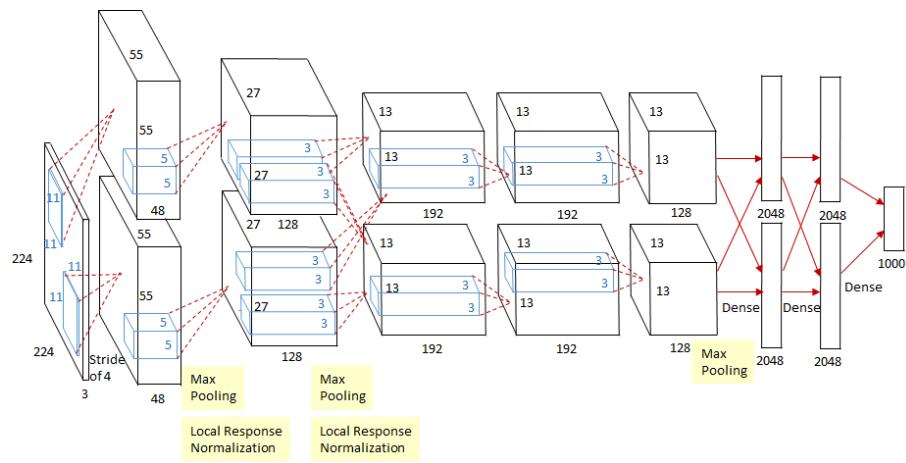
## (2). 更新参数

$$LP_{ij}^{\text{new}} = LP_{ij} - \eta \cdot \frac{\partial \text{Loss}}{\partial L_{P_{ij}}}$$

