并行计算课程期末报告

李冯福　201418000206054

2015 年 12 月 12 日

Contents

1 课题选择

1.1 科研前沿

在我目前的研究领域—机器学习，当下最前沿、最热门的方向当属深度学习和人工智能了。最近几年，基于深度卷积神经网络的机器学习模型取得了突破性的进展，在很多应用领域的任务上如 ImageNet 物体识别、语音识别等都取得了当下最好的性能。现实中基于深度学习的产品也越来愈多，如出门问问语音助手、百度识图、科大讯飞语音输入法、Face++ 人脸识别等产品都在深度学习算法的帮助下大幅提高了性能 [1]。

然而深度学习的一大难点在于它的模型规模往往很大，参数可以达到数百万的量级。另一方面，为了训练这种大规模的网络，需要处理跟模型规模匹配的数以百万计的数据。这两方面导致深度学习往往是很耗时的一个过程。传统的单机多 CPU 训练系统对这种规模的问题已经无能为力。

为了应对这种大规模模型、大数据量的深度学习训练问题，基于 GPU 加速的思想被看成是一种最简单有效的解决方案。最近两年，最火的深度学习开源框架 CAFFE[2] 就是基于 GPU 加速开发的。另外，最近几个月刚发布的 MXNET[3]、TensorFlow[4]（谷歌）等深度学习框架都支持多 GPU 运算。这些基于 GPU 加速的深度学习框架的发布对于深度学习的发展有特别大的促进作用。正是从 2014 年开始，深度学习的一个应用—计算机视觉领域涌现出了一大批初创公司，并且获得了相当不菲的投资。

1.2 目标

鉴于 GPU 加速的巨大威力，这篇报告的目标是用 GPU 加速一种常用的非监督深度学习模型—卷积玻尔兹曼机 (Convolutional Restricted Boltzmann Machine，**CRBM**)[5]。由于 CRBM 是卷积深度信念网络 (Convolutional Deep Belief Network) 的基础单元，因此，也相当于用 GPU 加速卷积深度信念网络模型的训练。

2 预备知识

2.1 机器学习和神经网络

机器学习，通俗的说就是让机器通过学习来改进自身的性能。它有三个基本的要素：

1. **经验**即学习资料。机器的经验就是收集的数据，包括语音、图片、文字等。

2. **性能**即评价标准。机器性能是通过机器对经验的表现与人为规定的指标的匹配度来衡量的。

3. **学习**即通过调节模型来改善性能。它是一个动态的过程。

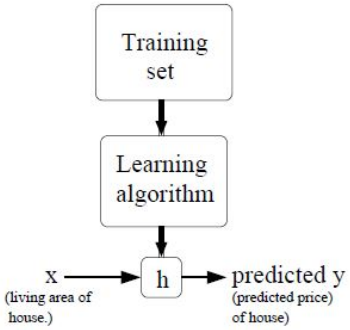


图 1: 有监督机器学习系统

图 1展示了机器学习中的一类最广泛的问题—有监督（机器）学习。它可以用如下的数学公式描述：

$$\hat{\theta} = \arg\min_{\theta} J(X, Y; \theta) = \frac{1}{2} \sum_{i=1}^{m} f(h_\theta(x^{(i)}), y^{(i)}) \tag{1}$$

