UFLDL 教程

ver 201406*

1 介绍

本教程将阐述无监督特征学习和深度学习的主要观点。通过学习，你也将实现多个功能学习/深度学习算法，能看到它们为你工作，并学习如何应用/适应这些想法到新问题上。

本教程假定机器学习的基本知识（特别是熟悉的监督学习，逻辑回归，梯度下降的想法），如果你不熟悉这些想法，我们建议你去这里 机器学习课程，并先完成第 II，III，IV 章（到逻辑回归）。

英文原文作者: Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen

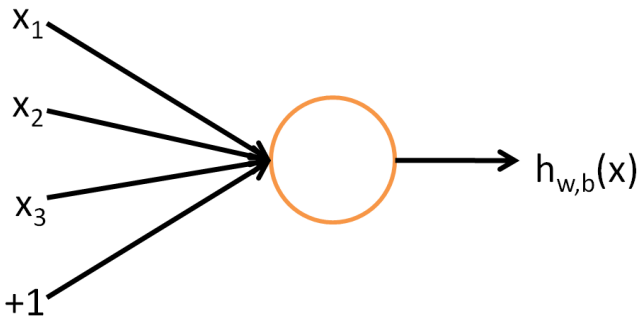
*本文档基于UFLDL 教程 2.15 版本编辑而成。你可以在GitHub 上获取本文档的最新版。如果你有什么建议，请在网站上留言。是的，这文档在排版上还不完美，如果你愿意，也可以自己动手修改，别忘了分享你的更新并通知我更新这个文档。谢谢！

2 稀疏自编码器

2.1 神经网络

以监督学习为例，假设我们有训练样本集 $(x^{(i)}, y^{(i)})$，那么神经网络算法能够提供一种复杂且非线性的假设模型 $h_{W,b}(x)$，它具有参数 W, b，可以以此参数来拟合我们的数据。

为了描述神经网络，我们先从最简单的神经网络讲起，这个神经网络仅由一个“神经元”构成，以下即是这个“神经元”的图示：



这个“神经元”是一个以 x_1, x_2, x_3 及截距 $+1$ 为输入值的运算单元，其输出为 $h_{W,b}(x) = f(W^T x) = f(\sum_{i=1}^{3} W_i x_i + b)$，其中函数 $f : \Re \mapsto \Re$ 被称为“激活函数”。在本教程中，我们选用 sigmoid 函数作为**激活函数** $f(\cdot)$

$$f(z) = \frac{1}{1 + \exp(-z)}.$$

可以看出，这个单一“神经元”的输入－输出映射关系其实就是一个逻辑回归（logistic regression）。虽然本系列教程采用 sigmoid 函数，但你也可以选择双曲正切函数（tanh）：

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

以下图 1 分别是 sigmoid 及 tanh 的函数图像

$\tanh(z)$ 函数是 sigmoid 函数的一种变体，它的取值范围为 $[-1, 1]$，而不是 sigmoid 函数的 $[0, 1]$。

注意，与其它地方（包括 OpenClassroom 公开课以及斯坦福大学 CS229 课程）不同的是，这里我们不再令 $x_0 = 1$。取而代之，我们用单独的参数 b 来表示截距。

最后要说明的是，有一个等式我们以后会经常用到：如果选择 $f(z) = 1/(1 + \exp(-z))$，也就是 sigmoid 函数，那么它的导数就是 $f'(z) = f(z)(1 - f(z))$（如果选择 tanh 函数，那它的导数就是 $f'(z) = 1 - (f(z))^2$，你可以根据 sigmoid（或 tanh）函数的定义自行推导这个等式。

2.1.1 神经网络模型

所谓神经网络 就是将许多个单一“神经元”联结在一起，这样，一个“神经元”的输出就可以是另一个“神经元”的输入。例如，下图就是一个简单的神经网络：

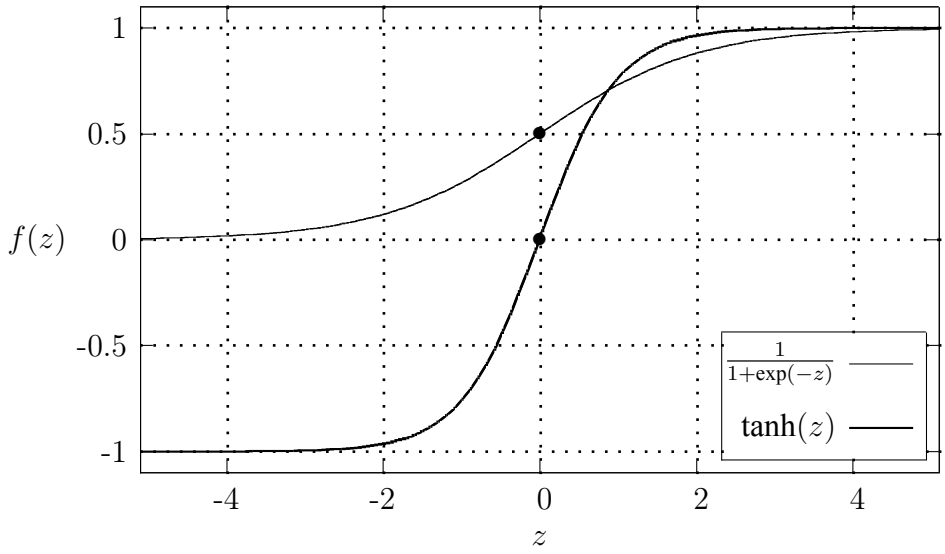


图 1: 激活函数

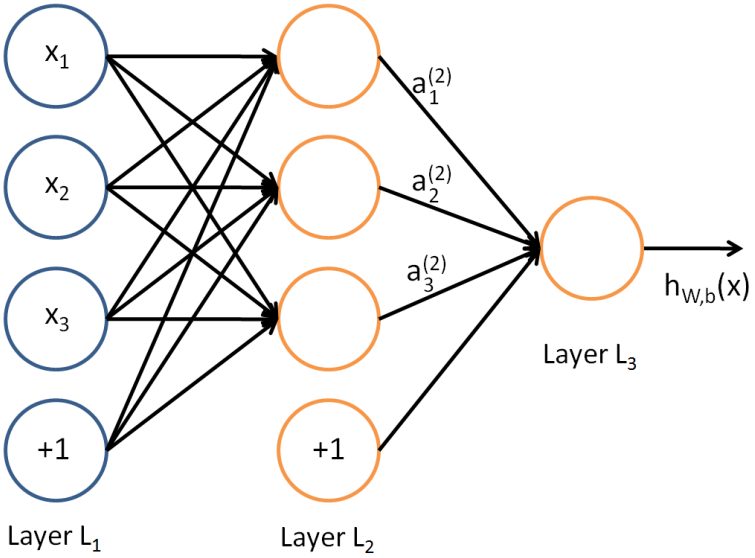
我们使用圆圈来表示神经网络的输入，标上“+1”的圆圈被称为**偏置节点**，也就是截距项。神经网络最左边的一层叫做**输入层**，最右的一层叫做**输出层**（本例中，输出层只有一个节点）。中间所有节点组成的一层叫做**隐藏层**，因为我们不能在训练样本集中观测到它们的值。同时可以看到，以上神经网络的例子中有 3 个输入单元（偏置单元不计在内），3 个隐藏单元及一个输出单元。

我们用 n_l 来表示网络的层数，本例中 $n_l = 3$，我们将第 l 层记为 L_l，于是 L_1 是输入层，输出层是 L_{n_l}。本例神经网络有参数 $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$，其中 $W_{ij}^{(l)}$（下面的式子中用到）是第 l 层第 j 单元与第 $l+1$ 层第 i 单元之间的联接参数（其实就是连接线上的权重，注意标号顺序），$b_i^{(l)}$ 是第 $l+1$ 层第 i 单元的偏置项。因此在本例中，$W^{(1)} \in \Re^{3\times 3}$，$W^{(2)} \in \Re^{1\times 3}$。注意，没有其他单元连向偏置单元 (即偏置单元没有输入)，因为它们总是输出 $+1$。同时，我们用 s_l 表示第 l 层的节点数（偏置单元不计在内）。

我们用 $a_i^{(l)}$ 表示第 l 层第 i 单元的激活值（输出值）。当 $l = 1$ 时，$a_i^{(1)} = x_i$，也就是第 i 个输入值（输入值的第 i 个特征）。对于给定参数集合 W, b，我们的神经网络就可以按照函数 $h_{W,b}(x)$ 来计算输出结果。本例神经网络的计算步骤如下:

$$
\begin{aligned}
a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\
a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\
a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\
h_{W,b}(x) &= a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})
\end{aligned}
$$

我们用 $z_i^{(l)}$ 表示第 l 层第 i 单元输入加权和 (包括偏置单元)，比如，$z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)}x_j + b_i^{(1)}$，



则 $a_i^{(l)} = f(z_i^{(l)})$。

这样我们就可以得到一种更简洁的表示法。这里我们将激活函数 $f(\cdot)$ 扩展为用向量（分量的形式）来表示，即 $f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$，那么，上面的等式可以更简洁地表示为：

$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h_{W,b}(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}
$$

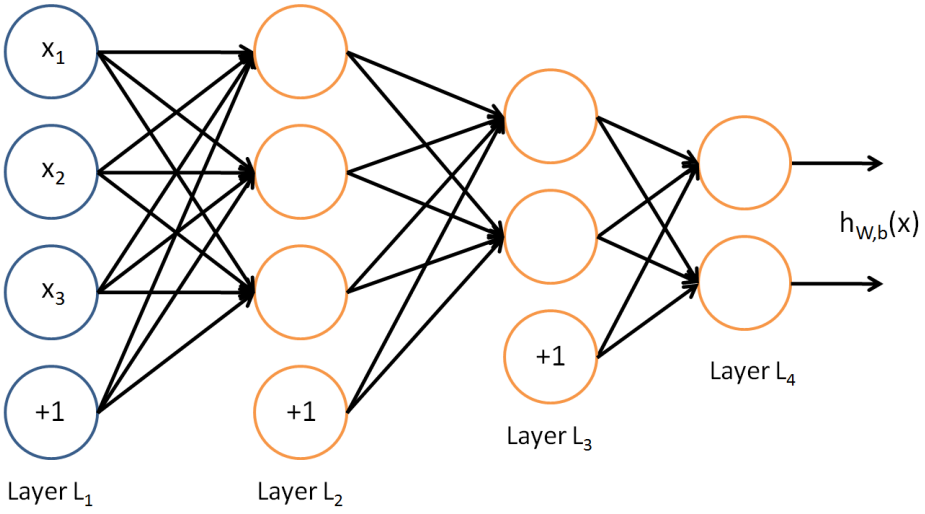
我们将上面的计算步骤叫作**前向传播**。回想一下，之前我们用 $a^{(1)} = x$ 表示输入层的激活值，那么给定第 l 层的激活值 $a^{(l)}$ 后，第 $l+1$ 层的激活值 $a^{(l+1)}$ 就可以按照下面步骤计算得到：

$$
\begin{aligned}
z^{(l+1)} &= W^{(l)}a^{(l)} + b^{(l)} \\
a^{(l+1)} &= f(z^{(l+1)})
\end{aligned}
$$

将参数矩阵化，使用矩阵－向量运算方式，我们就可以利用线性代数的优势对神经网络进行快速求解。

目前为止，我们讨论了一种神经网络，我们也可以构建另一种结构的神经网络（这里结构指的是神经元之间的联接模式），也就是包含多个隐藏层的神经网络。最常见的一个例子是 n_l 层的神经网络，第 1 层是输入层，第 n_l 层是输出层，中间的每个层 l 与层 $l+1$ 紧密相联。这种模式下，要计算神经网络的输出结果，我们可以按照之前描述的等式，按部就班，进行前向传播，逐一计算第 L_2 层的所有激活值，然后是第 L_3 层的激活值，以此类推，直到第 L_{n_l} 层。这是一个前馈神经网络的例子，因为这种联接图没有闭环或回路。

神经网络也可以有多个输出单元。比如，下面的神经网络有两层隐藏层：L_2 及 L_3，输出层 L_4 有两个输出单元。



要求解这样的神经网络，需要样本集 $(x^{(i)}, y^{(i)})$，其中 $y^{(i)} \in \Re^2$。如果你想预测的输出是多个的，那这种神经网络很适用。（比如，在医疗诊断应用中，患者的体征指标就可以作为向量 x 的输入值，而不同的输出值 y_i 可以表示不同的疾病存在与否。）

2.2 反向传导算法

假设我们有一个固定样本集 $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$，它包含 m 个样例。我们可以用批量梯度下降法来求解神经网络。具体来讲，对于单个样例 (x, y)，其代价函数为：

$$J(W, b; x, y) = \frac{1}{2} \left\| h_{W,b}(x) - y \right\|^2 . \tag{1}$$

这是一个（二分之一的）方差代价函数。给定一个包含 m 个样例的数据集，我们可以定义整体代价函数为：

$$\begin{aligned} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 \qquad (2) \\ &= \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_{W,b}(x^{(i)}) - y^{(i)} \right\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 \qquad (3) \end{aligned}$$

以上公式中的第一项 $J(W, b)$ 是一个均方差项。第二项是一个规则化项（也叫**权重衰减项**），其目的是减小权重的幅度，防止过度拟合。[1]

[1] 通常权重衰减的计算并不使用偏置项 $b_i^{(l)}$，比如我们在 $J(W, b)$ 的定义中就没有使用。一般来说，将偏置项包含在权重衰减项中只会对最终的神经网络产生很小的影响。如果你在斯坦福选修过 CS229（机器学习）课程，或者在 YouTube 上看过课程视频，你会发现这个权重衰减实际上是课上提到的贝叶斯规则化方法的变种。在贝叶斯规则化方法中，我们将高斯先验概率引入到参数中计算 MAP（极大后验）估计（而不是极大似然估计）。

权重衰减参数 λ 用于控制公式中两项的相对重要性。在此重申一下这两个复杂函数的含义：$J(W,b;x,y)$ 是针对单个样例计算得到的方差代价函数；$J(W,b)$ 是整体样本代价函数，它包含权重衰减项。

以上的代价函数经常被用于分类和回归问题。在分类问题中，我们用 $y=0$ 或 1，来代表两种类型的标签（回想一下，这是因为 sigmoid 激活函数的值域为 $[0,1]$；如果我们使用双曲正切型激活函数，那么应该选用 -1 和 $+1$ 作为标签）。对于回归问题，我们首先要变换输出值域（译者注：也就是 y），以保证其范围为 $[0,1]$（同样地，如果我们使用双曲正切型激活函数，要使输出值域为 $[-1,1]$）。

我们的目标是针对参数 W 和 b 来求其函数 $J(W,b)$ 的最小值。为了求解神经网络，我们需要将每一个参数 $W_{ij}^{(l)}$ 和 $b_i^{(l)}$ 初始化为一个很小的、接近零的随机值（比如说，使用正态分布 $Normal(0,\epsilon^2)$ 生成的随机值，其中 ϵ 设置为 0.01 ），之后对目标函数使用诸如批量梯度下降法的最优化算法。因为 $J(W,b)$ 是一个非凸函数，梯度下降法很可能会收敛到局部最优解；但是在实际应用中，梯度下降法通常能得到令人满意的结果。最后，需要再次强调的是，要将参数进行随机初始化，而不是全部置为 0。如果所有参数都用相同的值作为初始值，那么所有隐藏层单元最终会得到与输入值有关的、相同的函数（也就是说，对于所有 i，$W_{ij}^{(1)}$ 都会取相同的值，那么对于任何输入 x 都会有：$a_1^{(2)}=a_2^{(2)}=a_3^{(2)}=\ldots$ ）。随机初始化的目的是使**对称失效**。

梯度下降法中每一次迭代都按照如下公式对参数 W 和 b 进行更新：

$$W_{ij}^{(l)}=W_{ij}^{(l)}-\alpha\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b) \tag{4}$$

$$b_i^{(l)}=b_i^{(l)}-\alpha\frac{\partial}{\partial b_i^{(l)}}J(W,b) \tag{5}$$

其中 α 是学习速率。其中关键步骤是计算偏导数。我们现在来讲一下**反向传播算法**，它是计算偏导数的一种有效方法。

我们首先来讲一下如何使用反向传播算法来计算 $\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b;x,y)$ 和 $\frac{\partial}{\partial b_i^{(l)}}J(W,b;x,y)$，这两项是单个样例 (x,y) 的代价函数 $J(W,b;x,y)$ 的偏导数。一旦我们求出该偏导数，就可以推导出整体代价函数 $J(W,b)$ 的偏导数：

$$\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b)=\left[\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b;x^{(i)},y^{(i)})\right]+\lambda W_{ij}^{(l)} \tag{6}$$

$$\frac{\partial}{\partial b_i^{(l)}}J(W,b)=\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial b_i^{(l)}}J(W,b;x^{(i)},y^{(i)}) \tag{7}$$

以上两行公式稍有不同，第一行比第二行多出一项，是因为权重衰减是作用于 W 而不是 b。

反向传播算法的思路如下：给定一个样例 (x,y)，我们首先进行“前向传导”运算，计算出网络中所有的激活值，包括 $h_{W,b}(x)$ 的输出值。之后，针对第 l 层的每一个节点 i，我们计算

出其“残差”$\delta_i^{(l)}$，该残差表明了该节点对最终输出值的残差产生了多少影响。对于最终的输出节点，我们可以直接算出网络产生的激活值与实际值之间的差距，我们将这个差距定义为 $\delta_i^{(n_l)}$（第 n_l 层表示输出层）。对于隐藏单元我们如何处理呢? 我们将基于节点（译者注：第 $l+1$ 层节点）残差的加权平均值计算 $\delta_i^{(l)}$，这些节点以 $a_i^{(l)}$ 作为输入。下面将给出反向传导算法的细节：

1. 进行前馈传导计算，利用前向传导公式，得到 $L_2, L_3, \ldots$ 直到输出层 L_{n_l} 的激活值。

2. 对于第 n_l 层（输出层）的每个输出单元 i，我们根据以下公式计算残差：

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)}) \tag{8}$$

[2]

3. 对 $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ 的各个层，

   计算第 l 层的第 i 个节点的残差：

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

[3]

4. 计算我们需要的偏导数，计算方法如下：

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)} \tag{17}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}. \tag{18}$$

最后，我们用矩阵 -向量表示法重写以上算法。我们使用“$\bullet$”表示向量乘积运算符（在 Matlab 或 Octave 里用“$.*$”表示，也称作阿达马乘积）。若 $a = b \bullet c$，则 $a_i = b_i c_i$。在上一个

[2]译者注：

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{n_l}} J(W, b; x, y) = \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 \tag{9}$$

$$= \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2} \sum_{j=1}^{S_{n_l}} (y_j - a_j^{(n_l)})^2 = \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2} \sum_{j=1}^{S_{n_l}} (y_j - f(z_j^{(n_l)}))^2 \tag{10}$$

$$= -(y_i - f(z_i^{(n_l)})) \cdot f'(z_i^{(n_l)}) = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)}) \tag{11}$$

[3]译者注：

教程中我们扩展了 $f(\cdot)$ 的定义，使其包含向量运算，这里我们也对偏导数 $f'(\cdot)$ 也做了同样的处理（于是又有 $f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$）。

那么，反向传播算法可表示为以下几个步骤：

1. 进行前馈传导计算，利用前向传导公式，得到 $L_2, L_3, \ldots$ 直到输出层 L_{n_l} 的激活值。

2. 对输出层（第 n_l 层），计算：

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)}) \tag{19}$$

3. 对于 $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ 的各层，计算：

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)}) \tag{20}$$

4. 计算最终需要的偏导数值：

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T, \tag{21}$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}. \tag{22}$$

$$\delta_i^{(n_l-1)} = \frac{\partial}{\partial z_i^{n_l-1}} J(W, b; x, y) = \frac{\partial}{\partial z_i^{n_l-1}} \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 = \frac{\partial}{\partial z_i^{n_l-1}} \frac{1}{2} \sum_{j=1}^{S_{n_l}} (y_j - a_j^{(n_l)})^2 \tag{12}$$

$$= \frac{1}{2} \sum_{j=1}^{S_{n_l}} \frac{\partial}{\partial z_i^{n_l-1}} (y_j - a_j^{(n_l)})^2 = \frac{1}{2} \sum_{j=1}^{S_{n_l}} \frac{\partial}{\partial z_i^{n_l-1}} (y_j - f(z_j^{(n_l)}))^2 \tag{13}$$

$$= \sum_{j=1}^{S_{n_l}} -(y_j - f(z_j^{(n_l)})) \cdot \frac{\partial}{\partial z_i^{(n_l-1)}} f(z_j^{(n_l)}) = \sum_{j=1}^{S_{n_l}} -(y_j - f(z_j^{(n_l)})) \cdot f'(z_j^{(n_l)}) \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{(n_l-1)}} \tag{14}$$

$$= \sum_{j=1}^{S_{n_l}} \delta_j^{(n_l)} \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{n_l-1}} = \sum_{j=1}^{S_{n_l}} \left(\delta_j^{(n_l)} \cdot \frac{\partial}{\partial z_i^{n_l-1}} \sum_{k=1}^{S_{n_l-1}} f(z_k^{n_l-1}) \cdot W_{jk}^{n_l-1} \right) \tag{15}$$

$$= \sum_{j=1}^{S_{n_l}} \delta_j^{(n_l)} \cdot W_{ji}^{n_l-1} \cdot f'(z_i^{n_l-1}) = \left(\sum_{j=1}^{S_{n_l}} W_{ji}^{n_l-1} \delta_j^{(n_l)} \right) f'(z_i^{n_l-1}) \tag{16}$$

将上式中的 $n_l - 1$ 与 n_l 的关系替换为 l 与 $l+1$ 的关系，就可以得到：

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

以上逐次从后向前求导的过程即为“反向传导”的本意所在。

实现中应注意：在以上的第 2 步和第 3 步中，我们需要为每一个 i 值计算其 $f'(z_i^{(l)})$。假设 $f(z)$ 是 sigmoid 函数，并且我们已经在前向传导运算中得到了 $a_i^{(l)}$。那么，使用我们早先推导出的 $f'(z)$ 表达式，就可以计算得到 $f'(z_i^{(l)}) = a_i^{(l)}(1 - a_i^{(l)})$。

最后，我们将对梯度下降算法做个全面总结。在下面的伪代码中，$\Delta W^{(l)}$ 是一个与矩阵 $W^{(l)}$ 维度相同的矩阵，$\Delta b^{(l)}$ 是一个与 $b^{(l)}$ 维度相同的向量。注意这里“$\Delta W^{(l)}$”是一个矩阵，而不是“Δ 与 $W^{(l)}$ 相乘”。下面，我们实现批量梯度下降法中的一次迭代：

1. 对于所有 l，令 $\Delta W^{(l)} := 0$, $\Delta b^{(l)} := 0$ （设置为全零矩阵或全零向量）

2. 对于 $i = 1$ 到 m，

   (a) 使用反向传播算法计算 $\nabla_{W^{(l)}} J(W, b; x, y)$ 和 $\nabla_{b^{(l)}} J(W, b; x, y)$。

   (b) 计算 $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$。

   (c) 计算 $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$。

3. 更新权重参数：

$$W^{(l)} = W^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right] \tag{23}$$

$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right] \tag{24}$$

现在，我们可以重复梯度下降法的迭代步骤来减小代价函数 $J(W, b)$ 的值，进而求解我们的神经网络。

2.3 梯度检验与高级优化

众所周知，反向传播算法很难调试得到正确结果，尤其是当实现程序存在很多难于发现的 bug 时。举例来说，索引的缺位错误（off-by-one error）会导致只有部分层的权重得到训练，再比如忘记计算偏置项。这些错误会使你得到一个看似十分合理的结果（但实际上比正确代码的结果要差）。因此，但从计算结果上来看，我们很难发现代码中有什么东西遗漏了。本节中，我们将介绍一种对求导结果进行数值检验的方法，该方法可以验证求导代码是否正确。另外，使用本节所述求导检验方法，可以帮助你提升写正确代码的信心。[4]

假设我们想要最小化以 θ 为自变量的目标函数 $J(\theta)$。假设 $J : \Re \mapsto \Re$，则 $\theta \in \Re$。在一维的情况下，一次迭代的梯度下降公式是

$$\theta := \theta - \alpha \frac{d}{d\theta} J(\theta). \tag{25}$$

[4]缺位错误（Off-by-one error）举例说明：比如 for 循环中循环 m 次，正确应该是 $for(i = 1;\ i <= m;\ i++)$，但有时程序员疏忽，会写成 $for(i = 1;\ i < m;\ i++)$，这就是缺位错误。

再假设我们已经用代码实现了计算 $\frac{d}{d\theta}J(\theta)$ 的函数 $g(\theta)$，接着我们使用 $\theta := \theta - \alpha g(\theta)$ 来实现梯度下降算法。那么我们如何检验 g 的实现是否正确呢?

回忆导数的数学定义：

$$\frac{d}{d\theta}J(\theta) = \lim_{\epsilon \to 0} \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}. \tag{26}$$

那么对于任意 θ 值，我们都可以对等式左边的导数用：

$$\frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}} \tag{27}$$

来近似。

实际应用中，我们常将 $EPSILON$ 设为一个很小的常量，比如在 10^{-4} 数量级（虽然 $EPSILON$ 的取值范围可以很大，但是我们不会将它设得太小，比如 10^{-20}，因为那将导致数值舍入误差。）

给定一个被认为能计算 $\frac{d}{d\theta}J(\theta)$ 的函数 $g(\theta)$，我们可以用下面的数值检验公式

$$g(\theta) \approx \frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}}. \tag{28}$$

计算两端是否一样来检验函数是否正确。

上式两端值的接近程度取决于 J 的具体形式。但是在假定 $\text{EPSILON} = 10^{-4}$ 的情况下，你通常会发现上式左右两端至少有 4 位有效数字是一样的（通常会更多）。

现在，考虑 $\theta \in \Re^n$ 是一个向量而非一个实数（那么就有 n 个参数要学习得到），并且 $J : \Re^n \mapsto \Re$。在神经网络的例子里我们使用 $J(W, b)$，可以想象为把参数 W, b 组合扩展成一个长向量 θ。现在我们将求导检验方法推广到一般化，即 θ 是一个向量的情况。

假设我们有一个用于计算 $\frac{\partial}{\partial\theta_i}J(\theta)$ 的函数 $g_i(\theta)$；我们想要检验 g_i 是否输出正确的求导结果。我们定义 $\theta^{(i+)} = \theta + \text{EPSILON} \times \vec{e}_i$，其中

$$\vec{e}_i = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \tag{29}$$

是第 i 个基向量（维度和 θ 相同，在第 i 行是“1”而其他行是“0”）。所以，$\theta^{(i+)}$ 和 θ 几乎相同，除了第 i 行元素增加了 $EPSILON$。类似地，$\theta^{(i-)} = \theta - \text{EPSILON} \times \vec{e}_i$ 得到的第 i 行减小了 $EPSILON$。然后我们可以对每个 i 检查下式是否成立，进而验证 $g_i(\theta)$ 的正确性：

$$g_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2 \times \text{EPSILON}}. \tag{30}$$

当用反射传播算法求解神经网络时，正确算法实现会得到：

$$\nabla_{W^{(l)}} J(W,b) = \left(\frac{1}{m}\Delta W^{(l)}\right) + \lambda W^{(l)} \tag{31}$$

$$\nabla_{b^{(l)}} J(W,b) = \frac{1}{m}\Delta b^{(l)}. \tag{32}$$

以上结果与反向传播算法 (2.2) 中的最后一段伪代码一致，都是计算梯度下降。为了验证梯度下降代码的正确性，使用上述数值检验方法计算 $J(W,b)$ 的导数，然后验证 $\left(\frac{1}{m}\Delta W^{(l)}\right) + \lambda W$ 与 $\frac{1}{m}\Delta b^{(l)}$ 是否能够给出正确的求导结果。

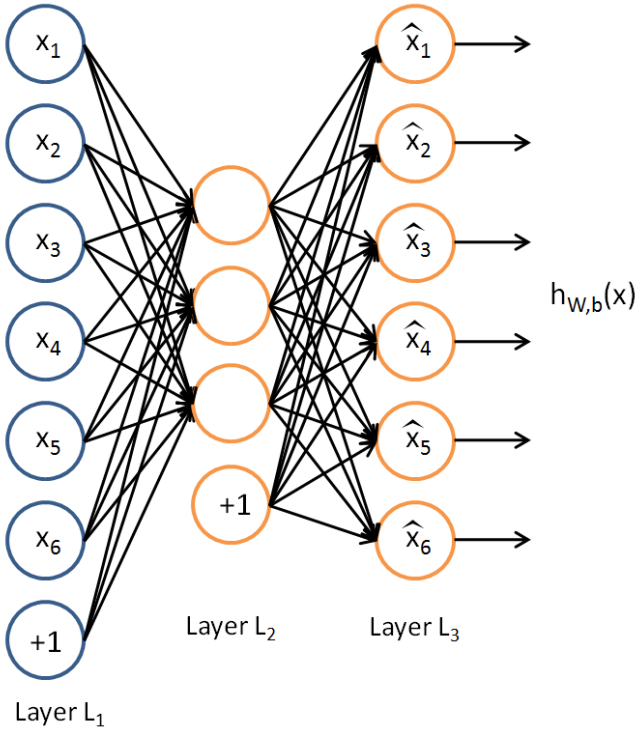
迄今为止，我们的讨论都集中在使用梯度下降法来最小化 $J(\theta)$。如果你已经实现了一个计算 $J(\theta)$ 和 $\nabla_\theta J(\theta)$ 的函数，那么其实还有更精妙的算法来最小化 $J(\theta)$。举例来说，可以想象这样一个算法：它使用梯度下降，并能够自动调整学习速率 α，以得到合适的步长值，最终使 θ 能够快速收敛到一个局部最优解。还有更妙的算法：比如可以寻找一个 Hessian 矩阵的近似，得到最佳步长值，使用该步长值能够更快地收敛到局部最优（和牛顿法类似）。此类算法的详细讨论已超出了这份讲义的范围，但是 L-BFGS 算法我们以后会有论述（另一个例子是共轭梯度算法）。你将在编程练习里使用这些算法中的一个。使用这些高级优化算法时，你需要提供关键的函数：即对于任一个 θ，需要你计算出 $J(\theta)$ 和 $\nabla_\theta J(\theta)$。之后，这些优化算法会自动调整学习速率/步长值 α 的大小（并计算 Hessian 近似矩阵等等）来自动寻找 $J(\theta)$ 最小化时 θ 的值。诸如 L-BFGS 和共轭梯度算法通常比梯度下降法快很多。