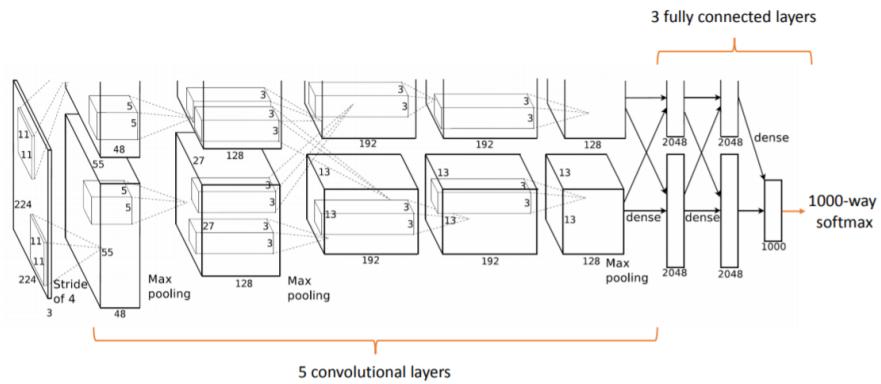
其中  $\eta$  为 学习率。

## 7. AlexNet 结构

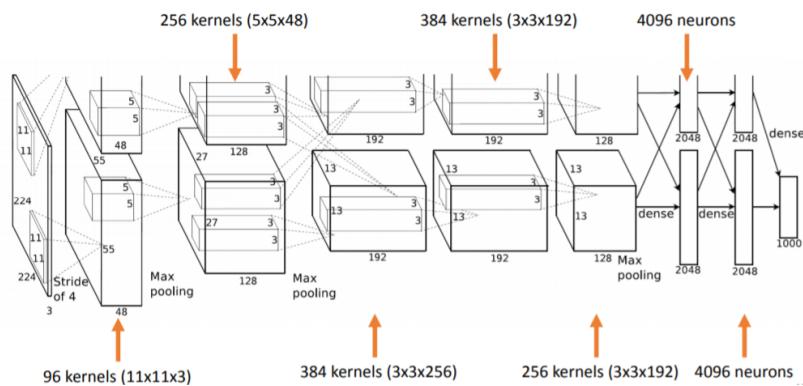
### (1). 总体结构



AlexNet 共有 8 层，前 5 层为卷积（含池化）层，后 3 层为全连接层。



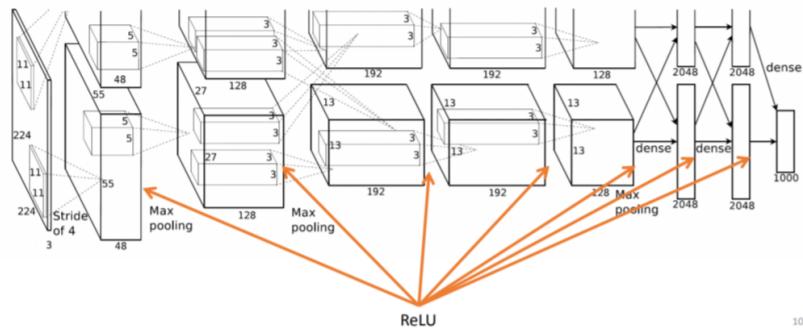
各层的参数数量如下图：



## (2). 激活函数

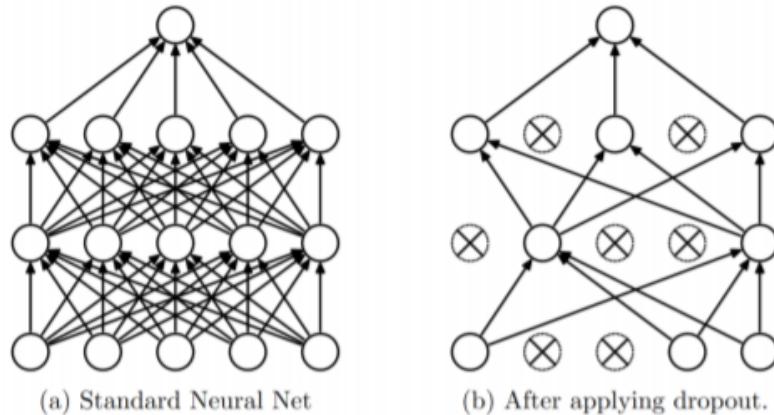
AlexNet 使用 ReLU 作为神经元的激活函数。

$$\text{ReLU}(x) = \max(x, 0)$$

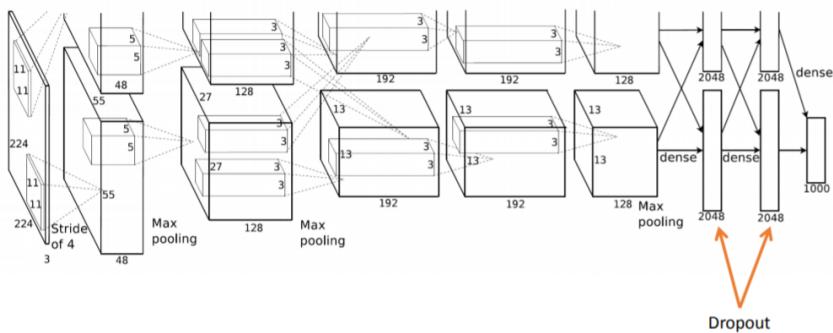


### (3). Dropout 层

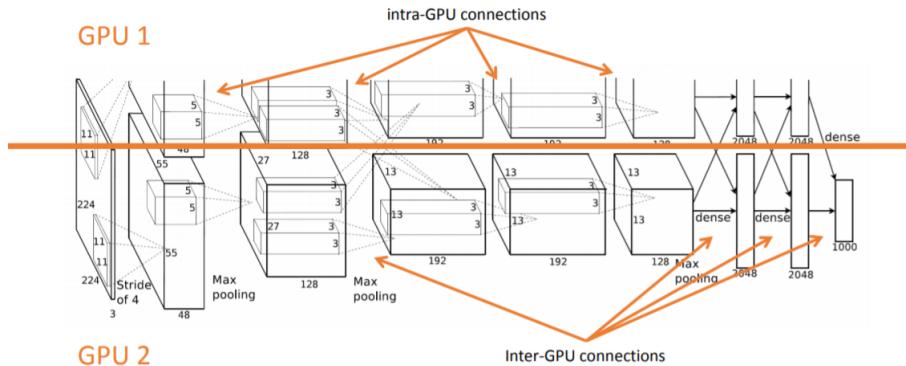
Dropout 是训练神经网络中常使用的防止过拟合的一种 trick (不得不提一下已被 Google 申请专利) , 原理是在每一次训练中随机选取一部分神经元不参与训练。 (如下图)