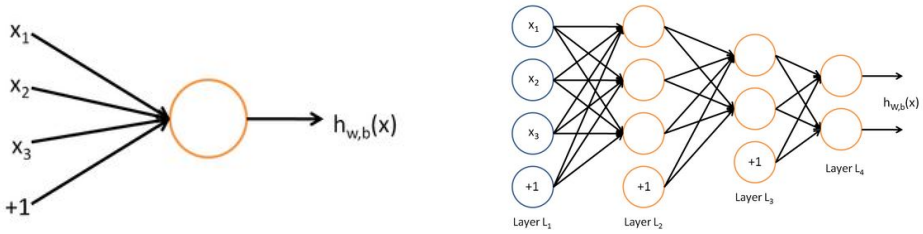
其中数据集 $X = \{x^{(1)}, x^{(2)}, \cdots, x^{(m)}\}$ 和标签集 $Y = \{y^{(1)}, y^{(2)}, \cdots, y^{(m)}\}$ 一起组成训练集 (**经验**)。$J(X, Y; \theta)$ 是**性能评价指标**，也称为代价函数 (cost function)。$h_\theta(x^{(i)})$ 是预测函数，它将给定的输入 $x^{(i)}$ 映射到标签集合 $\hat{y}^{(i)} = h_\theta(x^{(i)})$。$f(\hat{y}, y)$ 是针对单个样本点的损失函数，一般是非负的，用来评价预测值 $\hat{y}$ 与真实值 y 的差别程度。对于二分类问题，f 可以选择为交叉熵，即 $f(\hat{y}, y) = -y \log(\hat{y})$；对于回归问题，$f$ 一般选择欧式距离的平方，即 $f(\hat{y}, y) = (\hat{y} - y)^2$。**学习**的过程就是找出最优的 θ，使得损失函数最小 (即预测值与真实值最接近)。

神经网络是机器学习的一个分支，它通过模拟人脑的神经元结构来建模。图 2(a)所示是一个人工神经元的网络模型，它是一个以 x_1, x_2, x_3 以及截距 $+1$ 为输入的运算单元，其输出为

$$h_{W,b}(x) = g(W^T x) = g(\sum_{i=1}^{3} W_i x_i + b) \tag{2}$$

这里 $\theta = \{W, b\}$；$g(x)$ 是非线性的激活函数，常用的有：

- sigmoid 激活函数, $g(x) = \frac{1}{1+e^{-x}}$
- tanh 激活函数, $g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$



(a) 人工神经元　　(b) 三层前馈神经网络

图 2: 人工神经网络示意图。

- relu 激活函数, $g(x) = x \cdot 1\{x > 0\}$

通过单个神经元的组合, 可以构造出复杂的多层神经网络模型。图 2(b)展示了一个三层的前馈神经网络模型。

2.2 （批）梯度下降算法

机器学习最核心的问题是如何求解优化问题 (1)。当 $f(h, y)$ 关于 h 可导, $h_\theta(x)$ 关于 θ 可导时，最常用的求解方法是梯度下降算法。该方法从给定的初始参数 θ_0 出发，通过负梯度方向来更新 θ 的值，直到目标函数 $J(X, Y; \theta)$ 最终收敛。其中最关键的是求解梯度方向：

$$\frac{\partial J(X, Y; \theta)}{\partial \theta} = \sum_{i=1}^{m} \frac{\partial f(h, y)}{\partial h} \frac{\partial h_\theta(x)}{\partial \theta} \tag{3}$$

然后利用更新公式

$$\theta^{\text{new}} := \theta - \epsilon \cdot \frac{\partial J(X, Y; \theta)}{\partial \theta} \tag{4}$$

来更新参数。这里 ϵ 是下降的步长，一般设为较小的值。

对于大规模数据问题, m 往往很大, 因此, 直接一次遍历所有数据点往往是件很耗时间的工作。一种改进的方法是每次处理一批 (batch) 数据，数量记为 m_b。通过对一批数据求解下降方向来近似总的梯度下降方向，从而极大的降低时间复杂度。算法 1描述了**批梯度下降算法**。一种特例是 $m_b = 1$, 此时称为**随机梯度下降**，因为每次更新参数都是对某个随机样本而言的。

对于神经网络模型，它的梯度下降算法有一个特殊的名字：**误差反传算法 (Error Backpropagation)**。这是由于神经网络模型具有很强的层次性, 它的预测函数 $h_\theta(x)$ 是关于每一层参数的复合函数。为了求解较前参数的导数值，只需要将下一层的导数值通过该层的参数传回来即可。该算法因此而得名。

算法 1 批梯度下降算法求解问题 (1)

输入: m_b 和 ϵ

输出: $\hat{\theta}$

1: 初始化: $t = 0$, $\theta^{(t)} := \theta_0$

2: **while** $J(X, Y; \theta)$ 没有收敛到最小值 **do**

3: 从 (X, Y) 中随机选取 m_b 个样本点组成集合 (X_b, Y_b) ;