2.4 自编码算法与稀疏性

目前为止，我们已经讨论了神经网络在有监督学习中的应用。在有监督学习中，训练样本是有类别标签的。现在假设我们只有一个没有带类别标签的训练样本集合 $\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots\}$，其中 $x^{(i)} \in \Re^n$。自编码神经网络是一种无监督学习算法，它使用了反向传播算法，并让目标值等于输入值，比如 $y^{(i)} = x^{(i)}$。

下图是一个自编码神经网络的示例。

自编码神经网络尝试学习一个 $h_{W,b}(x) \approx x$ 的函数。换句话说，它尝试逼近一个恒等函数，从而使得输出 $\hat{x}$ 接近于输入 x。恒等函数虽然看上去不太有学习的意义，但是当我们为自编码神经网络加入某些限制，比如限定隐藏神经元的数量，我们就可以从输入数据中发现一些有趣的结构。举例来说，假设某个自编码神经网络的输入 x 是一张 10×10 图像（共 100 个像素）的像素灰度值，于是 $n = 100$，其隐藏层 L_2 中有 50 个隐藏神经元。注意，输出也是 100 维的 $y \in \Re^{100}$。由于只有 50 个隐藏神经元，我们迫使自编码神经网络去学习输入数据的压缩表示，也就是说，它必须从 50 维的隐藏神经元激活度向量 $a^{(2)} \in \Re^{50}$ 中重构出 100 维的像素灰度值输



入 x。如果网络的输入数据是完全随机的，比如每一个输入 x_i 都是一个跟其它特征完全无关的独立同分布高斯随机变量，那么这一压缩表示将会非常难学习。但是如果输入数据中隐含着一些特定的结构，比如某些输入特征是彼此相关的，那么这一算法就可以发现输入数据中的这些相关性。事实上，这一简单的自编码神经网络通常可以学习出一个跟主元分析（PCA）结果非常相似的输入数据的低维表示。

我们刚才的论述是基于隐藏神经元数量较小的假设。但是即使隐藏神经元的数量较大（可能比输入像素的个数还要多），我们仍然通过给自编码神经网络施加一些其他的限制条件来发现输入数据中的结构。具体来说，如果我们给隐藏神经元加入稀疏性限制，那么自编码神经网络即使在隐藏神经元数量较多的情况下仍然可以发现输入数据中一些有趣的结构。

稀疏性可以被简单地解释如下。如果当神经元的输出接近于 1 的时候我们认为它被激活，而输出接近于 0 的时候认为它被抑制，那么使得神经元大部分的时间都是被抑制的限制则被称作稀疏性限制。这里我们假设的神经元的激活函数是 sigmoid 函数。如果你使用 tanh 作为激活函数的话，当神经元输出为 -1 的时候，我们认为神经元是被抑制的。

注意到 $a_j^{(2)}$ 表示隐藏神经元 j 的激活度，但是这一表示方法中并未明确指出哪一个输入 x 带来了这一激活度。所以我们将使用 $a_j^{(2)}(x)$ 来表示在给定输入为 x 情况下，自编码神经网络隐藏神经元 j 的激活度。进一步，让

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} \left[a_j^{(2)}(x^{(i)}) \right] \tag{33}$$

表示隐藏神经元 j 的平均活跃度（在训练集上取平均）。我们可以近似的加入一条限制

$$\hat{\rho}_j = \rho, \tag{34}$$

其中，ρ 是稀疏性参数，通常是一个接近于 0 的较小的值（比如 $\rho = 0.05$）。换句话说，我们想要让隐藏神经元 j 的平均活跃度接近 0.05。为了满足这一条件，隐藏神经元的活跃度必须接近于 0。

为了实现这一限制，我们将会在我们的优化目标函数中加入一个额外的惩罚因子，而这一惩罚因子将惩罚那些 $\hat{\rho}_j$ 和 ρ 有显著不同的情况从而使得隐藏神经元的平均活跃度保持在较小范围内。惩罚因子的具体形式有很多种合理的选择，我们将会选择以下这一种：

$$\sum_{j=1}^{s_2} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}. \tag{35}$$

这里，s_2 是隐藏层中隐藏神经元的数量，而索引 j 依次代表隐藏层中的每一个神经元。如果你对相对熵（KL divergence）比较熟悉，这一惩罚因子实际上是基于它的。于是惩罚因子也可以被表示为

$$\sum_{j=1}^{s_2} \mathrm{KL}(\rho || \hat{\rho}_j), \tag{36}$$

其中 $\mathrm{KL}(\rho||\hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$ 是一个以 ρ 为均值和一个以 $\hat{\rho}_j$ 为均值的两个伯努利随机变量之间的相对熵。相对熵是一种标准的用来测量两个分布之间差异的方法。（如果你没有见过相对熵，不用担心，所有你需要知道的内容都会被包含在这份笔记之中。）

这一惩罚因子有如下性质，当 $\hat{\rho}_j = \rho$ 时 $\mathrm{KL}(\rho||\hat{\rho}_j) = 0$，并且随着 $\hat{\rho}_j$ 与 ρ 之间的差异增大而单调递增。举例来说，在下图中，我们设定 $\rho = 0.2$ 并且画出了相对熵值 $\mathrm{KL}(\rho||\hat{\rho}_j)$ 随着 $\hat{\rho}_j$ 变化的变化。

我们可以看出，相对熵在 $\hat{\rho}_j = \rho$ 时达到它的最小值 0，而当 $\hat{\rho}_j$ 靠近 0 或者 1 的时候，相对熵则变得非常大（其实是趋向于 ∞）。所以，最小化这一惩罚因子具有使得 $\hat{\rho}_j$ 靠近 ρ 的效果。

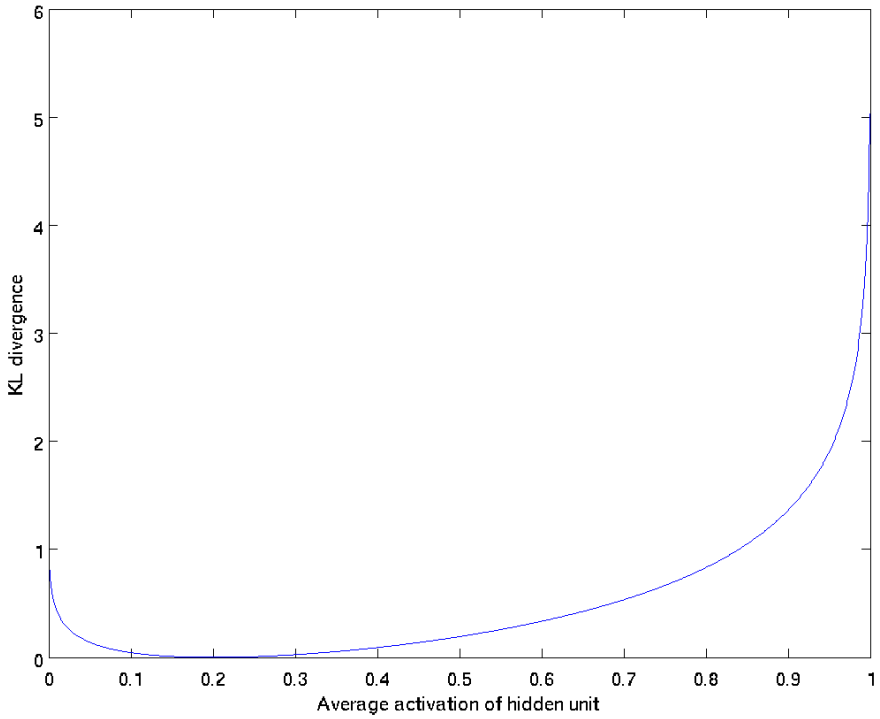
现在，我们的总体代价函数可以表示为

$$J_{\text{sparse}}(W,b) = J(W,b) + \beta \sum_{j=1}^{s_2} \mathrm{KL}(\rho||\hat{\rho}_j), \tag{37}$$

其中 $J(W,b)$ 如之前所定义，而 β 控制稀疏性惩罚因子的权重。$\hat{\rho}_j$ 项则也（间接地）取决于 W,b，因为它是隐藏神经元 j 的平均激活度，而隐藏层神经元的激活度取决于 W,b。

为了对相对熵进行导数计算，我们可以使用一个易于实现的技巧，这只需要在你的程序中稍作改动即可。具体来说，前面在后向传播算法中计算第二层（$l = 2$）更新的时候我们已经计算了

$$\delta_i^{(2)} = \left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) f'(z_i^{(2)}), \tag{38}$$



现在我们将其换成

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i} \right) \right) f'(z_i^{(2)}). \tag{39}$$

就可以了。

有一个需要注意的地方就是我们需要知道 $\hat{\rho}_i$ 来计算这一项更新。所以在计算任何神经元的后向传播之前，你需要对所有的训练样本计算一遍前向传播，从而获取平均激活度。如果你的训练样本可以小到被整个存到内存之中（对于编程作业来说，通常如此），你可以方便地在你所有的样本上计算前向传播并将得到的激活度存入内存并且计算平均激活度。然后你就可以使用事先计算好的激活度来对所有的训练样本进行后向传播的计算。如果你的数据量太大，无法全部存入内存，你就可以扫过你的训练样本并计算一次前向传播，然后将获得的结果累积起来并计算平均激活度 $\hat{\rho}_i$ （当某一个前向传播的结果中的激活度 $a_i^{(2)}$ 被用于计算平均激活度 $\hat{\rho}_i$ 之后就可以将此结果删除）。然后当你完成平均激活度 $\hat{\rho}_i$ 的计算之后，你需要重新对每一个训练样本做一次前向传播从而可以对其进行后向传播的计算。对于后一种情况，你对每一个训练样本需要计算两次前向传播，所以在计算上的效率会稍低一些。

证明上面算法能达到梯度下降效果的完整推导过程不再本教程的范围之内。不过如果你想要使用经过以上修改的后向传播来实现自编码神经网络，那么你就会对目标函数 $J_{\text{sparse}}(W, b)$ 做梯度下降。使用梯度验证方法，你可以自己来验证梯度下降算法是否正确。

2.5 可视化自编码器训练结果

训练完（稀疏）自编码器，我们还想把这自编码器学到的函数可视化出来，好弄明白它到底学到了什么。我们以在 10×10 图像（即 n=100）上训练自编码器为例。在该自编码器中，每个隐藏单元 i 对如下关于输入的函数进行计算：

$$a_i^{(2)} = f\left(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_i^{(1)}\right). \tag{40}$$

我们将要可视化的函数，就是上面这个以 2D 图像为输入、并由隐藏单元 i 计算出来的函数。它是依赖于参数 $W_{ij}^{(1)}$ 的（暂时忽略偏置项 b_i）。需要注意的是，$a_i^{(2)}$ 可看作输入 x 的非线性特征。不过还有个问题：什么样的输入图像 x 可让 $a_i^{(2)}$ 得到最大程度的激励?（通俗一点说，隐藏单元 i 要找个什么样的特征?）。这里我们必须给 x 加约束，否则会得到平凡解。若假设输入有范数约束 $||x||^2 = \sum_{i=1}^{100} x_i^2 \leq 1$，则可证（请读者自行推导）令隐藏单元 i 得到最大激励的输入应由下面公式计算的像素 x_j 给出（共需计算 100 个像素，$j = 1, \ldots, 100$）：

$$x_j = \frac{W_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}}. \tag{41}$$

当我们用上式算出各像素的值、把它们组成一幅图像、并将图像呈现在我们面前之时，隐藏单元 i 所追寻特征的真正含义也渐渐明朗起来。

假如我们训练的自编码器有 100 个隐藏单元，可视化结果就会包含 100 幅这样的图像——每个隐藏单元都对应一幅图像。审视这 100 幅图像，我们可以试着体会这些隐藏单元学出来的整体效果是什么样的。

当我们对稀疏自编码器进行上述可视化处理之后（100 个隐藏单元，在 10X10 像素的输入上训练），[5] 结果如下所示：

上图的每个小方块都给出了一个（带有有界范数的）输入图像 x，它可使这 100 个隐藏单元中的某一个获得最大激励。我们可以看到，不同的隐藏单元学会了在图像的不同位置和方向进行边缘检测。

显而易见，这些特征对物体识别等计算机视觉任务是十分有用的。若将其用于其他输入域（如音频），该算法也可学到对这些输入域有用的表示或特征。

2.6 稀疏自编码器符号一览表

下面是我们在推导 sparse autoencoder 时使用的符号一览表：

[5] The learned features were obtained by training on whitened natural images. Whitening is a preprocessing step which removes redundancy in the input, by causing adjacent pixels to become less correlated.

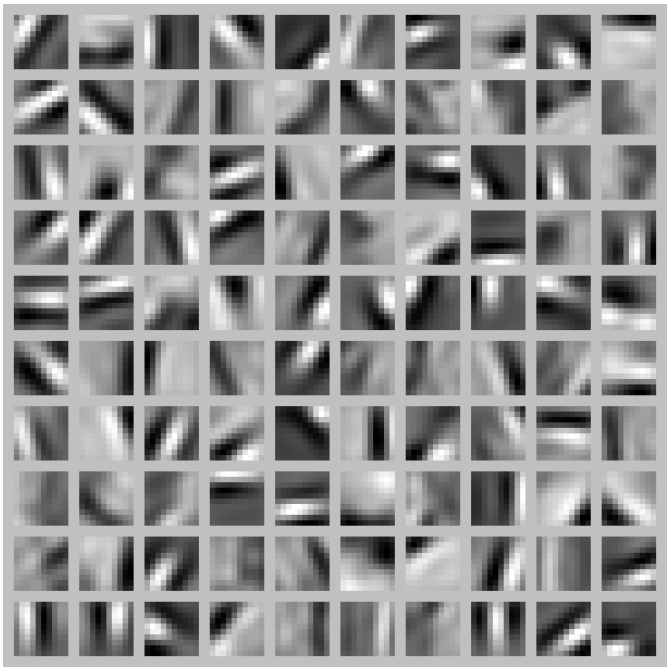


表 1: 稀疏自编码器符号一览表

符号	含义
x	训练样本的输入特征，$x \in \Re^n$.
y	输出值/目标值. 这里 y 可以是向量. 在 autoencoder 中，$y = x$.
$(x^{(i)}, y^{(i)})$	第 i 个训练样本
$h_{W,b}(x)$	输入为 x 时的假设输出，其中包含参数 W, b. 该输出应当与目标值 y 具有相同的维数.
$W_{ij}^{(l)}$	连接第 l 层 j 单元和第 $l+1$ 层 i 单元的参数.
$b_i^{(l)}$	第 $l+1$ 层 i 单元的偏置项. 也可以看作是连接第 l 层偏置单元和第 $l+1$ 层 i 单元的参数.
θ	参数向量. 可以认为该向量是通过将参数 W, b 组合展开为一个长的列向量而得到.
$a_i^{(l)}$	网络中第 l 层 i 单元的激活（输出）值. 另外，由于 L_1 层是输入层，所以 $a_i^{(1)} = x_i$.
$f(\cdot)$	激活函数. 本文中我们使用 $f(z) = \tanh(z)$.
$z_i^{(l)}$	第 l 层 i 单元所有输入的加权和. 因此有 $a_i^{(l)} = f(z_i^{(l)})$.
	续下页 ...

符号	含义
α	学习率
s_l	第 l 层的单元数目（不包含偏置单元）.
n_l	网络中的层数. 通常 L_1 层是输入层，L_{n_l} 层是输出层.
λ	权重衰减系数.
$\hat{x}$	对于一个 autoencoder，该符号表示其输出值；亦即输入值 x 的重构值. 与 $h_{W,b}(x)$ 含义相同.
ρ	稀疏值，可以用它指定我们所需的稀疏程度
$\hat{\rho}_i$	（sparse autoencoder 中）隐藏单元 i 的平均激活值.
β	（sparse autoencoder 目标函数中）稀疏值惩罚项的权重.

2.7 练习：稀疏自编码器

可以从 http://nlp.stanford.edu/~socherr/sparseAutoencoder_2011new.pdf 和 http://www.stanford.edu/class/cs294a/cs294a_2011-assignment.pdf 下载文档。

稀疏自编码器的实现

In this problem set, you will implement the sparse autoencoder algorithm, and show how it discovers that edges are a good representation for natural images. (Images provided by Bruno Olshausen.) The sparse autoencoder algorithm is described in the lecture notes found on the course website.

In the file http://ufldl.stanford.edu/wiki/resources/sparseae_exercise.zip , we have provided some starter code in Matlab. You should write your code at the places indicated in the files ("YOUR CODE HERE"). You have to complete the following files: sampleIMAGES.m, sparseAutoencoderCost.m, computeNumericalGradient.m. The starter code in train.m shows how these functions are used.

Specifically, in this exercise you will implement a sparse autoencoder, trained with 8×8 image patches using the L-BFGS optimization algorithm.

A note on the software: The provided .zip file includes a subdirectory minFunc with 3rd party software implementing L-BFGS, that is licensed under a Creative Commons, Attribute, Non-Commercial license. If you need to use this software for commercial purposes, you can download and use a different function (fminlbfgs) that can serve the same purpose, but runs $3\times$ slower for this exercise (and thus is less recommended). You can read more about this in the http:

//deeplearning.stanford.edu/wiki/index.php/Fminlbfgs_Details page.

2.7.1 第一步：生成训练集

The first step is to generate a training set. To get a single training example x, randomly pick one of the 10 images, then randomly sample an 8×8 image patch from the selected image, and convert the image patch (either in row-major order or column-major order; it doesn't matter) into a 64-dimensional vector to get a training example $x \in \Re^{64}$.

Complete the code in `sampleIMAGES.m`. Your code should sample 10000 image patches and concatenate them into a 64×10000 matrix.

To make sure your implementation is working, run the code in "Step 1" of train.m. This should result in a plot of a random sample of 200 patches from the dataset.

Implementational tip: When we run our implemented `sampleImages()`, it takes under 5 seconds. If your implementation takes over 30 seconds, it may be because you are accidentally making a copy of an entire 512×512 image each time you're picking a random image. By copying a 512×512 image 10000 times, this can make your implementation much less efficient. While this doesn't slow down your code significantly for this exercise (because we have only 10000 examples), when we scale to much larger problems later this quarter with 10^6 or more examples, this will significantly slow down your code. Please implement `sampleIMAGES` so that you aren't making a copy of an entire 512×512 image each time you need to cut out an 8×8 image patch.

2.7.2 第二步：稀疏自编码器

Implement code to compute the sparse autoencoder cost function $J_{sparse}(W, b)$ (Section 3 of the lecture notes) and the corresponding derivatives of J_{sparse} with respect to the different parameters. Use the sigmoid function for the activation function, $f(z) = \frac{1}{1+e^{-z}}$. In particular, complete the code in `sparseAutoencoderCost.m`.

The sparse autoencoder is parameterized by matrices $W^{(1)} \in \Re^{s_1 \times s_2}, W^{(2)} \in \Re^{s_2 \times s_3}$ vectors $b^{(1)} \in \Re^{s_2}, b^{(2)} \in \Re^{s_3}$. However, for subsequent notational convenience, we will "unroll" all of these parameters into a very long parameter vector θ with $s_1 s_2 + s_2 s_3 + s_2 + s_3$ elements. The code for converting between the $(W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)})$ and the θ parameterization is already provided in the starter code.

Implementational tip: The objective $J_{sparse}(W, b)$ contains 3 terms, corresponding to the squared error term, the weight decay term, and the sparsity penalty. You're welcome to implement this however you want, but for ease of debugging, you might implement the cost function and derivative computation (backpropagation) only for the squared error term first (this corresponds to setting $\lambda = \beta = 0$), and implement the gradient checking method in the next section to first verify that this code is correct. Then only after you have verified that the objective and derivative calculations corresponding to the squared error term are working, add in code to

compute the weight decay and sparsity penalty terms and their corresponding derivatives.

2.7.3 第三步：梯度检测

Following Section 2.3 of the lecture notes, implement code for gradient checking. Specifically, complete the code in `computeNumericalGradient.m`. Please use EPSILON $= 10^{-4}$ as described in the lecture notes.

We've also provided code in checkNumericalGradient.m for you to test your code. This code defines a simple quadratic function $h : \Re^2 \mapsto \Re$ given by $h(x) = x_1^2 + 3x_1x_2$, and evaluates it at the point $x = (4, 10)^T$. It allows you to verify that your numerically evaluated gradient is very close to the true (analytically computed) gradient.

After using checkNumericalGradient.m to make sure your implementation is correct, next use computeNumericalGradient.m to make sure that your sparseAutoencoderCost.m is computing derivatives correctly. For details, see Steps 3 in train.m. We strongly encourage you not to proceed to the next step until you've verified that your derivative computations are correct.

Implementational tip: If you are debugging your code, performing gradient checking on smaller models and smaller training sets (e.g., using only 10 training examples and 1-2 hidden units) may speed things up.

2.7.4 第四步：训练稀疏自编码器

Now that you have code that computes Jsparse and its derivatives, we're ready to minimize Jsparse with respect to its parameters, and thereby train our sparse autoencoder.

We will use the L-BFGS algorithm. This is provided to you in a function called minFunc (code provided by Mark Schmidt) included in the starter code. (For the purpose of this assignment, you only need to call minFunc with the default parameters. You do not need to know how L-BFGS works.) We have already provided code in train.m (Step 4) to call minFunc. The minFunc code assumes that the parameters to be optimized are a long parameter vector; so we will use the "θ" parameterization rather than the "$(W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)})$" parameterization when passing our parameters to it.

Train a sparse autoencoder with 64 input units, 25 hidden units, and 64 output units. In our starter code, we have provided a function for initializing the parameters. We initialize the biases $b_i^{(l)}$ to zero, and the weights $W_{ij}^{(l)}$ to random numbers drawn uniformly from the interval $\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}} + 1}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}} + 1}}\right]$, where n_{in} is the fan-in (the number of inputs feeding into a node) and n_{out} is the fan-in (the number of units that a node feeds into).

The values we provided for the various parameters (λ, β, ρ, etc.) should work, but feel free to play with different settings of the parameters as well.

Implementational tip: Once you have your backpropagation implementation correctly computing the derivatives (as verified using gradient checking in Step 3), when you are now using

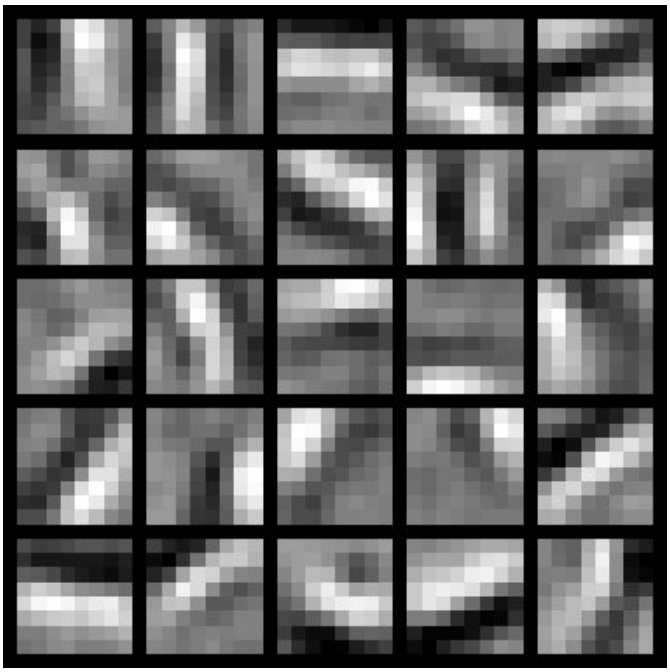
it with L-BFGS to optimize $J_{sparse}(W, b)$, make sure you're not doing gradient-checking on every step. Backpropagation can be used to compute the derivatives of Jsparse(W,b) fairly efficiently, and if you were additionally computing the gradient numerically on every step, this would slow down your program significantly.

2.7.5 第五步：可视化

After training the autoencoder, use `display_network.m` to visualize the learned weights. (See `train.m`, Step 5.) Run "`print -djpeg weights.jpg`" to save the visualization to a file "`weights.jpg`" (which you will submit together with your code).

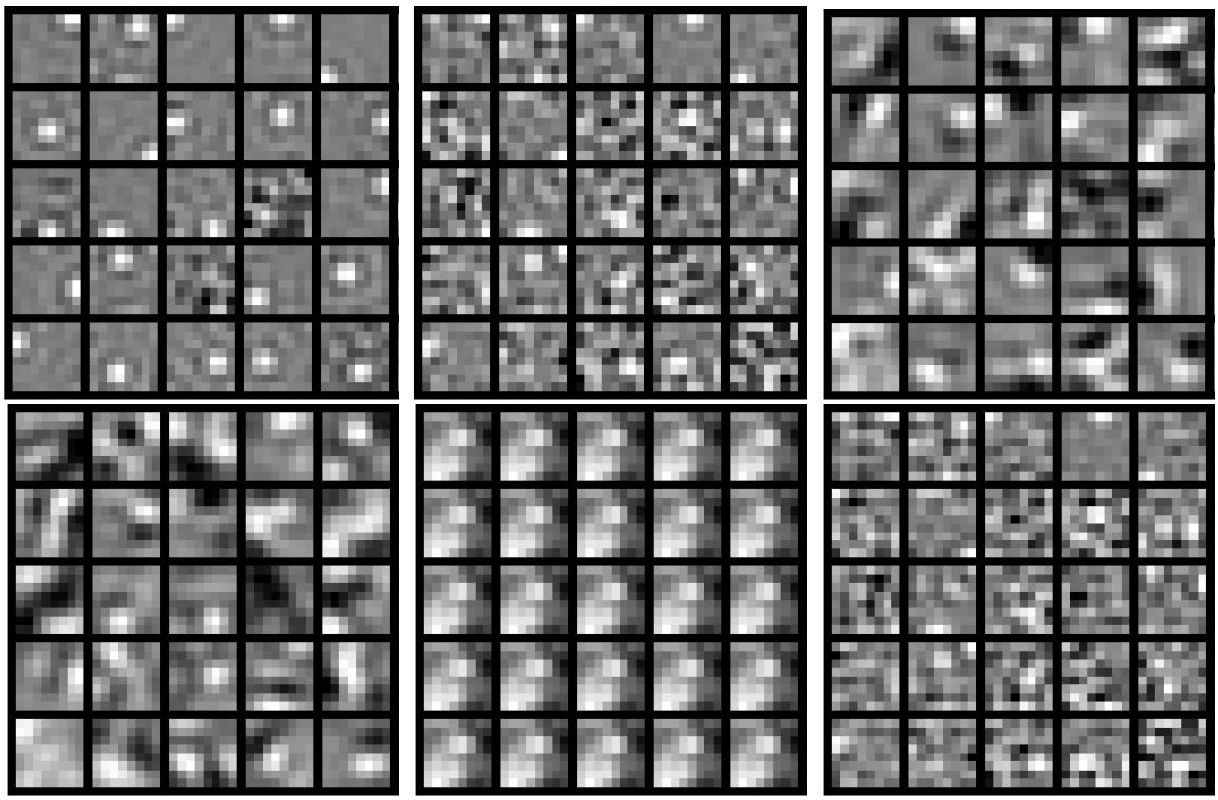
结果

To successfully complete this assignment, you should demonstrate your sparse autoencoder algorithm learning a set of edge detectors. For example, this was the visualization we obtained:



Our implementation took around 5 minutes to run on a fast computer. In case you end up needing to try out multiple implementations or different parameter values, be sure to budget enough time for debugging and to run the experiments you'll need.

