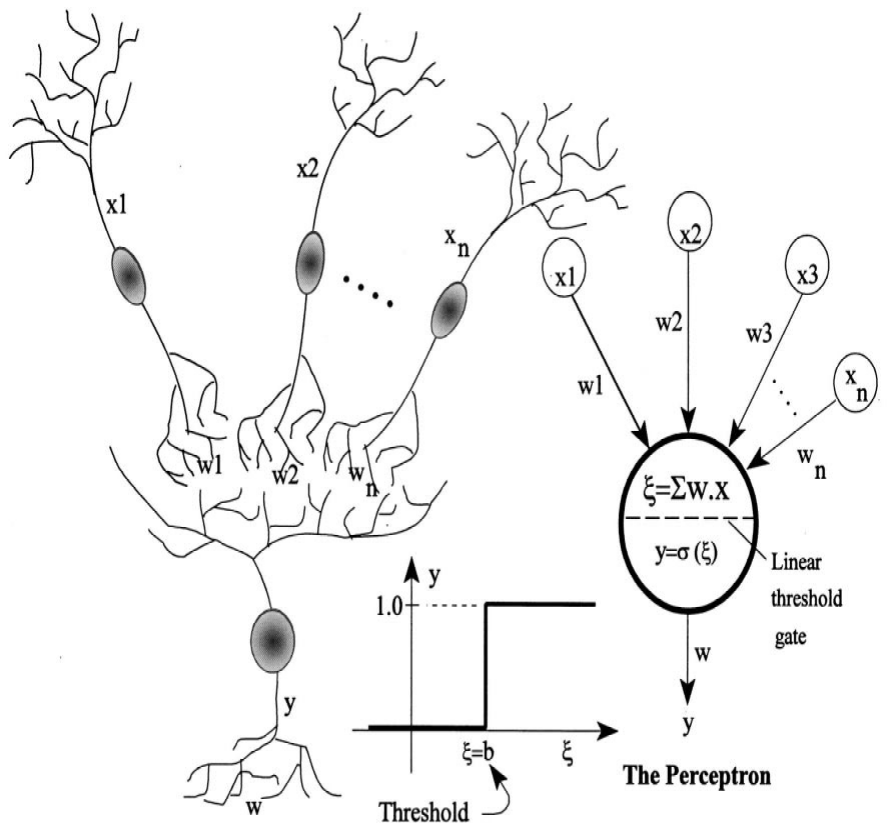
【异构多核是主流，加速器是异构多核的核心】

由于电压控制的存在，芯片上只有少量的晶体管会被同时使用得到，这引发了dark silicon这样的变革，这就导致现有的通过增加核心数量来提升并行度的方法受到了挑战，这就引起了加速程序性能的其他方法的出现。目前最有前途的方法是定制化的方案：传统方式是把程序或者算法分成很多个独立的线程，每一个芯片实际上就是一个专有的硬件，称之为加速器，可以各自执行这些特定的算法，并且比使用通用的芯片来的效率更高

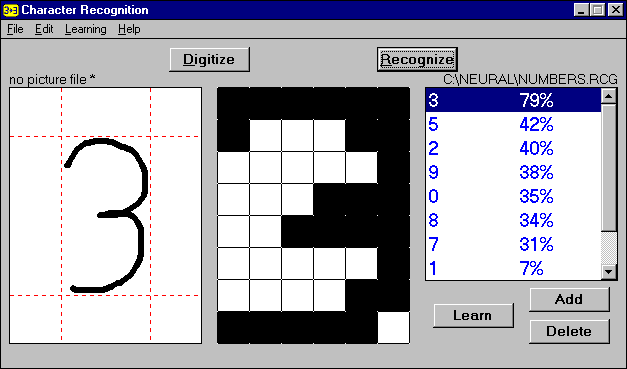
【神经网络算法的重要性，可能需要加上benchnn】

在计算机科学以及相关的计算科学当中，神经网络模型被从生物的中央神经系统当中启发并且抽象出来，现在常用于模式识别以及机器学习领域。动物的神经系统是大量的神经元或者神经细胞相连而成，将末梢神经元产生的刺激信号不断传递；而类似的，人工神经网络一般使用大量的结点互联而成，从一定的输入数据开始，每个结点都会得到相应的输入数据产生相应的输出数据作为下一个结点的输入，通过这样的方式将信息不断的传递到每一个结点，直到获得输出。同样的，动物的神经系统在学习的时候会不断调整神经元之间的突触连接[[1](#_ENREF_1)]，人工神经网络也是要调整结点之间的连接权重等。



《《动物神经网络与人工神经网络的对照图》》（已准备好）

在人工神经网络当中一个比较常见的例子就是手写识别，这也是模式识别的一个具体应用场景：每张手写图片的每一个像素的灰度值作为一个8bit值或者1bit值都可以做为神经网络的输入结点的输入值，而输入值通过激活神经网络当中的其他结点，通过结点之间连接权值的作用，以及结点内部的作用函数，最后输出结点会被激活，通过输出信息反应这张原始图片是哪个数字或者字母的手写图片。



《《手写识别中具象的手写图片通过神经网络重建出抽象的数字》》（已做）

【神经网络发展历史】

神经网络的研究看起来是最近若干年才兴起的研究课题，但是实际上其发展历史绵长且坎坷。

早在1943年时候Warren McCulloch与Walter Pits[[2](#_ENREF_2)]就根据他们自己对当时生物神经网络发展现状的理解发明了人工神经元，他们的神经元设计非常的朴素，处理的信号都是有着固定阈值的二元信号。在他们二人研究成果的基础上，来自IBM的Farley与Clark联合来自McGill大学的生物神经网络专家继续了人工神经网络的研究，这也奠定了以后神经网络的两条发展方向：生理研究与人工智能研究。前者强调不断探索人类大脑的信号处理方式，从而抽象并修复人工神经网络模型，而后者则强调从人工智能的应用场景出发调整人工神经网络模型。

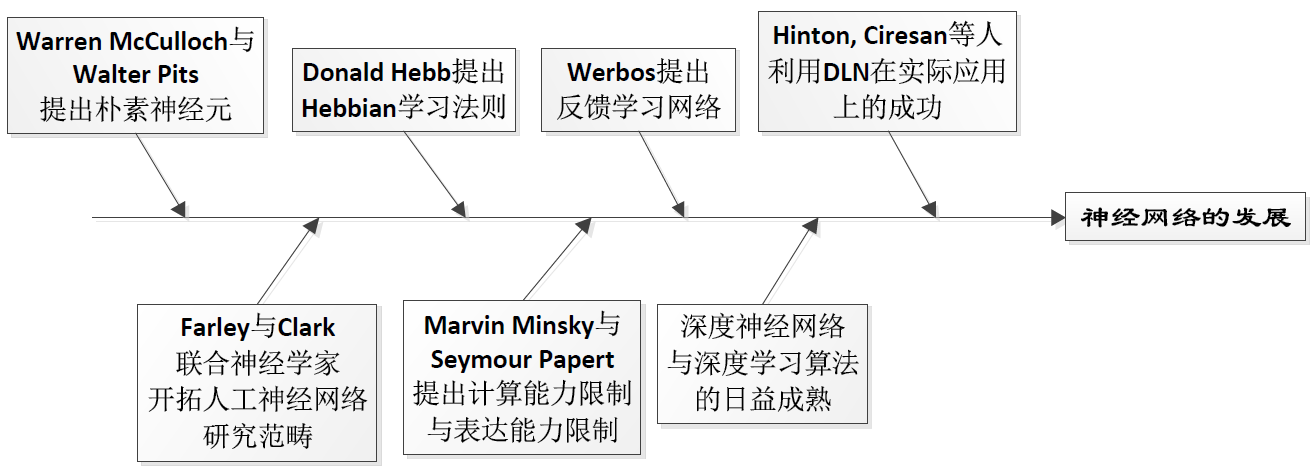
同时在理论上，Donald Hebb根据生物神经元的变动规律提出了人工神经网络的神经元学习假设，现在称之为Hebbian学习法则。该法则成为了典型的无监督学习法则。而Frank Rosenblatt则提出了神经元（perceptron）的模型。

在1969年，Marvin Minsky与Seymour Papert发表了神经网络领域著名的论文，使得神经网络相关的研究都停下了脚步，论文中提出的局限主要在于单层神经网络的模型能力有限，甚至连基本的异或电路都不能模拟，另外当时的计算能力完全不足以支撑大型的神经网络的长时间执行。

虽然不久之后Werbos提出了反馈学习算法就解决了上面提到的第一个问题，并且在二十世纪八十年代中期兴起的并行分布式处理也可以用来模拟神经网络模型，但是在实际研究以及应用的领域，更为简单的支持向量机（support vector machines，svm）以及其他线性分类器的使用则更为广泛。

直到2000年之后深度神经网络算法的兴起，神经网络的研究才重新步入了大众的视线。Geoff Hinton[[3](#_ENREF_3)]以及多伦多大学的研究小组使用的各种反馈神经网络的变形算法都可以被用来训练深度的高度非线性化神经系统。现在常见的深度前馈网络，例如卷积神经网络一般结构都是在卷积层和择优层之外包含分类层。Ciresan[[4](#_ENREF_4), [5](#_ENREF_5)]等人曾经利用GPU实现的深度神经网络获得过两项模式识别方面的比赛的冠军，这足以证明神经网络在现阶段的发展与广泛的应用。

《《神经网络发展鱼刺图》》（已做）



【神经网络形式化描述】

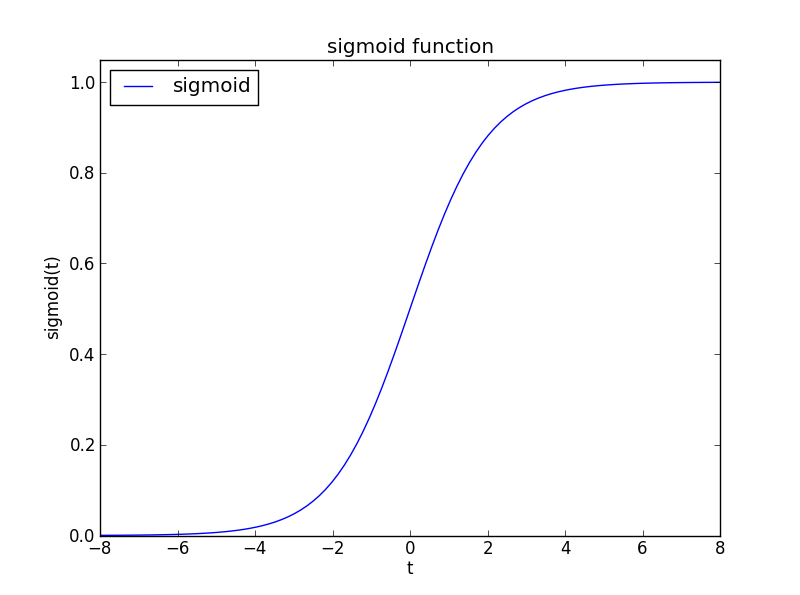
人工神经网络当中的网络主要是指系统当中不同层次的神经元之间的连接，如果系统如下图所示有三层的话，那么第一层仅有输入神经元，通过将输入的信号值通过突触，也就是神经元之间的连接传递给第二层上的神经元。然后通过更多的突触传递给第三层也就是输出层的神经元。更复杂的网络可能会包含更多层次的神经元，也可能会有更多的输入神经元和输出神经元，突触当中存储的数值称之为权值用来对传输的数据的计算进行影响和修正。

一个神经网络是由三组数据确定下来的：突触的权重，权值更新学习的速率以及每一个神经元的激活函数。其中激活函数是每个神经元内部的处理函数，用来将该神经元接收到的输入数据转换为输出数据。