4: 用链式法则求 $J(X_b, Y_b; \theta)$ 关于 θ 的导数 :

$$\frac{\partial J(X_b, Y_b; \theta)}{\partial \theta} = \sum_{i=1}^{m_b} \frac{\partial f(h, y)}{\partial h} \frac{\partial h_\theta(x^{(i)})}{\partial \theta} \tag{5}$$

5: 更新 θ:

$$\theta^{(t+1)} := \theta^{(t)} - \epsilon \cdot \frac{\partial J(X_b, Y_b; \theta)}{\partial \theta} \tag{6}$$

6: 更新迭代步数: $t := t + 1$

7: **end while**

8: 输出 $\hat{\theta} = \theta^{(t)}$

2.3 卷积玻尔兹曼机 (CRBM)

传统的神经网络模型对于处理低维输入并且各个特征之间无关联的数据非常有效。然而，对于图像的处理，传统的神经网络模型并不能取得很好的效果。这一方面是因为图像的维度往往较高 (e.g. 100*100 = 1 万维)，另一方面则是图像具有特殊的二维结构信息。因此，一种加入了这种结构信息的神经网络模型—卷积玻尔兹曼机 (Convolutional Restricted Boltzmann Machine) —被提出并被用来处理图像输入。

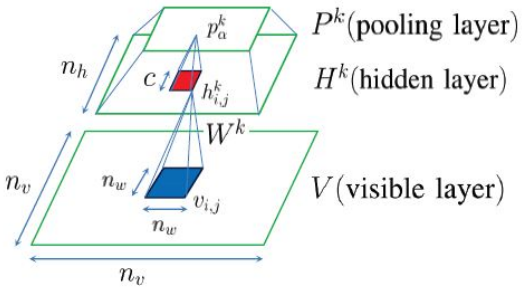


图 3: 卷积玻尔兹曼机示意图

图 3 所示是卷积玻尔兹曼机的一部分。它一般由 K_{in} 个输入特征 V 和

K_{out} 个隐含层特征 H 组成。这里 $K_{in}=1$，$k=1,2,\cdots,K_{out}$。除去这种结构上的差别，卷积玻尔兹曼机的求解仍然可以用传统的梯度下降算法来近似求解。只不过它的推断函数将公式 (2) 中的 $W_i x_i$ 换成了二维卷积的形式：

$$h_{i,j}^k = (W^k * v)_{ij} + b_k \tag{7}$$

这里 * 是二维卷积算子；h 的尺寸比 v 小，因为超过边界值的卷积被忽略。这种卷积一般称为 valid 卷积。

从隐含层 H 到下采样层 P 要经过下采样操作，它将 H 层的 $c \times c$ 区域采样成某个值。例如，如果选择极大采样的方法，则只要该 $c \times c$ 区域中有一个值为 1，则采样值为 1，否则为 0。

卷积玻尔兹曼机还提供反向的推断，即通过给定 H 层的值来推断 V。这时只需要将公式 7中的 v、h 位置互换即可。对边界的处理可以通过补充 0 来实现。这种卷积一般称为 full 卷积。

3 问题求解的两种并行思路

3.1 模型并行

模型并行是解决大规模模型的一种常用解决方案。注意到在公式 (7) 中，各个 $h_{i,j}^k$ 之间的求解是不会互相干扰的。因此可以并行的求解 $h_{i,j}^k\,(1 \le i,j \le n_h,\, k=1,2,\cdots,K_{out})$。反之，从 H 推断 V 的时候也是一样的思路。

3.2 数据并行

数据并行是有效解决大数据的一种加速方法。如公式 (5) 所示，由于各个数据得到的导数 $\frac{\partial f(x^{(i)},y^{(i)};\theta)}{\partial \theta}$ 是独立的，因此可以并行的求解各个数据得到的导数，然后求平均得到总的导数。以此来更新参数 θ。