AlexNet 中 Dropout 用在两个全连接层中。



### (4). 双 GPU 并行计算



两路之间不进行通信，数据最后在 Softmax 层中进行汇总。

## 8. Agilio SmartNIC 在训练 AlexNet 上的优势

1. 数据流架构与 NPU。
2. 可编程的数据通路。

## 三、AlexNet 在 Agilio SmartNIC 的实现

### 1. 在 Agilio SmartNIC 上实现 AlexNet 的难点

- 真正的 AlexNet 参数量巨大，约 650K 个神经元，60M 个参数，630M 次连接，复现极为复杂，更何况我们需要从头造轮子，时间上不允许。
- AlexNet 训练过程中使用的 Dropout 实现起来很困难。
- 涉及大量矩阵运算，尚未找到明确资料说明如何编程利用智能网卡对矩阵运算的支持单元。
- 最大的问题是参数的训练：BP 算法涉及到对损失函数求偏导，这在网络比较小时，可以采用提前求好算式把其写在程序里的方式实现，但是网络大时这种方式不可行。
- Agilio SmartNIC 上程序大小受到严格限制，存储能力有限，并且多核间通信以及保持数据一致性是一个很大的问题。
- 需要相对准确的时序控制来保证多个核心之间的正常协作，需要考虑给多核分配不同的任务，并且在合适的时间进行计算与记录。

### 2. 实现思路

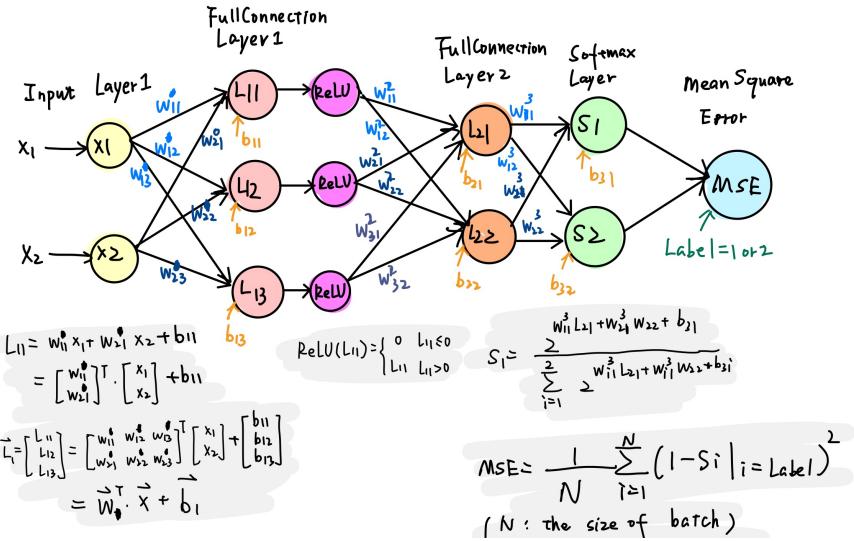
“怕什么真理无穷，进一寸有一寸的欢喜。”

——胡适

#### (1). 简化结构

我们的目标是实现一个基础的卷积神经网络，其结构如下：

输入 → 卷积层1 → ReLU → 池化层1 → 全连接层1 → Softmax → 输出



- 损失函数

采用均方误差函数 (Mean Square Error)