数学上一般用一个类似的神经网络函数f(x)来表示一个确定的人工神经网络，它是由一组其它函数gi(x)组合而成，一个广泛使用的神经网络函数表示形式如下：

其中函数K是预先设定的函数（称之为激活函数），一般使用sigmoid函数或者双曲正切函数，主要目的是为了将变量的范围缩小到0到1之间或者-1到1之间。

Sigmoid函数及其曲线



《《sigmoid函数的曲线》》（已做）

【神经网络的学习过程】

神经网络学习的过程主要是指给定待解决的问题，我们需要找到一组合适的函数集合来解决该问题并且要在某方面达到较有的标准。

在学习的过程当中往往会有一定的耗费函数来衡量特定的函数集合f在学习过程中的表现，当然任何可能的解都不会比最优解的耗费来的低，即。在学习的过程中耗费函数是一个非常重要的概念，它可以告诉我们当前的解与所求的最优解还有多大的差距。神经网络的学习的过程就是通过搜素最合适的函数集合使得其耗散值最小。

有三种常见的抽象学习任务：有监督学习、无监督学习、强化学习。

有监督学习当中，我们会被提前给出一组数对，，我们系统神经网络能够实现这样的映射，来满足之前给出的数对。或者说我们希望找到的神经网络能够表达的映射和实际情况下X与Y之间的映射的差距尽可能的小。这种情况下，我们普遍使用的误差函数一般为均方差函数，希望能够使神经网络的输出结果f(x)与预先给定的输出结果y的均方差尽可能的小。多层神经元的模型一般用来实现这样的神经网络，而梯度下降方法则是人们用来减少误差的常见方式，该神经网络一般用这种学习方式来实现反馈，修改神经网络各神经元之间连接的权重。

一般模式识别问题（或者称为分类问题）和回归问题（或者称为函数拟合问题）都会使用到有监督学习。由于先验知识（已经给定的数对）的存在，有监督学习有时候又被称为有教师学习，用来在给定的数对的基础上不断找到更精确的映射关系。

无监督学习的训练集合没有人为的标定，也就是说，无监督学习只有输入以及特定的限定规则。由于在无监督学习当中网络的调整不受到外界的干预，但是需要有额外的组织规则来负责网络内部的调整，可以认为网络所输出的各个模式是相互关联的，如果特定输入引发了特定模式的出现，那么激发该种模式的网络内部特性都会得到增强，反之则网络内部特性就会得到抑制。无监督学习重用的神经网络模型是自组织图（self organized map）等，无监督学习一般适用于聚类问题以及统计分布估计问题，这些问题往往更倾向于数据在这些神经网络下面的后验信息。

还有其他类型的学习策略如强化学习、混合式学习等，都是能够在实际当中应用起来的学习策略。但是主流的连接主义的学习方法主要还是以上两种策略。

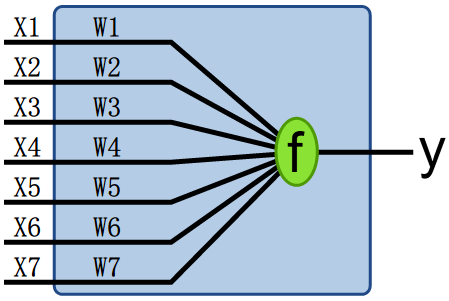
神经网络最为成功的地方就是能够从已经观察到的数据建立起所研究问题的模型，而不需要知道所研究问题的本质特征或者因果关系。但是神经网络在使用的时候并不是那么简单，一般首先要根据要解决的问题以及数据规模数据类型来选择合适的网络模型，并且选择合适的学习算法，不同的学习算法在特定的数据应用和前提假设的情况之下都会有非常出色的表现，但是要在未知的数据上面调整算法以及参数需要非常大量的实验。

下面从两种最为常见的神经网络入手来介绍神经网络具体的使用。

【介绍perceptron及cnn】

神经网络有监督分类学习当中最常见和最基本的模型是perceptron模型，该模型能够如此广泛的应用一来是由于它结构简单，二来是因为对于需要精确推理的问题perceptron模型能够提供足够的健壮性[[6](#_ENREF_6), [7](#_ENREF_7)]。

Perceptron模型的一般结构如下：



《perceptron示意图》

Perceptron是一个二进制分类器可以将输入数据向量映射为单一的二进制输出值f()

其中是实际权值的向量表示，上述公式当中的点乘形式可以得到一个乘积之和，而b则是偏置（bias），这是预先设定好的常量，与输入数据没有关系。

的结果是将分类为正类或者反类的标示，如果b是一个负值，那么点积就需要是一个较大的正值，且其绝对值应超过|b|才能使最终分类结果为正类。因此来说，b实际上是一个决策门限。

而在复杂的人工神经网络当中，perceptron模型是构成神经网络每个结点的基本模型，有时候perceptron模型也被称作单层perceptron。作为线性分类器，单层perceptron是最简单的前馈神经网络。

在介绍单层perceptron的训练之前首先来明确一些概念：

y=f()表示对输入向量的输出结果

b是决策门限，在我们介绍的例子当中设为0

D={(,d1), …, (,ds)}是s个训练样本的集合，其中是一个n维向量，而ds则是对于特定输入的期望输出

表示的是第j次训练样本的第i个结点的值，并且=1

为了表示权重值，代表了权值向量的第i个值，会与输入向量的第i个输入结点相乘，并且由于预先已经设定了=1，因此实际上是决策门限。

为了表示权重随着时间的变化，或者称之为随着训练样本的变化，我们可以采用来表示t时刻的第i个权重值。

在学习的过程当中还会引入学习速率的概念，它表示着权重调整的速率，用来表示，一般取值是0<<1。

因此一般的perceptron的使用过程可以表示为：

1. 初始化决策门限与权值向量，一般选择为0或者接近0的小随机数。
2. 对于训练样本集合D中的第j个样本(,dj)，首先可以计算出实际的输出是：



然后更新权值：

，对于i有0

1. 对于离线训练来说，第二步停止的前提是迭代的方差，作为判定的标准，小于一定的门限。或者迭代的次数已经达到了预先设定的次数上限了。

但是perceptron的表现能力又是极其有限的，它的训练过程在输入向量不是线性可分的情况下永远无法停止，这也就意味着它对于非线性分类是无能为力的，一个最具代表性的例子就是对异或问题的训练。

由于异或问题是线性不可分的，因此简单的perceptron模型并不能够描述并解决足够多的问题。

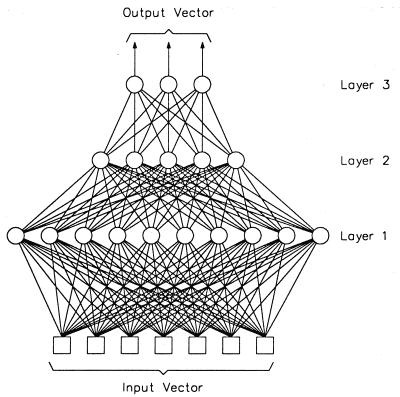
《可以给出异或问题的神经网络中的图示，配上说明》

多层神经元模型应运而生。

多层次神经元模型是在单层次神经元的基础上发展而来，具有更为丰富的描述能力，可以对非线性分类问题进行描述和操作。其基本的训练过程与单层perceptron也是非常相似，前馈的过程也是一致的，唯一的不同在反馈传播的过程。

多层神经元模型采用了成为反向传播（backpropagation）的反馈策略来调整权值，通过这种方法，输入数据可以被反复的呈现给神经网络进行计算和调整，每一次计算得到的实际输出都会与期望的输出进行比较得到误差，根据这一误差神经网络当中的权值可以被调整，使得下一次同样的输入数据通过神经网络计算得到的结果具有更小的误差。这一不断重复的过程就称之为训练的过程。

从具体的问题与神经网络的结构出发可以按照如下步骤来对多层神经元模型或者多层神经网络进行描述：



《多层神经网络的典型模型并带有输入》

如果多层神经网络当中的每一个神经元都仅含有线性激活函数的话，那么多层的神经网络就都可以简化为两层的神经网络了。而多层神经网络之所以内容这么丰富，表述能力这么强大，一个关键的原因在于其每个神经元都有非线性的激活函数，这些激活函数相当于人类大脑中的神经元的作用，在数学上来要求，这些人为规定的非线性激活函数需要能够归一化且可导。一般我们会选择诸如下面的sigmoid函数来作为激活函数：

 或者 

这两个函数一般都会作用在权重向量与输入向量的点积和之上，然后作为输出。其他还有一些比较特别的激活函数适用于特定类型的神经网络。

一般多层神经网络至少是指三层，输入层输出层和至少一层隐含层，当然这些层都是由含有非线性激活函数的结点构成的。每一层次中的每个结点都通过权值来和下一层次当中的结点进行互联。

多层神经网络的一个重要特点就是通过反馈传播进行训练和学习。一般反馈传播的算法如下：

//开始算法描述

对权值进行随机初始化

重复

重复 //每个epoch

选择训练集合当中的一个样本

输入到神经网络

计算出神经网络的输出

将实际输出与理想输出做比较计算

修改权值

直到训练集合当中的所有样本都被应用到

直到全局误差小于预先设定的标准

//结束算法描述

而反馈传播算法的关键就在于定义全局使用的神经网络能量来衡量神经网络的误差，反馈传播的目的就是尽量减少这一网络能量。一般来说我们定义能量函数如下：



其中

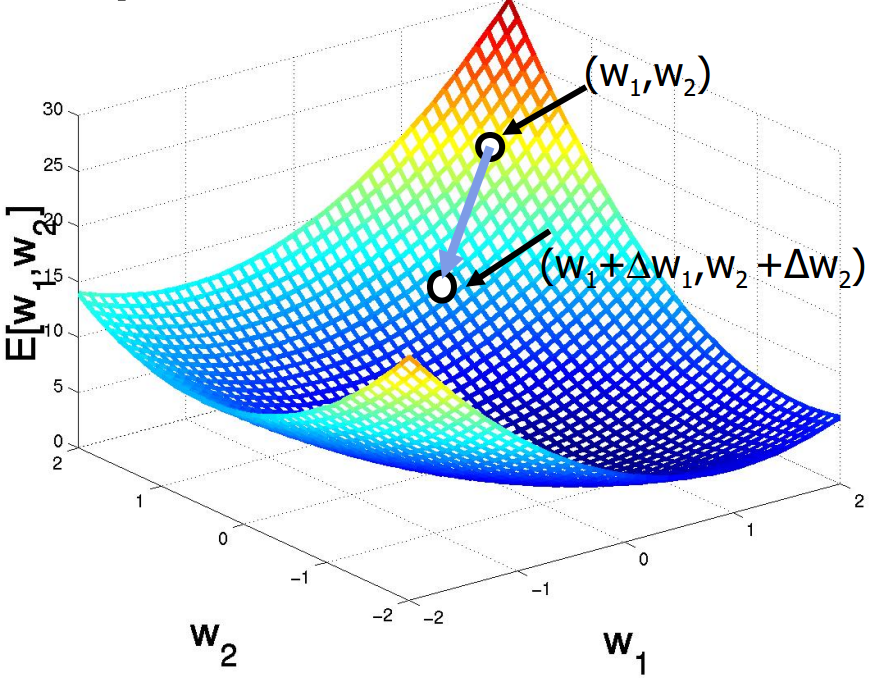


能量实际上是下面的参数：



其中权值是变量，而输入数据则是定值。

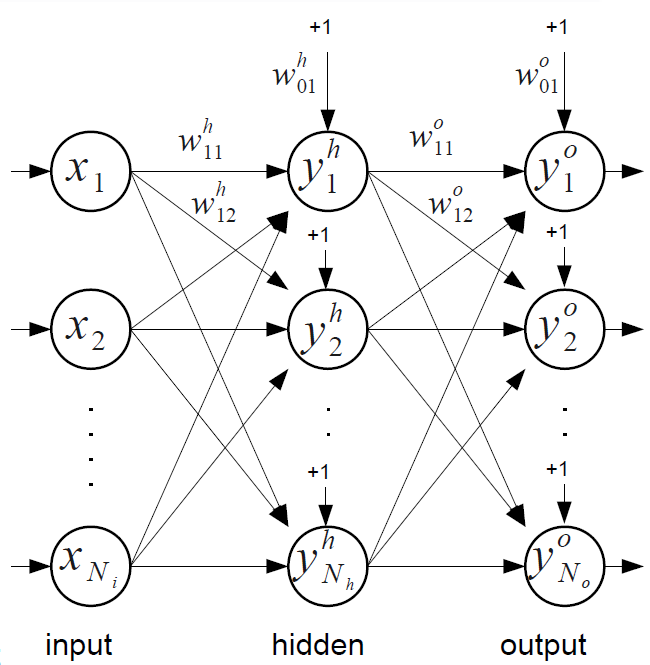
每次输入样本输入我们都能得到一个网络能量的值，我们的目标就是通过修改权值和阈值门限让这一值尽可能减小。因此我们选取了梯度下降算法来降低调整权重降低误差。