Also, by way of comparison, here are some visualizations from implementations that we do not consider successful (either a buggy implementation, or where the parameters were poorly tuned):



3 矢量化编程实现

3.1 矢量化编程

当使用学习算法时，一段更快的代码通常意味着项目进展更快。例如，如果你的学习算法需要花费 20 分钟运行完成，这意味着你每个小时能“尝试”3 个新主意。但是假如你的程序需要 20 个小时来运行，这意味着你一天只能“尝试”一个新主意，因为你需要花费这么长时间来等待程序的反馈。对于后者，假如你可以提升代码的效率让其只需要运行 10 个小时，那么你的效率差不多提升一倍。

矢量化编程是提高算法速度的一种有效方法。为了提升特定数值运算操作（如矩阵相乘、矩阵相加、矩阵 -向量乘法等）的速度，数值计算和并行计算的研究人员已经努力了几十年。矢量化编程的思想就是尽量使用这些被高度优化的数值运算操作来实现我们的学习算法。

例如，假设 $x \in \Re^{n+1}$ 和 $\theta \in \Re^{n+1}$ 为向量，需要计算 $z = \theta^T x$，那么可以按以下方式实现（使用 Matlab）：

```
z = 0;
for i=1:(n+1),
  z = z + theta(i) * x(i);
end;
```

或者可以更加简单的写为：

```
z = theta' * x;
```

第二段程序代码不仅简单，而且**运行速度更快**。

通常，一个编写 Matlab/Octave 程序的诀窍是：

代码中尽可能避免显式的 for 循环。

上面的第一段代码使用了一个显式的 `for` 循环。通过不使用 `for` 循环实现相同功能，可以显著提升运行速度。对 Matlab/Octave 代码进行矢量化的工作很大一部分集中在避免使用 `for` 循环上，因为这可以使得 Matlab/Octave 更多地利用代码中的并行性，同时其解释器的计算开销更小。

关于编写代码的策略，开始时你会觉得矢量化代码更难编写、阅读和调试，但你需要在编码和调试的便捷性与运行时间之间做个权衡。因此，刚开始编写程序的时候，你可能会选择不使用太多矢量化技巧来实现你的算法，并验证它是否正确（可能只在一个小问题上验证）。在确定它正确后，你可以每次只矢量化一小段代码，并在这段代码之后暂停，以验证矢量化后的代码计算结果和之前是否相同。最后，你会有望得到一份正确的、经过调试的、矢量化且有效率的代码。

一旦对矢量化常见的方法和技巧熟悉后，你将会发现对代码进行矢量化通常并不太费劲。矢量化可以使你的代码运行的更快，而且在某些情况下，还简化了你的代码。

3.2 逻辑回归的向量化实现样例

我们想用批量梯度上升法对 logistic 回归分析模型进行训练，其模型如下：

$$h_\theta(x) = \frac{1}{1+\exp(-\theta^T x)}, \tag{42}$$

让我们遵从公开课程视频与 CS229 教学讲义的符号规范，设 $x_0 = 1$，于是 $x \in \Re^{n+1}$，$\theta \in \Re^{n+1}$，θ_0 为截距。假设我们有 m 个训练样本 $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$，而批量梯度上升法的更新法则是：$\theta := \theta + \alpha \nabla_\theta \ell(\theta)$, 这里的 $\ell(\theta)$ 是对数似然函数，$\nabla_\theta \ell(\theta)$ 是其导函数。

[注：下文的符号规范与 < 公开课程视频 > 或 < 教学讲义 CS229：机器学习 > 中的相同，详细内容可以参见公开课程视频或教学讲义 #1 http://cs229.stanford.edu/]

于是，我们需要如下计算梯度：

$$\nabla_\theta \ell(\theta) = \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}. \tag{43}$$

我们用 Matlab/Octave 风格变量 `x` 表示输入数据构成的样本矩阵，`x(:,i)` 代表第 i 个训练样本 $x^{(i)}$，`x(j,i)` 就代表 $x_j^{(i)}$ [6]。同样，用 Matlab/Octave 风格变量 `y` 表示由训练样本集合的全体类别标号所构成的行向量，则该向量的第 i 个元素 `y(i)` 就代表上式中的 $y^{(i)} \in \{0,1\}$。（注意这里跟公开课程视频及 CS229 的符号规范不同，矩阵 x 按列而不是按行存放输入训练样本，同样，`y`$\in R^{1\times m}$ 是行向量而不是列向量。）

以下是梯度运算代码的一种实现，非常恐怖，速度极慢：

```
% Implementation 1
grad = zeros(n+1,1);
for i=1:m,
  h = sigmoid(theta'*x(:,i));
  temp = y(i) - h;
  for j=1:n+1,
    grad(j) = grad(j) + temp * x(j,i);
  end;
end;
```

嵌套的 `for` 循环语句使这段代码的运行非常缓慢。以下是更典型的实现方式，它对算法进行部分向量化，带来更优的执行效率：

```
% Implementation 2
grad = zeros(n+1,1);
for i=1:m,
  grad = grad + (y(i) - sigmoid(theta'*x(:,i)))* x(:,i);
end;
```

[6] 译者注：第 i 个训练样本向量的第 j 个元素

但是，或许可以向量化得更彻底些。如果去除 `for` 循环，我们就可以显著地改善代码执行效率。特别的，假定 `b` 是一个列向量，`A` 是一个矩阵，我们用以下两种方式来计算 `A*b`:

```
% Slow implementation of matrix-vector multiply
grad = zeros(n+1,1);
for i=1:m,
  grad = grad + b(i) * A(:,i);  % more commonly written A(:,i)*b(i)
end;

% Fast implementation of matrix-vector multiply
grad = A*b;
```

我们看到，代码 2 是用了低效的 `for` 循环语句执行梯度上升[7]运算，将 `b(i)` 看成 `(y(i) - sigmoid(theta'*x(:,i)))`，`A` 看成 `x`，我们就可以使用以下高效率的代码:

```
% Implementation 3
grad = x * (y- sigmoid(theta'*x))';
```

这里我们假定 Matlab/Octave 的 `sigmoid(z)` 函数接受一个向量形式的输入 `z`，依次对输入向量的每个元素施行 `sigmoid` 函数，最后返回运算结果，因此 `sigmoid(z)` 的输出结果是一个与 `z` 有相同维度的向量。

当训练数据集很大时，最终的实现[8]充分发挥了 Matlab/Octave 高度优化的数值线性代数库的优势来进行矩阵 -向量操作，因此，比起之前代码要高效得多。

想采用向量化实现并非易事，通常需要周密的思考。但当你熟练掌握向量化操作后，你会发现，这里面有固定的设计模式（对应少量的向量化技巧），可以灵活运用到很多不同的代码片段中。

3.3 神经网络向量化

在本节，我们将引入神经网络的向量化版本。在前面关于神经网络 (2.1) 介绍的章节中，我们已经给出了一个部分向量化的实现，它在一次输入一个训练样本时是非常有效率的。下边我们看看如何实现同时处理多个训练样本的算法。具体来讲，我们将把正向传播、反向传播这两个步骤以及稀疏特征集学习扩展为多训练样本版本。

[7]译者注：原文是下降
[8]译者注：代码 3

3.3.1 正向传播

考虑一个三层网络 (一个输入层、一个隐含层、以及一个输出层)，并且假定 x 是包含一个单一训练样本 $x^{(i)} \in \Re^n$ 的列向量。则向量化的正向传播步骤如下：

$$z^{(2)} = W^{(1)}x + b^{(1)} \tag{44}$$

$$a^{(2)} = f(z^{(2)}) \tag{45}$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \tag{46}$$

$$h_{W,b}(x) = a^{(3)} = f(z^{(3)}) \tag{47}$$

This is a fairly efficient implementation for a single example. If we have m examples, then we would wrap a `for` loop around this.

更具体点来说，参照逻辑回归向量化的例子 (3.2)，我们用 Matlab/Octave 风格变量 x 表示包含输入训练样本的矩阵，`x(:,i)` 代表第 i 个训练样本。则正向传播步骤可如下实现：

```
% Unvectorized implementation
for i=1:m,
  z2 = W1 * x(:,i) + b1;
  a2 = f(z2);
  z3 = W2 * a2 + b2;
  h(:,i) = f(z3);
end;
```

这个 `for` 循环能否去掉呢? 对于很多算法而言，我们使用向量来表示计算过程中的中间结果。例如在前面的非向量化实现中，`z2,a2,z3` 都是列向量，分别用来计算隐层和输出层的激励结果。为了充分利用并行化和高效矩阵运算的优势，我们希望算法能同时处理多个训练样本。让我们先暂时忽略前面公式中的 `b1` 和 `b2`(把它们设置为 0)，那么可以实现如下：

```
% Vectorized implementation (ignoring b1, b2)
z2 = W1 * x;
a2 = f(z2);
z3 = W2 * a2;
h = f(z3)
```

在这个实现中，`z2,a2,z3` 都是矩阵，每个训练样本对应矩阵的一列。在对多个训练样本实现向量化时常用的设计模式是，虽然前面每个样本对应一个列向量（比如 `z2`），但我们可把这些列向量堆叠成一个矩阵以充分享受矩阵运算带来的好处。这样，在这个例子中，`a2` 就成了一个 $s_2 \times m$ 的矩阵 (s_2 是网络第二层中的神经元数，m 是训练样本个数)。矩阵 `a2` 的物理含义是，当第 i 个训练样本 `x(:i)` 输入到网络中时，它的第 i 列就表示这个输入信号对隐神经元 (网络第二层) 的激励结果。

在上面的实现中，我们假定激活函数 `f(z)` 接受矩阵形式的输入 `z`，并对输入矩阵按列分

别施以激活函数。需要注意的是，你在实现 `f(z)` 的时候要尽量多用 Matlab/Octave 的矩阵操作，并尽量避免使用 `for` 循环。假定激活函数采用 Sigmoid 函数，则实现代码如下所示:

```
% Inefficient, unvectorized implementation of the activation function
function output = unvectorized_f(z)
output = zeros(size(z))
for i=1:size(z,1),
  for j=1:size(z,2),
    output(i,j) = 1/(1+exp(-z(i,j)));
  end;
end;
end

% Efficient, vectorized implementation of the activation function
function output = vectorized_f(z)
output = 1./(1+exp(-z));     % "./" is Matlab/Octave's element-wise division
   operator.
end
```

最后，我们上面的正向传播向量化实现中忽略了 `b1` 和 `b2`，现在要把他们包含进来，为此我们需要用到 Matlab/Octave 的内建函数 `repmat`:

```
% Vectorized implementation of forward propagation
z2 = W1 * x + repmat(b1,1,m);
a2 = f(z2);
z3 = W2 * a2 + repmat(b2,1,m);
h = f(z3)
```

`repmat(b1,1,m)` 的运算效果是，它把列向量 `b1` 拷贝 m 份，然后堆叠成如下矩阵:

$$\begin{bmatrix} | & | & & | \\ \texttt{b1} & \texttt{b1} & \cdots & \texttt{b1} \\ | & | & & | \end{bmatrix}.$$

这就构成一个 $s_2 \times m$ 的矩阵。它和 `W1 * x` 相加，就等于是把 `W1 * x` 矩阵[9]的每一列加上 `b1`。如果不熟悉的话，可以参考 Matlab/Octave 的帮助文档获取更多信息 (输入 "`help repmat`")。`rampat` 作为 Matlab/Octave 的内建函数，运行起来是相当高效的，远远快过我们自己用 `for` 循环实现的效果。

3.3.2 反向传播

现在我们来描述反向传播向量化的思路。在阅读这一节之前，强烈建议各位仔细阅读前面介绍的正向传播的例子代码，确保你已经完全理解。下边我们只会给出反向传播向量化实现的大致纲要，而由你来完成具体细节的推导（见向量化练习 3.4）。

[9]译者注：这里 x 是训练矩阵而非向量, 所以 `W1 * x` 代表两个矩阵相乘，结果还是一个矩阵

对于监督学习，我们有一个包含 m 个带类别标号样本的训练集 $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$。(对于自编码网络，我们只需令 $y^{(i)} = x^{(i)}$ 即可，但这里考虑的是更一般的情况。)

假定网络的输出有 s_3 维，因而每个样本的类别标号向量就记为 $y^{(i)} \in \Re^{s_3}$。在我们的 Matlab/Octave 数据结构实现中，把这些输出按列合在一起形成一个 Matlab/Octave 风格变量 `y`，其中第 i 列 `y(:,i)` 就是 $y^{(i)}$。

现在我们要计算梯度项 $\nabla_{W^{(l)}} J(W,b)$ 和 $\nabla_{b^{(l)}} J(W,b)$。对于梯度中的第一项，就像过去在反向传播算法 (2.2) 中所描述的那样，对于每个训练样本 (x,y)，我们可以这样来计算：

$$\delta^{(3)} = -(y - a^{(3)}) \bullet f'(z^{(3)}), \tag{48}$$

$$\delta^{(2)} = ((W^{(2)})^T \delta^{(3)}) \bullet f'(z^{(2)}), \tag{49}$$

$$\nabla_{W^{(2)}} J(W,b;x,y) = \delta^{(3)} (a^{(2)})^T, \tag{50}$$

$$\nabla_{W^{(1)}} J(W,b;x,y) = \delta^{(2)} (a^{(1)})^T. \tag{51}$$

在这里 $\bullet$ 表示对两个向量按对应元素相乘的运算[10]。为了描述简单起见，我们这里暂时忽略对参数 $b^{(l)}$ 的求导，不过在你真正实现反向传播时，还是需要计算关于它们的导数的。

假定我们已经实现了向量化的正向传播方法，如前面那样计算了矩阵形式的变量 `z2`, `a2`, `z3` 和 `h`，那么反向传播的非向量化版本可如下实现：

```
gradW1 = zeros(size(W1));
gradW2 = zeros(size(W2));
for i=1:m,
  delta3 = -(y(:,i) - h(:,i)) .* fprime(z3(:,i));
  delta2 = W2'*delta3(:,i) .* fprime(z2(:,i));

  gradW2 = gradW2 + delta3*a2(:,i)';
  gradW1 = gradW1 + delta2*a1(:,i)';
end;
```

在这个实现中，有一个 `for` 循环。而我们想要一个能同时处理所有样本、且去除这个 `for` 循环的向量化版本。

为做到这一点，我们先把向量 `delta3` 和 `delta2` 替换为矩阵，其中每列对应一个训练样本。我们还要实现一个函数 `fprime(z)`，该函数接受矩阵形式的输入 `z`，并且对矩阵的按元素分别执行 $f'(\cdot)$。这样，上面 `for` 循环中的 4 行 Matlab 代码中每行都可单独向量化，以一行新的（向量化的）Matlab 代码替换它（不再需要外层的 `for` 循环）。

在向量化练习 (3.4) 中，我们要求你自己去推导出这个算法的向量化版本。如果你已经能从上面的描述中了解如何去做，那么我们强烈建议你去实践一下。虽然我们已经为你准备了反向传播的向量化实现提示 (3.3)，但还是鼓励你在不看提示的情况下自己去推导一下。

[10] 译者注：其结果还是一个向量

3.3.3 稀疏自编码网络

稀疏自编码网络中包含一个额外的稀疏惩罚项，目的是限制神经元的平均激活率，使其接近某个（预设的）目标激活率 ρ。其实在对单个训练样本上执行反向传播时，我们已经考虑了如何计算这个稀疏惩罚项，如下所示：

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_2} W_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i} \right) \right) f'(z_i^{(2)}). \tag{52}$$

在**非向量化**的实现中，计算代码如下：

```
% Sparsity Penalty Delta
sparsity_delta = - rho ./ rho_hat + (1 - rho) ./ (1 - rho_hat);
for i=1:m,
  ...
  delta2 = (W2'*delta3(:,i) + beta*sparsity_delta).* fprime(z2(:,i));
  ...
end;
```

但在上面的代码中，仍旧含有一个需要在整个训练集上运行的 `for` 循环，这里 `delta2` 是一个列向量。

作为对照，回想一下在向量化的情况下，`delta2` 现在应该是一个有 m 列的矩阵，分别对应着 m 个训练样本。还要注意，稀疏惩罚项 `sparsity_delta` 对所有的训练样本一视同仁。这意味着要向量化实现上面的计算，只需在构造 `delta2` 时，往矩阵的每一列上分别加上相同的值即可。因此，要向量化上面的代码，我们只需简单的用 `repmat` 命令把 `sparsity_delta` 加到 `delta2` 的每一列上即可[11]。

3.4 练习：矢量化

In the previous problem set, we implemented a sparse autoencoder for patches taken from natural images. In this problem set, you will vectorize your code to make it run much faster, and further adapt your sparse autoencoder to work on images of handwritten digits. Your network for learning from handwritten digits will be much larger than the one you'd trained on the natural images, and so using the original implementation would have been painfully slow. But with a vectorized implementation of the autoencoder, you will be able to get this to run in a reasonable amount of computation time.

3.4.1 Support Code/Data

The following additional files are required for this exercise:

[11] 译者注：这里原文描述得不是很清楚，看似应加到上面代码中 `delta2` 行等号右边第一项，即 `W2' * delta3` 上

- MNIST Dataset (Training Images) http://yann.lecun.com/exdb/mnist/train-images-idx
  gz
- MNIST Dataset (Training Labels) http://yann.lecun.com/exdb/mnist/train-labels-idx
  gz
- Support functions for loading MNIST in Matlab A

3.4.2 Step 1: Vectorize your Sparse Autoencoder Implementation

Using the ideas from Vectorization(3.1) and Neural Network Vectorization(3.3), vectorize your implementation of `sparseAutoencoderCost.m`. In our implementation, we were able to remove all for-loops with the use of matrix operations and repmat. (If you want to play with more advanced vectorization ideas, also type `help bsxfun`. The `bsxfun` function provides an alternative to `repmat` for some of the vectorization steps, but is not necessary for this exercise). A vectorized version of our sparse autoencoder code ran in under one minute on a fast computer (for learning 25 features from 10000 8×8 image patches).

(Note that you do not need to vectorize the code in the other files.)

3.4.3 Step 2: Learn features for handwritten digits

Now that you have vectorized the code, it is easy to learn larger sets of features on medium sized images. In this part of the exercise, you will use your sparse autoencoder to learn features for handwritten digits from the MNIST dataset.

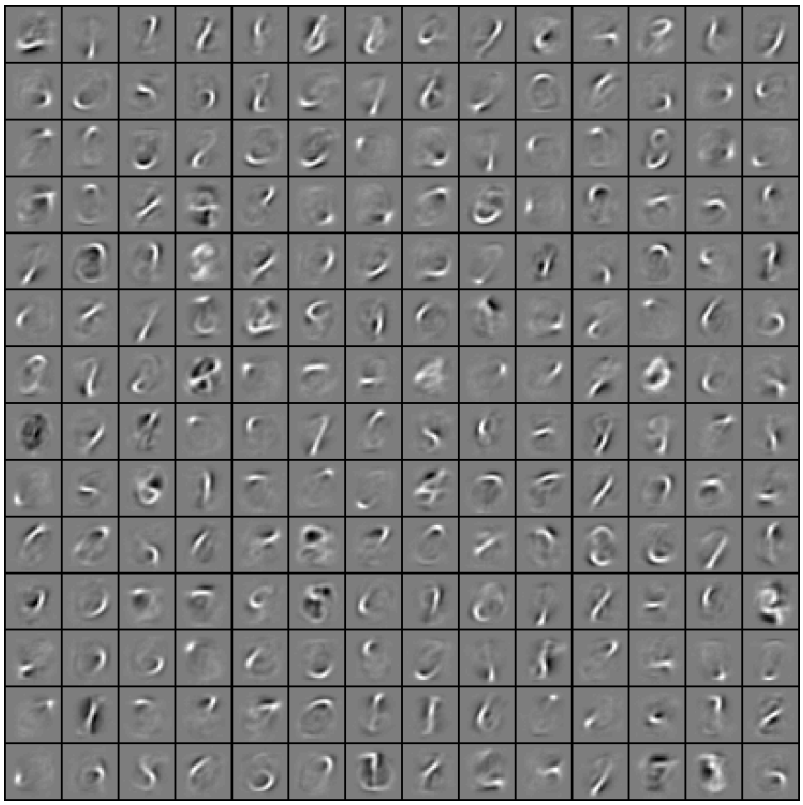
The MNIST data is available at http://yann.lecun.com/exdb/mnist/. Download the file train-images-idx3-ubyte.gz and decompress it. After obtaining the source images, you should use helper functions that we provide to load the data into Matlab as matrices. While the helper functions that we provide(A) will load both the input examples x and the class labels y, for this assignment, you will only need the input examples x since the sparse autoencoder is an unsupervised learning algorithm. (In a later assignment, we will use the labels y as well.)

The following set of parameters worked well for us to learn good features on the MNIST dataset:

```
visibleSize = 28*28
hiddenSize = 196
sparsityParam = 0.1
lambda = 3e-3
beta = 3
patches = first 10000 images from the MNIST dataset
```

After 400 iterations of updates using minFunc, your autoencoder should have learned features that resemble pen strokes. In other words, this has learned to represent handwritten characters in

terms of what pen strokes appear in an image. Our implementation takes around 15-20 minutes on a fast machine. Visualized, the features should look like the following image:



If your parameters are improperly tuned, or if your implementation of the autoencoder is buggy, you may get one of the following images instead:

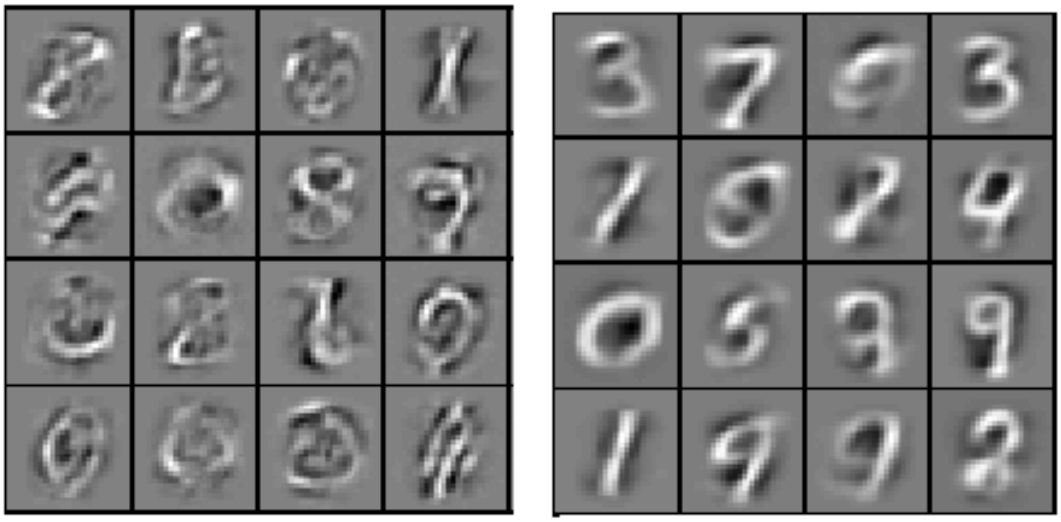
3.5 反向传播矢量化提示

Here, we give a few hints on how to vectorize the Backpropagation step. The hints here specifically build on our earlier description of how to vectorize a neural network(3.3).

Assume we have already implemented the vectorized forward propagation steps, so that the matrix-valued `z2`, `a2`, `z3` and `h` have already been computed. Here was our unvectorized implementation of `backprop`:

```
gradW1 = zeros(size(W1));
gradW2 = zeros(size(W2));
for i=1:m,
  delta3 = -(y(:,i) - h(:,i)) .* fprime(z3(:,i));
  delta2 = W2'*delta3(:,i) .* fprime(z2(:,i));

  gradW2 = gradW2 + delta3*a2(:,i)';
  gradW1 = gradW1 + delta2*a1(:,i)';
end;
```



Assume that we have implemented a version of `fprime(z)` that accepts matrix-valued inputs. We will use matrix-valued `delta3, delta2`. Here, `delta3` and `delta2` will have m columns, with one column per training example. We want to compute `delta3, delta2, gradW2` and `gradW1`.

Consider the computation for the matrix `delta3`, which can now be written:

```
for i=1:m,
  delta3(:,i) = -(y(:,i) - h(:,i)) .* fprime(z3(:,i));
end;
```

Each iteration of the for loop computes one column of `delta3`. You should be able to find a single line of Matlab to compute `delta3` as a function of the matrices `y, h` and `z3`. This lets you compute `delta3`. Similarly, you should also be able to find a single line of code to compute the entire matrix `delta2`, as a function of `W2, delta3` (which is now a matrix), and `z2`.

Next, consider the computation for `gradW2`. We can now write this as:

```
gradW2 = zeros(size(W2));
for i=1:m,
  gradW2 = gradW2 + delta3(:,i)*a2(:,i)';
end;
```

You should be able to find a single line of Matlab that replaces this for loop, and computes `gradW2` as a function of the matrices `delta3` and `a2`. If you're having trouble, take another look at the Logistic Regression Vectorization Example(3.2), which uses a related (but slightly different) vectorization step to get to the final implementation. Using a similar method, you will also be able to compute `gradW1` with a single line of code.

When you complete the derivation, you should be able to replace the unvectorized backpropagation code example above with just 4 lines of Matlab/Octave code.

4 预处理：主成分分析与白化

4.1 主成分分析

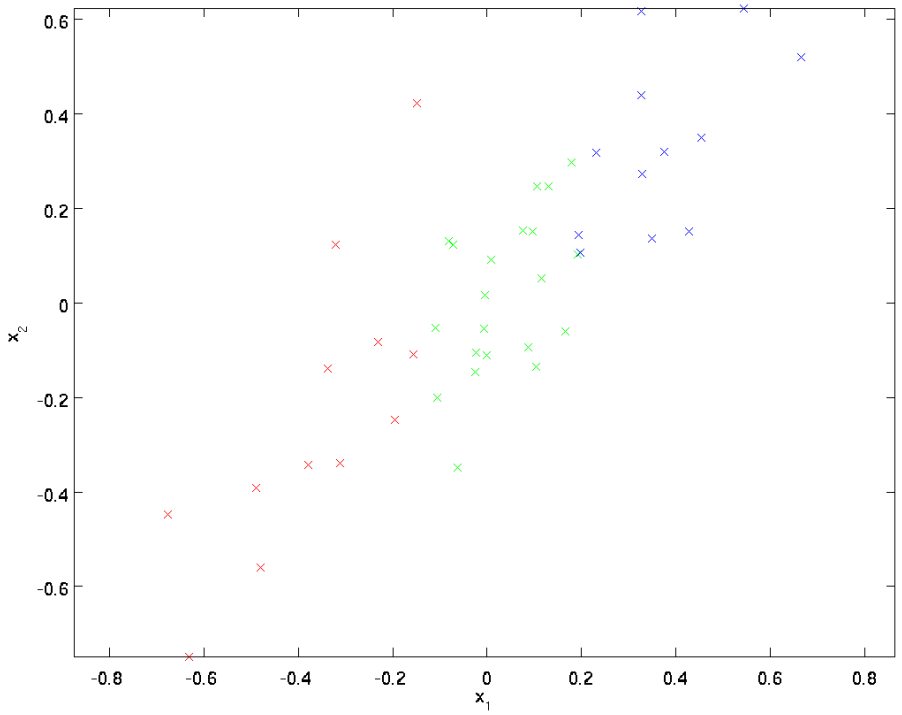
4.1.1 引言

主成分分析（PCA）是一种能够极大提升无监督特征学习速度的数据降维算法。更重要的是，理解 PCA 算法，对实现白化算法有很大的帮助，很多算法都先用白化算法作预处理步骤。

假设你使用图像来训练算法，因为图像中相邻的像素高度相关，输入数据是有一定冗余的。具体来说，假如我们正在训练的 16×16 灰度值图像，记为一个 256 维向量 $x \in \Re^{256}$，其中特征值 x_j 对应每个像素的亮度值。由于相邻像素间的相关性，PCA 算法可以将输入向量转换为一个维数低很多的近似向量，而且误差非常小。

4.1.2 实例和数学背景

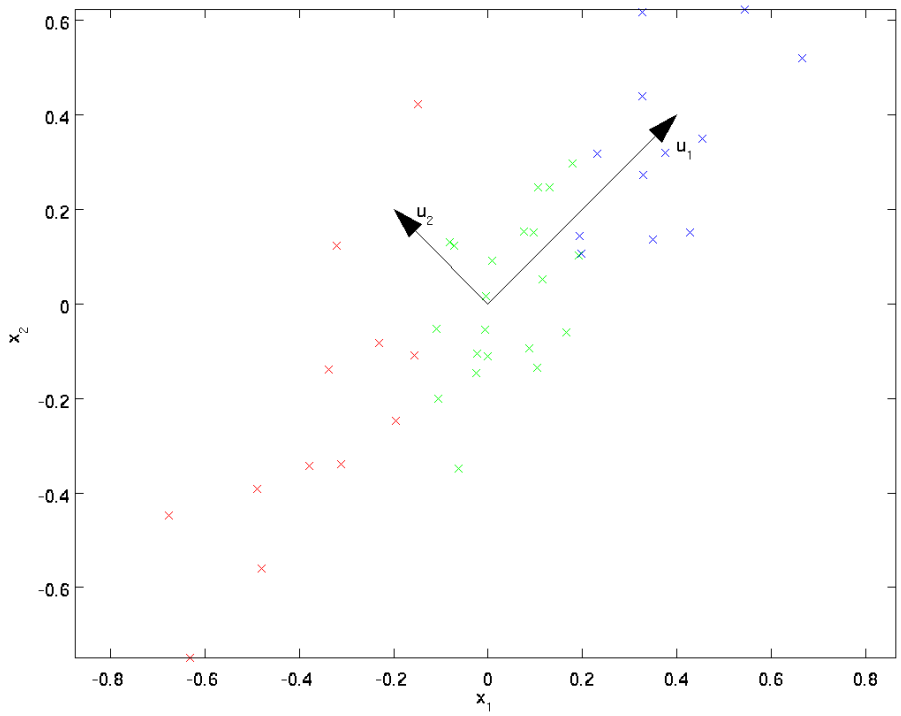
在我们的实例中，使用的输入数据集表示为 $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$，维度 $n = 2$ 即 $x^{(i)} \in \Re^2$。假设我们想把数据从 2 维降到 1 维。（在实际应用中，我们也许需要把数据从 256 维降到 50 维；在这里使用低维数据，主要是为了更好地可视化算法的行为）。下图是我们的数据集：



这些数据已经进行了预处理，使得每个特征 x_1 和 x_2 具有相同的均值（零）和方差。

为方便展示，根据 x_1 值的大小，我们将每个点分别涂上了三种颜色之一，但该颜色并不用于算法而仅用于图解。

PCA 算法将寻找一个低维空间来投影我们的数据。从下图中可以看出，u_1 是数据变化的主方向，而 u_2 是次方向。



也就是说，数据在 u_1 方向上的变化要比在 u_2 方向上大。为更形式化地找出方向 u_1 和 u_2，我们首先计算出矩阵 Σ，如下所示：

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})(x^{(i)})^T. \tag{53}$$

假设 x 的均值为零，那么 Σ 就是 x 的协方差矩阵。（符号 Σ，读 “Sigma"，是协方差矩阵的标准符号。虽然看起来与求和符号 $\sum_{i=1}^{n} i$ 比较像，但它们其实是两个不同的概念。）

可以证明，数据变化的主方向 u_1 就是协方差矩阵 Σ 的主特征向量，而 u_2 是次特征向量。

注：如果你对如何得到这个结果的具体数学推导过程感兴趣，可以参看 CS229（机器学习）PCA 部分的课件（链接在本页底部）。但如果仅仅是想跟上本课，可以不必如此。

你可以通过标准的数值线性代数运算软件求得特征向量（见实现说明）. 我们先计算出协方差矩阵 Σ 的特征向量，按列排放，而组成矩阵 U：

$$U = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_n \\ | & | & & | \end{bmatrix} \tag{54}$$

此处，u_1 是主特征向量(对应最大的特征值)，u_2 是次特征向量。以此类推，另记 $\lambda_1, \lambda_2, \ldots, \lambda_n$ 为相应的特征值。

在本例中，向量 u_1 和 u_2 构成了一个新基，可以用来表示数据。令 $x \in \Re^2$ 为训练样本，那么 $u_1^T x$ 就是样本点 x 在维度 u_1 上的投影的长度（幅值）。

同样的，$u_2^T x$ 是 x 投影到 u_2 维度上的幅值。

4.1.3 旋转数据

至此，我们可以把 x 用 (u_1, u_2) 基表达为：

$$x_{\text{rot}} = U^T x = \begin{bmatrix} u_1^T x \\ u_2^T x \end{bmatrix} \tag{55}$$

（下标“rot”来源于单词“rotation”，意指这是原数据经过旋转（也可以说成映射）后得到的结果）对数据集中的每个样本 i 分别进行旋转：$x_{\text{rot}}^{(i)} = U^T x^{(i)}$ for every i，然后把变换后的数据 x_{rot} 显示在坐标图上，可得：

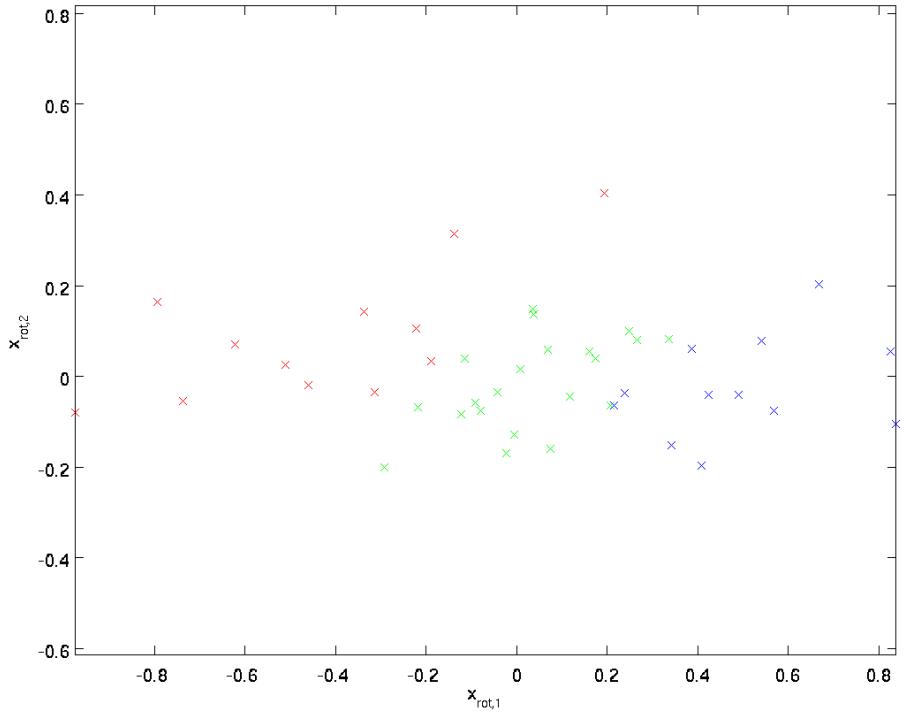


图 2:

这就是把训练数据集旋转到 u_1,u_2 基后的结果。一般而言，运算 $U^T x$ 表示旋转到基 $u_1, u_2, \ldots, u_n$ 之上的训练数据。

矩阵 U 有正交性，即满足 $U^TU = UU^T = I$，所以若想将旋转后的向量 x_{rot} 还原为原始数据 x，将其左乘矩阵 U 即可：$x = Ux_{\text{rot}}$, 验算一下：$Ux_{\text{rot}} = UU^Tx = x$.

$$x = Ux_{\text{rot}}, \tag{56}$$

because $Ux_{\text{rot}} = UU^Tx = x$.