$$f_{Loss} = \frac{\sum_{i=1}^N (1 - S_i |_{i=Label})^2}{N}$$

其中 N 是一组训练样本 (batch) 的大小。

- Softmax

如前所述，使用 2 替代公式中的 e

$$\text{Softmax}^*(x_i) = \frac{2^{x_i}}{\sum_{j=0}^n 2^{x_j}} \in [0, 1]$$

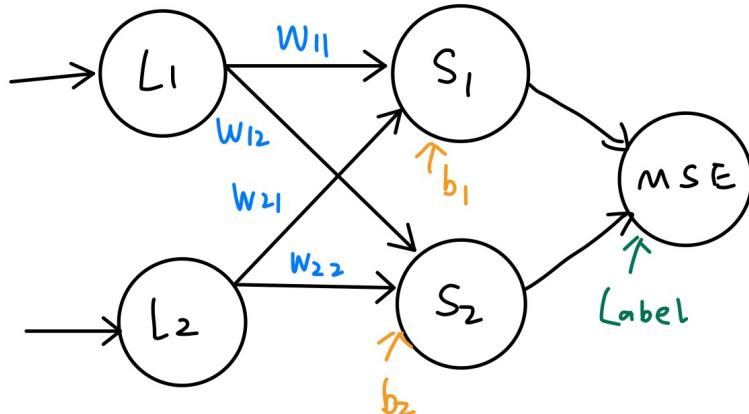
则

$$S_i = \frac{2^{W_{1i}^3 L_{21} + W_{2i}^3 L_{22} + b_{3i}}}{\sum_{i=1}^2 2^{W_{1i}^3 L_{21} + W_{2i}^3 L_{22} + b_{3i}}}$$

- 训练方法

由于网络结构简单，可以由 BP 算法直接写出各参数的更新公式。

以全连接层 2 为例：



$$\begin{aligned}
 \frac{\partial f_{Loss}}{\partial \vec{W}} &= \left[ \frac{\partial f_{Loss}}{\partial W_{11}}, \frac{\partial f_{Loss}}{\partial W_{12}}, \frac{\partial f_{Loss}}{\partial W_{21}}, \frac{\partial f_{Loss}}{\partial W_{22}} \right] \\
 &= \frac{\partial f_{Loss}}{\partial \vec{S}} \frac{\partial \vec{S}}{\partial \vec{W}} \\
 &= \frac{2}{N} \times \left[ n_1 \frac{d(1 - S_1)}{dS_1}, n_2 \frac{d(1 - S_2)}{dS_2} \right] \begin{bmatrix} \frac{\partial S_1}{\partial W_{11}} & \frac{\partial S_1}{\partial W_{12}} & \frac{\partial S_1}{\partial W_{21}} & \frac{\partial S_1}{\partial W_{22}} \\ \frac{\partial S_2}{\partial W_{11}} & \frac{\partial S_2}{\partial W_{12}} & \frac{\partial S_2}{\partial W_{21}} & \frac{\partial S_2}{\partial W_{22}} \end{bmatrix}
 \end{aligned}$$

采用差分方式代替偏导。并将各参数放大  $2^7 = 128$  倍以提高精度。

$$\frac{\partial S_1}{\partial W_{11}} = \frac{S_1(W_{11}) - S_1(W_{11} - \Delta W_{11})}{W_{11} - \Delta W_{11}}$$

然后利用梯度下降公式更新参数：

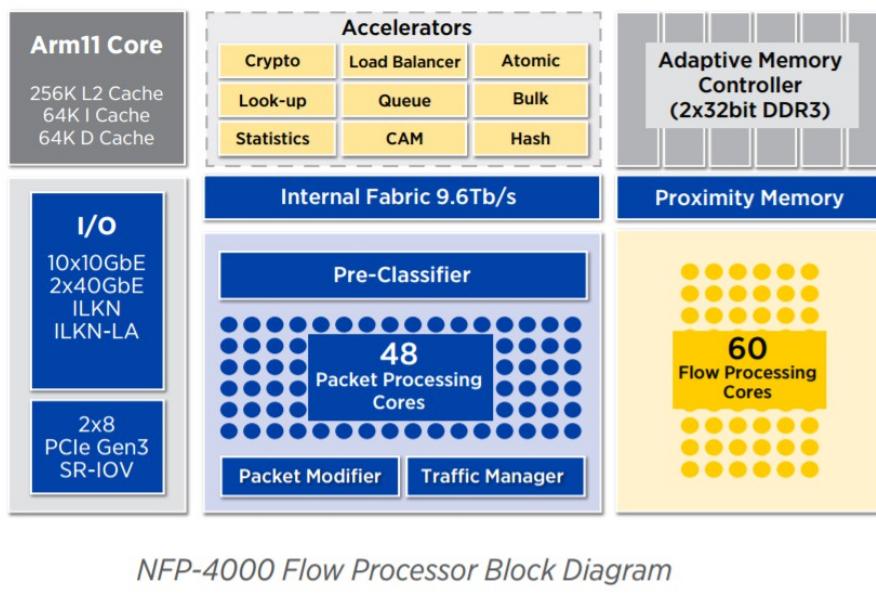
$$W' = W - \eta \times \frac{\partial f_{\text{Loss}}}{\partial W}$$