3.3 编程考虑

编程实现时，考虑到 CUDA 的两级并行框架，即 block 与 grid，可以与上面的两种并行很好的对应。首先，模型并行对应于 CUDA 的 thread block。每个 block 中开辟固定尺寸的线程块，来计算某一些 $h_{ij}^k\,(1 \le i,j \le n_h)$。注意到硬件的限制，每个 block 中容纳的线程上限是有限制的。这里我用的 GPU 是 K20C，它每个 block 中最多容纳 1024 个线程。因此，我将 block 尺寸设为 16×16 可以达到要求 (注意：设置太小，例如 8×8，容易导致资源闲置，设置太大，例如 32×32，则容易导致 block 内的同步时间太长)。由于 block 尺寸的限制，当 $n_h > 16$ 时，某些线程一次最多需要计算 $(\lfloor (n_h-1)/16 \rfloor + 1)^2$ 个 $h_{ij}^k\,(1 \le i,j \le n_h)$ 的值。

kernel 函数名	block 尺寸	grid 尺寸
valid 卷积	(16, 16)	(K_{in}, K_{out}, m_b)
full 卷积	(16, 16)	(K_{out}, K_{in}, m_b)
pooling	(16, 16)	(K_{out}, m_b)
element-wise opration	256	(K_{out}, m_b)

表 1: 主要 kernel 函数的配置参数

其次，数据并行对应于 CUDA 的 block grid。grid 的尺寸对应执行一次内核所处理的数据量。不同于 block 尺寸的限制，grid 的尺寸往往可以大很多。因此，可以将 $h^k_{ij}\,(1 \le i, j \le n_h)$ 的 K_{out} 个特征分配给不同的 block 来处理。按照这种处理，grid 的尺寸可以设置为 $K_{out} \times m_b$。

表 1 给出了主要内核函数 (kernel function) 的配置参数。

作为一个例子，下面用伪代码描述由 V 和 W 得到 H 的 valid 卷积的具体的并行思路：

```
================================================================
#define WIDTH 8  // 预先固定共享内存的宽度
__global__ void conv4d_valid_kernel
(float *d_H, float *d_V, float *d_W, int nv, int nw){
    // 获取线程 id 和 block id
    const int tx = threadIdx.x;  // 对应于 i
    const int ty = threadIdx.y;  // 对应于 j
    const int bx = blockIdx.x;   // 对应于 v: Kin id
    const int by = blockIdx.y;   // 对应于 k: Kout id
    const int bz = blockIdx.z;   // 对应于 l: mb id

    // 声明共享内存Ws并将 W 对应的部分读入 Ws 供当前 block 使用
    __shared__ float Ws[WIDTH][WIDTH];

    if(ty < nw && tx < nw){
        Ws[ty][tx] = d_W[tx, ty, bx, by];
    }
    __syncthreads();  // 同步；因为后面都得用到这个共享的 Ws

    const int nh = nv - nw + 1;
    // tiles_x：当前 block 沿 x 方向平移来覆盖 nh*nh 的区域
    const int tiles_x = nh / blockDim.x + 1;
    // tiles_y：当前 block 沿 y 方向平移来覆盖 nh*nh 的区域
    const int tiles_y = nh / blockDim.y + 1;
```

```
    for iy = 0 to tiles_y-1
        for ix = 0 to tiles_x-1
            // 当前计算的 H 中元素对应的位置
            int idH_y = ty + iy * blockDim.y;
            int idH_x = tx + ix * blockDim.x;

            // 线程越界则抛弃
            if(idH_x >= nh || idH_y >= nh)
                continue;

            // 计算 V 与 Ws 的卷积
            float temp = conv(d_V, Ws);

            // 将当前卷积值加入到H中
            // 对 Kout 个 block 都得执行这一步, 因此用原子操作
            atomicAdd(&d_H[idH_x, idH_y, by, bz], temp);
        endfor
    endfor
}
================================================================
```

4 实验

4.1 数据和模型

为便于对比，实验分两部分进行，一部分是不用并行的单 CPU 进行运算 (作为对照实验)，另一部分是用 GPU 并行加速的运算。最后根据加速比来评价加速效果。

下面是实验的数据和主要参数：

- **数据**：10,000 个 31×31 的灰度图像。
- **迭代次数**：50,000 次 (尽可能大的消除随机性的影响)
- K_{out} **尺寸**：从 2 变化到 256，以 2 倍速增长
- m_b **大小**：从 1 变化到 64，以 2 倍速增长
- **卷积核大小**：8×8