《梯度下降算法示意马鞍图》【done】energy/error landscape



如上面公式所示，我们希望权值的变化方向是和梯度（梯度本身是有方向的向量，这里展示的是每一个方向上的偏导）的方向是相反的，其中是学习速率，表示权值调整的速度。



《bp网络层参数示例》【done】

在标准的神经网络一般结构当中：

|  |  |
| --- | --- |
|  | 神经元j和神经元k之间的权值 |
|  | m层中神经元k的阈值 |
|  | m-1层与m层之间连接的权值 |
|  | m层中神经元k的内部加权和 |
|  | m层中神经元k的输出 |
|  | 第k个输入 |
|  | 输入层、隐藏层、输出层的规模 |

而对于能量函数，有：



其中可以用

以及

（其中）

来做替代，得到



进一步分析，在神经网络的输出层有：



带入E的定义，有



前半部分，后半部分记做，为激活函数的微分，带入输入层的权值调整公式有：



而对于隐藏层的权值来说



后面一部分可以进行展开



后半部分和前面的输出层是类似的，都是激活函数的微分：，前半部分则可以展开为：



根据前面推导的结果，可以将上式化简为：



带入原来的式子有：





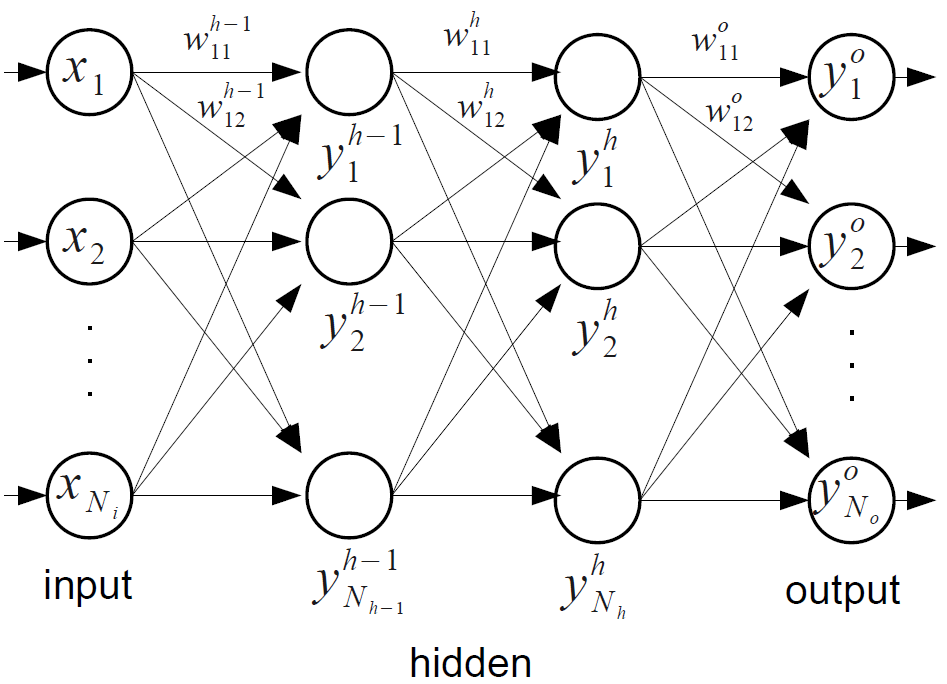
可以得到隐藏层的权值调整为：



关键的问题都在于激活函数的微分，而我们选取的激活函数，无论是sigmoid函数，还是双曲正切函数都是连续且可分的。拿sigmoid函数为例子：



而对于多隐藏层的神经网络来说，权值调整的策略也是一样的，都是通过后面权值来影响前面的权值



《多隐藏层bp神经网络》【done】

如图上所示，h-1层的权值是由h层的来决定的。

在采用sigmoid作为激活函数的情况下，输出层权值调整为：



隐藏层权值调整为：



权值调整规则为：



BP规则非常的流行和经典，现在很多实际实用的神经网络都采用了BP规则作为学习策略。

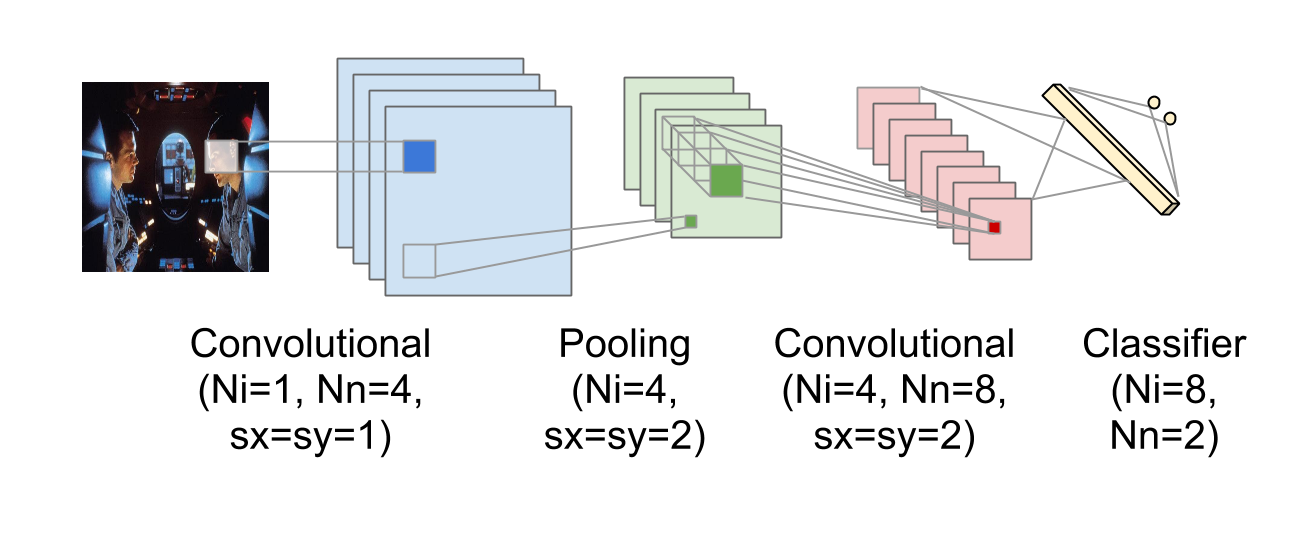
并且多层神经网络相比于单层神经元就有了更广泛的应用，例如建模与控制、机器诊断、文件管理、医疗诊断、信用评级等。

而我们在本篇论文当中主要研究的卷积神经网络（convolution neural network）是受到生物学上的启发，从多层次神经元（multi-layer perceptron，mlp）变形而来的一种神经网络。通过对猫视觉皮层的研究观察[[8](#_ENREF_8)]，我们发现视觉皮层的神经细胞的排列有着极为复杂的规则：这些细胞对于视觉输入的一部分，称之为接受域，并且通过推进平铺这一区域来覆盖整个视觉范围，这样的处理方式能够揭示自然形象的空间关联。

卷积神经网络发展的背景也很有趣，正如前面提到的在2006年，随着神经科学领域模型的发展以及计算机视觉领域[[9](#_ENREF_9)]的兴起，伴随着诸多应用一起进入大家视线的是一系列的神经网络模型，这些模型能够得到相当不错的准确率。其中最有代表性的就是卷积神经网络（convolution neural network）以及深度神经网络（deep neural network）等。这两种神经网络模型有非常高的相关程度，只是在其实现形式以及卷积层的性质上略有差异。

在本篇论文当中我们主要实现的是处理过程而不是训练过程，这一选择是基于技术和市场的综合考虑之后做出的。技术上，虽然有很多人都认为在线的训练和学习过程是非常重要且必须的，但是实际上，对于很多工业上的应用来说，线下训练就已经足够了，神经网络在线下用数据集进行大量的训练，在调整好权值阈值等参数之后交递给用户。例如，对于手写数字的识别，对于车牌号的识别，对于面部识别和物体识别等应用场景当中，充足的线下训练有着非常充裕的时间可以完成，甚至可以定期在后台进行线下重复训练，最后展现给使用者的只是训练好的神经网络模型，而不需要在极少的时间内进行少量的线上训练，那样得到的准确率也非常的低。纵观今日的市场，希望发展加速训练过程的机器学习专家和工程师只是代表了市场的一小部分，面向终端用户的市场则是一个很大的研究方向，他们需要的是高效率的前馈网络。有趣的是，最近一些关注与硬件加速器的机器学习专家[[10](#_ENREF_10)]也做出了同样的选择。另外还有一点值得注意的是，从计算上来看，训练过程，尤其是反馈传播过程的计算形式前馈过程非常相似，在后续的研究过程当中我可以考虑将训练的反馈过程加入到前馈过程之后作为补充。

尽管卷积神经网络和深度神经网络在实现当中有着各种各样的形式与变化，但是它们都有一些共性可以梳理出来，这些模型都是由大量的网络层次堆叠起来的；这些网络层次是串行执行的，之间可以被认为是相互独立的；每一个网络层都会包含数个网络次层，称之为输入特征阵列（input feature maps）和输出特征阵列（output feature maps）等。总体来看，总共有三种网络层次：主要的模型架构是由卷积层（convolution layer）和归并层（pooling layer，又称为二次抽样层subsampling layer），在模型的最上面有分类层（classifier layer）。



《含有卷积层、pooling层、mlp层的神经网络架构》【done】

卷积层

卷积层的作用是将前一层的数据应用到滤波器上，考虑到输入数据是一副图像的情况下，卷积层的作用就是将一个Kx\*Ky大小的窗口作用到一个同样大小的数据字块上的二维变换，如图所示。最核心的数据是输入层和输出层之间的权值

一般在输入层中，往往会包含多个特征数据阵列（feature map），而输出层的特征点的计算往往是将多个特征数据阵列的同一位置的窗口数据都应用卷积操作，其核心是一个三维操作，其规模为Kx\*Ky\*Ni，其中Ni就是输入特征数据阵列的数量。在这种情况之下，与全连接多层神经网络不同，这里的连接一般情况之下是稀疏的，对于每一个输出特征点来说，并不是每一个输入数据都被使用到。卷积层还有一个比较重要的特征是两个连续的窗口之间是有交叠的，如同所示代码当中的x，y循环的sx，sy的步骤。

在有些情况（主要是在卷积神经网络）之下，输入层的所有的Kx\*Ky个窗口都使用同一个核特征阵列（kernel），权重数值是完全一样的，相当于对于整个输入的特征数据阵列来说核特征阵列式共享的，称之为共享核（shared kernel）。而在深度神经网络之中，核特征阵列在不同的输入层窗口之间是各自独立私有的，称之为私有核（private kernel）。

归并层

归并层所起到的作用是把相邻输入数据的信息给集合起来，仍然拿图像问题来举例子，归并层主要起到的作用就是输入数据的每一个窗口之内的最显著的特征给保留下来，正如上图所示，并且这样归并带来的一个非常重要的副作用就是大大降低了输入数据的数据规模和维度。要注意到每一个特征数据阵列都是被归并层独立作用到的，也就是说和卷积层的三维操作不用，归并层采用的是二维操作。归并层所采用的保留特征的方式也是多种多样，有对窗口内所有数据取平均值的，有选取最大值的。而在归并层之后是否添加非线性函数也是可选的。