4.1.4 数据降维

数据的主方向就是旋转数据的第一维 $x_{\text{rot},1}$。因此，若想把这数据降到一维，可令：

$$\tilde{x}^{(i)} = x^{(i)}_{\text{rot},1} = u_1^Tx^{(i)} \in \Re. \tag{57}$$

更一般的，假如想把数据 $x \in \Re^n$ 降到 k 维表示 $\tilde{x} \in \Re^k$ （令 $k < n$），只需选取 x_{rot} 的前 k 个成分，分别对应前 k 个数据变化的主方向。

PCA 的另外一种解释是：x_{rot} 是一个 n 维向量，其中前几个成分可能比较大（例如，上例中大部分样本第一个成分 $x^{(i)}_{\text{rot},1} = u_1^Tx^{(i)}$ 的取值相对较大），而后面成分可能会比较小（例如，上例中大部分样本的 $x^{(i)}_{\text{rot},2} = u_2^Tx^{(i)}$ 较小）。

PCA 算法做的其实就是丢弃 x_{rot} 中后面（取值较小）的成分，就是将这些成分的值近似为零。具体的说，设 $\tilde{x}$ 是 x_{rot} 的近似表示，那么将 x_{rot} 除了前 k 个成分外，其余全赋值为零，就得到：

$$\tilde{x} = \begin{bmatrix} x_{\text{rot},1} \\ \vdots \\ x_{\text{rot},k} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \approx \begin{bmatrix} x_{\text{rot},1} \\ \vdots \\ x_{\text{rot},k} \\ x_{\text{rot},k+1} \\ \vdots \\ x_{\text{rot},n} \end{bmatrix} = x_{\text{rot}} \tag{58}$$

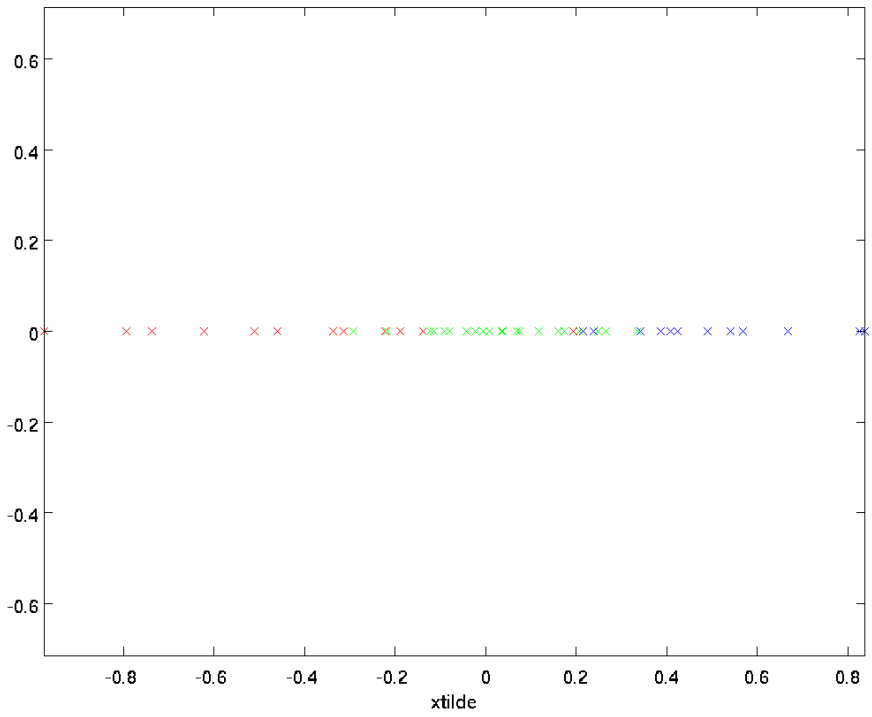
在本例中，可得 $\tilde{x}$ 的点图如下（取 $n = 2, k = 1$ ）：

然而，由于上面 $\tilde{x}$ 的后 $n - k$ 项均为零，没必要把这些零项保留下来。所以，我们仅用前 k 个（非零）成分来定义 k 维向量 $\tilde{x}$。

这也解释了我们为什么会以 $u_1, u_2, \ldots, u_n$ 为基来表示数据：要决定保留哪些成分变得很简单，只需取前 k 个成分即可。这时也可以说，我们“保留了前 k 个 PCA（主）成分”。

4.1.5 还原近似数据

现在，我们得到了原始数据 $x \in \Re^n$ 的低维“压缩”表征量 $\tilde{x} \in \Re^k$，反过来，如果给定 $\tilde{x}$，我们应如何还原原始数据 x 呢？查看以往章节 (4.1.3) 可知，要转换回来，只需 $x = Ux_{\text{rot}}$ 即可。进一步，我们把 $\tilde{x}$ 看作将 x_{rot} 的最后 $n - k$ 个元素被置 0 所得的近似表示，因此如果给定



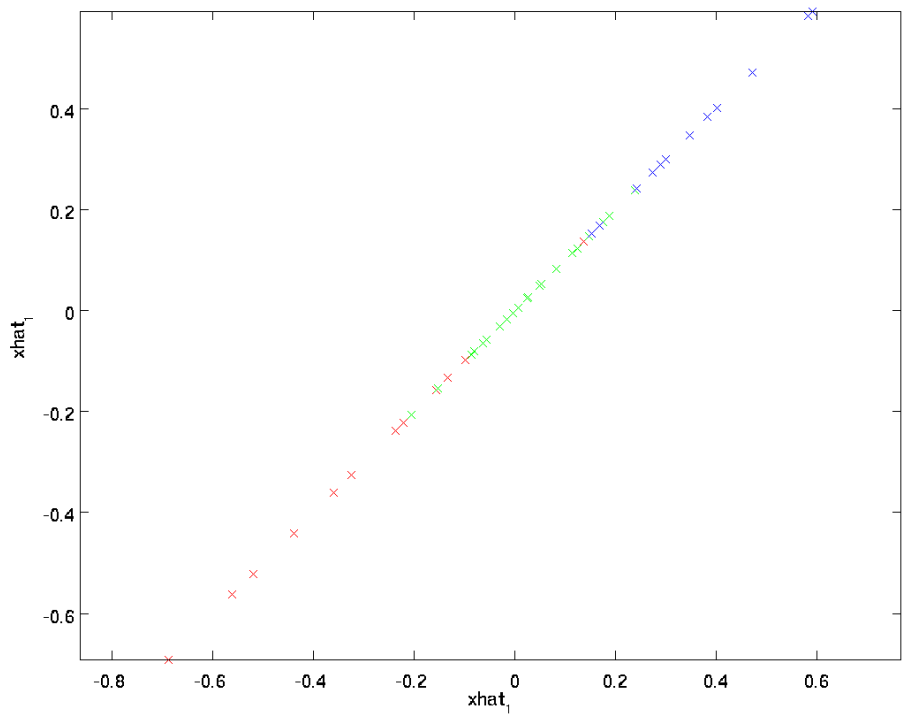
$\tilde{x} \in \Re^k$，可以通过在其末尾添加 $n-k$ 个 0 来得到对 $x_{\text{rot}} \in \Re^n$ 的近似，最后，左乘 U 便可近似还原出原数据 x。具体来说，计算如下：

$$\hat{x} = U \begin{bmatrix} \tilde{x}_1 \\ \vdots \\ \tilde{x}_k \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \sum_{i=1}^{k} u_i \tilde{x}_i. \tag{59}$$

上面的等式基于先前 (4.1.2) 对 U 的定义。在实现时，我们实际上并不先给 $\tilde{x}$ 填 0 然后再左乘 U，因为这意味着大量的乘 0 运算。我们可用 $\tilde{x} \in \Re^k$ 来与 U 的前 k 列相乘，即上式中最右项，来达到同样的目的。将该算法应用于本例中的数据集，可得如下关于重构数据 $\hat{x}$ 的点图：

由图可见，我们得到的是对原始数据集的一维近似重构。

在训练自动编码器或其它无监督特征学习算法时，算法运行时间将依赖于输入数据的维数。若用 $\tilde{x} \in \Re^k$ 取代 x 作为输入数据，那么算法就可使用低维数据进行训练，运行速度将显著加快。对于很多数据集来说，低维表征量 $\tilde{x}$ 是原数据集的极佳近似，因此在这些场合使用 PCA 是很合适的，它引入的近似误差的很小，却可显著地提高你算法的运行速度。



4.1.6 选择主成分个数

我们该如何选择 k，即保留多少个 PCA 主成分？在上面这个简单的二维实验中，保留第一个成分看起来是自然的选择。对于高维数据来说，做这个决定就没那么简单：如果 k 过大，数据压缩率不高，在极限情况 $k = n$ 时，等于是在使用原始数据（只是旋转投射到了不同的基）；相反地，如果 k 过小，那数据的近似误差太太。

决定 k 值时，我们通常会考虑不同 k 值**可保留的方差百分比**。具体来说，如果 $k = n$，那么我们得到的是对数据的完美近似，也就是保留了 100% 的方差，即原始数据的所有变化都被保留下来；相反，如果 $k = 0$，那等于是使用零向量来逼近输入数据，也就是只有 0% 的方差被保留下来。

一般而言，设 $\lambda_1, \lambda_2, \ldots, \lambda_n$ 表示 Σ 的特征值（按由大到小顺序排列），使得 λ_j 为对应于特征向量 u_j 的特征值。那么如果我们保留前 k 个成分，则保留的方差百分比可计算为：

$$\frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{n} \lambda_j}. \tag{60}$$

在上面简单的二维实验中，$\lambda_1 = 7.29, \lambda_2 = 0.69$。因此，如果保留 $k = 1$ 个主成分，等于我们保留了 $7.29/(7.29 + 0.69) = 0.913$，即 91.3% 的方差。

对保留方差的百分比进行更正式的定义已超出了本教程的范围，但很容易证明，$\lambda_j = \sum_{i=1}^{m} x_{\text{rot},j}^2$。因此，如果 $\lambda_j \approx 0$，则说明 $x_{\text{rot},j}$ 也就基本上接近于 0，所以用 0 来近似它并不会产生多大损失。这也解释了为什么要保留前面的主成分（对应的 λ_j 值较大）而不是末尾的那些。

这些前面的主成分 $x_{\text{rot},j}$ 变化性更大，取值也更大，如果将其设为 0 势必引入较大的近似误差。

以处理图像数据为例，一个惯常的经验法则是选择 k 以保留 99% 的方差，换句话说，我们选取满足以下条件的最小 k 值：

$$\frac{\sum_{j=1}^{k}\lambda_j}{\sum_{j=1}^{n}\lambda_j} \geq 0.99. \tag{61}$$

对其它应用，如不介意引入稍大的误差，有时也保留 90-98% 的方差范围。若向他人介绍 PCA 算法详情，告诉他们你选择的 k 保留了 95% 的方差，比告诉他们你保留了前 120 个（或任意某个数字）主成分更好理解。

4.1.7 对图像数据应用 PCA 算法

为使 PCA 算法能有效工作，通常我们希望所有的特征 $x_1, x_2, \ldots, x_n$ 都有相似的取值范围（并且均值接近于 0）。如果你曾在其它应用中使用过 PCA 算法，你可能知道有必要单独对每个特征做预处理，即通过估算每个特征 x_j 的均值和方差，而后将其取值范围规整化为零均值和单位方差。但是，对于大部分图像类型，我们却不需要进行这样的预处理。假定我们将在自然图像上训练算法，此时特征 x_j 代表的是像素 j 的值。所谓“自然图像”，不严格的说，是指人或动物在他们一生中所见的那种图像。

注：通常我们选取含草木等内容的户外场景图片，然后从中随机截取小图像块（如 16×16 像素）来训练算法。在实践中我们发现，大多数特征学习算法对训练图片的确切类型并不敏感，所以大多数用普通照相机拍摄的图片，只要不是特别的模糊或带有非常奇怪的人工痕迹，都可以使用。

在自然图像上进行训练时，对每一个像素单独估计均值和方差意义不大，因为（理论上）图像任一部分的统计性质都应该和其它部分相同，图像的这种特性被称作**平稳性**（stationarity）。

具体而言，为使 PCA 算法正常工作，我们通常需要满足以下要求：(1) 特征的均值大致为 0；(2) 不同特征的方差值彼此相似。对于自然图片，即使不进行方差归一化操作，条件 (2) 也自然满足，故而我们不再进行任何方差归一化操作（对音频数据, 如声谱, 或文本数据, 如词袋向量，我们通常也不进行方差归一化）。实际上，PCA 算法对输入数据具有缩放不变性，无论输入数据的值被如何放大（或缩小），返回的特征向量都不改变。更正式的说：如果将每个特征向量 x 都乘以某个正数（即所有特征量被放大或缩小相同的倍数），PCA 的输出特征向量都将不会发生变化。

既然我们不做方差归一化，唯一还需进行的规整化操作就是均值规整化，其目的是保证所有特征的均值都在 0 附近。根据应用，在大多数情况下，我们并不关注所输入图像的整体明亮程度。比如在对象识别任务中，图像的整体明亮程度并不会影响图像中存在的是什么物体。更为正式地说，我们对图像块的平均亮度值不感兴趣，所以可以减去这个值来进行均值规整化。

具体的步骤是，如果 $x^{(i)} \in \Re^n$ 代表 16×16 的图像块的亮度（灰度）值（$n = 256$），可用如

下算法来对每幅图像进行零均值化操作：

$$\mu^{(i)} := \frac{1}{n}\sum_{j=1}^{n} x_j^{(i)}$$

$$x_j^{(i)} := x_j^{(i)} - \mu^{(i)}, \text{ forall } j$$

请注意：1）对每个输入图像块 $x^{(i)}$ 都要单独执行上面两个步骤，2）这里的 $\mu^{(i)}$ 是指图像块 $x^{(i)}$ 的平均亮度值。尤其需要注意的是，这和为每个像素 x_j 单独估算均值是两个完全不同的概念。

如果你处理的图像并非自然图像（比如，手写文字，或者白背景正中摆放单独物体），其他规整化操作就值得考虑了，而哪种做法最合适也取决于具体应用场合。但对自然图像而言，对每幅图像进行上述的零均值规整化，是默认而合理的处理。

4.1.8 参考文献

http://cs229.stanford.edu/

4.2 白化

4.2.1 介绍

我们已经了解了如何使用 PCA 降低数据维度。在一些算法中还需要一个与之相关的预处理步骤，这个预处理过程称为**白化**（一些文献中也叫 sphering）。举例来说，假设训练数据是图像，由于图像中相邻像素之间具有很强的相关性，所以用于训练时输入是冗余的。白化的目的就是降低输入的冗余性；更正式的说，我们希望通过白化过程使得学习算法的输入具有如下性质：(i) 特征之间相关性较低；(ii) 所有特征具有相同的方差。

4.2.2 2D 的例子

下面我们先用前文的 2D 例子描述白化的主要思想，然后分别介绍如何将白化与平滑和 PCA 相结合。

如何消除输入特征之间的相关性？在前文计算 $x_{\text{rot}}^{(i)} = U^T x^{(i)}$ 时实际上已经消除了输入特征 $x^{(i)}$ 之间的相关性。得到的新特征 x_{rot} 的分布如图 2 所示：

这个数据的协方差矩阵如下：

$$\begin{bmatrix} 7.29 & 0 \\ 0 & 0.69 \end{bmatrix}. \tag{62}$$

(注: 严格地讲, 这部分许多关于“协方差”的陈述仅当数据均值为 0 时成立。下文的论述都隐式地假定这一条件成立。不过即使数据均值不为 0，下文的说法仍然成立，所以你无需担心这个。)

x_{rot} 协方差矩阵对角元素的值为 λ_1 和 λ_2 绝非偶然。并且非对角元素值为 0; 因此, $x_{\text{rot},1}$ 和 $x_{\text{rot},2}$ 是不相关的, 满足我们对白化结果的第一个要求 (特征间相关性降低)。

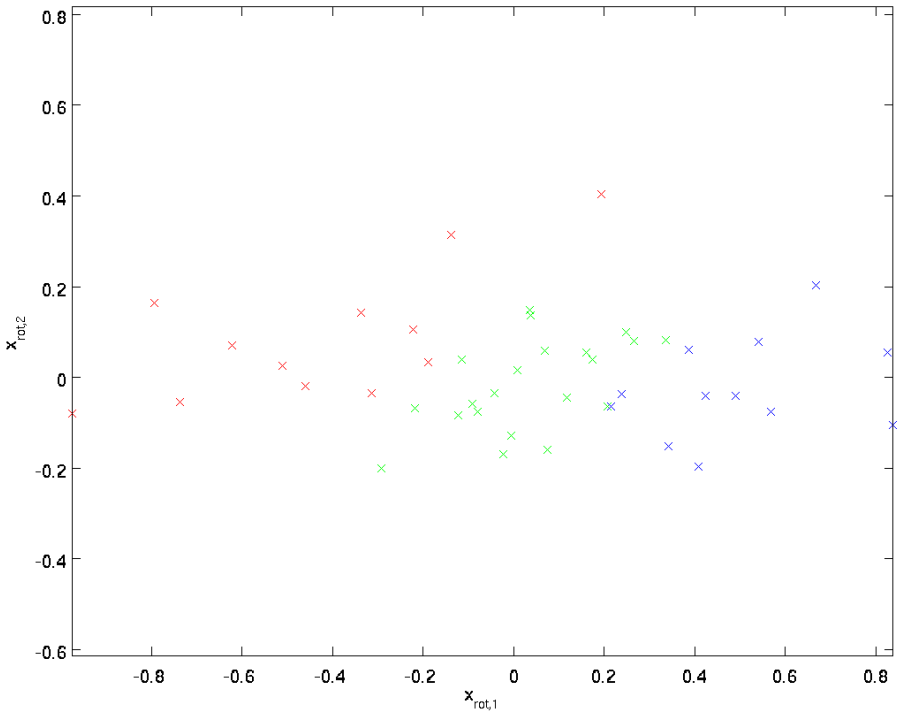


图 3:

为了使每个输入特征具有单位方差，我们可以直接使用 $1/\sqrt{\lambda_i}$ 作为缩放因子来缩放每个特征 $x_{\text{rot},i}$。具体地，我们定义白化后的数据 $x_{\text{PCAwhite}} \in \Re^n$ 如下：

$$x_{\text{PCAwhite},i} = \frac{x_{\text{rot},i}}{\sqrt{\lambda_i}}. \tag{63}$$

绘制出 x_{PCAwhite}，我们得到:

这些数据现在的协方差矩阵为单位矩阵 I。我们说，x_{PCAwhite} 是数据经过 PCA **白化**后的版本: x_{PCAwhite} 中不同的特征之间不相关并且具有单位方差。

白化与降维相结合。如果你想要得到经过白化后的数据，并且比初始输入维数更低, 可以仅保留 x_{PCAwhite} 中前 k 个成分。当我们把 PCA 白化和正则化结合起来时 (在稍后讨论)，x_{PCAwhite} 中最后的少量成分将总是接近于 0，因而舍弃这些成分不会带来很大的问题。

4.2.3 ZCA 白化

最后要说明的是，使数据的协方差矩阵变为单位矩阵 I 的方式并不唯一。具体地，如果 R 是任意正交矩阵，即满足 $RR^T = R^T R = I$ (说它正交不太严格，R 可以是旋转或反射矩阵), 那么 $R\,x_{\text{PCAwhite}}$ 仍然具有单位协方差。在 ZCA **白化**中，令 $R = U$。我们定义 ZCA 白化的结果为:

$$x_{\text{ZCAwhite}} = U x_{\text{PCAwhite}} \tag{64}$$

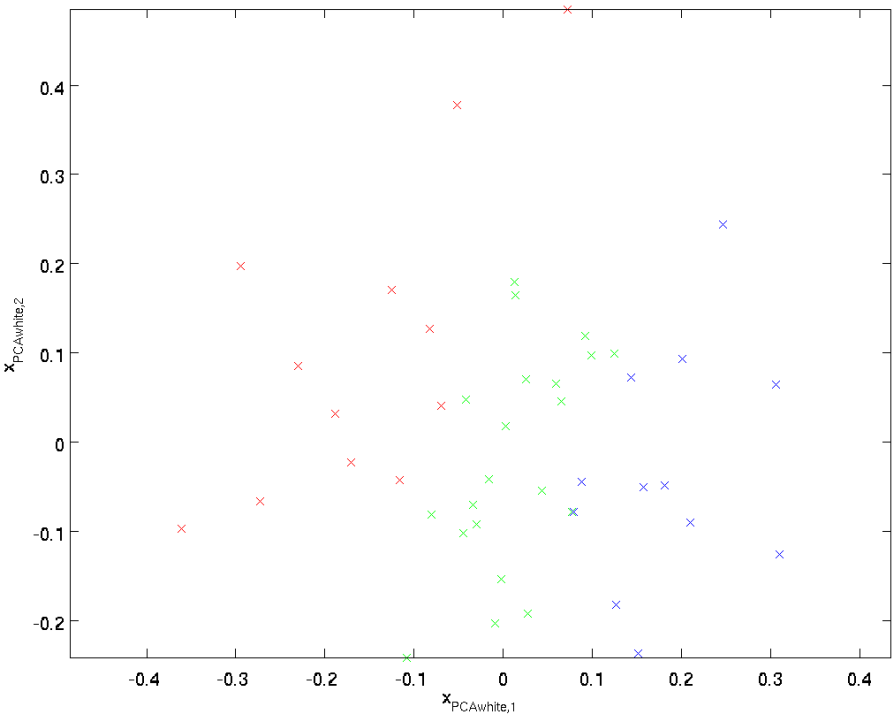


图 4:

绘制 x_{ZCAwhite}，得到:

可以证明，对所有可能的 R，这种旋转使得 x_{ZCAwhite} 尽可能地接近原始输入数据 x。

当使用 ZCA 白化时 (不同于 PCA 白化)，我们通常保留数据的全部 n 个维度，不尝试去降低它的维数。

4.2.4 正则化

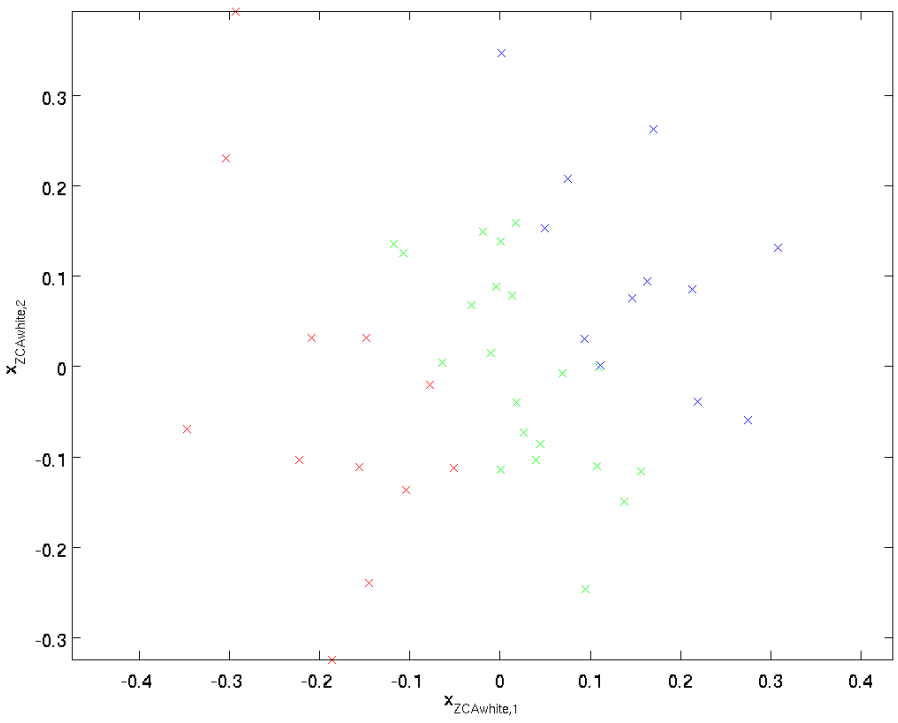
实践中需要实现 PCA 白化或 ZCA 白化时，有时一些特征值 λ_i 在数值上接近于 0，这样在缩放步骤时我们除以 $\sqrt{\lambda_i}$ 将导致除以一个接近 0 的值；这可能使数据上溢 (赋为大数值) 或造成数值不稳定。因而在实践中，我们使用少量的正则化实现这个缩放过程，即在取平方根和倒数之前给特征值加上一个很小的常数 ϵ:

$$x_{\text{PCAwhite},i} = \frac{x_{\text{rot},i}}{\sqrt{\lambda_i + \epsilon}}. \tag{65}$$

当 x 在区间 $[-1, 1]$ 上时, 一般取值为 $\epsilon \approx 10^{-5}$。

对图像来说, 这里加上 ϵ，对输入图像也有一些平滑 (或低通滤波) 的作用。这样处理还能消除在图像的像素信息获取过程中产生的噪声，改善学习到的特征 (细节超出了本文的范围)。

ZCA 白化是一种数据预处理方法，它将数据从 x 映射到 x_{ZCAwhite}。事实证明这也是一种生物眼睛 (视网膜) 处理图像的粗糙模型。具体而言，当你的眼睛感知图像时，由于一幅图像中相



邻的部分在亮度上十分相关，大多数临近的“像素”在眼中被感知为相近的值。因此，如果人眼需要分别传输每个像素值（通过视觉神经）到大脑中，会非常不划算。取而代之的是，视网膜进行一个与 ZCA 中相似的去相关操作 (这是由视网膜上的 ON-型和 OFF-型光感受器细胞将光信号转变为神经信号完成的)。由此得到对输入图像的更低冗余的表示，并将它传输到大脑。

4.3 实现主成分分析和白化

在这一节里，我们将总结 PCA, PCA 白化和 ZCA 白化算法，并描述如何使用高效的线性代数库来实现它们。

首先，我们需要确保数据的均值（近似）为零。对于自然图像，我们通过减去每个图像块 (patch) 的均值（近似地）来达到这一目标。

为此，我们计算每个图像块的均值，并从每个图像块中减去它的均值。(译注：参见 PCA 一章中“对图像数据应用 PCA 算法”一节)。Matlab 实现如下：

```
avg = mean(x, 1);     % Compute the mean pixel intensity value separately for each
    patch.
x = x - repmat(avg, size(x, 1), 1);
```

下面，我们要计算 $\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})(x^{(i)})^T$，如果你在 Matlab 中实现（或者在 C++, Java 等中实现，但可以使用高效的线性代数库），直接求和效率很低。不过，我们可以这样一气呵成。

```
sigma = x * x' / size(x, 2);
```

（自己推导一下看看）这里，我们假设 x 为一数据结构，其中每列表示一个训练样本（所以 x 是一个 $n \times m$ 的矩阵）。

接下来，PCA 计算 Σ 的特征向量。你可以使用 Matlab 的 eig 函数来计算。但是由于 Σ 是对称半正定的矩阵，用 svd 函数在数值计算上更加稳定。具体来说，如果你使用

```
[U,S,V] = svd(sigma);
```

那矩阵 U 将包含 Sigma 的特征向量（一个特征向量一列，从主向量开始排序），矩阵 S 对角线上的元素将包含对应的特征值（同样降序排列）。矩阵 V 等于 U 的转置，可以忽略。

（注意 svd 函数实际上计算的是一个矩阵的奇异值和奇异向量，就对称半正定矩阵的特殊情况来说，它们对应于特征值和特征向量，这里我们也只关心这一特例。关于奇异向量和特征向量的详细讨论超出了本文范围。）

最后，我们可以这样计算 x_{rot} 和 $\tilde{x}$:

```
xRot = U' * x;          % rotated version of the data.
xTilde = U(:,1:k)' * x; % reduced dimension representation of the data,
                        % where k is the number of eigenvectors to keep
```

这以 $\tilde{x} \in \Re^k$ 的形式给出了数据的 PCA 表示。顺便说一下，如果 x 是一个包括所有训练数据的 $n \times m$ 矩阵，这也是一种向量化的实现方式，上面的式子可以让你一次对所有的训练样本计算出 x_{rot} 和 $\tilde{x}$。得到的 x_{rot} 和 $\tilde{x}$ 中，每列对应一个训练样本。

为计算 PCA 白化后的数据 x_{PCAwhite}，可以用

```
xPCAwhite = diag(1./sqrt(diag(S) + epsilon)) * U' * x;
```

因为 S 的对角线包括了特征值 λ_i，这其实就是同时为所有样本 i 计算 $x_{\text{PCAwhite},i} = \frac{x_{\text{rot},i}}{\sqrt{\lambda_i}}$ 的简洁表达。

最后，你也可以这样计算 ZCA 白化后的数据 x_{ZCAwhite}:

```
xZCAwhite = U * diag(1./sqrt(diag(S) + epsilon)) * U' * x;
```

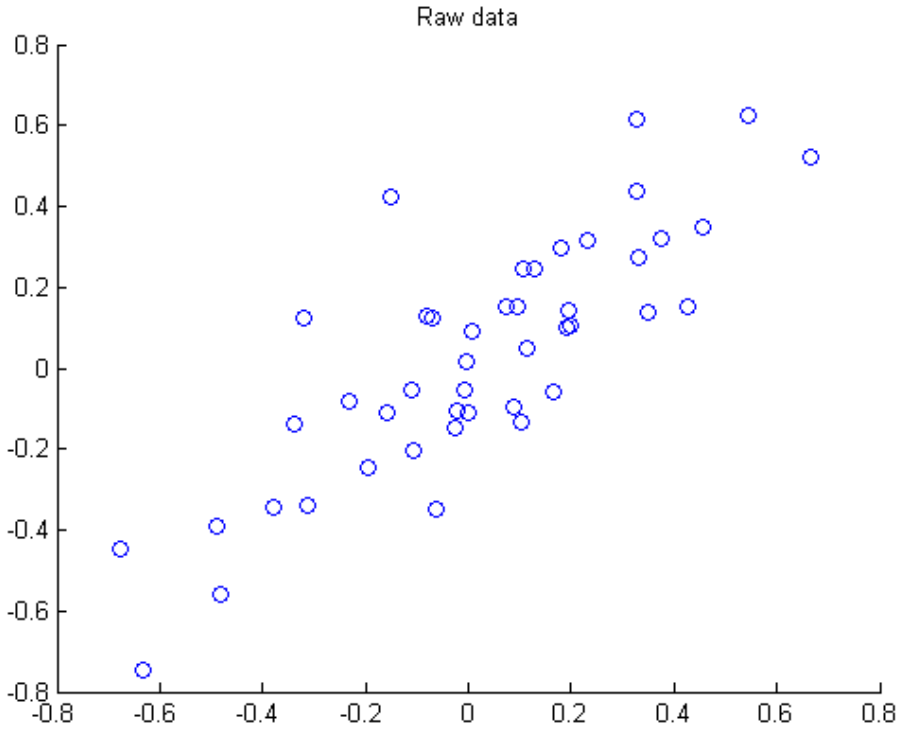
4.4 练习：2D 中的 PCA

PCA, PCA whitening and ZCA whitening in 2D

In this exercise you will implement PCA, PCA whitening and ZCA whitening, as described in the earlier sections of this tutorial, and generate the images shown in the earlier sections yourself. You will build on the starter code that has been provided at http://ufldl.stanford.edu/wiki/resources/pca_2d.zip. You need only write code at the places indicated by "YOUR CODE HERE" in the files. The only file you need to modify is pca_2d.m. Implementing this exercise will make the next exercise significantly easier to understand and complete.