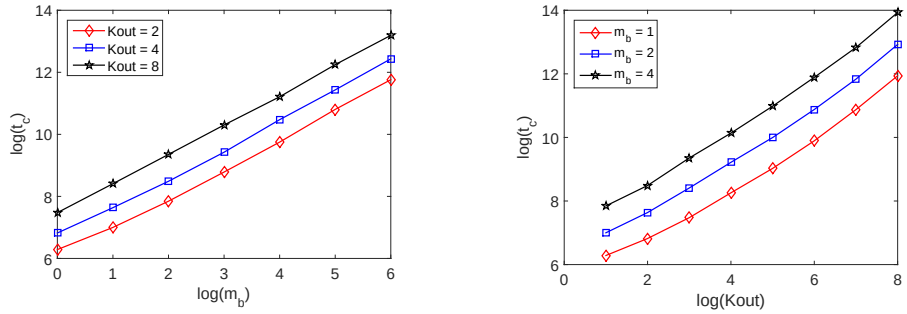
4.2 CPU 时间

CPU 配置：Intel(R) Core(TM) i7-4700 CPU @ 3.4GHz。

理论上，CPU 的运行时间 t_c 应该跟 K_{out} 的个数和 m_b 的大小都近似成正比例关系，因为这里关于 K_{out} 和 m_b 相当于两层 for 循环。下面分别来验证这两点。

我们先来考察 t_c 与批数据的尺寸 m_b 的关系。图 4(a)展示了 $K_{out} = 2$, 4, 和 8 时 t_c 随 m_b 变化而变化的曲线。从图中可以看出，$\log(t_c)$ 与 $\log(m_b)$ 近似成线性关系。从而 t_c 与 m_b 也近似成线性关系。这个结果与预期相符。

另一方面，我们也考察 t_c 与输出特征个数 K_{out} 之间的关系。图 4(b) 展示了 $m_b = 1, 2, 4$ 时 t_c 随 Kout 变化而变化的曲线。可以看出，$\log(t_c)$ 与 $\log(K_{out})$ 也近似成线性关系。从而 t_c 与 K_{out} 也近似成线性关系。



(a) m_b - t_c 曲线　　(b) K_{out} - t_c 曲线

图 4: CPU 运算时间 t_c 随 m_b 和 K_{out} 的变化关系图。

4.3 GPU 时间

GPU 配置：Tesla K20c，参数见表 2 或 K20c.pdf 文件。

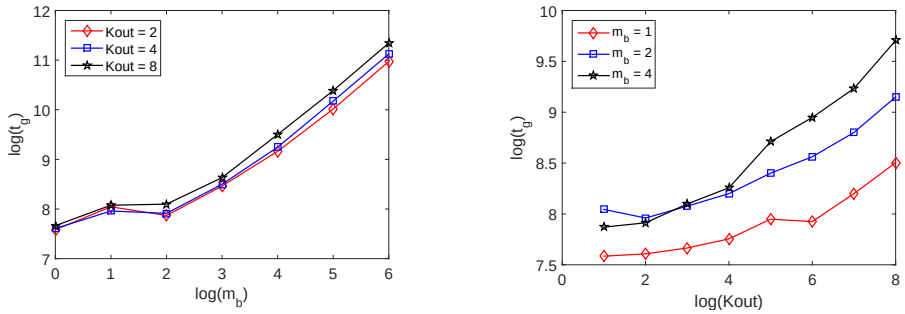
不同于 CPU 的单核心运算，GPU 往往有成千上万个核心 (这里是 2496 个)。当模型规模较小、数据规模较小时，并不一定需要调动所有的 GPU 核心一起运算。因此，K_{out} 和 m_b 较小时 GPU 的运算时间 t_g 的增长并不会与 K_{out} 或 m_b 成正比例关系，而是几乎不增长或者增长非常缓慢，直到达到计算能力的上限。此后，随着 K_{out} 或 m_b 的进一步增长，t_g 也应该是近似线性的增长。同样，我们分两部分来验证上述预期。