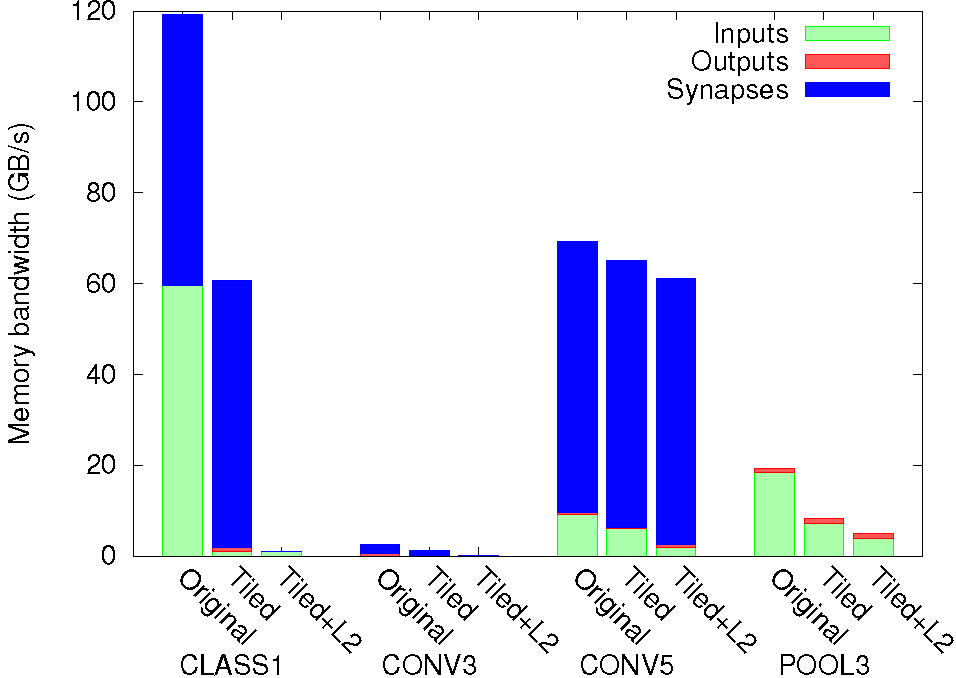
分类层

卷积层和归并层一般都是在卷积网络架构的底层，而在这一架构的顶层往往会搭建一个分类器（classifier），这样的分类器可以是一个简单的线性分类器或者是多层神经元（一般是两层）。与卷积层类似，分类层也需要在神经元的输出之后加上一个非线性函数进行归一化处理，例如，但是和卷积层不同的地方是，分类层对所有的特征数值阵列一视同仁，在分类层当中也就没有了特征数值阵列这一概念了。

加速大规模神经网络的关键在于潜在的高内存传输，在下面这一段文字当中我将就不同神经网络层的数据特性进行具体的分析。对于本段当中的带宽分析，我们使用了一个连接在一个理想计算框架下的cache模拟器上进行的，这个理想计算框架在每一个cycle当中可以计算带有Ti个权值的Tn个神经元。Cache模拟器是仿照Intel Core i7设计的，其参数如下：

|  |  |  |
| --- | --- | --- |
|  | L1 | L2（可选） |
| 大小 | 32KB | 2MB |
| 块大小 | 64-byte | 64-byte |
| 路数 | 8 | 8 |

和core i7的设置不同，这里的cache模拟器有足够多的端口来提供输入神经元的Tn\*4字节以及Tn\*Ti\*4字节大小的数据，在本文的实验当中我们选取Tn=16，Ti=16，虽然在实际应用当中，Tn与Ti都会选择较大的数据。并且这样的cache也是很难设计的，但是对我们的研究工作来说，Tn与Ti的选取已经是足够大的了。

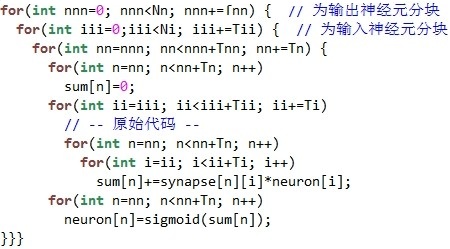


《典型网络层次的带宽研究》

分类层

//分类层代码

或者是《分类层代码截图》



Ii层循环和nn层循环反映的是每个神经元都带有Ti个突触权值的Tn个神经元的计算框架，因此从上面代码当中可以看出总的内存数据迁移数量是Ni\*Nn+Ni\*Nn+Nn个（载入输入数据+载入突触权值数据+写入输出数据），对于benchmark当中的CLASS1来说，对应的内存带宽有120GB/s，是相当巨大的计算量。



《分类层划分示意图》

再次分析上面原始代码以及图【】，可以看到对于各自的输出神经元来说，每一个输入神经元的数据都会被重复使用，可以考虑到将输入神经元的数据放到L1 cache当中。但是一般情况之下，输入神经元的个数可能从几十个到几十万个不等，因此很可能会出现L1 cache当中存放不下的情况。因此我们队输入神经元ii进行了划分，每次划分的规模是Ti，这种方式带来的一个缺点是在提升了输入神经元数据复用性的同时，增加了加权和的复用距离，也就是说对于每个sum[n]进行计算的时候，可能会牵涉到在不同划分当中进行取值，额外增加了缓存与内存之间的数据传输的开销。因此我们也考虑了对nnn循环进行划分，划分规模是Tnn，专门只对部分和进行操作，消除了加权和与输入权值之间的额外距离。这种划分方式能够极大的减少输入神经元的带宽需求，输出神经元的带宽限制得到了放松，这时候带宽的主要限制就放到了突触权值上面。

在神经元模型当中，所有的权值都是独立的，因此神经层内部是没有数据复用的。但是另一方面来看，突触权值在神经层次之间是有复用的，这就成了我们下手的着眼点。例如：对于每一组新输入神经网络的数据之间，突触权值是存在复用的。因此我们一般都会选择一个足够大的L2 cache来存放突触权值信息来利用这种复用。对于使用私有核的深度神经网络来说，其突触权值的数量能达到上亿个，这种想法也是不够现实的；但是对于使用共享核的深度神经网络和卷积神经网络模型来说，总共的突触数量是在百万量级，一个足够大的L2 cache还是能够容纳的了这么大规模的数据的。如图【】所示的CLASS1-Tiled+L2的数据，在我们考虑了这么多的数据复用性并且采取了L1 cache和L2 cache的帮助之下，我们对于内存带宽的要求实际上已经降到了非常低的水平。

卷积层

《卷积层代码截图》【done】



我们考虑一个二维的卷积神经网络层，与分类层的不同点在于，卷积层的输入和输出是多个特征阵列数据，并且拥有kx\*ky的核阵列。



《卷积层划分示意图》

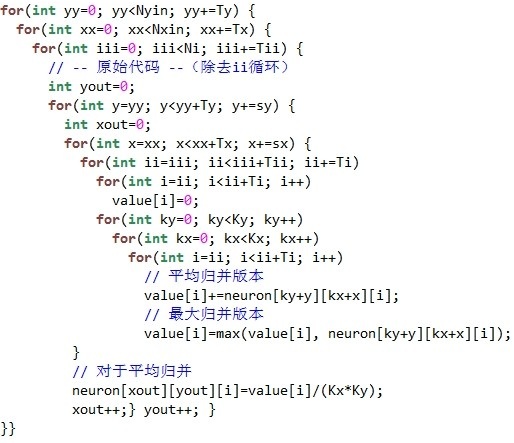
对于输入输出有两个可以重复使用的机会：滑动窗口可以扫描输入层数据造成的滑动窗口间数据复用，以及Nn个输出特征阵列之间的数据复用。前者涉及到的复用次数最多为，后者涉及到的数据复用次数为Nn个。针对面前一种情况，正如源代码当中描述的一样，我们使用了Tx与Ty的划分，但是对于后一种情况，我们没有采取任何措施，其原因在于一般Kx与Ky大小一般在20以下，而Ni的大小在数十到数百之间，因此Kx\*Ky\*Ni的核阵列权值数据都是能够在L1 cache当中放置的。如果这三个参数的值比较大的时候，我们也有补救办法，可以对输入的特征阵列进行划分，正如分类层当中引入的对ii的划分iii一样。

对于使用共享核的卷积层来说，所有的xout，yout输出特征阵列都会使用同样的核阵列权值参数，这本身占用的内存带宽就非常的低了，正如图【】当中CONV3列示意。但是鉴于整个共享核的大小达到了Kx\*Ky\*Ni\*No，这超出了L1 cache的容量，因此我们仍然需要对No进行划分（划分规模为Tnn），使得共享核的大小减小到Kx\*Ky\*Ni\*Tnn，这样就可以进一步减少卷积层整体的内存带宽需求。

对于使用私有核的卷积层来说，所有的突触权值都是独有的，不存在复用的可能性，其内存带宽需求如图【】中的CONV5所示，与分类层的行为非常的相似，然后对于分类层来说，如果L2 cache足够大的话，在层次之间仍然存在着复用的可能性。而对于卷积层，虽然在卷积层当中，有（sx，sy）这一步骤的存在以及稀疏的输入数据与输出数据之间的关联，从网络结构上大大减少了私有核潜在的突触权值的数量，但是必须的突触权值数量依然有上亿字节的规模，远远超过了L2 cache的容量，造成的超高的内存带宽也是无法避免的。

但是值得注意的是，在学术圈依然有大量关于私有核与共享核的辩论[[11](#_ENREF_11), [12](#_ENREF_12)]，在机器学习领域关于私有核取代共享核的重要性也不是非常明确。就加速器领域内，选用私有核还是选用共享核在统一架构下有着截然不同的表现，这反而给讨论中选用支持私有核一方增添了一份砝码。

归并层



《归并层源代码截图》



《归并层划分》

与卷积层不同，归并层的输入特征阵列的数量与输出层的特征阵列的数量是相同的，更重要的是，卷积层中没有私有核与共享核的概念，也就不需要存储这额外的突触权值。每一个输出特征阵列的数据完全是由Kx\*Ky个输入特征阵列的二维数据窗口来决定的，这就造成了数据的复用性只能在滑动窗口当中才能体现（而不是由输出特征阵列和滑动窗口共同决定的）。通过上面的分析也能看到，可以利用的数据复用较少，相比于卷积层，归并层的输入神经元要求的内存带宽要相对较高，而对Tx，Ty进行划分操作带来的提升效果也相对较小。

我们选择16位的定点算数指令替代了32位的浮点算数指令来作为整体的评价标准，这也是由于最后要比较的神经网络加速器的选择决定的。虽然看起来很不可思议，但是很多文献[[13-15](#_ENREF_13)]当中都有充分的证据表明即便是位数很小的操作指令（8位甚至更少）对于神经网络的准确度也仅有很小的影响。为了进一步验证和揭示这一事实，我们训练并且测试了加州大学欧文分校的机器学习算法集合，其结果如图所示。另外我们还训练了标准的MNIST机器学习测试程序[[16](#_ENREF_16)]（手写数字识别程序），在同时比较了16位和32位操作指令后的结果如表格所示（在测试当中使用了交叉验证方法）。对于定点运算符来说，我们定义其前六位为整数部分，后十位为浮点部分。以上实验的结果都说明了我们采用16位运算符带来的影响微乎其微。

|  |  |
| --- | --- |
| 类型 | 错误率 |
| 32位浮点操作符 | 0.0311 |
| 16位定点操作符 | 0.0337 |

《在MNIST当中使用32位浮点操作符与16位定点操作符准确性比较》

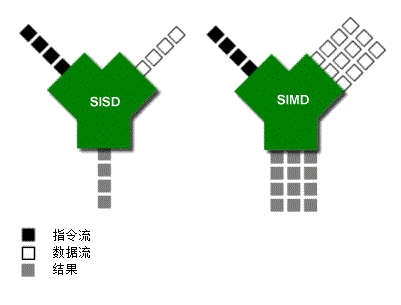


《UCI数据集合上的32位浮点操作符与16位定点操作符准确性比较》

Simd介绍

单指令流多数据流（Single Instruction Multi Data）是针对计算机系统结构进行分类的费林分类法（Flynn’s taxonomy）的一个种类，simd描述的计算机体系结构拥有多个处理部件，可以同时对多个数据进行同一种操作，这样的结构能够实现数据流并行：有并行的数据处理，但是每次只能执行1条指令。一般simd结构或者处理过程对于数字图像或者是数字音频非常常用，这也是我们首先选用simd作为神经网络层次实现的重要原因。