## (2). 算法实现

Agilio SmartNIC 上有很多相对独立的流处理核心，可以给每个核心分配不同的计算任务，使其充当上述神经网络中某一个节点或者计算对应的梯度，并且给两个隐藏层分配一定的存储空间来储存权重矩阵和偏置参数以及中间数据。此外还应有整体控制模块，用来发出信号，控制每个核心的工作顺序。

计算节点在接收到信号后开始从固定位置获取信息，并完成该节点的计算任务，然后在合适的时间存储，以供下一层节点使用。

整个迭代过程分为输入数据，前向传播，计算梯度，更新参数。其中后面三个阶段均会涉及到多个处理核心之间的通信，以及核与存储之间大量数据读写。



上图是 NFP-4000 流处理器的主要功能部件示意图，其包含 60 个可编程的流处理核心，19MB 的片上存储，并且支持 2\*32b 或 64b 的 DDR3。

## 数据存储

整个网络的数据包含两个主要部分：权重、偏置等参数（大型矩阵）以及当前正在计算的中间结果、梯度（向量）。前者需要更多的存储空间（MB 级别），并且在每次反向传播完成后才会修改。因此，将权重、偏置等参数存放在外部的 Adaptive Memory 中，而当前正在计算的数据记录在寄存器中。

官方文档（“Network Flow C Compiler User's Guide”）中给出了寄存器和内存的介绍：

### a). 寄存器

- 通用寄存器 General Purpose Register: GPR

- 传输寄存器 Transfer Registers: XFR
- 邻居寄存器 Next Neighbor Register: NN
- 不定寄存器 Volatile Register

每一个 NFP 支持 256 个通用寄存器，这些寄存器被划分成两个 banks - A&B，这里需要注意的是每个指令周期内只能读取一个 bank 中的一个寄存器，如二元运算  $w = r1 + r2$ ，若  $r1$  与  $r2$  在同一个 bank，编译器会在编译时隐性增加数据转移指令将其中一个数据先移到不同 bank。

每个NFP还支持 512 个传输寄存器（其中256个 Transfer\_In registers for I/O, 256 个 Transfer\_Out registers for I/O）

并且每个NFP有128个邻居寄存器（NN）。NN 可以用于两个相邻NFP核心之间的通信，是我们需要重点关注的。NN有两种工作模式，可以对 CTX\_ENABLE CSR 的NN\_MODE 位进行修改。

当 NN\_MODE = 0 时，核心 A 不能向自己的NN中写数据而只能读，但可以向相邻的核心 B 的NN中写数据；NN\_MODE=1时，核心A只能读写自己的NN。

### b). Memory

包含了处理器核心内的 Local Memory 以及处理器外部的：

- SRAM (为了向后兼容而留下)
- MEM (包含IMEM, EMEM和CTM)
- Cluster Local scratch (CLS)

每个NPU都有一个私有的Local Memory，大小是1024 longwords。

需要注意的是，寄存器与 Momory 数据交换时需要使用传输寄存器（XFR）

XFR有 read XFR (作为Memory source) , write XFR (作为Memory destination) (P45)。需要注意的是，C代码中涉及对内存数据的读写时，编译器会自动保证数据的同步性。 (P44)

此外在官方文档 (“Microcode Standard Library Reference Manual”) 中提供有 `buf_alloc()` 和 `buf_free()` 函数，可以在程序内分配和释放 S/DRAM 的存储空间。以及控制 sram 读写的以及直接对存储内容增减的函数，包括 `sram_read()`、`sram_write`、`sram_bits_clr()`、`sram_bits_set()`、`sram_add()`…… (见文档 2.24) 并且提供了实现队列的一系列函数。

## 数据计算

计算主要涉及到前馈中的矩阵运算、Softmax 函数的指数运算、高精度除法和大规模的累加。

查阅文档并未发现有专门针对矩阵运算的函数，因此可以考虑利用硬件结构进行矩阵乘法的优化。注意到多个流处理核心可以并行工作而且计算完成后的到的是一个  $n*1$  的向量，因此可以并行计算矩阵乘法中每行的乘加计算，并直接存放到向量中元素对应位置，从而实现时间复杂度为  $O(n)$  的矩阵乘法。