4.4.1 Step 0: Load data

The starter code contains code to load 45 2D data points. When plotted using the scatter function, the results should look like the following:



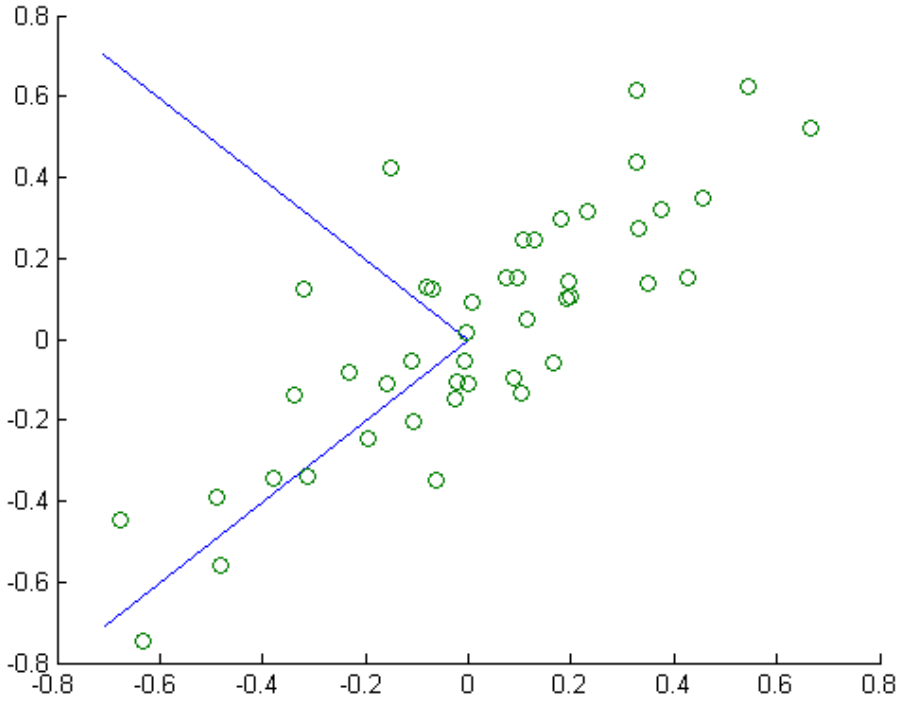
4.4.2 Step 1: Implement PCA

In this step, you will implement PCA to obtain xrot, the matrix in which the data is "rotated" to the basis comprising $u_1, \ldots, u_n$ made up of the principal components. As mentioned in the implementation notes, you should make use of MATLAB's svd function here.

4.4.3 Step 1a: Finding the PCA basis

Find u_1 and u_2, and draw two lines in your figure to show the resulting basis on top of the given data points. You may find it useful to use MATLAB's `hold on` and `hold off` functions. (After calling `hold on`, plotting functions such as `plot` will draw the new data on top of the previously existing figure rather than erasing and replacing it; and `hold off` turns this off.) You can use `plot([x1,x2], [y1,y2], '-')` to draw a line between `(x1,y1)` and `(x2,y2)`. Your figure should look like this:

If you are doing this in Matlab, you will probably get a plot that's identical to ours. However, eigenvectors are defined only up to a sign. I.e., instead of returning u_1 as the first eigenvector, Matlab/Octave could just as easily have returned $-u_1$, and similarly instead of u_2 Matlab/Octave



could have returned $-u_2$. So if you wound up with one or both of the eigenvectors pointing in a direction opposite (180 degrees difference) from what's shown above, that's okay too.

4.4.4 Step 1b: Check xRot

Compute `xRot`, and use the `scatter` function to check that `xRot` looks as it should, which should be something like the following:

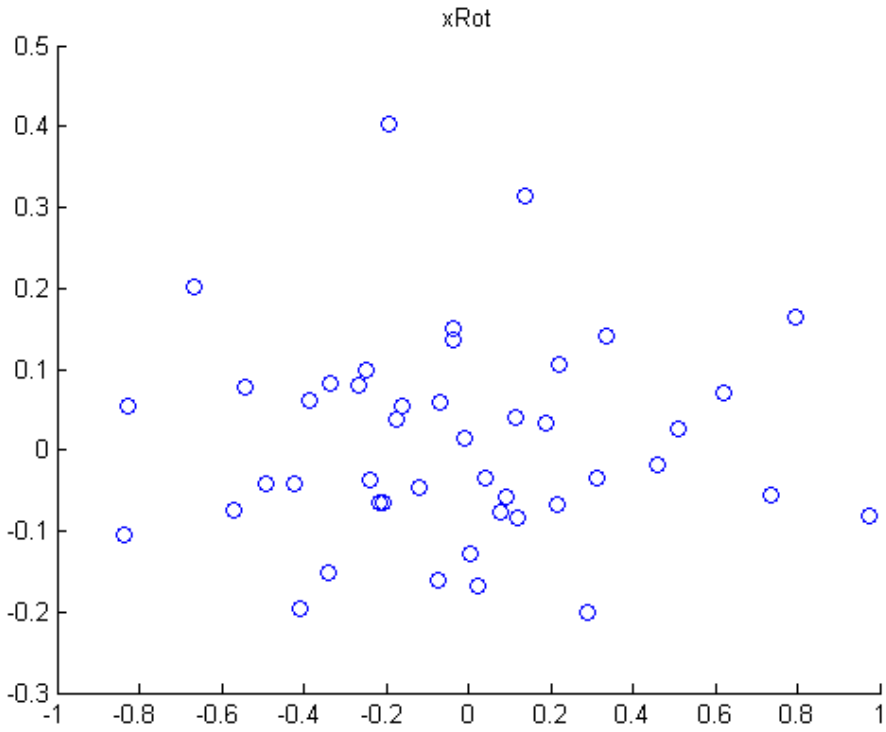
Because Matlab/Octave could have returned $-u_1$ and/or $-u_2$ instead of u_1 and u_2, it's also possible that you might have gotten a figure which is "flipped" or "reflected" along the x- and/or y-axis; a flipped/reflected version of this figure is also a completely correct result.

4.4.5 Step 2: Dimension reduce and replot

In the next step, set k, the number of components to retain, to be 1 (we have already done this for you). Compute the resulting xHat and plot the results. You should get the following (this figure should not be flipped along the x- or y-axis):

4.4.6 Step 3: PCA Whitening

Implement PCA whitening using the formula from the notes. Plot `xPCAWhite`, and verify that it looks like the following (a figure that is flipped/reflected on either/both axes is also correct):



4.4.7 Step 4: ZCA Whitening

Implement ZCA whitening and plot the results. The results should look like the following (this should not be flipped/reflected along the x- or y-axis):

4.5 练习：PCA 和白化

PCA and Whitening on natural images

In this exercise, you will implement PCA, PCA whitening and ZCA whitening, and apply them to image patches taken from natural images.

You will build on the MATLAB starter code which we have provided in `http://ufldl.stanford.edu/wiki/resources/pca_exercise.zip`pca_exercise.zip. You need only write code at the places indicated by "YOUR CODE HERE" in the files. The only file you need to modify is `pca_gen.m`.

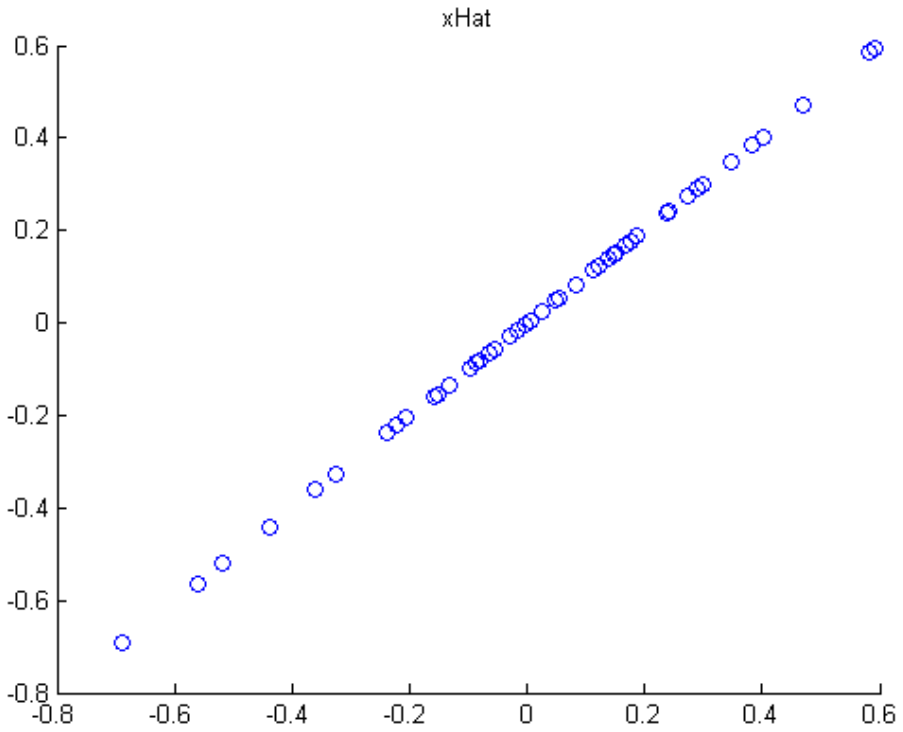
4.5.1 Step 0: Prepare data

Step 0a: Load data

The starter code contains code to load a set of natural images and sample 12×12 patches from them. The raw patches will look something like this:

These patches are stored as column vectors $x^{(i)} \in \mathbb{R}^{144}$ in the 144×10000 matrix `x`.

Step 0b: Zero mean the data



First, for each image patch, compute the mean pixel value and subtract it from that image, this centering the image around zero. You should compute a different mean value for each image patch.

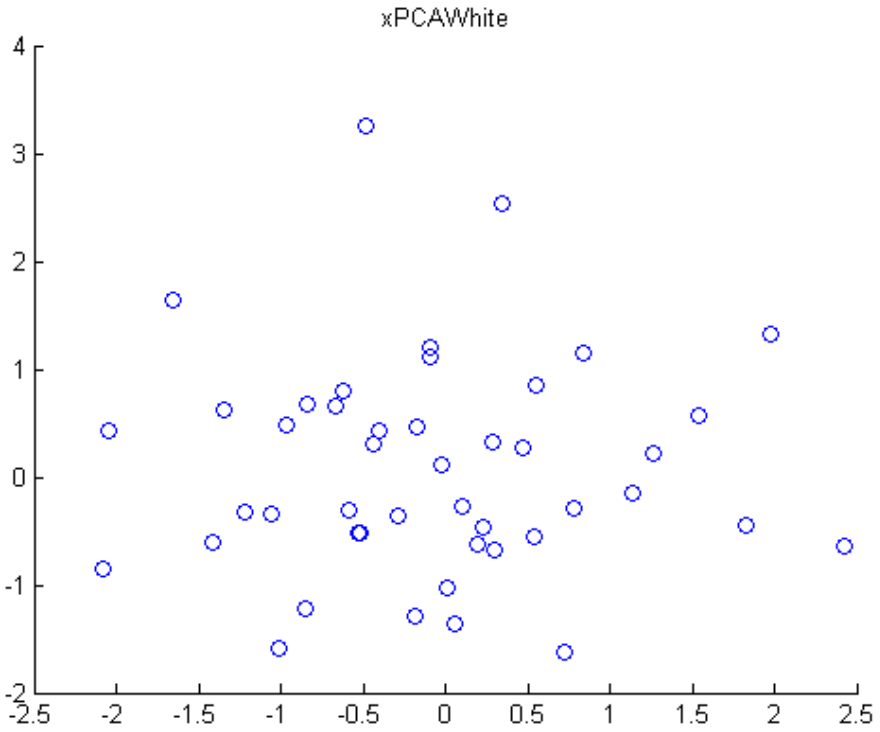
4.5.2 Step 1: Implement PCA

Step 1a: Implement PCA

n this step, you will implement PCA to obtain x_{rot}, the matrix in which the data is "rotated" to the basis comprising the principal components (i.e. the eigenvectors of Σ). Note that in this part of the exercise, you should not whiten the data.

Step 1b: Check covariance

To verify that your implementation of PCA is correct, you should check the covariance matrix for the rotated data x_{rot}. PCA guarantees that the covariance matrix for the rotated data is a diagonal matrix (a matrix with non-zero entries only along the main diagonal). Implement code to compute the covariance matrix and verify this property. One way to do this is to compute the covariance matrix, and visualise it using the MATLAB command imagesc. The image should show a coloured diagonal line against a blue background. For this dataset, because of the range of the diagonal entries, the diagonal line may not be apparent, so you might get a figure like the one show below, but this trick of visualizing using imagesc will come in handy later in this exercise.



4.5.3 Step 2: Find number of components to retain

Next, choose k, the number of principal components to retain. Pick k to be as small as possible, but so that at least 99% of the variance is retained. In the step after this, you will discard all but the top k principal components, reducing the dimension of the original data to k.

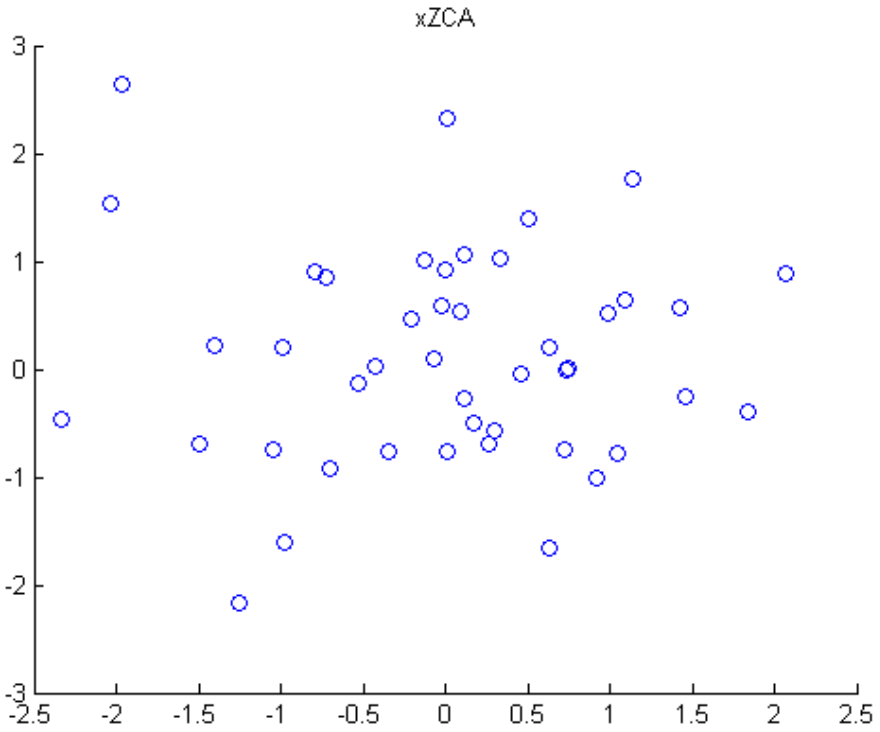
4.5.4 Step 3: PCA with dimension reduction

Now that you have found k, compute $\tilde{x}$, the reduced-dimension representation of the data. This gives you a representation of each image patch as a k dimensional vector instead of a 144 dimensional vector. If you are training a sparse autoencoder or other algorithm on this reduced-dimensional data, it will run faster than if you were training on the original 144 dimensional data.

To see the effect of dimension reduction, go back from $\tilde{x}$ to produce the matrix $\hat{x}$, the dimension-reduced data but expressed in the original 144 dimensional space of image patches. Visualise $\hat{x}$ and compare it to the raw data, x. You will observe that there is little loss due to throwing away the principal components that correspond to dimensions with low variation. For comparison, you may also wish to generate and visualise $\hat{x}$ for when only 90% of the variance is retained.

4.5.5 Step 4: PCA with whitening and regularization

Step 4a: Implement PCA with whitening and regularization



Now implement PCA with whitening and regularization to produce the matrix `xPCAWhite`. Use the following parameter value:

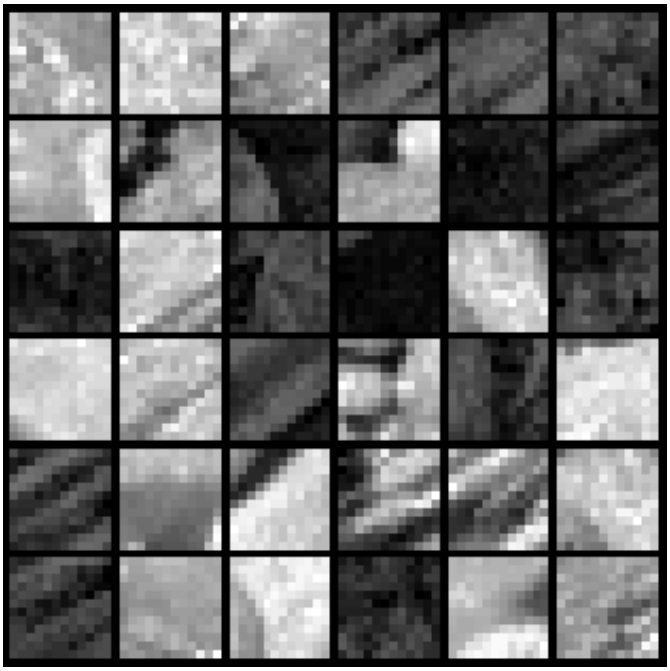
```
epsilon = 0.1
```

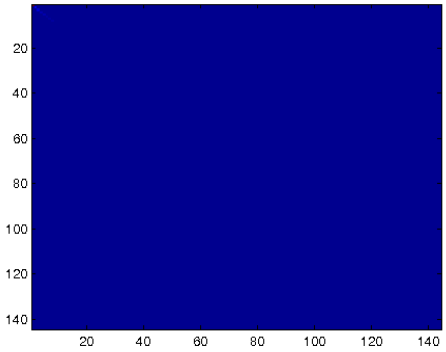
Step 4b: Check covariance

Similar to using PCA alone, PCA with whitening also results in processed data that has a diagonal covariance matrix. However, unlike PCA alone, whitening additionally ensures that the diagonal entries are equal to 1, i.e. that the covariance matrix is the identity matrix.

That would be the case if you were doing whitening alone with no regularization. However, in this case you are whitening with regularization, to avoid numerical/etc. problems associated with small eigenvalues. As a result of this, some of the diagonal entries of the covariance of your `xPCAwhite` will be smaller than 1.

To verify that your implementation of PCA whitening with and without regularization is correct, you can check these properties. Implement code to compute the covariance matrix and verify this property. (To check the result of PCA without whitening, simply set epsilon to 0, or close to 0, say 1e-10). As earlier, you can visualise the covariance matrix with imagesc. When visualised as an image, for PCA whitening without regularization you should see a red line across the diagonal (corresponding to the one entries) against a blue background (corresponding to the zero entries); for PCA whitening with regularization you should see a red line that slowly turns





blue across the diagonal (corresponding to the 1 entries slowly becoming smaller).

4.5.6 Step 5: ZCA whitening

Now implement ZCA whitening to produce the matrix `xZCAWhite`. Visualize `xZCAWhite` and compare it to the raw data, x. You should observe that whitening results in, among other things, enhanced edges. Try repeating this with epsilon set to 1, 0.1, and 0.01, and see what you obtain. The example shown below (left image) was obtained with epsilon = 0.1.

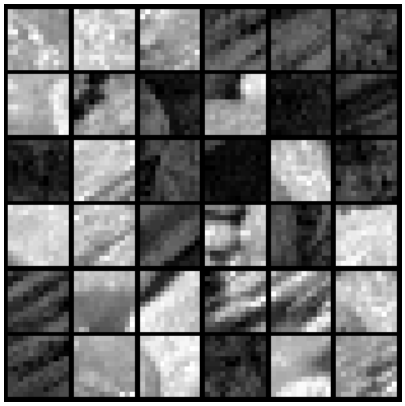


图 5: Raw images

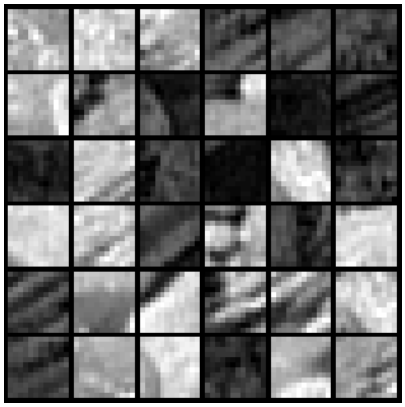


图 6: PCA dimension-reduced images (99% variance)

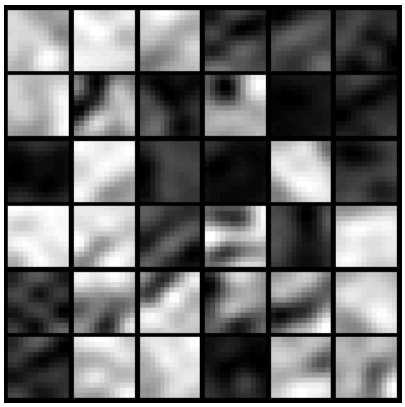


图 7: PCA dimension-reduced images (90% variance)

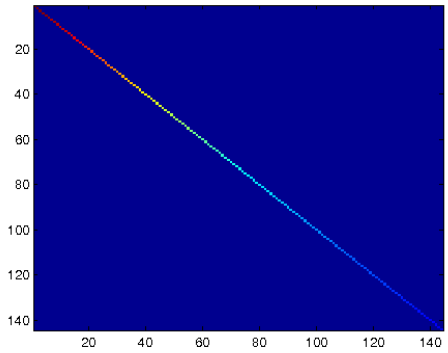


图 8: Covariance for PCA whitening with regularization

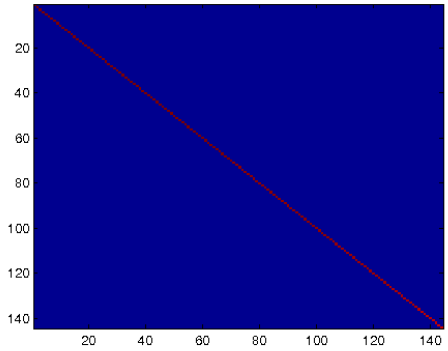


图 9: Covariance for PCA whitening without regularization

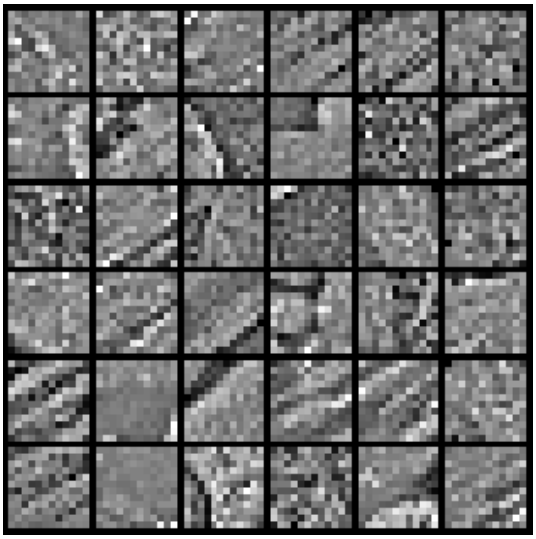


图 10: ZCA whitened images

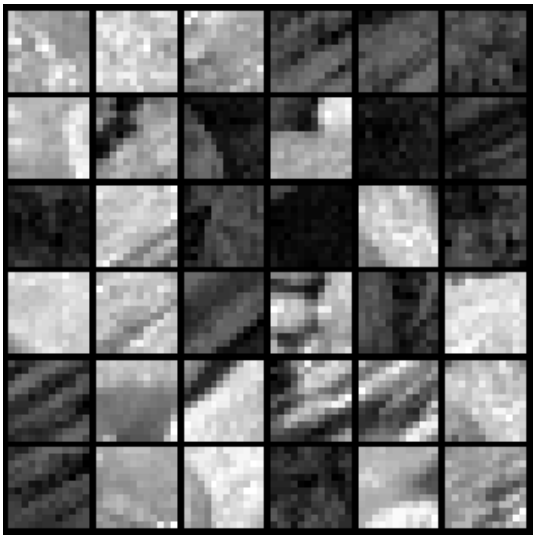


图 11: Raw images

5 Softmax 回归

5.1 Softmax 回归

5.1.1 介绍

在本节中，我们介绍 Softmax **回归**模型，该模型是 logistic 回归模型在多分类问题上的推广，在多分类问题中，类标签 y 可以取两个以上的值。Softmax 回归模型对于诸如 MNIST[12] 手写数字分类等问题是很有用的，该问题的目的是辨识 10 个不同的单个数字。Softmax 回归是有监督的，不过后面也会介绍它与深度学习/无监督学习方法的结合。

回想一下在 logistic 回归中,我们的训练集由 m 个已标记的样本构成:$\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$，其中输入特征 $x^{(i)} \in \Re^{n+1}$。(我们对符号的约定如下：特征向量 x 的维度为 $n+1$，其中 $x_0 = 1$ 对应截距项。) 由于 logistic 回归是针对二分类问题的，因此类标记 $y^{(i)} \in \{0, 1\}$。假设函数 (hypothesis function) 如下：

$$h_\theta(x) = \frac{1}{1 + \exp(-\theta^T x)}, \tag{66}$$

我们将训练模型参数 θ，使其能够最小化代价函数：

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] \tag{67}$$

在 softmax 回归中，我们解决的是多分类问题（相对于 logistic 回归解决的二分类问题），类标 y 可以取 k 个不同的值（而不是 2 个）。因此，对于训练集 $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$，我们有 $y^{(i)} \in \{1, 2, \ldots, k\}$。(注意此处的类别下标从 1 开始，而不是 0)。例如，在 MNIST 数字识别任务中，我们有 $k = 10$ 个不同的类别。

对于给定的测试输入 x，我们想用假设函数针对每一个类别 j 估算出概率值 $p(y = j|x)$。也就是说，我们想估计 x 的每一种分类结果出现的概率。因此，我们的假设函数将要输出一个 k 维的向量（向量元素的和为 1）来表示这 k 个估计的概率值。具体地说，我们的假设函数 $h_\theta(x)$ 形式如下：

$$h_\theta(x^{(i)}) = \begin{bmatrix} p(y^{(i)} = 1|x^{(i)}; \theta) \\ p(y^{(i)} = 2|x^{(i)}; \theta) \\ \vdots \\ p(y^{(i)} = k|x^{(i)}; \theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix} \tag{68}$$

其中 $\theta_1, \theta_2, \ldots, \theta_k \in \Re^{n+1}$ 是模型的参数。请注意 $\frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}}$ 这一项对概率分布进行归一化，使得所有概率之和为 1 。

[12]译者注：MNIST 是一个手写数字识别库，由 NYU 的 Yann LeCun 等人维护。http://yann.lecun.com/exdb/mnist/.