早期的微处理器并没有浮点计算能力，它们只能算是整形处理工具，而浮点计算是在单独定制的硬件上完成的，这样的硬件被被称为协处理器。后来随着晶体管的不断发展，浮点处理单元是能够被放置在CPU片上的，现代所见到的整形/浮点处理器才真正的诞生。当然，引入了额外的浮点硬件就代表着引入了额外的浮点处理指令，例如，当时的x86体系结构就引入了x87浮点架构以及其基于栈的寄存器模型。和x87协处理器为x86架构引入了全新的名字、全新的计算能力、全新的指令集类似，SIMD硬件和指令集就是在原来的SISD（Single Instruction Single Data，单指令流单数据流）基础上引入的额外的功能和内容。

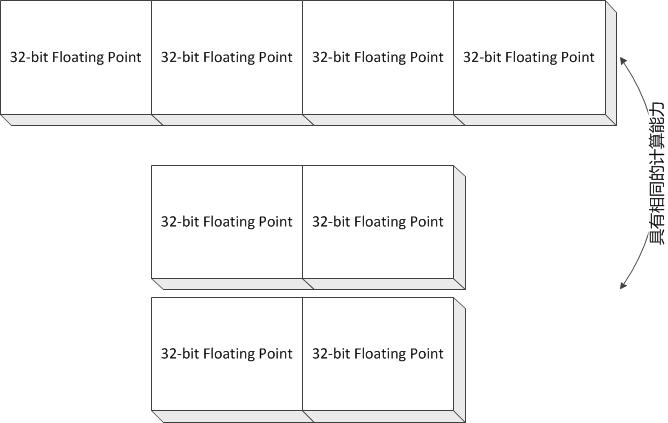


《SISD vs SIMD》

如上图所示，传统的SISD架构将一条操作执行到一条数据上面，而SISD架构致力于发掘数据流的并行性（Data parallelism）。数据流并行性是指具有统一类型的大规模数据，它们都需要完全相同的指令来进行处理。典型的数据并行性的例子就是在数字图像处理当中，将RGB图像反色：图像当中的每一个像素值都需要被遍历到，并且执行相同的数据操作——多数据单操作。而现代对于高级的SISD架构，人们还会挖掘另一种性质，即指令流并行性（Instruction Level Parallelism），主要目的致力于在相同的数据流上面执行多条指令。

SIMD架构当中最基本的单元是向量（vector），这也就是SIMD计算也被称作向量处理的原因，向量仅仅是一组独立的数字或者标量的集合。传统的CPU是作用于标量的（超标量CPU是指可以同时在多个标量上执行不同操作的架构），而SIMD当中的以向量为基本操作单元就是指对向量当中含有的多个同一类型的标量，同时执行完全相同的数据操作。

这些向量在操作时被称为打包向量格式，数据都是以字节（byte，8-bit）或者字（word，16-bit）的形式存储，而后打包成向量继续操作。在设计SIMD架构时候一个很重要的问题在于同时执行多少个数据元素最为合适。如果想要执行32-bit的单精度浮点（single precision floating point）操作的话，那么在此SIMD架构之下可以同时对4个数据进行4路操作，也可以用2个64-bit的向量进行2路的单精度浮点操作。因此每个向量的大小就决定了使用者可以同时处理特定类型数据的数量。



《SIMD当中向量大小的影响》

对于我们本次要使用到的SIMD指令集SSE（Streaming SIMD Extensions），是intel在x86架构下设计的SIMD扩展指令集，它是Intel在1999年为了回应AMD发布的3DNow!在MMX（Multi Media eXtensions）指令集的基础上做的进一步扩展。

Intel最开始的IA-32基础上的SIMD指令集是MMX指令集，但是其存在两个问题：它复用了现有的浮点数寄存器而不是单独使用自己的SIMD寄存器，这就导致了它不能同时进行浮点指令操作与SIMD指令操作；MMX指令集也只能作用到整形数据上。SSE指令集引入了全新的寄存器组（XMM寄存器组），并且可以使用更多的整形指令与浮点指令，更易于应用到数字信号处理与图像处理领域。并且SSE指令集还有后续的扩展指令集，我们本次使用到的指令基本到SSE2就足够使用了。

SSE编程环境

如果要使用SSE指令集，首先需要接触到的就是SSE指令集当中的寄存器组。图展示了SSE2扩展指令集的编程环境，SSE2扩展指令集并没有引入新的寄存器，而是使用了原来MMX引入的MMX寄存器组，SSE引入的XMM寄存器组，以及传统的IA-32通用寄存器组：

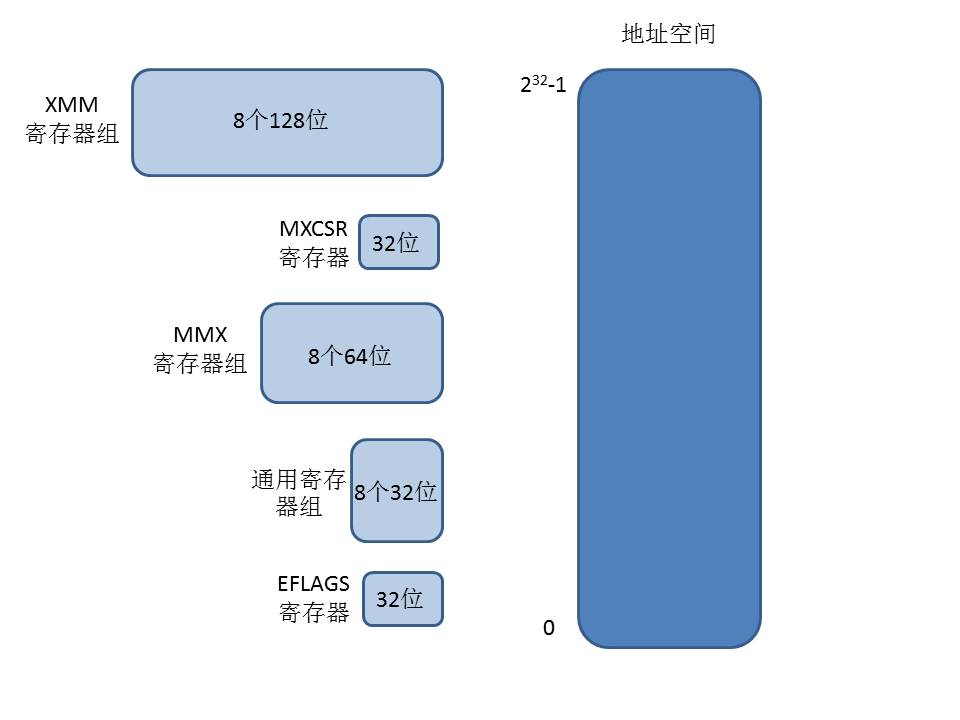
XMM寄存器组：总共有8个，均为128位，可以处理组合的或者标量的双精度浮点操作，其中标量双精度浮点操作是指可以处理存放在XMM寄存器当中低64位的双精度浮点数。XMM寄存器还可以处理128位的组合整形数据，我们在本次实验当中主要使用的就是这一类型的操作。这8个寄存器的命名方式是XMM0到XMM7

MXCSR寄存器：仅有1个，32位寄存器，可以显示在浮点操作当中的状态为和控制位。例如其中的denormals-to-zeros位与flush-to-zero位可以为处理类似溢出的非常规操作提供更高效的处理方法。

MMX寄存器组：共有8个，均为64位，可以用来处理组合整形数据操作。也可以为了数据在MMX寄存器和XMM寄存器之间进行数据转移提供过渡，其命名方式为MM0到MM7。

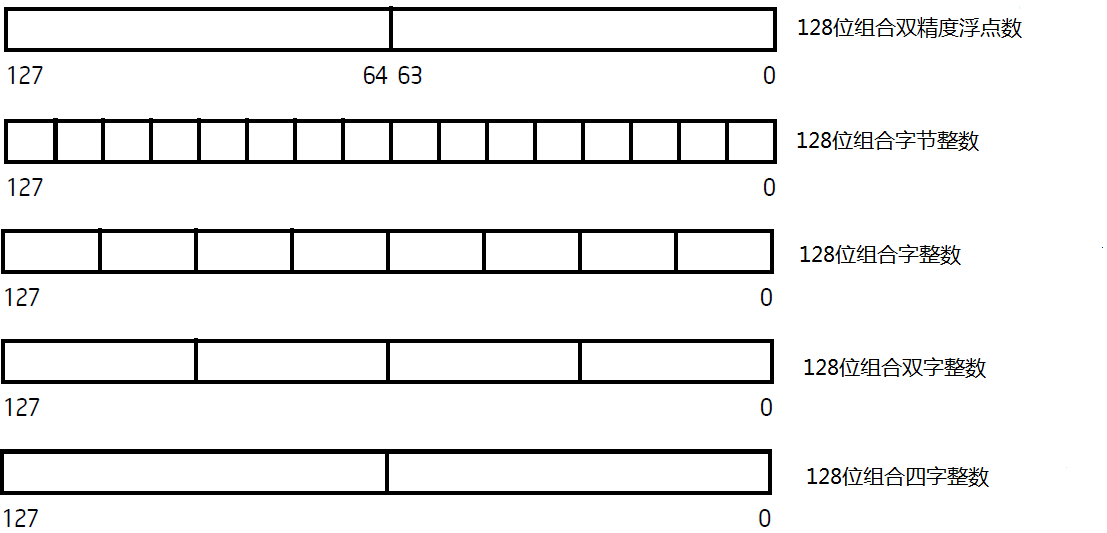
通用寄存器组：这就是最常见的通用IA-32寄存器，适用于IA-32架构的地址模型及寻址操作。可以被用来存储一些SSE指令的操作数，根据功能不同分别命名为EAX，EBX，ECX，EDX，EBP，ESI，EDI，ESP。

EFLAGS寄存器：32位寄存器，用来存储一些比较操作的结果。



《SSE编程环境》

SSE当中的数据类型如图所示，主要是考虑到待处理数据类型的位数以及寄存器的位数综合决定最后采用的数据类型。在本次实验当中我们所要处理的是16位定点型数据，所要采用的是128位组合双字整数的数据结构。这种数据结构在一个寄存器当中可以容纳8个16位定点数据。