对于乘法计算，

查阅文档 (Micro-C Standard Library Reference Manual) , 以使用 intrinsic function 实现乘法运算:

```

1 // 16b * 16b
2 unsigned int multiply_16x16(unsigned int x, unsigned
int y)
3
4 //取低32b
5 unsigned int multiply_32x32_lo(unsigned int x, unsigned
int y)
6
7 //取高32b
8 unsigned int multiply_32x32_hi(unsigned int x, unsigned
int y)

```

例如:

$$y_1 = w_1 \times x_1 + b_1$$

$$y_2 = \text{ReLU}(y_1)$$

```

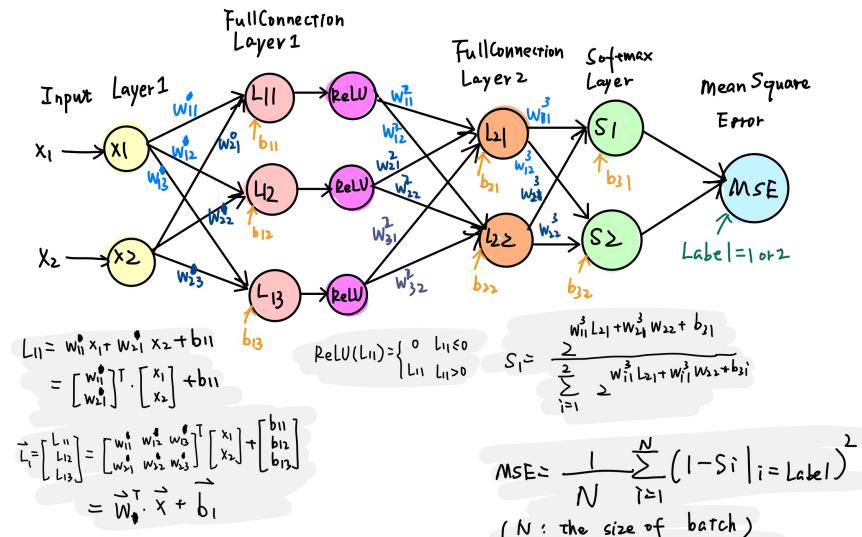
1 __declspec(gp_reg) unsigned int y1,y2,w1,b1;
2
3 y1= multiply_16x16(w1,x1)+b1;
4
5 y2=(y1>0)?y1:0;

```

对于 Softmax 函数, 可以将 e 指数对数运算简化为 2 次幂, 以及 2 对数, 在文档 2.15 提供了相应的除法和对数运算 `math_log2(out_result, in_arg, IN_ROUND)`、`math_int_div(out_q, in_numerator, in_denominator)`。

### 核心之间数据共享与同步

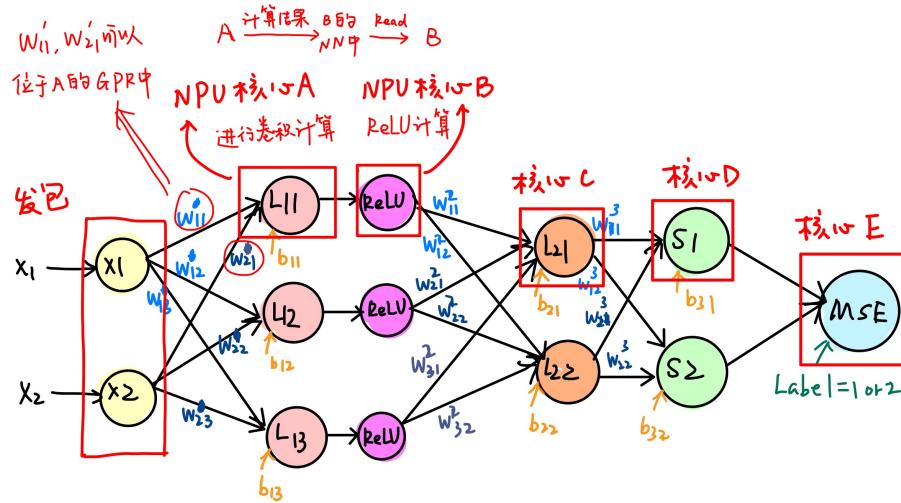
我们知道, 多个计算核心之间需要进行通信, 比如下图中:  $L_{11}$  节点与ReLU 节点需要通信