我们先来考察 t_g 与批数据的尺寸 m_b 的关系。图 5(a)展示了 $K_{out} = 2$, 4, 和 8 时 t_g 随 m_b 变化而变化的曲线。图中可以看出，$\log(t_g)$ 与 $\log(m_b)$ 不再是线性的关系。当 m_b 比较小 ($m_b \leq 2^2$) 时，t_g 的变化非常平缓；而当 m_b 比较大时，t_g 与 m_b 近似成线性关系。

参数描述	参数值
Total amount of global memory	4800 MBytes
(13) Multiprocessors, (192) CUDA Cores/MP	2496 CUDA Cores
GPU Clock rate	706 MHz
Memory Clock rate	2600 Mhzs
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Max dimension size of a thread block (x,y,z)	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z)	(2147483647, 65535, 65535)

表 2: Tesla K20c 主要配置参数

另一方面,我们也考察 t_g 与 K_{out} 之间的关系。图 5(b) 展示了 $m_b = 1, 2, 4$ 时 t_g 随 $Kout$ 变化而变化的曲线。图中 $m_b = 2$ 对应的 t_g 在 K_{out} 较小时偶尔会比 $m_b = 4$ 时的更大，这可能是实验误差引起的。同样，当 K_{out} 比较小时，t_g 的增长比较缓慢；当 K_{out} 超过某个值 (根据 m_b 的不同而不同) 时，t_g 开始关于 K_{out} 成近似线性的关系增长。



(a) m_b - t_g (b) Kout - t_g

图 5: GPU 运算时间 t_g 随 m_b 和 K_{out} 的变化关系图。

4.4 加速比曲线

加速比定义为

$$\text{speed up} \quad := \quad \frac{t_c}{t_g}$$

图 6 给出了不同 K_{out} 下加速比随 m_b 的变化而变化曲线。从图中可以看出，当 K_{out} 比较小时，加速比的变化并不明显。随着 K_{out} 取值的变大，m_b-加速比曲线的变化越明显。