SSE编程数据结构

一般SSE指令集当中的指令都可以按照功能分为以下几组：

数据搬移指令

算术指令

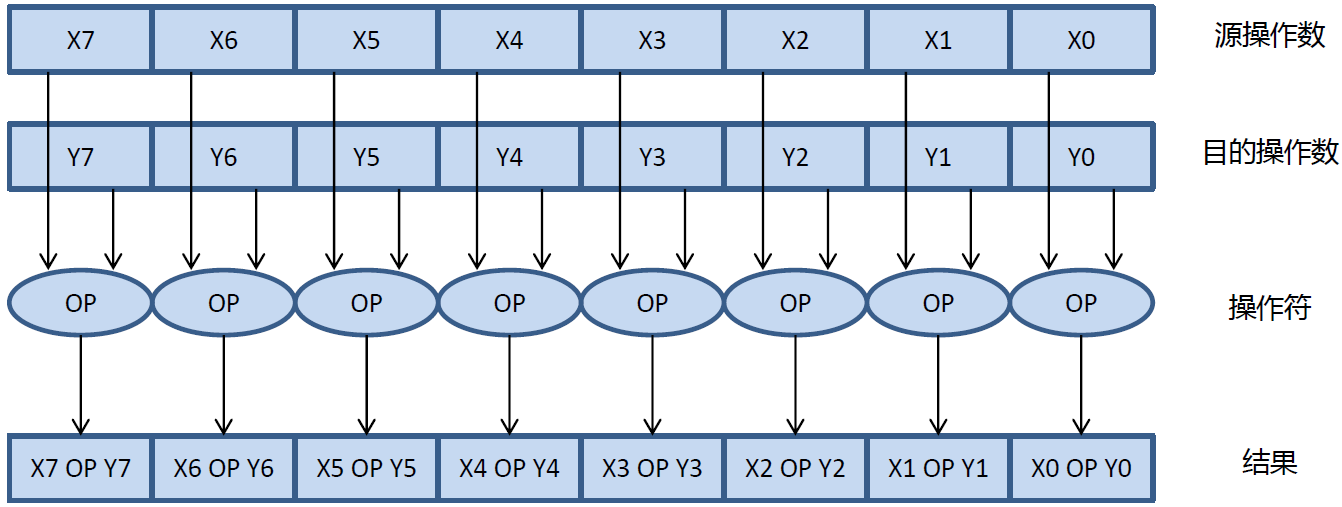
比较指令

转换指令

逻辑指令

随机指令

典型的16位定点数的SIMD指令形式如图所示：



对于组合操作来说，源操作数（即源操作寄存器）都包含着8个16位定点数据，而目的操作数（即目的操作寄存器）也包含着8个16位定点数据，在每次操作的过程当中16个并行的操作可以同时进行，并且写入到目的操作寄存器的相应位置。

而标量操作则是和传统的x86浮点操作相似，目的寄存器的高位不发生变化，而低位数据则和源寄存器进行算数或者逻辑等操作，写入到目的寄存器的低位。我们这里主要使用的是组合操作指令。

以MLP层为例说明SIMD的执行过程：

//SIMD执行MLP算法

初始化xmm0作为临时加和存储寄存器xmm0[0-7]=0

循环开始

Step1. 将输入神经元权值载入到xmm1寄存器xmm1[0-7]

Step2. 将突触神经元权值载入到xmm2寄存器xmm2[0-7]

Step3. 做SIMD16位组合定点数乘法xmm2[i]=xmm2[i]\*xmm1[i]

Step4. 做垂直加法xmm0[0]=xmm0[0]+xmm0[1] xmm0[1]=xmm0[2]+xmm0[3] xmm0[2]=xmm0[4]+xmm0[5] xmm0[3]=xmm0[6]+xmm0[7] xmm0[4]=xmm1[0]+xmm1[1] xmm0[5]=xmm1[2]+xmm1[3] xmm0[6]=xmm1[4]+xmm1[5] xmm0[7]=xmm1[6]+xmm1[7]

循环结束

对xmm0寄存器做3次垂直加法，xmm0[0]即为所求的加权和

将xmm0寄存器数据载入到内存

对加权和进行stepwise sigmoid操作

//算法结束

由于要想对数据进行操作，首先要将数据从内存当中搬运到寄存器当中，因此在程序当中首先要使用的就是数据搬移指令。我们这里需要的是将8个非对齐16位定点数从内存当中搬移到寄存器当中，因此需要对需要处理的输入数据数组以及突触上的权值数组进行对齐及组合操作，保证数据搬移指令的正确处理。

然后需要对寄存器当中的数据进行算数指令操作。值得注意的是上面step1-step4每一个step都是一个操作即可完成，这也是SIMD相比于传统x86架构的最大优势，并行性得到了充分的保证。SSE指令集当中不仅有上面提到的标准SIMD指令（如PADDSW，PMULLW等），也提供了类似PHADDSW垂直加法指令这样的特殊SIMD指令来供调用，充分实现了并行化（如step4当中展示）。

在实现以及调优的时候还需要考虑到SSE指令集当中每一条被调用指令的延迟（latency）和吞吐率（throughput），延迟是指该条指令的结果在多少个周期（cycle）之后才能被使用，吞吐率是指在几个cycle之后才能再次发射同一条该种指令。过长的延迟和过高的吞吐率都会给系统带来性能瓶颈，在调优时候利用循环展开调整指令顺序都方式都可以提升性能。

另外在系统当中由于需要使用sigmoid函数，而sigmoid函数当中大多涉及到了指数操作，这在SSE指令集当中没有相应的支持，并且在传统x86的指数指令的执行速度要远低于乘加操作的处理速度，因此我们决定对sigmoid函数进行分段线性化。

//sigmoid算法

Func sigmoid(t)

x=int(t)

查表得到k与b

执行y=k\*x+b

返回y

//sigmoid算法结束

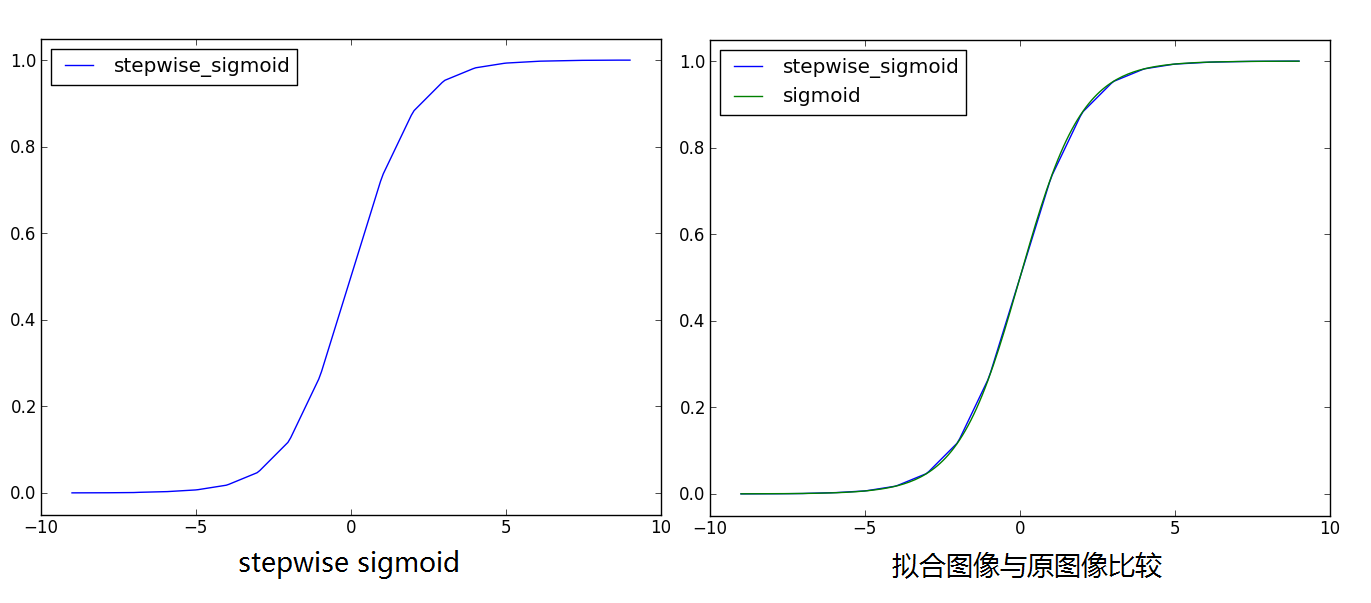
对于

其中的t首先要进行近似为整数的操作x=int(t)，对于x要进行查表计算K(t)，表格如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | k | b | x | k | b |
| 小于等于-8 | 0 | 0 | 1 | 0.1498 | 0.5813 |
| -7 | 0.0006 | 0.0050 | 2 | 0.0718 | 0.7373 |
| -6 | 0.0016 | 0.0118 | 3 | 0.0299 | 0.8643 |
| -5 | 0.0042 | 0.0278 | 4 | 0.0113 | 0.9369 |
| -4 | 0.0113 | 0.0631 | 5 | 0.0042 | 0.9722 |
| -3 | 0.0299 | 0.1357 | 6 | 0.0016 | 0.9882 |
| -2 | 0.0718 | 0.2627 | 7 | 0.0006 | 0.9950 |
| -1 | 0.1498 | 0.4187 | 大于等于8 | 0 | 1 |
| 0 | 0.2311 | 0.5 |  |  |  |

《Stepwise sigmoid线性拟合系数表》

表中系数是按照最小二乘法结合原始sigmoid函数的各点分布拟合出来的结果。按照上面所示的线性步进sigmoid函数做出图像并且与原始的sigmoid函数的图像做出比较，得到下面的图像。



从图中可以看到，线性拟合的sigmoid函数与原始sigmoid函数的行为非常相似，完全可以取代原来的sigmoid函数在神经网络当中被使用。

因此在上面mlp层算法的最后，在每一层的加权和从SSE寄存器当中迁移到内存之后，就可以按照上面的步进sigmoid的方式线性计算出该层输出。

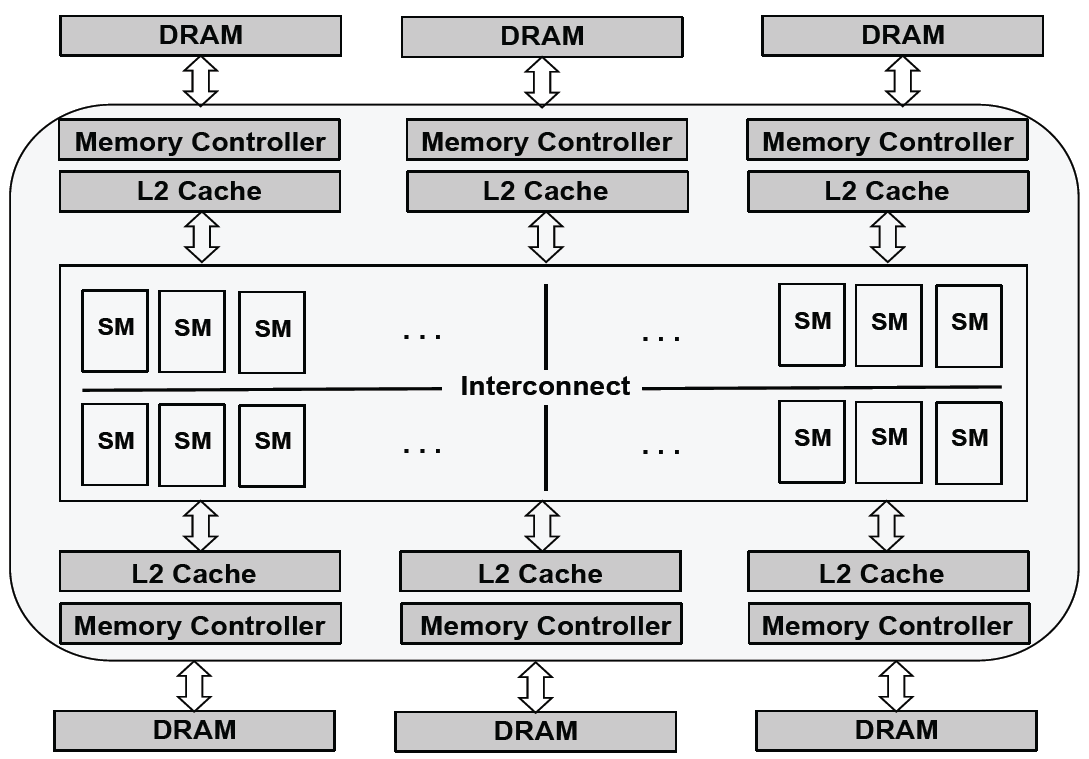
其它神经网络层的实现方式都与MLP层类似，关键的问题主要在于发掘各自网络层的计算的并行性，并结合SSE指令集提供的指令选择合适的指令，充分利用数据并行性。上面提到的步进sigmoid函数也能够为整体实现提升速度。