设计如下图:

通信参考：

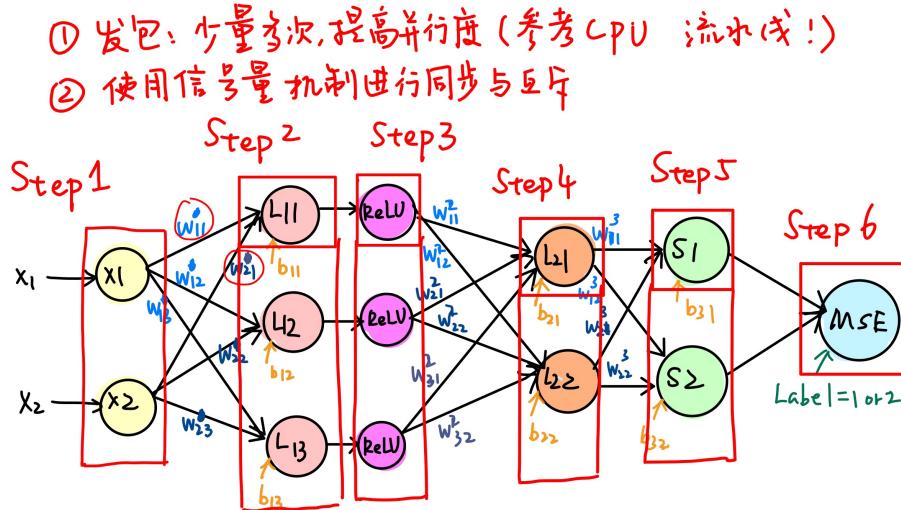
1. Next Neighbor Register
2. 一种名叫 Reflector 的运算 (UG\_nfp6000\_nfcc.pdf P46)



分析后发现：

$(w_{11}^1, w_{21}^1, b_{11})$ 这三个参数可以存储在核心A ( $L_{11}$ 节点) 的GPR中。因为这三个参数并未被其他节点所使用。

但还有一个核心计算先后顺序的问题。如下图：



信号量机制参考：

1. Signals (UG\_nfp6000\_nfcc.pdf P46)
2. Semaphore Library (UG\_nfp6000\_nfcc.pdf)

## 多核心协作

如前所述，若想充分利用多个核心性能，需要同时处理不同计算任务，因此要考虑将多个线程分配给不同的核心处理，这就涉及到多个线程的同步、互斥，并且需要有一个主控程序控制算法整体的运作。

注意到整个算法包含了前馈和反馈两个相互独立的阶段，需要进一步考虑是主控制程序仅负责时序控制，即事先给每个核心分配单一任务，仅在合适的时间开始执行计算任务；还是由主控制程序动态分配每个核心执行的任务，并且负责数据收集和存储。