为了方便起见，我们同样使用符号 θ 来表示全部的模型参数。在实现 Softmax 回归时，将 θ 用一个 $k \times (n+1)$ 的矩阵来表示会很方便，该矩阵是将 $\theta_1, \theta_2, \ldots, \theta_k$ 按行罗列起来得到的，如下所示：

$$\theta = \begin{bmatrix} —\theta_1^T— \\ —\theta_2^T— \\ \vdots \\ —\theta_k^T— \end{bmatrix}$$

5.1.2 代价函数

现在我们来介绍 softmax 回归算法的代价函数。在下面的公式中，$1\{\cdot\}$ 是示性函数，其取值规则为：$1\{\textbf{值为真的表达式}\} = 1$，$1\{\textbf{值为假的表达式}\} = 0$。举例来说，表达式 $1\{2+2=4\}$ 的值为 1 ，$1\{1+1=5\}$ 的值为 0。我们的代价函数为：

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{k} 1\left\{y^{(i)}=j\right\}\log\frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}\right] \tag{69}$$

值得注意的是，上述公式是 logistic 回归代价函数的推广。logistic 回归代价函数可以改为：

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}(1-y^{(i)})\log(1-h_\theta(x^{(i)})) + y^{(i)}\log h_\theta(x^{(i)})\right] \tag{70}$$

$$= -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=0}^{1} 1\left\{y^{(i)}=j\right\}\log p(y^{(i)}=j|x^{(i)};\theta)\right] \tag{71}$$

可以看到，Softmax 代价函数与 logistic 代价函数在形式上非常类似，只是在 Softmax 损失函数中对类标记的 k 个可能值进行了累加。注意在 Softmax 回归中将 x 分类为类别 j 的概率为：$p(y^{(i)}=j|x^{(i)};\theta) = \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}$ 。

对于 $J(\theta)$ 的最小化问题，目前还没有闭式解法。因此，我们使用迭代的优化算法（例如梯度下降法，或 L-BFGS）。经过求导，我们得到梯度公式如下：

$$\nabla_{\theta_j} J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[x^{(i)}\left(1\{y^{(i)}=j\} - p(y^{(i)}=j|x^{(i)};\theta)\right)\right] \tag{72}$$

让我们来回顾一下符号 "∇_{θ_j}" 的含义。$\nabla_{\theta_j} J(\theta)$ 本身是一个向量，它的第 l 个元素 $\frac{\partial J(\theta)}{\partial \theta_{jl}}$ 是 $J(\theta)$ 对 θ_j 的第 l 个分量的偏导数。

有了上面的偏导数公式以后，我们就可以将它代入到梯度下降法等算法中，来最小化 $J(\theta)$。例如，在梯度下降法的标准实现中，每一次迭代需要进行如下更新: $\theta_j := \theta_j - \alpha\nabla_{\theta_j} J(\theta)$ ($j = 1, \ldots, k$)。

当实现 softmax 回归算法时，我们通常会使用上述代价函数的一个改进版本。具体来说，就是和权重衰减 (weight decay) 一起使用。我们接下来介绍使用它的动机和细节。

5.1.3 Softmax 回归模型参数化的特点

Softmax 回归有一个不寻常的特点：它有一个“冗余”的参数集。为了便于阐述这一特点，假设我们从参数向量 θ_j 中减去了向量 ψ，这时，每一个 θ_j 都变成了 $\theta_j - \psi$ $(j = 1, \ldots, k)$。此时假设函数变成了以下的式子：

$$p(y^{(i)} = j|x^{(i)}; \theta) = \frac{e^{(\theta_j - \psi)^T x^{(i)}}}{\sum_{l=1}^{k} e^{(\theta_l - \psi)^T x^{(i)}}} \tag{73}$$

$$= \frac{e^{\theta_j^T x^{(i)}} e^{-\psi^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}} e^{-\psi^T x^{(i)}}} \tag{74}$$

$$= \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}. \tag{75}$$

换句话说，从 θ_j 中减去 ψ 完全不影响假设函数的预测结果！这表明前面的 softmax 回归模型中存在冗余的参数。更正式一点来说，Softmax 模型被过度参数化了。对于任意一个用于拟合数据的假设函数，可以求出多组参数值，这些参数得到的是完全相同的假设函数 h_θ。

进一步而言，如果参数 $(\theta_1, \theta_2, \ldots, \theta_k)$ 是代价函数 $J(\theta)$ 的极小值点，那么 $(\theta_1 - \psi, \theta_2 - \psi, \ldots, \theta_k - \psi)$ 同样也是它的极小值点，其中 ψ 可以为任意向量。因此使 $J(\theta)$ 最小化的解不是唯一的。（有趣的是，由于 $J(\theta)$ 仍然是一个凸函数，因此梯度下降时不会遇到局部最优解的问题。但是 Hessian 矩阵是奇异的/不可逆的，这会直接导致采用牛顿法优化就遇到数值计算的问题）

注意，当 $\psi = \theta_1$ 时，我们总是可以将 θ_1 替换为 $\theta_1 - \psi = \vec{0}$（即替换为全零向量），并且这种变换不会影响假设函数。因此我们可以去掉参数向量 θ_1 （或者其他 θ_j 中的任意一个）而不影响假设函数的表达能力。实际上，与其优化全部的 $k \times (n+1)$ 个参数 $(\theta_1, \theta_2, \ldots, \theta_k)$ （其中 $\theta_j \in \Re^{n+1}$），我们可以令 $\theta_1 = \vec{0}$，只优化剩余的 $(k-1) \times (n+1)$ 个参数，这样算法依然能够正常工作。

在实际应用中，为了使算法实现更简单清楚，往往保留所有参数 $(\theta_1, \theta_2, \ldots, \theta_n)$，而不任意地将某一参数设置为 0。但此时我们需要对代价函数做一个改动：加入权重衰减。权重衰减可以解决 softmax 回归的参数冗余所带来的数值问题。

5.1.4 权重衰减

我们通过添加一个权重衰减项 $\frac{\lambda}{2} \sum_{i=1}^{k} \sum_{j=0}^{n} \theta_{ij}^2$ 来修改代价函数，这个衰减项会惩罚过大的参数值，现在我们的代价函数变为：

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^{m} \sum_{j=1}^{k} 1\left\{ y^{(i)} = j \right\} \log \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}} \right] + \frac{\lambda}{2} \sum_{i=1}^{k} \sum_{j=0}^{n} \theta_{ij}^2 \tag{76}$$

有了这个权重衰减项以后 $(\lambda > 0)$，代价函数就变成了严格的凸函数，这样就可以保证得到唯一的解了。此时的 Hessian 矩阵变为可逆矩阵，并且因为 $J(\theta)$ 是凸函数，梯度下降法和 L-BFGS 等算法可以保证收敛到全局最优解。

为了使用优化算法，我们需要求得这个新函数 $J(\theta)$ 的导数，如下：

$$\nabla_{\theta_j} J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[x^{(i)}(1\{y^{(i)}=j\} - p(y^{(i)}=j|x^{(i)};\theta))\right] + \lambda\theta_j \tag{77}$$

通过最小化 $J(\theta)$，我们就能实现一个可用的 softmax 回归模型。

5.1.5 Softmax 回归与 Logistic 回归的关系

当类别数 $k=2$ 时，softmax 回归退化为 logistic 回归。这表明 softmax 回归是 logistic 回归的一般形式。具体地说，当 $k=2$ 时，softmax 回归的假设函数为：

$$h_\theta(x) = \frac{1}{e^{\theta_1^T x} + e^{\theta_2^T x^{(i)}}}\begin{bmatrix} e^{\theta_1^T x} \\ e^{\theta_2^T x} \end{bmatrix} \tag{78}$$

利用 softmax 回归参数冗余的特点，我们令 $\psi = \theta_1$，并且从两个参数向量中都减去向量 θ_1，得到:

$$h(x) = \frac{1}{e^{\vec{0}^T x} + e^{(\theta_2-\theta_1)^T x^{(i)}}}\begin{bmatrix} e^{\vec{0}^T x} \\ e^{(\theta_2-\theta_1)^T x} \end{bmatrix} \tag{79}$$

$$= \begin{bmatrix} \frac{1}{1+e^{(\theta_2-\theta_1)^T x^{(i)}}} \\ \frac{e^{(\theta_2-\theta_1)^T x}}{1+e^{(\theta_2-\theta_1)^T x^{(i)}}} \end{bmatrix} \tag{80}$$

$$= \begin{bmatrix} \frac{1}{1+e^{(\theta_2-\theta_1)^T x^{(i)}}} \\ 1 - \frac{1}{1+e^{(\theta_2-\theta_1)^T x^{(i)}}} \end{bmatrix} \tag{81}$$

因此，用 θ' 来表示 $\theta_2 - \theta_1$，我们就会发现 softmax 回归器预测其中一个类别的概率为 $\frac{1}{1+e^{(\theta')^T x^{(i)}}}$，另一个类别概率的为 $1 - \frac{1}{1+e^{(\theta')^T x^{(i)}}}$，这与 logistic 回归是一致的。

5.1.6 Softmax 回归 vs. k 个二元分类器

如果你在开发一个音乐分类的应用，需要对 k 种类型的音乐进行识别，那么是选择使用 softmax 分类器呢，还是使用 logistic 回归算法建立 k 个独立的二元分类器呢?

这一选择取决于你的类别之间是否互斥，例如，如果你有四个类别的音乐，分别为：古典音乐、乡村音乐、摇滚乐和爵士乐，那么你可以假设每个训练样本只会被打上一个标签（即：一首歌只能属于这四种音乐类型的其中一种），此时你应该使用类别数 $k=4$ 的 softmax 回归。（如果在你的数据集中，有的歌曲不属于以上四类的其中任何一类，那么你可以添加一个“其他类”，并将类别数 k 设为 5。）

如果你的四个类别如下：人声音乐、舞曲、影视原声、流行歌曲，那么这些类别之间并不是互斥的。例如：一首歌曲可以来源于影视原声，同时也包含人声。这种情况下，使用 4 个二分类的 logistic 回归分类器更为合适。这样，对于每个新的音乐作品，我们的算法可以分别判断它是否属于各个类别。

现在我们来看一个计算视觉领域的例子，你的任务是将图像分到三个不同类别中。(i) 假设这三个类别分别是：室内场景、户外城区场景、户外荒野场景。你会使用 sofmax 回归还是 3 个 logistic 回归分类器呢？(ii) 现在假设这三个类别分别是室内场景、黑白图片、包含人物的图片，你又会选择 softmax 回归还是多个 logistic 回归分类器呢？

在第一个例子中，三个类别是互斥的，因此更适于选择 softmax 回归分类器。而在第二个例子中，建立三个独立的 logistic 回归分类器更加合适。

5.2 练习：Softmax 回归

In this problem set, you will use softmax regression(5) to classify MNIST images. The goal of this exercise is to build a softmax classifier that you will be able to reuse in the future exercises and also on other classification problems that you might encounter.

In the file http://ufldl.stanford.edu/wiki/resources/softmax_exercise.zip,we have provided some starter code. You should write your code in the places indicated by "YOUR CODE HERE" in the files.

In the starter code, you will need to modify `softmaxCost.m` and `softmaxPredict.m` for this exercise.

We have also provided `softmaxExercise.m` that will help walk you through the steps in this exercise.

5.2.1 Dependencies

The following additional files are required for this exercise:

- MNIST Dataset http://yann.lecun.com/exdb/mnist/
- Support functions for loading MNIST in Matlab (A)
- Starter Code http://ufldl.stanford.edu/wiki/resources/softmax_exercise.zip

You will also need:

- `computeNumericalGradient.m` from Exercise:Sparse Autoencoder (2.7)

If you have not completed the exercises listed above, we strongly suggest you complete them first.

5.2.2 Step 0: Initialize constants and parameters

We've provided the code for this step in `softmaxExercise.m`.

Two constants, inputSize and numClasses, corresponding to the size of each input vector and the number of class labels have been defined in the starter code. This will allow you to reuse

your code on a different data set in a later exercise. We also initialize lambda, the weight decay parameter here.

5.2.3 Step 1: Load data

The starter code loads the MNIST images and labels into inputData and labels respectively. The images are pre-processed to scale the pixel values to the range $[0, 1]$, and the label 0 is remapped to 10 for convenience of implementation, so that the labels take values in $\{1, 2, \ldots, 10\}$. You will not need to change any code in this step for this exercise, but note that your code should be general enough to operate on data of arbitrary size belonging to any number of classes.

5.2.4 Step 2: Implement softmaxCost

In softmaxCost.m, implement code to compute the softmax cost function $J(\theta)$. Remember to include the weight decay term in the cost as well. Your code should also compute the appropriate gradients, as well as the predictions for the input data (which will be used in the cross-validation step later).

It is important to vectorize your code so that it runs quickly. We also provide several implementation tips below:

Note: In the provided starter code, `theta` is a matrix where each the jth row is θ_j^T

Implementation Tip: Computing the ground truth matrix - In your code, you may need to compute the ground truth matrix `M`, such that `M(r, c)` is 1 if $y^{(c)} = r$ and 0 otherwise. This can be done quickly, without a loop, using the MATLAB functions `sparse` and `full`. Specifically, the command `M = sparse(r, c, v)` creates a sparse matrix such that `M(r(i), c(i)) = v(i)` for all `i`. That is, the vectors `r` and `c` give the position of the elements whose values we wish to set, and `v` the corresponding values of the elements. Running full on a sparse matrix gives a "full" representation of the matrix for use (meaning that Matlab will no longer try to represent it as a sparse matrix in memory). The code for using `sparse` and `full` to compute the ground truth matrix has already been included in `softmaxCost.m`.

Implementation Tip: Preventing overflows - in softmax regression, you will have to compute the hypothesis

$$h(x^{(i)}) = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix} \tag{82}$$

When the products $\theta_i^T x^{(i)}$ are large, the exponential function $e^{\theta_i^T x^{(i)}}$ will become very large and possibly overflow. When this happens, you will not be able to compute your hypothesis. However, there is an easy solution - observe that we can multiply the top and bottom of the hypothesis by

some constant without changing the output:

$$h(x^{(i)}) = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix} \tag{83}$$

$$= \frac{e^{-\alpha}}{e^{-\alpha} \sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix} \tag{84}$$

$$= \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)} - \alpha}} \begin{bmatrix} e^{\theta_1^T x^{(i)} - \alpha} \\ e^{\theta_2^T x^{(i)} - \alpha} \\ \vdots \\ e^{\theta_k^T x^{(i)} - \alpha} \end{bmatrix} \tag{85}$$

$$\tag{86}$$

Hence, to prevent overflow, simply subtract some large constant value from each of the $\theta_j^T x^{(i)}$ terms before computing the exponential. In practice, for each example, you can use the maximum of the $\theta_j^T x^{(i)}$ terms as the constant. Assuming you have a matrix `M` containing these terms such that `M(r, c)` is $\theta_r^T x^{(c)}$, then you can use the following code to accomplish this:

```
% M is the matrix as described in the text
M = bsxfun(@minus, M, max(M, [], 1));
```

`max(M)` yields a row vector with each element giving the maximum value in that column. `bsxfun` (short for binary singleton expansion function) applies minus along each row of `M`, hence subtracting the maximum of each column from every element in the column.

Implementation Tip: Computing the predictions - you may also find `bsxfun` useful in computing your predictions - if you have a matrix `M` containing the $e^{\theta_j^T x^{(i)}}$ terms, such that `M(r, c)` contains the $e^{\theta_r^T x^{(c)}}$ term, you can use the following code to compute the hypothesis (by dividing all elements in each column by their column sum):

```
% M is the matrix as described in the text
M = bsxfun(@rdivide, M, sum(M))
```

The operation of `bsxfun` in this case is analogous to the earlier example.

5.2.5 Step 3: Gradient checking

Once you have written the softmax cost function, you should check your gradients numerically. In general, whenever implementing any learning algorithm, you should always check your gradients numerically before proceeding to train the model. The norm of the difference between the numerical gradient and your analytical gradient should be small, on the order of 10^{-9}.

Implementation Tip: Faster gradient checking - when debugging, you can speed up gradient checking by reducing the number of parameters your model uses. In this case, we have included code for reducing the size of the input data, using the first 8 pixels of the images instead of the full 28×28 images. This code can be used by setting the variable `DEBUG` to true, as described in step 1 of the code.

5.2.6 Step 4: Learning parameters

Now that you've verified that your gradients are correct, you can train your softmax model using the function `softmaxTrain` in `softmaxTrain.m`. `softmaxTrain` which uses the L-BFGS algorithm, in the function `minFunc`. Training the model on the entire MNIST training set of 60000 28×28 images should be rather quick, and take less than 5 minutes for 100 iterations.

Factoring `softmaxTrain` out as a function means that you will be able to easily reuse it to train softmax models on other data sets in the future by invoking the function with different parameters.

Use the following parameter when training your softmax classifier:

```
lambda = 1e-4
```

5.2.7 Step 5: Testing

Now that you've trained your model, you will test it against the MNIST test set, comprising 10000 28×28 images. However, to do so, you will first need to complete the function `softmaxPredict` in `softmaxPredict.m`, a function which generates predictions for input data under a trained softmax model.

Once that is done, you will be able to compute the accuracy (the proportion of correctly classified images) of your model using the code provided. Our implementation achieved an accuracy of 92.6%. If your model's accuracy is significantly less (less than 91%), check your code, ensure that you are using the trained weights, and that you are training your model on the full 60000 training images. Conversely, if your accuracy is too high (99-100%), ensure that you have not accidentally trained your model on the test set as well.

6 自我学习与无监督特征学习

6.1 自我学习

6.1.1 综述

如果已经有一个足够强大的机器学习算法，为了获得更好的性能，最靠谱的方法之一是给这个算法以更多的数据。机器学习界甚至有个说法：“有时候胜出者并非有最好的算法，而是有更多的数据。”

人们总是可以尝试获取更多的已标注数据，但是这样做成本往往很高。例如研究人员已经花了相当的精力在使用类似 AMT(Amazon Mechanical Turk) 这样的工具上，以期获取更大的训练数据集。相比大量研究人员通过手工方式构建特征，用众包的方式让多人手工标数据是一个进步，但是我们可以做得更好。具体的说，如果算法能够从未标注数据中学习，那么我们就可以轻易地获取大量无标注数据，并从中学习。自学习和无监督特征学习就是这种的算法。尽管一个单一的未标注样本蕴含的信息比一个已标注的样本要少，但是如果能获取大量无标注数据（比如从互联网上下载随机的、无标注的图像、音频剪辑或者是文本），并且算法能够有效的利用它们，那么相比大规模的手工构建特征和标数据，算法将会取得更好的性能。

在自学习和无监督特征学习问题上，可以给算法以大量的未标注数据，学习出较好的特征描述。在尝试解决一个具体的分类问题时，可以基于这些学习出的特征描述和任意的（可能比较少的）已标注数据，使用有监督学习方法完成分类。

在一些拥有大量未标注数据和少量的已标注数据的场景中，上述思想可能是最有效的。即使在只有已标注数据的情况下（这时我们通常忽略训练数据的类标号进行特征学习），以上想法也能得到很好的结果。

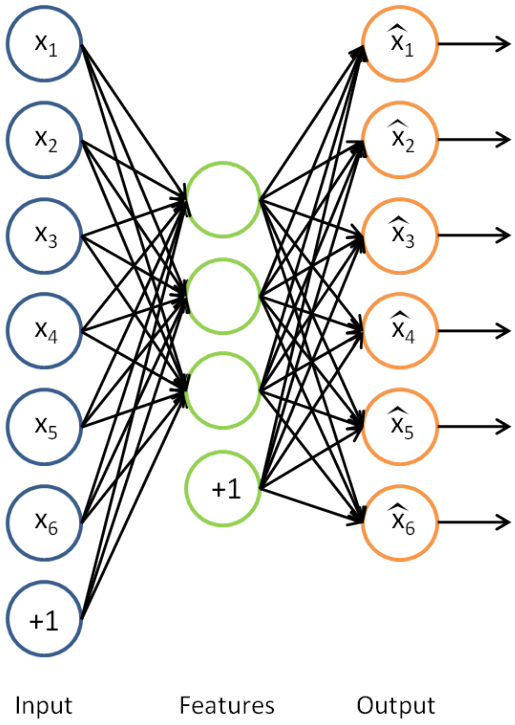
6.1.2 特征学习

我们已经了解到如何使用一个自编码器（autoencoder）从无标注数据中学习特征。具体来说，假定有一个无标注的训练数据集 $\{x_u^{(1)}, x_u^{(2)}, \ldots, x_u^{(m_u)}\}$（下标 u 代表“不带类标”）。现在用它们训练一个稀疏自编码器（可能需要首先对这些数据做白化或其它适当的预处理）。

利用训练得到的模型参数 $W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$，给定任意的输入数据 x，可以计算隐藏单元的激活量（activations）a。如前所述，相比原始输入 x 来说，a 可能是一个更好的特征描述。下图的神经网络描述了特征（激活量 a）的计算。

这实际上就是之前得到的稀疏自编码器，在这里去掉了最后一层。

假定有大小为 m_l 的已标注训练集 $\{(x_l^{(1)}, y^{(1)}), (x_l^{(2)}, y^{(2)}), \ldots (x_l^{(m_l)}, y^{(m_l)})\}$（下标 l 表示“带类标”），我们可以为输入数据找到更好的特征描述。例如，可以将 $x_l^{(1)}$ 输入到稀疏自编码器，得到隐藏单元激活量 $a_l^{(1)}$。接下来，可以直接使用 $a_l^{(1)}$ 来**代替**原始数据 $x_l^{(1)}$ （“替代表示”,Replacement Representation）。也可以合二为一，使用新的向量 $(x_l^{(1)}, a_l^{(1)})$ 来代替原始数据 $x_l^{(1)}$（“级联表示”,Concatenation Representation）。



经过变换后，训练集就变成 $\{(a_l^{(1)}, y^{(1)}), (a_l^{(2)}, y^{(2)}), \ldots (a_l^{(m_l)}, y^{(m_l)})\}$ 或者是 $\{((x_l^{(1)}, a_l^{(1)}), y^{(1)}), ((x_l^{(2)}, a_l^{(1)}), y$（取决于使用 $a_l^{(1)}$ 替换 $x_l^{(1)}$ 还是将二者合并）。在实践中，将 $a_l^{(1)}$ 和 $x_l^{(1)}$ 合并通常表现的更好。但是考虑到内存和计算的成本，也可以使用替换操作。

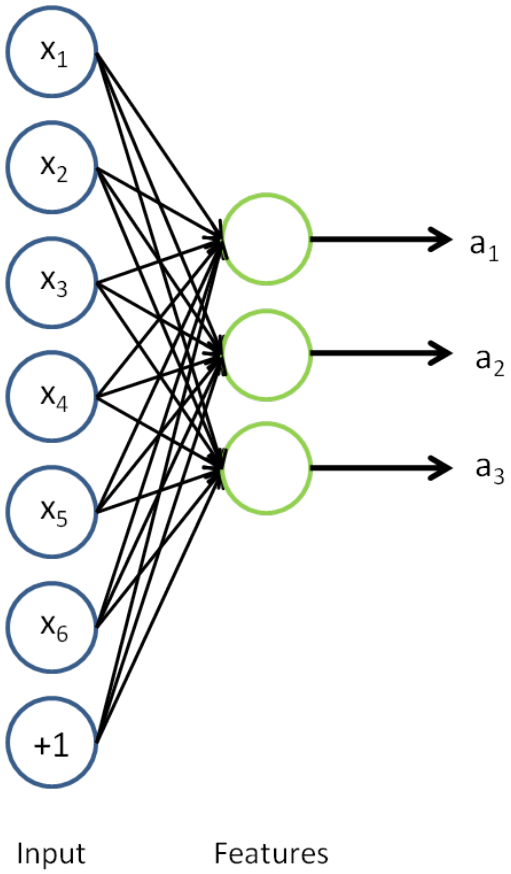
最终，可以训练出一个有监督学习算法（例如 svm, logistic regression 等），得到一个判别函数对 y 值进行预测。预测过程如下：给定一个测试样本 x_{test}，重复之前的过程，将其送入稀疏自编码器，得到 a_{test}。然后将 a_{test} （或者 $(x_{\text{test}}, a_{\text{test}})$ ）送入分类器中，得到预测值。

6.1.3 数据预处理

在特征学习阶段，我们从未标注训练集 $\{x_u^{(1)}, x_u^{(2)}, \ldots, x_u^{(m_u)}\}$ 中学习，这一过程中可能计算了各种数据预处理参数。例如计算数据均值并且对数据做均值标准化（mean normalization）；或者对原始数据做主成分分析（PCA），然后将原始数据表示为 $U^T x$ (又或者使用 PCA 白化或 ZCA 白化)。这样的话，有必要将这些参数保存起来，并且在后面的训练和测试阶段使用同样的参数，以保证数据进入稀疏自编码神经网络之前经过了同样的变换。例如，如果对未标注数据集进行 PCA 预处理，就必须将得到的矩阵 U 保存起来，并且应用到有标注训练集和测试集上；而不能使用有标注训练集重新估计出一个不同的矩阵 U （也不能重新计算均值并做均值标准化），否则的话可能得到一个完全不一致的数据预处理操作，导致进入自编码器的数据分布迥异于训练自编码器时的数据分布。

6.1.4 无监督特征学习的术语

有两种常见的无监督特征学习方式，区别在于你有什么样的未标注数据。**自学习** (self-taught learning) 是其中更为一般的、更强大的学习方式，它不要求未标注数据 x_u 和已标注数据 x_l 来



自同样的分布。另外一种带限制性的方式也被称为**半监督学习**，它要求 x_u 和 x_l 服从同样的分布。下面通过例子解释二者的区别。

假定有一个计算机视觉方面的任务，目标是区分汽车和摩托车图像；也即训练样本里面要么是汽车的图像，要么是摩托车的图像。哪里可以获取大量的未标注数据呢？最简单的方式可能是从互联网上下载一些随机的图像数据集，在这些数据上训练出一个稀疏自编码器，从中得到有用的特征。这个例子里，未标注数据完全来自于一个和已标注数据不同的分布（未标注数据集中，或许其中一些图像包含汽车或者摩托车，但是不是所有的图像都如此）。这种情形被称为自学习。

相反，如果有大量的未标注图像数据，要么是汽车图像，要么是摩托车图像，仅仅是缺失了类标号（没有标注每张图片到底是汽车还是摩托车）。也可以用这些未标注数据来学习特征。这种方式，即要求未标注样本和带标注样本服从相同的分布，有时候被称为半监督学习。在实践中，常常无法找到满足这种要求的未标注数据（到哪里找到一个每张图像不是汽车就是摩托车，只是丢失了类标号的图像数据库?）因此，自学习在无标注数据集的特征学习中应用更广。

6.2 练习：自学习

6.2.1 综述

In this exercise, we will use the self-taught learning paradigm with the sparse autoencoder and softmax classifier to build a classifier for handwritten digits.

You will be building upon your code from the earlier exercises. First, you will train your sparse autoencoder on an "unlabeled" training dataset of handwritten digits. This produces feature that are penstroke-like. We then extract these learned features from a labeled dataset of handwritten digits. These features will then be used as inputs to the softmax classifier that you wrote in the previous exercise.

Concretely, for each example in the the labeled training dataset x_l, we forward propagate the example to obtain the activation of the hidden units $a^{(2)}$. We now represent this example using $a^{(2)}$ (the "replacement" representation), and use this to as the new feature representation with which to train the softmax classifier.

Finally, we also extract the same features from the test data to obtain predictions.

In this exercise, our goal is to distinguish between the digits from 0 to 4. We will use the digits 5 to 9 as our "unlabeled" dataset which which to learn the features; we will then use a labeled dataset with the digits 0 to 4 with which to train the softmax classifier.

In the starter code, we have provided a file `stlExercise.m` that will help walk you through the steps in this exercise.

6.2.2 Dependencies

The following additional files are required for this exercise:

- MNIST Dataset `http://yann.lecun.com/exdb/mnist/`
- Support functions for loading MNIST in Matlab (A)
- Starter Code stl_exercise.zip)

You will also need your code from the following exercises:

- Exercise:Sparse Autoencoder (2.7)
- Exercise:Vectorization (3.4)
- Exercise:Softmax Regression (5.2)

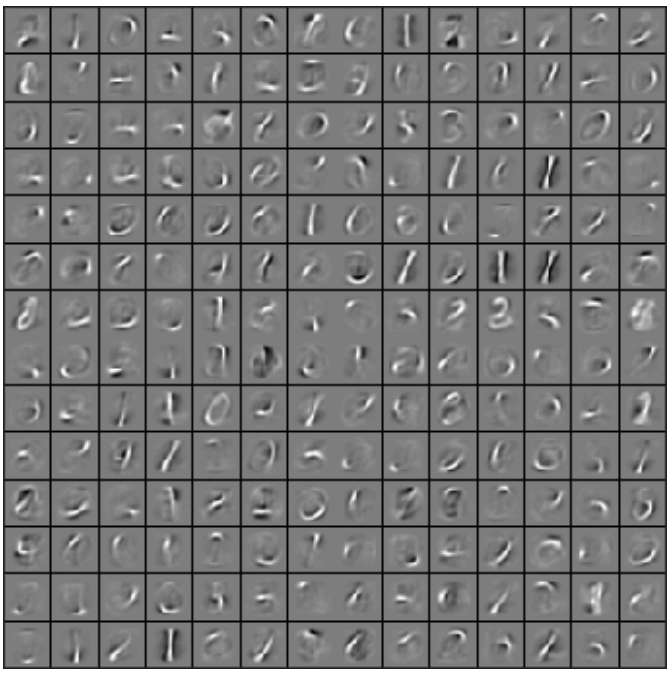
If you have not completed the exercises listed above, we strongly suggest you complete them first.

6.2.3 Step 1: Generate the input and test data sets

Download and decompress stl_exercise.zip, which contains starter code for this exercise. Additionally, you will need to download the datasets from the MNIST Handwritten Digit Database for this project.

6.2.4 Step 2: Train the sparse autoencoder

Next, use the unlabeled data (the digits from 5 to 9) to train a sparse autoencoder, using the same `sparseAutoencoderCost.m` function as you had written in the previous exercise. (From the earlier exercise, you should have a working and vectorized implementation of the sparse autoencoder.) For us, the training step took less than 25 minutes on a fast desktop. When training is complete, you should get a visualization of pen strokes like the image shown below:



Informally, the features learned by the sparse autoencoder should correspond to penstrokes.

6.2.5 Step 3: Extracting features

After the sparse autoencoder is trained, you will use it to extract features from the handwritten digit images.

Complete `feedForwardAutoencoder.m` to produce a matrix whose columns correspond to activations of the hidden layer for each example, i.e., the vector $a^{(2)}$ corresponding to activation of layer 2. (Recall that we treat the inputs as layer 1).

After completing this step, calling `feedForwardAutoencoder.m` should convert the raw image data to hidden unit activations $a^{(2)}$.

6.2.6 Step 4: Training and testing the logistic regression model

Use your code from the softmax exercise (`softmaxTrain.m`) to train a softmax classifier using the training set features (`trainFeatures`) and labels (`trainLabels`).

6.2.7 Step 5: Classifying on the test set

Finally, complete the code to make predictions on the test set (`testFeatures`) and see how your learned features perform! If you've done all the steps correctly, you should get an accuracy of about 98% percent.

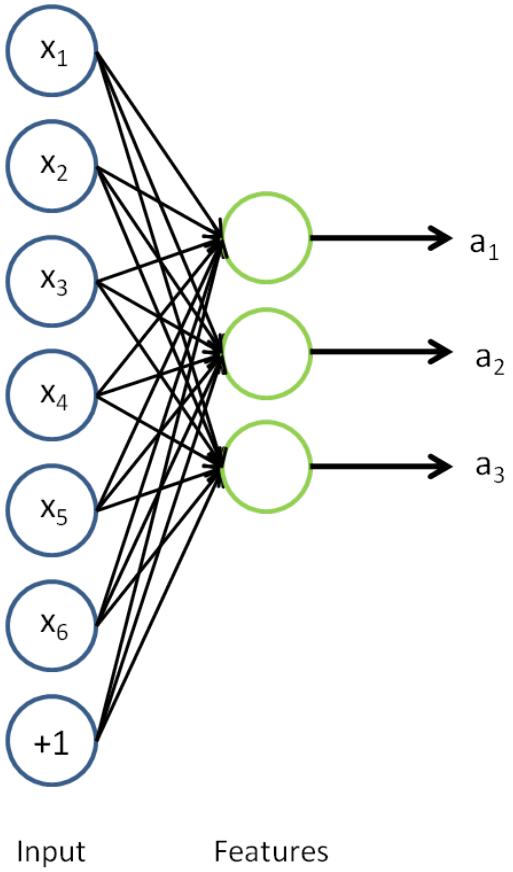
As a comparison, when raw pixels are used (instead of the learned features), we obtained a test accuracy of only around 96% (for the same train and test sets).