今年来，很多实现卷积神经网络和深度神经网络算法的方式都是在GPU上实现的[[17](#_ENREF_17)]，这也是由于这些算法本身的特征决定的。因此我们在选择神经网络加速器实现方式的时候，GPU也是不能错过的一个环节。

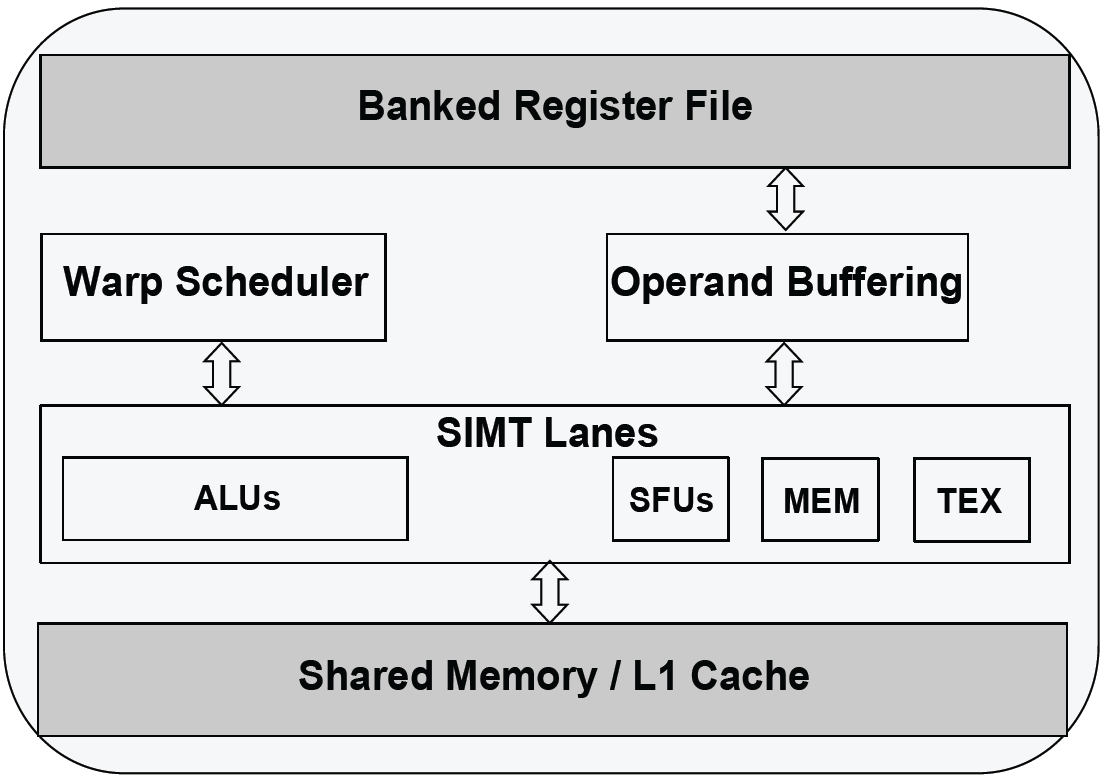
一般我们使用到的是GPGPU，其全称是通用图形处理单元（General Purpose Graphic Processing Unit），主要适用于计算机图形的处理工作，替代了原来传统CPU的部分工作。所有的GPU都提供了一套完整的操作集合来处理所有的数值及逻辑等操作。最为重要的是，GPU在现代计算机当中对于图形操作等并行操作能够比传统CPU更好地发掘其并行性，提高程序执行效率。

本次采用的主要是GPU的fermi架构，而GPU在最近的发展历程上也经历了G80架构到GT200架构到fermi架构到kepler架构的转变。但是现代GPU的整体架构的框架确一直没有发生大的变化，下面就可以以fermi架构为例简单介绍一下本次使用到的实验环境当中的GPU架构。

现代的通用GPU架构是由很多固定的图形函数流水线构成的，这些流水线要么包含着处理几何图形的硬件要么包含着处理像素的渲染器，几何处理器可以处理一些涉及到原点、直线、三角形等的原语操作，而像素渲染器则能够处理一些用于填充图形内部光影效果的光栅插值。但是在Tesla架构推出之后推动的统一处理（Unified Processing）模型下面，现在的GPU架构更着重关注与更抽象的设计元素与更高效的资源利用率。



图中展示了现代GPU架构的框架，GPU是由SM（Streaming Multiprocessor），DRAM控制器以及片上的二级cache构成的。根据GPU型号的不同，每个GPU当中的SM的数量，以及每个SM中流处理器（Streaming Processor，SP）的数量是各不相同的。一般每个SM当中会有32个单指令流多线程流（Single Instruction Multiple Thread，SIMT）处理通道（SIMT lane），每个SIMT通道都可以在一个cycle下面在一个线程当中发射一条指令，这样每个SM在每个cycle当中就可以同时发射32条指令。线程也会被每32个被划分为Warp，在GPU程序执行的时候，程序是按照warp为最小粒度进行调度执行的，每一个warp当中所有的32个线程是共用一个程序计数器的（Program Counter，PC）。每一个SIMT通道都各自有自己的寄存器文件，其读写速度是非常快的，并且不同SIMT通道能够同时访问一个低延迟的片上共享存储，成为shared cache。对于寄存器文件的合理安排能够为SM提供足够的带宽，来实现每一个操作符在每一个cycle当中都有两个输入和一个输出。



关于GPU的存储模型也是需要关注的一个重点，整体来说在GPU当中每个SM都会有相应的一级缓存，而GPU上的所有操作则是共用统一的二级缓存。分开来说：

寄存器：每一个SM都有32K个寄存器，在程序执行时候，每一个线程就会被分配到一定数量的寄存器，这些寄存器是不会被其它线程所使用的。一般在CUDA的核函数当中所允许使用的寄存器数量是63个，如果程序需要划分为相当多数量的线程来执行的话，那么每一个线程的核函数当中允许使用的寄存器数量最多则将为了21个。寄存器文件都有非常高的读写带宽，一般约为8000GB/s。

一级缓存（L1 cache）与共享缓存（shared cache）：这两种缓存要放在一起是因为这种存储本身都是片上存储，但是其用途可以作为一级缓存来存放每个线程各自独立的数据也可以作为共享缓存来存放多个线程需要共享的数据。一般片上存储有64KB大小，可以被配置为48KB的一级缓存加上16KB的共享缓存或者配置为16KB的一级缓存加上48KB的共享缓存。共享缓存是我们本次实现神经网络当中需要重点关注的对象，因为它可以在被同一个程序块（block）当中的所有线程来共享数据，非常方便每个中间结点都要共享输入数据或者在私有核情况下共享突触权值的情况。这种共享数据的方式可以让在一个线程块当中的线程协调合作，重用片上数据，能够大量地减少核与存储之间的通信开销。一级缓存与共享存储有着较短的延迟，一般为10到20个cycle，较高的带宽，一般为1600GB/s。对于神经网络这样需要大量空间来存放中间数据的程序来说，共享存储的存在可以大大减少中间数据在存储与计算单元之间传输的开销。

设备存储（Device Memory）：本地存储是用来存放没有被存放在寄存器以及共享缓存当中的数据的。有时候线程可能需要存放大量的变量，而上面提到的寄存器文件是不能满足需求的，这就是本地存储发挥功用的时候。一般设备存储还支持存放以下类型的数据，常量大小的数组类型数据，超过寄存器文件支持上线的数组类型数据，从本次存储当中读取的需要应用到各个线程块上面的规模较大的数据等。有较大的读写延迟，一般为400到800个cycle。

二级缓存：二级缓存是多个SM共用的，可以作为从全局存储当中进行读写的中间过渡存储，二级存储也支持原子操作，但是这些原子操作所涉及的数据必须能够在不同线程块甚至核函数之间进行共享。

宿主存储（Host Memory）：CPU存放数据的地方，一般在程序开始时候会在CPU上开辟空间存放数据，然后搬移到GPU上的设备存储作为输入数据，最后也会开辟空间来接受从GPU迁出的输出数据。

CUDA核（CUDA cores）一般指得是GPU上最核心的数据处理的模块，一般分为：

整数算术逻辑单元（Integer Arithmetic Logic Unit）：支持32位精度的所有符合传统编程需求的定点操作，并且也对64位以及更多扩展位的算术逻辑操作的精度进行了优化。这也就是说对于传统编程环境当中所有可用的指令与操作在GPU上都是可以实现的。

浮点处理单元（Floating Point Unit）：实现了最新的IEEE 754-2008标准规定的所有浮点标准操作，并且提供单精度版本和双精度版本的聚合乘加（Fused Multiply-add）指令。在每一个cycle当中，每个sm当中最多可以处理16个符合乘加操作。

Warp调度功能也是Fermi架构实现的一个功能，这是通过两层的分布式线程调度器完成的。每一个sm都有两个发射指令单元，可以让两个warp同时相互独立的并行发射，可以让两个warp的一条指令分别在16个一组的CUDA core上进行计算，或者是16个访存操作，或者在4个SFU上执行。这样的调度机制可以让GPU在硬件的计算能力达到极致。

GPU本身的设计是非常适用于图形处理的，因此其在操作上和可编程性上都有着很多的限制要求，根据GPU的架构设计，对于那些可以流处理的问题GPU能够较为高效的解决方案，但是其硬件的使用也必须要遵循规则。

流处理是在并行化处理当中发掘的一种可以突破传统限制的处理方法，能够直接使用多个计算单元来实现计算并行化，加速处理过程，而不需要非常复杂的分配存储、管理同步、通讯交流等额外的开销。正如我们在传统SIMD概念当中描述的那样，我们需要对一组数据流中的每一个数据都执行相同的操作。最典型的就是对数据流当中数据的操作都组合到一个核函数（kernel function）当中。一般核函数都是可以被流水线化的，这样就可以重复使用片上存储而避免了额外的内存带宽，这也符合了GPU设计的理念。

在GPU当中可以处理独立的向量，也能够高效地并行化地处理大量的向量。而面对数据流处理这样的问题，GPU就作为流处理器就可以充分发挥其功效——并行化地将同一个核函数作用到数据流当中的每一个记录当中，当然其前提是数据流本身已经提供了数据并行性，这在神经网络各层次的输入数据当中都能得到充分的保证。并且神经网络同一层次的数据之间（例如输入层数据之间，神经突触层权值之间）各自独立，我们可以同时对各个输入数据进行处理，处理完成之后写入到各自的存储当中，而不需要考虑到是否有存储需要同时被读写。并且由上面的分析也能够看到，各个神经网络层次在计算的时候的计算密度都是并行化的前提，对于GPU来说，如果执行的计算过程有着较高的计算密度，那么对于应用的加速将会非常有利，否则就会成为计算性能提升的瓶颈。

从上面的GPU的架构以及处理模型当中可以看到，虽然GPU也是并行执行但是更倾向于将同一个指令作用到多个线程上面，然后由每一个线程对该线程上面的数据进行操作，而传统的SIMD模型，包括Intel的MMX以及SSE指令集都是将单个指令直接作用到多个数据上面，虽然SIMT的模型是从SIMD更进一步修改得到，但是两者在并行性上面还是有着相当大的差别：