文档在 2.25 给出了线程同步的机制和相关函数，可以在后续尝试中选择合适的实现方式。

#### 4. C 伪代码示例

```
1 //输入: 8x8(每个像素点0-255, __u8)(只有一层的灰度图)
2 // __u8 image[8][8];
3 //卷积核, 大小暂定5*5
4 // __u8 filter[5][5]= random_initial(); //随机初始化
5
6 //卷积:
7 __u8 ** Convolution(__u8 *image[], __u8 *filter[]){
8     //卷积核 大小5x5, 步长1, 这样卷积出来的结果是一个4x4的矩阵
9     //卷积核中25个参数是要训练得到的
10    __u8 conv_result[4][4]
11    for(int i=0;i<16){ //卷积核移动
12        //矩阵乘法
13        for(int j=0;j<5;j++){
14            for(int k=0;k<5;k++){
15                result[i/4][i%4] += filter[j]
16                    [k]*image[i/4+k][i%4+j];
17            }
18        }
19        return conv_result;
20    }
21 //ReLU 激活函数
22 __u8** ReLU(__u8 *x[], int n){
23     __u8 result[n][n];
24     for(int i=0;i<n;i++){
25         for(int j=0;j<n;j++){
26             result[i][j] = (x[i][j]>0)?x[i][j]:0;
27         }
28     }
29     return result;
30 }
31
32 //池化: AlexNet中采用最大值池化
33 //卷积的结果是一个4x4的矩阵, 池化后变成2x2的
34 __u8 ** Pooling(__u8 *conv_result[]){
35     __u8 pool_result[2][2];
36     for(int i=0;i<2;i++){
37         for(int j=0;j<2;j++){
38             pool_result[i][j] = max(
39                 conv_result[2*i][2*j],
40                 conv_result[2*i+1][2*j],
```

```

41         conv_result[2*i][2*j+1],
42         conv_result[2*i+1][2*j+1]);
43     }
44 }
45 return pool_result;
46 }
47
48 //全连接层，返回值是一维数组
49 __u8 *FullConnectLayer(__u8 *pool_result[], __u8
FCL_filter0* [], __u8 FCL_filter2* [], ..., __u8
FCL_filter9* [], ){
50     //全连接层应该有10个神经元：对应数字识别
51     __u8 neuron[10];
52     for(i=0;i<10;i++){
53
      Neuron[i]=**Convolution(pool_result,FCL_filter{i});
54     }
55     return neuron;
56 }
57
58 //Softmax层，输出0-9的识别概率
59 int *Softmax(__u8 neuron[]){
60     int probability[10];
61     int sum=0;
62     for(int i=0;i<10;i++){
63         sum+=exp(neuron[i]); //需要改成2
64     }
65     for(int i=0;i<10;i++){
66         probability[i]=exp(neuron[i])/sum;
67     }
68     return probability;
69 }
70 //选出概率最大的作为预测结果
71 int Argmax(int x[],int n){
72     int max=0,arg;
73     for(int i=0;i<n;i++){
74         if(x[i]>max){max=x[i];arg=i;}
75     }
76     return arg;
77 }
78
79 int main(){
80     //一张图片
81     __u8 image[8][8];
82
83     /* input image */
84     //偏置，是不需要训练（?）的参数，先设置为0.1
85     __u8 Bias[4][4]={0.1,...};
86     //学习率，超参数，人为设定，比如说0.4
87     const __u8 eta=0.4;
88     //卷积核初始化，可以全赋值为1
89     __u8 filter[5][5]= random_initial();

```

```

90     __u8 FCL_filter1,...9[2][2]= random_initial();
91
92     //若对数字识别: result=0,1,2,...,9
93     //搭建神经网络:
94     int result=Argmax(
95         Softmax(
96             FullConnectLayer(
97
98                 Pooling(ReLU(Convolution(image,filter)+Bias,4))
99                     FCL_filter0,...,FCL_filter9)),10
100                );
101
102    /*训练: 进行验证, 误差反向传播, 使用BP算法训练参数 */
103
104    //误差可以采用均方误差 (交叉熵要用log, 算了)
105    //每训练一组 (batch) , 一组n张图, 计算一次loss, 然后用
106    //BP算法调参
107    double loss=(sum((result-true_value)*(result-
108    true_value))/n;
109
110    //BP 算法, 需事先把偏导式写出
111    //这里要调整的参数有: 卷积核5x5=25 + FCL卷积核
112    //10x2x2=40 =65个参数
113    wi-=eta*(A*wi+B*wj+C*wk+...);
114
115    printf("Pridiction is %d\n",result);
116 }
```

## 四、参考文献

1. ImageNet Classification with Deep Convolutional Neural Networks
2. 实例详解神经网络的back propagation过程
3. 多类别神经网络 (Multi-Class Neural Networks): Softmax
4. ImageNet
5. AlexNet 结构:
  - a. [http://cvml.ist.ac.at/courses/DLWT\\_W17/material/AlexNet.pdf](http://cvml.ist.ac.at/courses/DLWT_W17/material/AlexNet.pdf)
  - b. [http://vision.stanford.edu/teaching/cs231b\\_spring1415/slides/alexnet\\_tugce\\_kyunghee.pdf](http://vision.stanford.edu/teaching/cs231b_spring1415/slides/alexnet_tugce_kyunghee.pdf)
6. 全连接层: <https://zhuanlan.zhihu.com/p/33841176>