7 建立分类用深度网络

7.1 从自我学习到深层网络

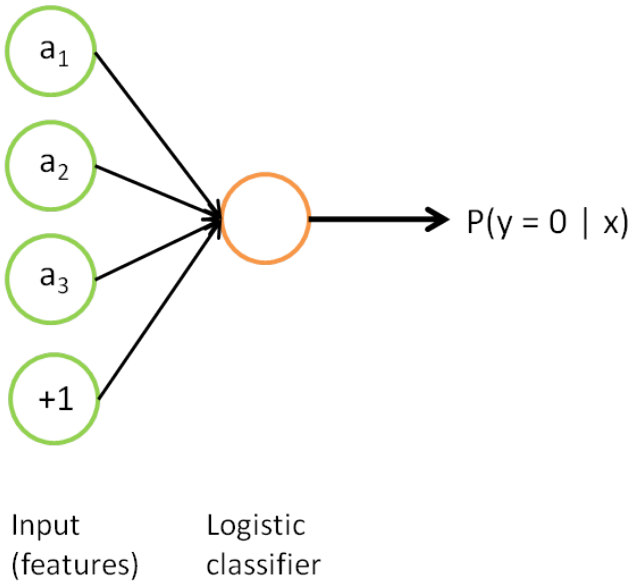
在前一节 (6) 中，我们利用自编码器来学习输入至 softmax 或 logistic 回归分类器的特征。这些特征仅利用未标注数据学习获得。在本节中，我们描述如何利用已标注数据进行**微调**，从而进一步优化这些特征。如果有大量已标注数据，通过微调就可以显著提升分类器的性能。

在自我学习中，我们首先利用未标注数据训练一个稀疏自编码器。随后，给定一个新样本 x，我们通过隐含层提取出特征 a。上述过程图示如下：



我们感兴趣的是分类问题，目标是预测样本的类别标号 y。我们拥有标注数据集 $\{(x_l^{(1)}, y^{(1)}), (x_l^{(2)}, y^{(2)}), \ldots (x_l^{(m_l)}, y^{(m_l)})\}$，包含 m_l 个标注样本。此前我们已经说明，可以利用稀疏自编码器获得的特征 $a^{(l)}$ 来替代原始特征。这样就可获得训练数据集 $\{(a^{(1)}, y^{(1)}), \ldots (a^{(m_l)}, y^{(m_l)})\}$。最终，我们训练出一个从特征 $a^{(i)}$ 到类标号 $y^{(i)}$ 的 logistic 分类器。为说明这一过程，我们按照神经网络一节 (2.1) 中的方式，用下图描述 logistic 回归单元（橘黄色）。

考虑利用这个方法所学到的分类器（输入 -输出映射）。它描述了一个把测试样本 x 映射到预测值 $p(y = 1|x)$ 的函数。将此前的两张图片结合起来，就得到该函数的图形表示。也即，最终的分类器可以表示为：



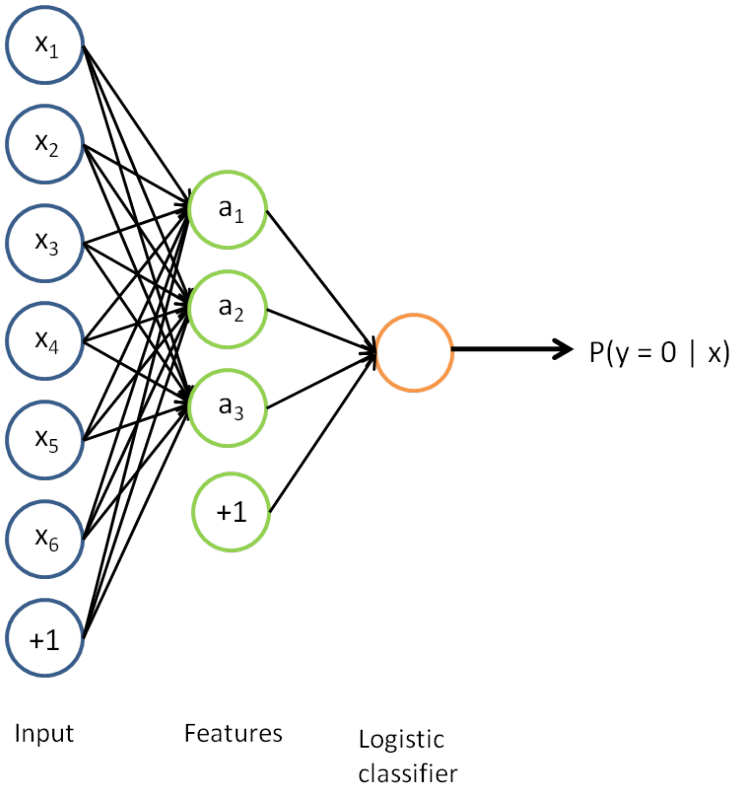
该模型的参数通过两个步骤训练获得：在该网络的第一层，将输入 x 映射至隐藏单元激活量 a 的权值 $W^{(1)}$ 可以通过稀疏自编码器训练过程获得。在第二层，将隐藏单元 a 映射至输出 y 的权值 $W^{(2)}$ 可以通过 logistic 回归或 softmax 回归训练获得。

这个最终分类器整体上显然是一个大的神经网络。因此，在训练获得模型最初参数（利用自动编码器训练第一层，利用 logistic/softmax 回归训练第二层）之后，我们可以进一步修正模型参数，进而降低训练误差。具体来说，我们可以对参数进行微调，在现有参数的基础上采用梯度下降或者 L-BFGS 来降低已标注样本集 $\{(x_l^{(1)}, y^{(1)}), (x_l^{(2)}, y^{(2)}), \ldots (x_l^{(m_l)}, y^{(m_l)})\}$ 上的训练误差。

使用微调时，初始的非监督特征学习步骤（也就是自动编码器和 logistic 分类器训练）有时候被称为**预训练**。微调的作用在于，已标注数据集也可以用来修正权值 $W^{(1)}$，这样可以对隐藏单元所提取的特征 a 做进一步调整。

到现在为止，我们描述上述过程时，都假设采用了“替代 (Replacement)”表示而不是“级联 (Concatenation)”表示。在替代表示中，logistic 分类器所看到的训练样本格式为 $(a^{(i)}, y^{(i)})$；而在级联表示中，分类器所看到的训练样本格式为 $((x^{(i)}, a^{(i)}), y^{(i)})$。对级联表示同样可以进行微调（在级联表示神经网络中，输入值 x_i 也直接被输入至 logistic 分类器。对此前的神经网络示意图稍加更改，即可获得其示意图。具体的说，第一层的输入节点除了与隐层联接之外，还将越过隐层，与第三层输出节点直接相连）。但是对于微调来说，级联表示相对于替代表示几乎没有优势。因此，如果需要开展微调，我们通常使用替代表示的网络（但是如果不开展微调，级联表示的效果有时候会好得多）。

在什么时候应用微调？通常仅在有大量已标注训练数据的情况下使用。在这样的情况下，微调能显著提升分类器性能。然而，如果有大量**未标注**数据集（用于非监督特征学习/预训练），却只有相对较少的已标注训练集，微调的作用非常有限。



7.2 深度网络概览

7.2.1 概览

在之前的章节中，你已经构建了一个包括输入层、隐藏层以及输出层的三层神经网络。虽然该网络对于 MNIST 手写数字数据库非常有效，但是它还是一个非常**“浅”**的网络。这里的“浅”指的是特征（隐藏层的激活值 $a^{(2)}$）只使用一层计算单元（隐藏层）来得到的。

在本节中，我们开始讨论**深度**神经网络，即含有多个隐藏层的神经网络。通过引入深度网络，我们可以计算更多复杂的输入特征。因为每一个隐藏层可以对上一层的输出进行非线性变换，因此深度神经网络拥有比“浅层”网络更加优异的表达能力（例如可以学习到更加复杂的函数关系）。

值得注意的是当训练深度网络的时候，每一层隐层应该使用**非线性**的激活函数 $f(\cdot)$。这是因为多层的线性函数组合在一起本质上也只有线性函数的表达能力（例如，将多个线性方程组合在一起仅仅产生另一个线性方程）。因此，在激活函数是线性的情况下，相比于单隐藏层神经网络，包含多隐藏层的深度网络并没有增加表达能力。

7.2.2 深度网络的优势

为什么我们要使用深度网络呢？使用深度网络最主要的优势在于，它能以更加紧凑简洁的方式来表达比浅层网络大得多的函数集合。正式点说，我们可以找到一些函数，这些函数可以用 k 层网络简洁地表达出来（这里的简洁是指隐层单元的数目只需与输入单元数目呈**多项式关**

系）。但是对于一个只有 $k-1$ 层的网络而言，除非它使用与输入单元数目呈指数关系的隐层单元数目，否则不能简洁表达这些函数。

举一个简单的例子，比如我们打算构建一个布尔网络来计算 n 个输入比特的奇偶校验码（或者进行异或运算）。假设网络中的每一个节点都可以进行逻辑“或”运算（或者“与非”运算），亦或者逻辑“与”运算。如果我们拥有一个仅仅由一个输入层、一个隐层以及一个输出层构成的网络，那么该奇偶校验函数所需要的节点数目与输入层的规模 n 呈指数关系。但是，如果我们构建一个更深点的网络，那么这个网络的规模就可做到仅仅是 n 的多项式函数。

当处理对象是图像时，我们能够使用深度网络学习到“部分 -整体”的分解关系。例如，第一层可以学习如何将图像中的像素组合在一起来检测边缘（正如我们在前面的练习中做的那样）。第二层可以将边缘组合起来检测更长的轮廓或者简单的“目标的部件”。在更深的层次上，可以将这些轮廓进一步组合起来以检测更为复杂的特征。

最后要提的一点是，大脑皮层同样是分多层进行计算的。例如视觉图像在人脑中是分多个阶段进行处理的，首先是进入大脑皮层的“V1”区，然后紧跟着进入大脑皮层“V2”区，以此类推。

7.2.3 训练深度网络的困难

虽然几十年前人们就发现了深度网络在理论上的简洁性和较强的表达能力，但是直到最近，研究者们也没有在训练深度网络方面取得多少进步。

问题原因在于研究者们主要使用的学习算法是：首先随机初始化深度网络的权重，然后使用有监督的目标函数在有标签的训练集 $\left\{\left(x_l^{(1)}, y^{(1)}\right), \ldots, \left(x_l^{(m_l)}, y^{(m_l)}\right)\right\}$ 上进行训练。例如通过使用梯度下降法来降低训练误差。然而，这种方法通常不是十分奏效。这其中有如下几方面原因：

数据获取问题

使用上面提到的方法，我们需要依赖于有标签的数据才能进行训练。然而有标签的数据通常是稀缺的，因此对于许多问题，我们很难获得足够多的样本来拟合一个复杂模型的参数。例如，考虑到深度网络具有强大的表达能力，在不充足的数据上进行训练将会导致过拟合。

局部极值问题

使用监督学习方法来对浅层网络（只有一个隐藏层）进行训练通常能够使参数收敛到合理的范围内。但是当用这种方法来训练深度网络的时候，并不能取得很好的效果。特别的，使用监督学习方法训练神经网络时，通常会涉及到求解一个高度非凸的优化问题（例如最小化训练误差 $\sum_i ||h_W\left(x^{(i)}\right) - y^{(i)}||^2$，其中参数 W 是要优化的参数。对深度网络而言，这种非凸优化问题的搜索区域中充斥着大量“坏”的局部极值，因而使用梯度下降法（或者像共轭梯度下降法，L-BFGS 等方法）效果并不好。

梯度弥散问题

梯度下降法（以及相关的 L-BFGS 算法等）在使用随机初始化权重的深度网络上效果不好的技术原因是：梯度会变得非常小。具体而言，当使用反向传播方法计算导数的时候，随着网络的深度的增加，反向传播的梯度（从输出层到网络的最初几层）的幅度值会急剧地减小。结

果就造成了整体的损失函数相对于最初几层的权重的导数非常小。这样，当使用梯度下降法的时候，最初几层的权重变化非常缓慢，以至于它们不能够从样本中进行有效的学习。这种问题通常被称为“梯度的弥散”.

与梯度弥散问题紧密相关的问题是：当神经网络中的最后几层含有足够数量神经元的时候，可能单独这几层就足以对有标签数据进行建模，而不用最初几层的帮助。因此，对所有层都使用随机初始化的方法训练得到的整个网络的性能将会与训练得到的浅层网络（仅由深度网络的最后几层组成的浅层网络）的性能相似。

7.2.4 逐层贪婪训练方法

那么，我们应该如何训练深度网络呢？**逐层贪婪训练**方法是取得一定成功的一种方法。我们会在后面的章节中详细阐述这种方法的细节。简单来说，逐层贪婪算法的主要思路是每次只训练网络中的一层，即我们首先训练一个只含一个隐藏层的网络，仅当这层网络训练结束之后才开始训练一个有两个隐藏层的网络，以此类推。在每一步中，我们把已经训练好的前 $k-1$ 层固定，然后增加第 k 层（也就是将我们已经训练好的前 $k-1$ 的输出作为输入）。每一层的训练可以是有监督的（例如，将每一步的分类误差作为目标函数），但更通常使用无监督方法（例如自动编码器，我们会在后边的章节中给出细节）。这些各层单独训练所得到的权重被用来初始化最终（或者说全部）的深度网络的权重，然后对整个网络进行“微调”（即把所有层放在一起来优化有标签训练集上的训练误差）.

逐层贪婪的训练方法取得成功要归功于以下几方面：

数据获取

虽然获取有标签数据的代价是昂贵的，但获取大量的无标签数据是容易的。自学习方法（self-taught learning）的潜力在于它能通过使用大量的无标签数据来学习到更好的模型。具体而言，该方法使用无标签数据来学习得到所有层（不包括用于预测标签的最终分类层）$W^{(l)}$ 的最佳初始权重。相比纯监督学习方法，这种自学习方法能够利用多得多的数据，并且能够学习和发现数据中存在的模式。因此该方法通常能够提高分类器的性能。

更好的局部极值

当用无标签数据训练完网络后，相比于随机初始化而言，各层初始权重会位于参数空间中较好的位置上。然后我们可以从这些位置出发进一步微调权重。从经验上来说，以这些位置为起点开始梯度下降更有可能收敛到比较好的局部极值点，这是因为无标签数据已经提供了大量输入数据中包含的模式的先验信息。

在下一节中，我们将会具体阐述如何进行逐层贪婪训练。

7.3 栈式自编码算法

7.3.1 概述

逐层贪婪训练法依次训练网络的每一层，进而预训练整个深度神经网络。在本节中，我们将会学习如何将自编码器“栈化”到逐层贪婪训练法中，从而预训练（或者说初始化）深度神经网

络的权重。

栈式自编码神经网络是一个由多层稀疏自编码器组成的神经网络，其前一层自编码器的输出作为其后一层自编码器的输入。对于一个 n 层栈式自编码神经网络，我们沿用自编码器一章的各种符号，假定用 $W^{(k,1)}, W^{(k,2)}, b^{(k,1)}, b^{(k,2)}$ 表示第 k 个自编码器对应的 $W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)}$ 参数，那么该栈式自编码神经网络的编码过程就是，按照从前向后的顺序执行每一层自编码器的编码步骤:

$$a^{(l)} = f(z^{(l)}) \tag{87}$$

$$z^{(l+1)} = W^{(l,1)}a^{(l)} + b^{(l,1)} \tag{88}$$

同理，栈式神经网络的解码过程就是，按照从后向前的顺序执行每一层自编码器的解码步骤:

$$a^{(n+l)} = f(z^{(n+l)}) \tag{89}$$

$$z^{(n+l+1)} = W^{(n-l,2)}a^{(n+l)} + b^{(n-l,2)} \tag{90}$$

其中，$a^{(n)}$ 是最深层隐藏单元的激活值，其包含了我们感兴趣的信息，这个向量也是对输入值的更高阶的表示。

通过将 $a^{(n)}$ 作为 softmax 分类器的输入特征，可以将栈式自编码神经网络中学到的特征用于分类问题。

训练

一种比较好的获取栈式自编码神经网络参数的方法是采用逐层贪婪训练法进行训练。即先利用原始输入来训练网络的第一层，得到其参数 $W^{(1,1)}, W^{(1,2)}, b^{(1,1)}, b^{(1,2)}$；然后网络第一层将原始输入转化成为由隐藏单元激活值组成的向量（假设该向量为 A），接着把 A 作为第二层的输入，继续训练得到第二层的参数 $W^{(2,1)}, W^{(2,2)}, b^{(2,1)}, b^{(2,2)}$；最后，对后面的各层同样采用的策略，即将前层的输出作为下一层输入的方式依次训练。

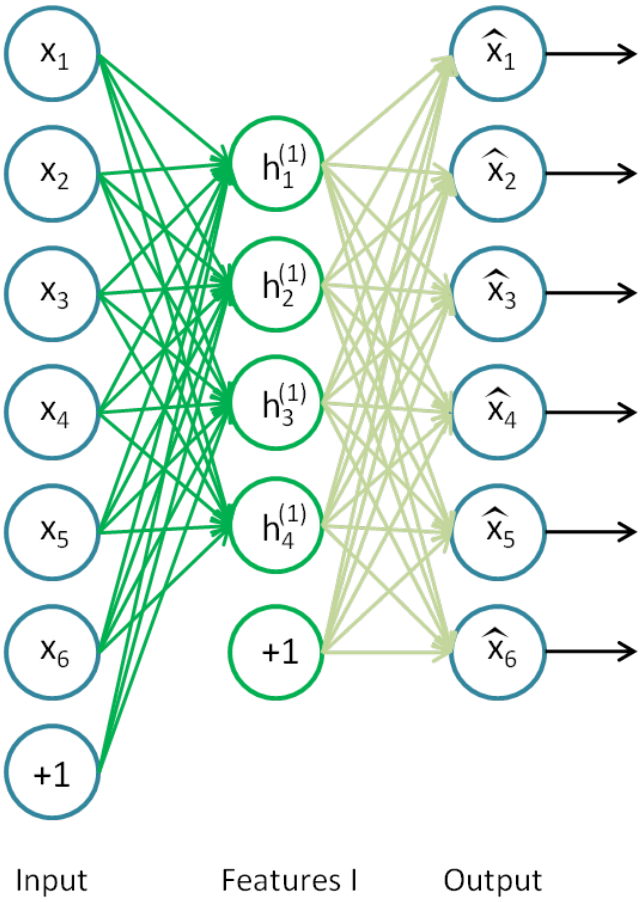
对于上述训练方式，在训练每一层参数的时候，会固定其它各层参数保持不变。所以，如果想得到更好的结果，在上述预训练过程完成之后，可以通过反向传播算法同时调整所有层的参数以改善结果，这个过程一般被称作“微调（fine-tuning）”。实际上，使用逐层贪婪训练方法将参数训练到快要收敛时，应该使用微调 (7.4)。反之，如果直接在随机化的初始权重上使用微调，那么会得到不好的结果，因为参数会收敛到局部最优。

如果你只对以分类为目的的微调感兴趣，那么惯用的做法是丢掉栈式自编码网络的“解码”层，直接把最后一个隐藏层的 $a^{(n)}$ 作为特征输入到 softmax 分类器进行分类，这样，分类器（softmax）的分类错误的梯度值就可以直接反向传播给编码层了。

具体实例

让我们来看个具体的例子，假设你想要训练一个包含两个隐含层的栈式自编码网络，用来进行 MNIST 手写数字分类（这将会是你的下一个练习）。

首先，你需要用原始输入 $x^{(k)}$ 训练第一个自编码器，它能够学习得到原始输入的一阶特征表示 $h^{(1)(k)}$（如下图所示）。



接着，你需要把原始数据输入到上述训练好的稀疏自编码器中，对于每一个输入 $x^{(k)}$，都可以得到它对应的一阶特征表示 $h^{(1)(k)}$。然后你再用这些一阶特征作为另一个稀疏自编码器的输入，使用它们来学习二阶特征 $h^{(2)(k)}$。(如下图所示)

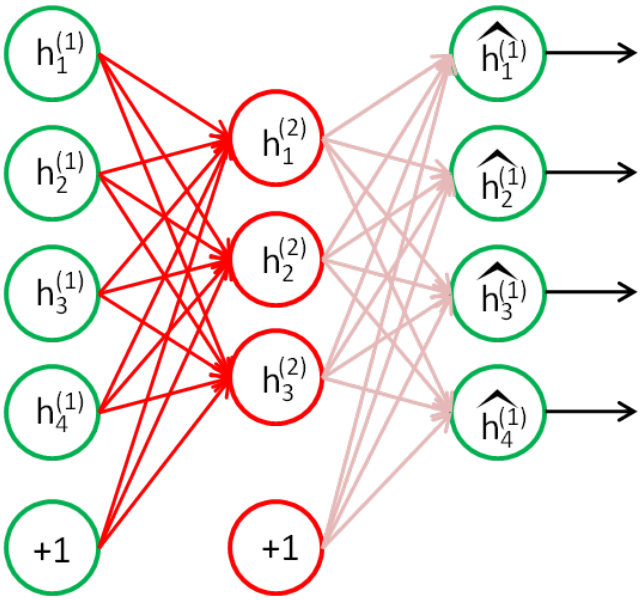
同样，再把一阶特征输入到刚训练好的第二层稀疏自编码器中，得到每个 $h^{(1)(k)}$ 对应的二阶特征激活值 $h^{(2)(k)}$。接下来，你可以把这些二阶特征作为 softmax 分类器的输入，训练得到一个能将二阶特征映射到数字标签的模型。

如下图所示，最终，你可以将这三层结合起来构建一个包含两个隐藏层和一个最终 softmax 分类器层的栈式自编码网络，这个网络能够如你所愿地对 MNIST 数字进行分类。

讨论

栈式自编码神经网络具有强大的表达能力及深度神经网络的所有优点。

更进一步，它通常能够获取到输入的“层次型分组”或者“部分 -整体分解”结构。为了弄清这一点，回顾一下，自编码器倾向于学习得到能更好地表示输入数据的特征。因此，栈式自编码神经网络的第一层会学习得到原始输入的一阶特征（比如图片里的边缘），第二层会学习得到二阶特征，该特征对应一阶特征里包含的一些模式（比如在构成轮廓或者角点时，什么样的边缘会共现）。栈式自编码神经网络的更高层还会学到更高阶的特征。举个例子，如果网络的输



入数据是图像，网络的第一层会学习如何去识别边，第二层一般会学习如何去组合边，从而构成轮廓、角等。更高层会学习如何去组合更形象且有意义的特征。例如，如果输入数据集包含人脸图像，更高层会学习如何识别或组合眼睛、鼻子、嘴等人脸器官。

7.4 微调多层自编码算法

7.4.1 介绍

微调是深度学习中的常用策略，可以大幅提升一个栈式自编码神经网络的性能表现。从更高的视角来讲，微调将栈式自编码神经网络的所有层视为一个模型，这样在每次迭代中，网络中所有的权重值都可以被优化。

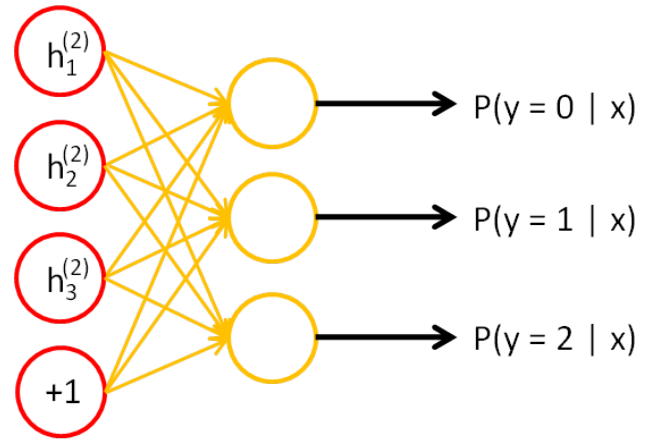
7.4.2 一般策略

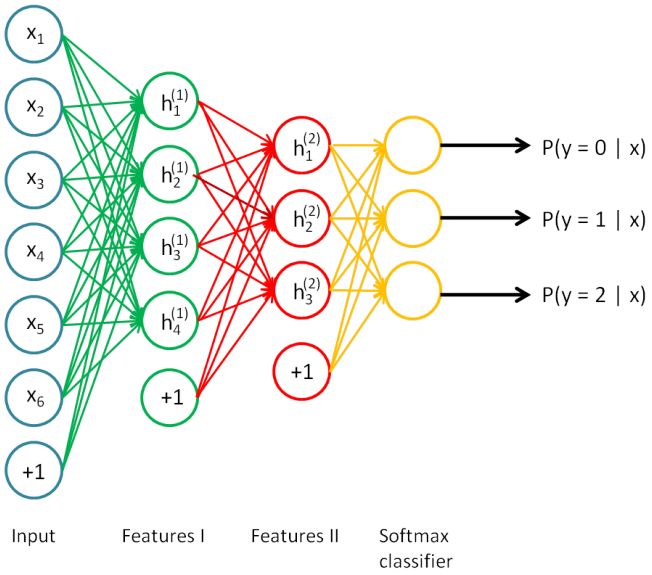
幸运的是，实施微调栈式自编码神经网络所需的工具都已齐备！为了在每次迭代中计算所有层的梯度，我们需要使用稀疏自动编码一节中讨论的反向传播算法 (2.2)。因为反向传播算法可以延伸应用到任意多层，所以事实上，该算法对任意多层的栈式自编码神经网络都适用。

7.4.3 使用反向传播法进行微调

为方便读者，以下我们简要描述如何实施反向传播算法：

1. 进行一次前馈传递，对 L_2 层、L_3 层直到输出层 L_{n_l}，使用前向传播步骤中定义的公式计算各层上的激活值（激励响应）。





2. 对输出层（n_l 层），令

$$\delta^{(n_l)} = -(\nabla_{a^{n_l}} J) \bullet f'(z^{(n_l)}) \tag{91}$$

（当使用 softmax 分类器时，softmax 层满足：$\nabla J = \theta^T(I - P)$，其中 I 为输入数据对应的类别标签，P 为条件概率向量。）

3. 对 $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$; 令

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)}) \tag{92}$$

4. 计算所需的偏导数：

$$\nabla_{W^{(l)}} J(W,b;x,y) = \delta^{(l+1)}(a^{(l)})^T, \tag{93}$$

$$\nabla_{b^{(l)}} J(W,b;x,y) = \delta^{(l+1)}. \tag{94}$$

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] \tag{95}$$

注：我们可以认为输出层 softmax 分类器是附加上的一层，但是其求导过程需要单独处理。具体地说，网络“最后一层”的特征会进入 softmax 分类器。所以，第二步中的导数由 $\delta^{(n_l)} = -(\nabla_{a^{n_l}} J) \bullet f'(z^{(n_l)})$ 计算，其中 $\nabla J = \theta^T(I-P)$。

7.5 练习：实现数字分类的深度网络

7.5.1 Overview

In this exercise, you will use a stacked autoencoder for digit classification. This exercise is very similar to the self-taught learning exercise, in which we trained a digit classifier using a autoencoder layer followed by a softmax layer. The only difference in this exercise is that we will be using two autoencoder layers instead of one and further finetune the two layers.

The code you have already implemented will allow you to stack various layers and perform layer-wise training. However, to perform fine-tuning, you will need to implement backpropogation through both layers. We will see that fine-tuning significantly improves the model's performance.

In the file stackedae_exercise.zip, we have provided some starter code. You will need to complete the code in `stackedAECost.m`, `stackedAEPredict.m` and `stackedAEExercise.m`. We have also provided `params2stack.m` and `stack2params.m` which you might find helpful in constructing deep networks.

7.5.2 Dependencies

The following additional files are required for this exercise:

- MNIST Dataset http://yann.lecun.com/exdb/mnist/
- Support functions for loading MNIST in Matlab (A)
- Starter Code (stackedae_exercise.zip)

You will also need your code from the following exercises:

- Exercise:Sparse Autoencoder (2.7)
- Exercise:Vectorization (3.4)
- Exercise:Softmax Regression (5.2)
- Exercise:Self-Taught Learning (6.2)

If you have not completed the exercises listed above, we strongly suggest you complete them first.

7.5.3 Step 0: Initialize constants and parameters

Open `stackedAEExercise.m`. In this step, we set meta-parameters to the same values that were used in previous exercise, which should produce reasonable results. You may to modify the meta-parameters if you wish.

7.5.4 Step 1: Train the data on the first stacked autoencoder

Train the first autoencoder on the training images to obtain its parameters. This step is identical to the corresponding step in the sparse autoencoder and STL assignments, complete this part of the code so as to learn a first layer of features using your `sparseAutoencoderCost.m` and `minFunc`.

7.5.5 Step 2: Train the data on the second stacked autoencoder

We first forward propagate the training set through the first autoencoder (using `feedForwardAutoenco` that you completed in Exercise:Self-Taught Learning(6.2)) to obtain hidden unit activations. These activations are then used to train the second sparse autoencoder. Since this is just an adapted application of a standard autoencoder, it should run similarly with the first. Complete this part of the code so as to learn a first layer of features using your `sparseAutoencoderCost.m` and `minFunc`.

This part of the exercise demonstrates the idea of greedy layerwise training with the same learning algorithm reapplied multiple times.

7.5.6 Step 3: Train the softmax classifier on the L2 features

Next, continue to forward propagate the L1 features through the second autoencoder (using `feedForwardAutoencoder.m`) to obtain the L2 hidden unit activations. These activations are then used to train the softmax classifier. You can either use `softmaxTrain.m` or directly use `softmaxCost.m` that you completed in Exercise:Softmax Regression(5.2) to complete this part of the assignment.

7.5.7 Step 4: Implement fine-tuning

To implement fine tuning, we need to consider all three layers as a single model. Implement `stackedAECost.m` to return the cost and gradient of the model. The cost function should be as defined as the log likelihood and a gradient decay term. The gradient should be computed using back-propagation as discussed earlier(2.2). The predictions should consist of the activations of the output layer of the softmax model.

To help you check that your implementation is correct, you should also check your gradients on a synthetic small dataset. We have implemented `checkStackedAECost.m` to help you check your gradients. If this checks passes, you will have implemented fine-tuning correctly.

Note: When adding the weight decay term to the cost, you should regularize only the softmax weights (do not regularize the weights that compute the hidden layer activations).

Implementation Tip: It is always a good idea to implement the code modularly and check (the gradient of) each part of the code before writing the more complicated parts.

7.5.8 Step 5: Test the model

Finally, you will need to classify with this model; complete the code in `stackedAEPredict.m` to classify using the stacked autoencoder with a classification layer.

After completing these steps, running the entire script in stackedAETrain.m will perform layer-wise training of the stacked autoencoder, finetune the model, and measure its performance on the test set. If you've done all the steps correctly, you should get an accuracy of about 87.7% before finetuning and 97.6% after finetuning (for the 10-way classification problem).