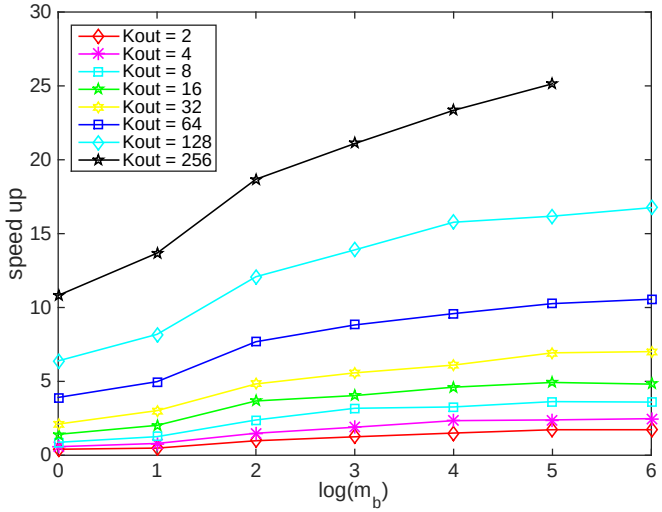


图 6: K_{out} 取不同值时的 m_b - 加速比 曲线

同时，我们也给出不同 m_b 下加速比随 K_{out} 变化而变化的曲线。见图 7。

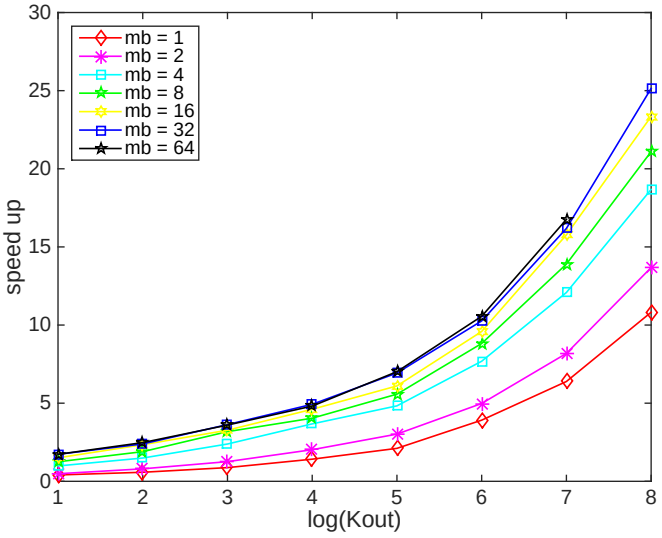


图 7: m_b 取不同值时的 K_{out} - 加速比 曲线

4.5 部分结果展示

最后学习到的参数 W 经过可视化后如图 8 所示。它是一种底层的 Garbor 特征提取器，可以用作高层特征的提取。

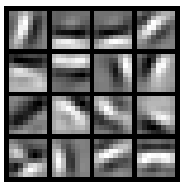
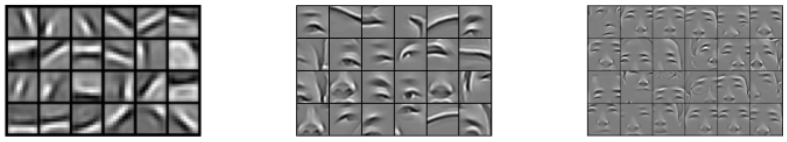


图 8: 参数 W 的可视化 ($K_{out} = 16, m_b = 1$)

作为拓展，图 9 也展示了我们工作中学习到的一些高层的人脸特征，请参考 [6]。



(a) (b) (c)

Fig. 7: Visualization of the first (a), second (b), and third (c) layer bases. The first layer bases are visualized as "images". The second and the third layers bases are visualized as the patterns (which are obtained by hierarchically inference of the CDBN) most liked by the second and third layer's hidden units respectively.

图 9: 高层特征的可视化

5 结论

通过上面的实验结果我们得出**结论**：随着问题规模 K_{out} 的扩大和数据量 m_b 的扩大，加速比的增长越来越明显。上述实验中最多实现了 **25 倍速**的加速比。根据加速比的增长曲线趋势可以看出，如果继续扩大 K_{out} 和 m_b，则加速比很有可能进一步增加。考虑到时间的限制，这里没有继续增大实验规模，而是把这个猜测留到实际应用中去检验。

最后，上面的实验只是一个相对简单的示例性结果。真正的应用中数据规模可以达到几十万到上百万，输入图片的尺寸可以达到 200×200 以上 (例如 ImageNet 数据集包含 120 万张 256×256 左右大小的自然图像)。这时传统的 CPU 运算根本不可能 (1 年甚至以上时间训练)，而基于 GPU 的

加速的可能只需要几天的时间。从而，使用 GPU 加速为实现大规模、大数据量下的卷积玻尔兹曼机训练问题提供了一种高效、可行的解决方案。

附录：代码说明

这里我们使用 matlab 作为前端的语言，提供 CPU 和 GPU 两套接口，分别调用 cpu 下的 crbm 程序和基于 CUDA C 的内核程序。

基于 GPU 的代码中间由 mex 文件连接，它让 matlab 能够调用 C/C++ 程序。总结起来就是：matlab 中的 crbm 调用 mex 函数，mex 函数调用 C/C++ 接口函数，C/C++ 接口函数调用各个内核来执行最终的运算。具体的代码见 source codes。我使用的操作系统是 ubuntu 12.04；mex 调用的 gcc/g++ 版本是 4.6.2；matlab 版本是 R2012 版。

参考资料

[1] http://www.csdn.net/article/2015-09-25/2825806

[2] http://caffe.berkeleyvision.org/

[3] http://mxnet.readthedocs.org/en/latest/

[4] https://www.tensorflow.org/

[5] http://dl.acm.org/citation.cfm?id=1553453

[6] H. Qiao, X. Xi, Y. Li, W. Wu, and F. Li, *Biologically Inspired Visual Model With Preliminary Cognition and Active Attention Adjustment*, IEEE Transactions on Cybernetics, 2014, vol. 45, pp. 2612 - 2624.