CPU并行化特点：

任务并行化

多个任务被映射到多个线程当中

不同的任务执行不同的指令

最多支持在几十个核心上支持几十个重量级线程并行执行

每一个线程都需要人为地精确管理与调度

每一个线程都需要分别编程其行为

GPU并行化特点：

SIMD模型

不同数据上执行相同指令

数百个核心上并行执行上万个轻量级线程

线程通过硬件来管理调度

编程框架已经实现了对线程的批量操作

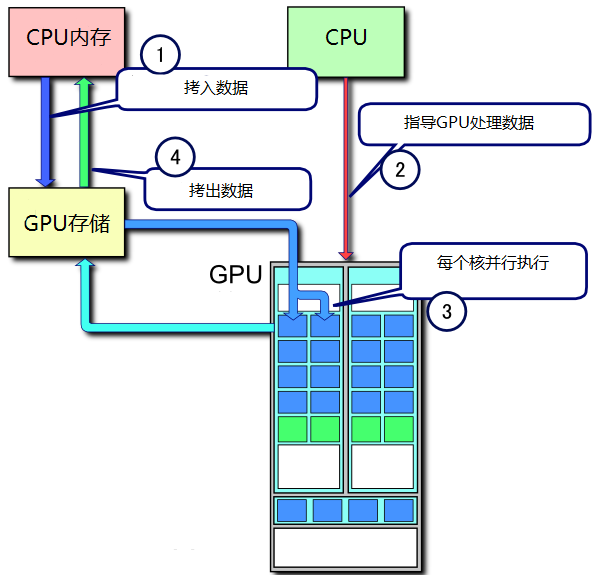
CPU并行与GPU并行对照表

GPU编程模型

CUDA（Computing Unified Device Architecture）是NVIDIA制定的并行编程平台和编程模型，作用在NVIDIA生产的GPU上。CUDA使得程序员能够使用到GPU当中的指令集，并且对片上以及片下的存储空间进行直接的操作。在CUDA的帮助下，传统的GPU除了能够在图形处理方面发挥作用之外，在通用的并行问题处理当中也能有相当不错的表现，这也就是GPGPU生产的直接原因。与CPU不同，GPU本身的设计架构更强调同时执行很多个线程，而CPU的设计架构则更强调能够更快的执行一个线程。

程序员可以使用CUDA提供的加速库、第三方提供的预编译指令以及传统的编程语言来在CUDA平台上发挥能力解决问题。根据GPU架构的不断改进，CUDA的版本也随之不断的更新，从最初2007年的CUDA SDK到现在的CUDA 6.0，虽然CUDA的API函数以及实现方式可能发生了较多的修改，但是NVIDIA公司明确表示CUDA是能够前向兼容的，也就是当初作用到G8x系列GPU上的程序在新版本的CUDA SDK当中依然适用，而不需要对程序做任何修改。

一般CUDA程序的执行流程如图所示



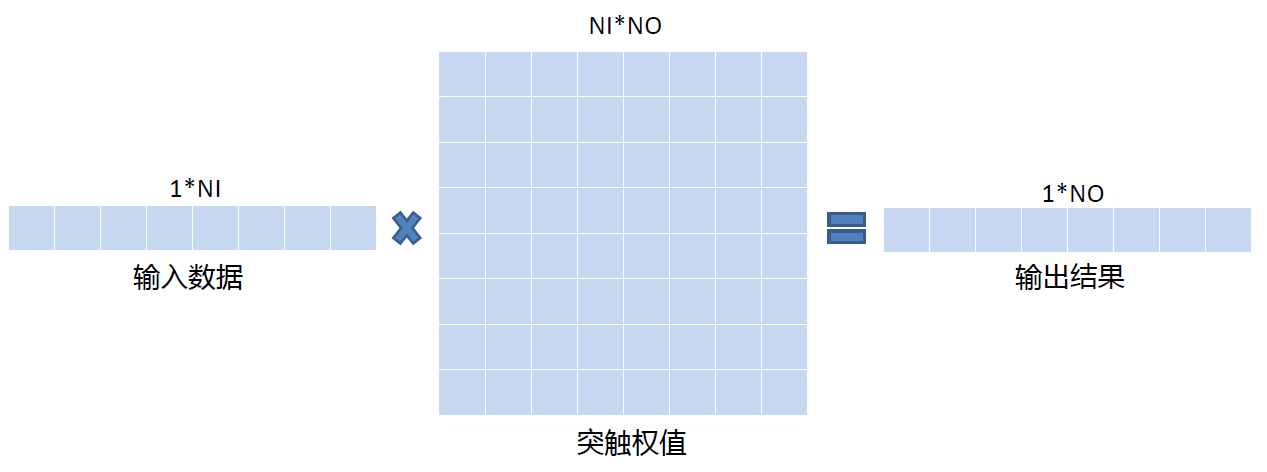
如图中所示，一般CUDA程序的执行步骤分为一下四步：

1. 拷入数据：将GPU需要处理的输入数据从CPU的内存当中（Host Memory）拷贝进入GPU的存储当中（Device Memory）。但是在这一过程当中，可能会因为系统总线的带宽上限和延迟带来一定的性能瓶颈。GPU本身也考虑了这个问题，一般采用GPU本身的DMA控制器来保证数据的异步传输，充分利用带宽，尽量掩盖传输延迟造成计算核心空等的影响。
2. CPU指导GPU处理数据：CPU决定对GPU的硬件采用怎样的并行化分组进行处理，如对大规模问题划分为多个线程块进行执行，而每个线程块当中又有若干个线程分别执行较小规模的同质化的问题。在CUDA API将数据从CPU传输到GPU之后，CPU就可以为GPU预先划分问题规模，在GPU执行阶段就可以按照如此的规模划分进行处理。
3. GPU各核之间并行执行：在GPU进行执行的时候，各个核心就按照预先写好的CUDA核函数进行处理和执行，各个线程按照warp的发射顺序被分配到流处理器核心上，执行指令和操作。一个优秀的kernel函数可以指导硬件行为，大大提升资源使用率，这也是通用GPU的一个劣势，对于没有经验且不了解GPU本身设计架构的CUDA程序员来说，面对复杂问题写出高效的解决方案基本是不可能的。
4. 拷出结果。在kernel函数执行完毕之后，在GPU存储上的数据就是我们需要的结果，我们可以将GPU当中的数据拷贝到CPU的存储当中。当然这一过程也会收到带宽的限制和延迟的影响。处理方法与步骤1当中的情况是类似的。

对于本次将要处理的神经网络层的GPU实现，主要困难在于各层次数据的划分与核函数的书写。

另外在传统的神经网络GPU实现当中，多数处理的是32位浮点程序，这在GPU当中有着大量的库函数可以使用，但本次我们由于面临的是16位定点数据的处理，就需要从最底层的数据划分与性能的优化来重新考虑每一个神经网络层的实现。这并不是毫无意义的重复性劳动，也不是闭门造车的行为。我们这样做一方面是一开始就选择了以16位定点数作为统一的处理标准，另一方面来说我们之前按照硬件思路对神经网络各层模型进行的重新划分与分析在传统的GPU实现当中也是找不到实现的，对于全新的模型和全新的数据标准，从头进行设计和实现也是非常自然的。

对于MLP程序，之前的分析可以看到，前一部分的部分和的计算与向量矩阵乘法非常的相似，后面的sigmoid函数并不是并行化的重点和难点，之前SIMD当中实现sigmoid的方式基本可以照搬到GPU的实现当中，这里我们主要考虑的是MLP的矩阵向量乘的实现。原本的实现如图所示，输入数据是一个1\*NI的向量，突触权值数据则相当于一个NI\*NO的矩阵，最后的输出则是之前向量与矩阵相乘得到的一个1\*NO的向量。



单纯的以每一个thread来计算每一个输出的结果是最自然的想法，但是这样的话无法充分利用GPU上大量的SM，导致SM的闲置，也导致了并行化程度不够高，最后的程序的效率自然也还是有充分的提升的空间。

图展示的就是改进之后的MLP层的GPU实现方式。如图中展示，执行过程被分为了两步，第一步是求部分和操作，第二部是对部分和进行累加的操作。

在第一部分当中，CPU在为GPU设置blocks维度时候，采用了二维的维度即（NO/BLOCKSIZE，NI/ROWNUM），其中BLOCKSIZE和ROWNUM是预先设定的变量值。而threads的维度则放弃了最自然的以每个输出节点计算若干个输入节点与相应突触权值乘积的划分方式，而是直接采用了对输出节点数量NO进行划分的BLOCKSIZE，这样做的目的也是为了配合前面的划分做出的选择。在每一个核函数当中，每一个线程需要对ROWNUM个输入数据进行乘法和加法操作，对于每一个输出节点来说，就相当于原来有NI次乘法操作之后得到NI个乘积，然后对这NI个乘积做加法操作，现在则只每ROWNUM个数据做乘法操作，然后对每ROWNUM个乘积做加法操作，得到NI/ROWNUM个临时和，并且将这些临时和存放到GPU存储上面。这样做的目的在于，原本每一个输出节点都要进行NI次乘法及NI次加法操作，这样的并行化程度只能达到原来的NO倍，而在现在的情况下由于临时和的存在，并行化程度能够达到原来的NO\*NI/ROWNUM倍，并且ROWNUM也可以根据具体的NO和NI做定制，一来是能够充分使用到硬件资源，二来则是根据程序的表现选择最优的参数。得到的中间和需要存放在一个NI/ROWNUM\*NO的数组当中，这一点在图中也得到了体现。



第二步则是对这些中间和做累加，划分的维度则是和第一步当中划分的维度以及得到的结果相呼应的，其blocks维度为NO/BLOCKSIZE，其threads维度为BLOCKSIZE。在核函数执行之前不需要对数据进行迁移，因为在第一步当中已经有了同步操作，所有的threads的计算任务都已经完成，部分和都已经存放在了GPU的存储当中。核函数所要做的就是每个thread都要对NI/ROWNUM个临时和进行加和操作，并不会涉及到任何存储上的传输操作。

一个比较遗憾的地方就是部分和如果可以存放到shared memory当中的话会对程序的性能还有进一步的提升，但是由于临时和的数量虽然相比元原始突触权值的数量有个大规模的减少，但是对于大规模的数据将临时和存放在shared memory当中还是不现实的事情，这里所做的妥协也是不可避免的。

[1] S. Christos and S. Dimitros, "What is a neural network," *retrieved from http//: docs. toc. com/doc/1505/neural-networks,* 2008.

[2] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics,* vol. 5, pp. 115-133, 1943.

[3] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation,* vol. 18, pp. 1527-1554, 2006.

[4] D. Cireşan, U. Meier, J. Masci, and J. Schmidhuber, "Multi-column deep neural network for traffic sign classification," *Neural Networks,* vol. 32, pp. 333-338, 2012.

[5] D. Ciresan, A. Giusti, and J. Schmidhuber, "Deep neural networks segment neuronal membranes in electron microscopy images," in *Advances in Neural Information Processing Systems 25*, 2012, pp. 2852-2860.

[6] R. T. McDonald and F. C. Pereira, "Online Learning of Approximate Dependency Parsing Algorithms," in *EACL*, 2006.

[7] Y. Zhang and S. Clark, "A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2008, pp. 562-571.

[8] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *The Journal of physiology,* vol. 195, pp. 215-243, 1968.

[9] T. Serre, L. Wolf, S. Bileschi, M. Riesenhuber, and T. Poggio, "Robust object recognition with cortex-like mechanisms," *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* vol. 29, pp. 411-426, 2007.

[10] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, 2011, pp. 109-116.

[11] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, 2013, pp. 8595-8598.

[12] P. Sermanet, S. Chintala, and Y. LeCun, "Convolutional neural networks applied to house numbers digit classification," in *Pattern Recognition (ICPR), 2012 21st International Conference on*, 2012, pp. 3288-3291.

[13] S. Draghici, "On the capabilities of neural networks using limited precision weights," *Neural networks,* vol. 15, pp. 395-414, 2002.

[14] J. Holi and J.-N. Hwang, "Finite precision error analysis of neural network hardware implementations," *Computers, IEEE Transactions on,* vol. 42, pp. 281-290, 1993.

[15] D. Larkin, A. Kinane, and N. O’Connor, "Towards hardware acceleration of neuroevolution for multimedia processing applications on mobile devices," in *Neural Information Processing*, 2006, pp. 1178-1188.

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE,* vol. 86, pp. 2278-2324, 1998.

[17] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, 2012, pp. 3642-3649.