8 自编码线性解码器

8.1 线性解码器

8.1.1 稀疏自编码重述

稀疏自编码器包含 3 层神经元，分别是输入层，隐含层以及输出层。从前面（神经网络）自编码器描述可知，位于神经网络中的神经元都采用相同的激励函数。在注解中，我们修改了自编码器定义，使得某些神经元采用不同的激励函数。这样得到的模型更容易应用，而且模型对参数的变化也更为鲁棒。

回想一下，输出层神经元计算公式如下：

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \tag{96}$$

$$a^{(3)} = f(z^{(3)}) \tag{97}$$

其中 $a^{(3)}$ 是输出. 在自编码器中, $a^{(3)}$ 近似重构了输入 $x = a^{(1)}$。

S 型激励函数输出范围是 $[0, 1]$，当 $f(z^{(3)})$ 采用该激励函数时，就要对输入限制或缩放，使其位于 $[0, 1]$ 范围中。一些数据集，比如 MNIST，能方便将输出缩放到 $[0, 1]$ 中，但是很难满足对输入值的要求。比如，PCA 白化处理的输入并不满足 $[0, 1]$ 范围要求，也不清楚是否有最好的办法可以将数据缩放到特定范围中。

8.1.2 线性解码器

设定 $a^{(3)} = z^{(3)}$ 可以很简单的解决上述问题。从形式上来看，就是输出端使用恒等函数 $f(z) = z$ 作为激励函数，于是有 $a^{(3)} = f(z^{(3)}) = z^{(3)}$。我们称该特殊的激励函数为线性激励函数（称为恒等激励函数可能更好些）。需要注意，神经网络中隐含层的神经元依然使用 S 型（或者 tanh）激励函数。这样隐含单元的激励公式为 $a^{(2)} = \sigma(W^{(1)}x + b^{(1)})$, 其中 $\sigma(\cdot)$ 是 S 型函数, x 是输入, $W^{(1)}$ 和 $b^{(1)}$ 分别是隐单元的权重和偏差项。我们仅在输出层中使用线性激励函数。

一个 S 型或 tanh 隐含层以及线性输出层构成的自编码器，我们称为线性解码器。

在这个线性解码器模型中，$\hat{x} = a^{(3)} = z^{(3)} = W^{(2)}a + b^{(2)}$。因为输出 $\hat{x}$ 是隐单元激励输出的线性函数，改变 $W^{(2)}$ ，可以使输出值 $a^{(3)}$ 大于 1 或者小于 0。这使得我们可以用实值输入来训练稀疏自编码器，避免预先缩放样本到给定范围。

随着输出单元的激励函数的改变，这个输出单元梯度也相应变化。回顾之前每一个输出单元误差项定义为：

$$\delta_i^{(3)} = \frac{\partial}{\partial z_i} \ \frac{1}{2} \left\| y - \hat{x} \right\|^2 = -(y_i - \hat{x}_i) \cdot f'(z_i^{(3)}) \tag{98}$$

其中 $y = x$ 是所期望的输出, $\hat{x}$ 是自编码器的输出, $f(\cdot)$ 是激励函数. 因为在输出层激励函数为 $f(z) = z$, 这样 $f'(z) = 1$，所以上述公式可以简化为

$$\delta_i^{(3)} = -(y_i - \hat{x}_i) \tag{99}$$

当然，若使用反向传播算法来计算隐含层的误差项时:

$$\delta^{(2)} = \left((W^{(2)})^T \delta^{(3)}\right) \bullet f'(z^{(2)}) \tag{100}$$

因为隐含层采用一个 S 型（或 tanh）的激励函数 f, 在上述公式中，$f'(\cdot)$ 依然是 S 型（或 tanh）函数的导数。

8.2 练习：用线性解码器学习颜色特性

In this exercise, you will implement a linear decoder (8.1.1, a sparse autoencoder whose output layer uses a linear activation function). You will then apply it to learn features on color images from the STL-10 dataset. These features will be used in an later exercise on convolution and pooling for classifying STL-10 images.

In the file linear_decoder_exercise.zip we have provided some starter code. You should write your code at the places indicated "YOUR CODE HERE" in the files.

For this exercise, you will need to copy and modify `sparseAutoencoderCost.m` from the sparse autoencoder exercise(2.7).

8.2.1 Dependencies

You will need:

`sparseAutoencoderCost.m` (and related functions) from Exercise:Sparse Autoencoder(2.7)

The following additional file is also required for this exercise:

Sampled 8×8 patches from the STL-10 dataset (stl10_patches_100k.zip)

If you have not completed the exercise listed above, we strongly suggest you complete it first.

8.2.2 Learning from color image patches

In all the exercises so far, you have been working only with grayscale images. In this exercise, you will get to work with RGB color images for the first time.

Conveniently, the fact that an image has three color channels (RGB), rather than a single gray channel, presents little difficulty for the sparse autoencoder. You can just combine the intensities from all the color channels for the pixels into one long vector, as if you were working with a grayscale image with $3\times$ the number of pixels as the original image.

8.2.3 Step 0: Initialization

In this step, we initialize some parameters used in the exercise (see starter code for details).

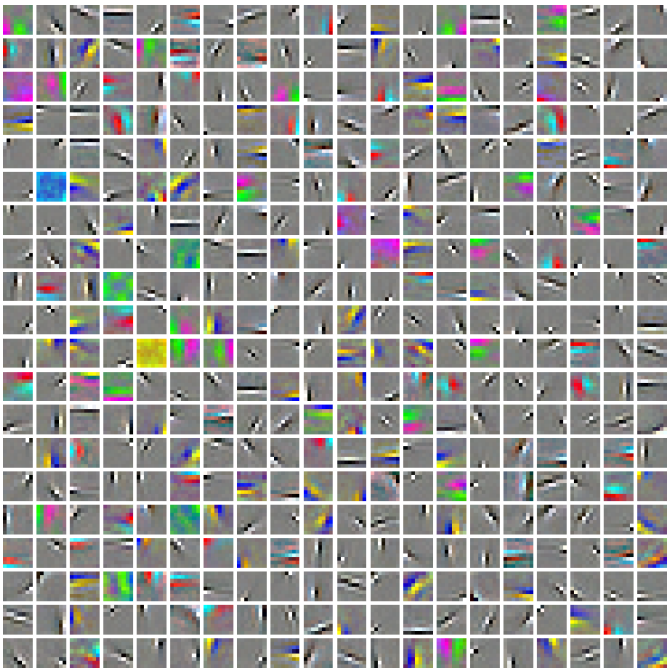
8.2.4 Step 1: Modify your sparse autoencoder to use a linear decoder

Copy `sparseAutoencoderCost.m` to the directory for this exercise and rename it to `sparseAutoencoderLinearCost.m`. Rename the function sparseAutoencoderCost in the file to sparseAutoencoderLinearCost, and modify it to use a linear decoder((8.1.1)). In particular, you should change the cost and gradients returned to reflect the change from a sigmoid to a linear decoder. After making this change, check your gradients to ensure that they are correct.

8.2.5 Step 2: Learn features on small patches

You will now use your sparse autoencoder to learn features on a set of 100,000 small 8×8 patches sampled from the larger 96×96 STL-10 images (The STL-10 dataset comprises 5000 training and 8000 test examples, with each example being a 96×96 labelled color image belonging to one of ten classes: airplane, bird, car, cat, deer, dog, horse, monkey, ship, truck.)

The code provided in this step trains your sparse autoencoder for 400 iterations with the default parameters initialized in step 0. This should take around 45 minutes. Your sparse autoencoder should learn features which when visualized, look like edges and "opponent colors," as in the figure below.



If your parameters are improperly tuned (the default parameters should work), or if your implementation of the autoencoder is buggy, you might instead get images that look like one of the following:

The learned features will be saved to `STL10Features.mat`, which will be used in the later exercise on convolution and pooling(9.3).

9 处理大型图像

9.1 卷积特征提取

9.1.1 概述

前面的练习中，解决了一些有关低分辨率图像的问题，比如：小块图像，手写数字小幅图像等。在这部分中，我们将把已知的方法扩展到实际应用中更加常见的大图像数据集。

9.1.2 Fully Connected Networks

在稀疏自编码章节中，我们介绍了把输入层和隐含层进行“全连接”的设计。从计算的角度来讲，在其他章节中曾经用过的相对较小的图像（如在稀疏自编码的作业中用到过的 8×8 的小块图像，在 MNIST 数据集中用到过的 28×28 的小块图像），从整幅图像中计算特征是可行的。但是，如果是更大的图像（如 96×96 的图像），要通过这种全联通网络的这种方法来学习整幅图像上的特征，从计算角度而言，将变得非常耗时。你需要设计 10 的 4 次方（=10000）个输入单元，假设你要学习 100 个特征，那么就有 10 的 6 次方个参数需要去学习。与 28×28 的小块图像相比较，96×96 的图像使用前向输送或者后向传导的计算方式，计算过程也会慢 10 的 2 次方（=100）倍。

9.1.3 Locally Connected Networks

解决这类问题的一种简单方法是对隐含单元和输入单元间的连接加以限制：每个隐含单元仅仅只能连接输入单元的一部分。例如，每个隐含单元仅仅连接输入图像的一小片相邻区域。（对于不同于图像输入的输入形式，也会有一些特别的连接到单隐含层的输入信号“连接区域”选择方式。如音频作为一种信号输入方式，一个隐含单元所需要连接的输入单元的子集，可能仅仅是一段音频输入所对应的某个时间段上的信号。）

网络部分连通的思想，也是受启发于生物学里面的视觉系统结构。视觉皮层的神经元就是局部接受信息的（即这些神经元只响应某些特定区域的刺激）。

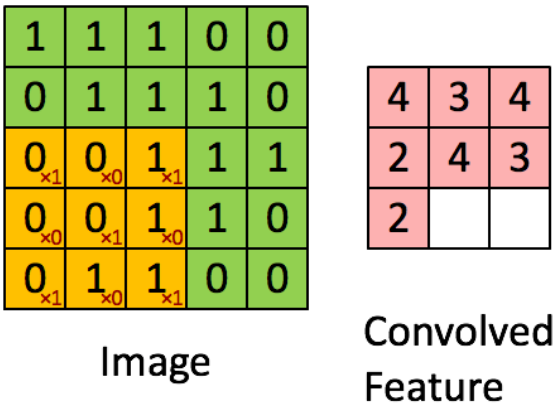
9.1.4 Convolutions

自然图像有其固有特性，也就是说，图像的一部分的统计特性与其他部分是一样的。这也意味着我们在这一部分学习的特征也能用在另一部分上，所以对于这个图像上的所有位置，我们都能使用同样的学习特征。

更恰当的解释是，当从一个大尺寸图像中随机选取一小块，比如说 8×8 作为样本，并且从这个小块样本中学习到了一些特征，这时我们可以把从这个 8×8 样本中学习到的特征作为探测器，应用到这个图像的任意地方中去。特别是，我们可以用从 8×8 样本中所学习到的特征跟原本的大尺寸图像作卷积，从而对这个大尺寸图像上的任一位置获得一个不同特征的激活值。

下面给出一个具体的例子：假设你已经从一个 96×96 的图像中学习到了它的一个 8×8 的样本所具有的特征，假设这是由有 100 个隐含单元的自编码完成的。为了得到卷积特征，需要

对 96×96 的图像的每个 8×8 的小块图像区域都进行卷积运算。也就是说，抽取 8×8 的小块区域，并且从起始坐标开始依次标记为 $(1,1),(1,2),\ldots(89,89)$，然后对抽取的区域逐个运行训练过的稀疏自编码来得到特征的激活值。在这个例子里，显然可以得到 100 个集合，每个集合含有 89×89 个卷积特征。



假设给定了 $r \times c$ 的大尺寸图像，将其定义为 x_{large}。首先通过从大尺寸图像中抽取的 $a \times b$ 的小尺寸图像样本 x_{small} 训练稀疏自编码，计算 $f = \sigma(W^{(1)}x_{small} + b^{(1)})$（$\sigma$ 是一个 sigmoid 型函数）得到了 k 个特征，其中 $W^{(1)}$ 和 $b^{(1)}$ 是可视层单元和隐含单元之间的权重和偏差值。对于每一个 $a \times b$ 大小的小图像 xs，计算出对应的值 $f_s = \sigma(W^{(1)}x_s + b^{(1)})$，对这些 $f_{convolved}$ 值做卷积，就可以得到 $k \times (r-a+1) \times (c-b+1)$ 个卷积后的特征的矩阵。

在接下来的章节里，我们会更进一步描述如何把这些特征汇总到一起以得到一些更利于分类的特征。

9.2 池化

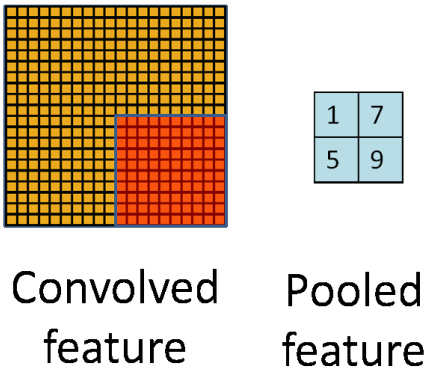
9.2.1 池化：概述

在通过卷积获得了特征 (features) 之后，下一步我们希望利用这些特征去做分类。理论上讲，人们可以用所有提取得到的特征去训练分类器，例如 softmax 分类器，但这样做面临计算量的挑战。例如：对于一个 96×96 像素的图像，假设我们已经学习得到了 400 个定义在 8×8 输入上的特征，每一个特征和图像卷积都会得到一个 $(96-8+1)*(96-8+1) = 7921$ 维的卷积特征，由于有 400 个特征，所以每个样例 (example) 都会得到一个 $89^2 * 400 = 3,168,400$ 维的卷积特征向量。学习一个拥有超过 3 百万特征输入的分类器十分不便，并且容易出现过拟合 (over-fitting)。

为了解决这个问题，首先回忆一下，我们之所以决定使用卷积后的特征是因为图像具有一种“静态性”的属性，这也就意味着在一个图像区域有用的特征极有可能在另一个区域同样适用。因此，为了描述大的图像，一个很自然的想法就是对不同位置的特征进行聚合统计，例如，人们可以计算图像一个区域上的某个特定特征的平均值 (或最大值)。这些概要统计特征不仅具有

低得多的维度 (相比使用所有提取得到的特征)，同时还会改善结果 (不容易过拟合)。这种聚合的操作就叫做池化 (pooling)，有时也称为平均池化或者最大池化 (取决于计算池化的方法)。

下图显示池化如何应用于一个图像的四块不重合区域。



9.2.2 池化的不变性

如果人们选择图像中的连续范围作为池化区域，并且只是池化相同 (重复) 的隐藏单元产生的特征，那么，这些池化单元就具有平移不变性 (translation invariant)。这就意味着即使图像经历了一个小的平移之后，依然会产生相同的 (池化的) 特征。在很多任务中 (例如物体检测、声音识别)，我们都更希望得到具有平移不变性的特征，因为即使图像经过了平移，样例 (图像) 的标记仍然保持不变。例如，如果你处理一个 MNIST 数据集的数字，把它向左侧或右侧平移，那么不论最终的位置在哪里，你都会期望你的分类器仍然能够精确地将其分类为相同的数字。

(*MNIST 是一个手写数字库识别库: http://yann.lecun.com/exdb/mnist/)

9.2.3 形式化描述

形式上，在获取到我们前面讨论过的卷积特征后，我们要确定池化区域的大小 (假定为 $m \times n$)，来池化我们的卷积特征。那么，我们把卷积特征划分到数个大小为 $m \times n$ 的不相交区域上，然后用这些区域的平均 (或最大) 特征来获取池化后的卷积特征。这些池化后的特征便可以用来做分类。

9.3 练习：卷积和池化

In this exercise you will use the features you learned on 8×8 patches sampled from images from the STL-10 dataset in the earlier exercise on linear decoders(8.2) for classifying images from a reduced STL-10 dataset applying convolution(9.1) and pooling(9.2). The reduced STL-10 dataset comprises 64×64 images from 4 classes (airplane, car, cat, dog).

In the file cnn_exercise.zip we have provided some starter code. You should write your code at the places indicated “YOUR CODE HERE" in the files.

For this exercise, you will need to modify cnnConvolve.m and cnnPool.m.

9.3.1 Dependencies

The following additional files are required for this exercise:

- A subset of the STL10 Dataset (stlSubset.zip)
- Starter Code (cnn_exercise.zip)

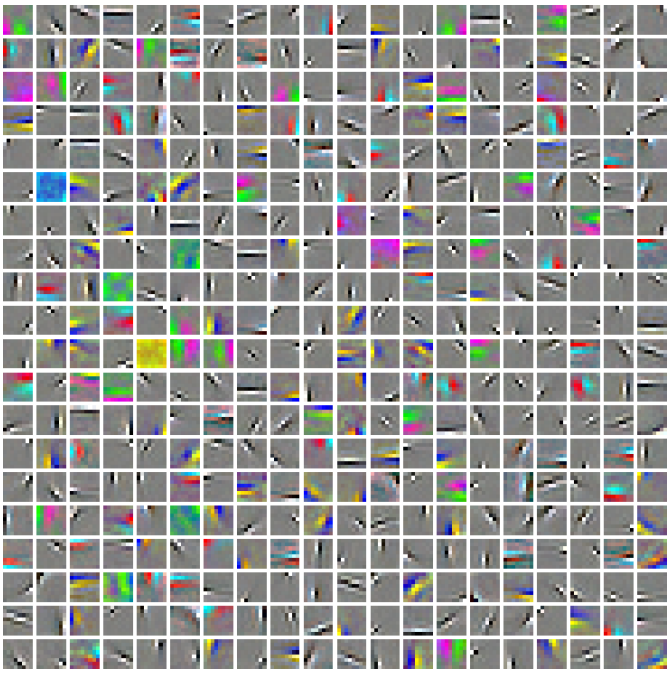
You will also need:

- `sparseAutoencoderLinear.m` or your saved features from Exercise:Learning color features with Sparse Autoencoders (8.2)
- `feedForwardAutoencoder.m` (and related functions) from Exercise:Self-Taught Learning (6.2)
- `softmaxTrain.m` (and related functions) from Exercise:Softmax Regression (5.2)

If you have not completed the exercises listed above, we strongly suggest you complete them first.

9.3.2 Step 1: Load learned features

In this step, you will use the features from Exercise:Learning color features with Sparse Autoencoders(8.2). If you have completed that exercise, you can load the color features that were previously saved. To verify that the features are good, the visualized features should look like the following:



9.3.3 Step 2: Implement and test convolution and pooling

In this step, you will implement convolution and pooling, and test them on a small part of the data set to ensure that you have implemented these two functions correctly. In the next step, you will actually convolve and pool the features with the STL-10 images.

9.3.4 Step 2a: Implement convolution

Implement convolution, as described in feature extraction using convolution(9.1), in the function cnnConvolve in cnnConvolve.m. Implementing convolution is somewhat involved, so we will guide you through the process below.

First, we want to compute $\sigma(Wx_{(r,c)} + b)$ for all valid (r, c) (valid meaning that the entire 8×8 patch is contained within the image; this is as opposed to a full convolution, which allows the patch to extend outside the image, with the area outside the image assumed to be 0), where W and b are the learned weights and biases from the input layer to the hidden layer, and $x_{(r,c)}$ is the 8×8 patch with the upper left corner at (r, c). To accomplish this, one naive method is to loop over all such patches and compute $\sigma(Wx_{(r,c)} + b)$ for each of them; while this is fine in theory, it can very slow. Hence, we usually use Matlab's built in convolution functions, which are well optimized.

Observe that the convolution above can be broken down into the following three small steps. First, compute $Wx_{(r,c)}$ for all (r, c). Next, add b to all the computed values. Finally, apply the sigmoid function to the resulting values. This doesn't seem to buy you anything, since the first step still requires a loop. However, you can replace the loop in the first step with one of MATLAB's optimized convolution functions, `conv2`, speeding up the process significantly.

However, there are two important points to note in using `conv2`.

First, `conv2` performs a 2-D convolution, but you have 5 "dimensions" - image number, feature number, row of image, column of image, and (color) channel of image - that you want to convolve over. Because of this, you will have to convolve each feature and image channel separately for each image, using the row and column of the image as the 2 dimensions you convolve over. This means that you will need three outer loops over the image number `imageNum`, feature number `featureNum`, and the channel number of the image channel. Inside the three nested for-loops, you will perform a `conv2` 2-D convolution, using the weight matrix for the `featureNum`-th feature and `channel`-th channel, and the image matrix for the `imageNum`-th image.

Second, because of the mathematical definition of convolution, the feature matrix must be "flipped" before passing it to `conv2`. The following implementat

Implementation tip: Using `conv2` and `convn`

Because the mathematical definition of convolution involves "flipping" the matrix to convolve with (reversing its rows and its columns), to use MATLAB's convolution functions, you must first

"flip" the weight matrix so that when MATLAB "flips" it according to the mathematical definition the entries will be at the correct place. For example, suppose you wanted to convolve two matrices `image` (a large image) and `W` (the feature) using `conv2(image, W)`, and `W` is a 3×3 matrix as below:

$$W = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

If you use `conv2(image, W)`, MATLAB will first "flip" `W`, reversing its rows and columns, before convolving `W` with `image`, as below:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \xrightarrow{flip} \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

If the original layout of `W` was correct, after flipping, it would be incorrect. For the layout to be correct after flipping, you will have to flip W before passing it into `conv2`, so that after MATLAB flips `W` in `conv2`, the layout will be correct. For `conv2`, this means reversing the rows and columns, which can be done with flipud and fliplr, as shown below:

```
% Flip W for use in conv2
W = flipud(fliplr(W));
```

Next, to each of the `convolvedFeatures`, you should then add `b`, the corresponding bias for the `featureNum`-th feature.

However, there is one additional complication. If we had not done any preprocessing of the input patches, you could just follow the procedure as described above, and apply the sigmoid function to obtain the convolved features, and we'd be done. However, because you preprocessed the patches before learning features on them, you must also apply the same preprocessing steps to the convolved patches to get the correct feature activations.

In particular, you did the following to the patches:

- subtract the mean patch, `meanPatch` to zero the mean of the patches
- ZCA whiten using the whitening matrix `ZCAWhite`.

These same three steps must also be applied to the input image patches.

Taking the preprocessing steps into account, the feature activations that you should compute is $\sigma(W(T(x-\bar{x}))+b)$, where T is the whitening matrix and $\bar{x}$ is the mean patch. Expanding this, you obtain $\sigma(WTx - WT\bar{x} + b)$, which suggests that you should convolve the images with WT rather than W as earlier, and you should add $(b - WT\bar{x})$, rather than just b to `convolvedFeatures`, before finally applying the sigmoid function.

9.3.5 Step 2b: Check your convolution

We have provided some code for you to check that you have done the convolution correctly. The code randomly checks the convolved values for a number of (feature, row, column) tuples by computing the feature activations using `feedForwardAutoencoder` for the selected features and patches directly using the sparse autoencoder.

9.3.6 Step 2c: Pooling

Implement pooling in the function `cnnPool` in `cnnPool.m`. You should implement mean pooling (i.e., averaging over feature responses) for this part.

9.3.7 Step 2d: Check your pooling

We have provided some code for you to check that you have done the pooling correctly. The code runs `cnnPool` against a test matrix to see if it produces the expected result.

9.3.8 Step 3: Convolve and pool with the dataset

In this step, you will convolve each of the features you learned with the full 64×64 images from the STL-10 dataset to obtain the convolved features for both the training and test sets. You will then pool the convolved features to obtain the pooled features for both training and test sets. The pooled features for the training set will be used to train your classifier, which you can then test on the test set.

Because the convolved features matrix is very large, the code provided does the convolution and pooling 50 features at a time to avoid running out of memory.

9.3.9 Step 4: Use pooled features for classification

In this step, you will use the pooled features to train a softmax classifier to map the pooled features to the class labels. The code in this section uses `softmaxTrain` from the softmax exercise to train a softmax classifier on the pooled features for 500 iterations, which should take around a few minutes.

9.3.10 Step 5: Test classifier

Now that you have a trained softmax classifier, you can see how well it performs on the test set. These pooled features for the test set will be run through the softmax classifier, and the accuracy of the predictions will be computed. You should expect to get an accuracy of around 80%.

10 稀疏编码

10.1 稀疏编码

稀疏编码算法是一种无监督学习方法，它用来寻找一组“超完备”基向量来更高效地表示样本数据。稀疏编码算法的目的就是找到一组基向量 ϕ_i ，使得我们能将输入向量 x 表示为这些基向量的线性组合：

$$\mathsf{x} = \sum_{i=1}^{k} a_i \phi_i \tag{101}$$

虽然形如主成分分析技术（PCA）能使我们方便地找到一组“完备”基向量，但是这里我们想要做的是找到一组**“超完备”**基向量来表示输入向量 $\mathsf{x} \in \mathbb{R}^n$ （也就是说，$k > n$）。超完备基的好处是它们能更有效地找出隐含在输入数据内部的结构与模式。然而，对于超完备基来说，系数 ai 不再由输入向量 x 唯一确定。因此，在稀疏编码算法中，我们另加了一个评判标准**“稀疏性”**来解决因超完备而导致的退化（degeneracy）问题。

这里，我们把“稀疏性”定义为：只有很少的几个非零元素或只有很少的几个远大于零的元素。要求系数 a_i 是稀疏的意思就是说：对于一组输入向量，我们只想有尽可能少的几个系数远大于零。选择使用具有稀疏性的分量来表示我们的输入数据是有原因的，因为绝大多数的感官数据，比如自然图像，可以被表示成少量基本元素的叠加，在图像中这些基本元素可以是面或者线。同时，比如与初级视觉皮层的类比过程也因此得到了提升。

我们把有 m 个输入向量的稀疏编码代价函数定义为：

$$\text{minimize}_{a_i^{(j)},\phi_i} \sum_{j=1}^{m} \left|\left| \mathsf{x}^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \phi_i \right|\right|^2 + \lambda \sum_{i=1}^{k} S(a_i^{(j)}) \tag{102}$$

此处 $S(\cdot)$ 是一个稀疏代价函数，由它来对远大于零的 ai 进行“惩罚”。我们可以把稀疏编码目标函式的第一项解释为一个重构项，这一项迫使稀疏编码算法能为输入向量 x 提供一个高拟合度的线性表达式，而公式第二项即“稀疏惩罚”项，它使 x 的表达式变得“稀疏”。常量 λ 是一个变换量，由它来控制这两项式子的相对重要性。

虽然“稀疏性”的最直接测度标准是“L_0" 范式 $(S(a_i) = \mathsf{1}(|a_i| > 0))$，但这是不可微的，而且通常很难进行优化。在实际中，稀疏代价函数 $S(\cdot)$ 的普遍选择是 L_1 范式代价函数 $S(a_i) = |a_i|_1$ 及对数代价函数 $S(a_i) = \log(1 + a_i^2)$。

此外，很有可能因为减小 a_i 或增加 ϕ_i 至很大的常量，使得稀疏惩罚变得非常小。为防止此类事件发生，我们将限制 $||\phi||^2$ 要小于某常量 C。包含了限制条件的稀疏编码代价函数的完整形式如下：

$$\begin{array}{rc} \text{minimize}_{a_i^{(j)},\phi_i} & \sum_{j=1}^{m} \left|\left| \mathsf{x}^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \phi_i \right|\right|^2 + \lambda \sum_{i=1}^{k} S(a_i^{(j)}) \\ \text{subject to} & ||\phi_i||^2 \leq C, \forall i = 1, ..., k \end{array}$$

10.1.1 概率解释

[基于 1996 年 Olshausen 与 Field 的理论]

到目前为止，我们所考虑的稀疏编码，是为了寻找到一个稀疏的、超完备基向量集，来覆盖我们的输入数据空间。现在换一种方式，我们可以从概率的角度出发，将稀疏编码算法当作一种“生成模型”。

我们将自然图像建模问题看成是一种线性叠加，叠加元素包括 k 个独立的源特征 ϕ_i 以及加性噪声 ν：

$$\mathsf{x} = \sum_{i=1}^{k} a_i \phi_i + \nu(\mathsf{x}) \tag{103}$$

我们的目标是找到一组特征基向量 ϕ，它使得图像的分布函数 $P(\mathsf{x} \mid \phi)$ 尽可能地近似于输入数据的经验分布函数 $P^*(\mathsf{x})$。一种实现方式是，最小化 $P^*(\mathsf{x})$ 与 $P(\mathsf{x} \mid \phi)$ 之间的 KL 散度，此 KL 散度表示如下：

$$D(P^*(\mathsf{x}) || P(\mathsf{x} \mid \phi)) = \int P^*(\mathsf{x}) \log \left(\frac{P^*(\mathsf{x})}{P(\mathsf{x} \mid \phi)} \right) d\mathsf{x} \tag{104}$$

因为无论我们如何选择 ϕ，经验分布函数 $P^*(\mathsf{x})$ 都是常量，也就是说我们只需要最大化对数似然函数 $P(\mathsf{x} \mid \phi)$。

假设 ν 是具有方差 σ^2 的高斯白噪音，则有下式：

$$P(\mathsf{x} \mid \mathsf{a}, \phi) = \frac{1}{Z} \exp \left(-\frac{(\mathsf{x} - \sum_{i=1}^{k} a_i \phi_i)^2}{2\sigma^2} \right) \tag{105}$$

为了确定分布 $P(\mathsf{x} \mid \phi)$，我们需要指定先验分布 $P(\mathsf{a})$。假定我们的特征变量是独立的，我们就可以将先验概率分解为：

$$P(\mathsf{a}) = \prod_{i=1}^{k} P(a_i) \tag{106}$$

此时，我们将“稀疏”假设加入进来——假设任何一幅图像都是由相对较少的一些源特征组合起来的。因此，我们希望 a_i 的概率分布在零值附近是凸起的，而且峰值很高。一个方便的参数化先验分布就是：

$$P(a_i) = \frac{1}{Z} \exp(-\beta S(a_i)) \tag{107}$$

这里 $S(a_i)$ 是决定先验分布的形状的函数。

当定义了 $P(\mathsf{x} \mid \mathsf{a}, \phi)$ 和 $P(\mathsf{a})$ 后，我们就可以写出在由 ϕ 定义的模型之下的数据 x 的概率分布：

$$P(\mathsf{x} \mid \phi) = \int P(\mathsf{x} \mid \mathsf{a}, \phi) P(\mathsf{a}) d\mathsf{a} \tag{108}$$

那么，我们的问题就简化为寻找：

$$\phi^* = \text{argmax}_\phi < \log(P(\mathsf{x} \mid \phi)) > \tag{109}$$

这里 $< \cdot >$ 表示的是输入数据的期望值。

不幸的是，通过对 a 的积分计算 $P(\mathsf{x} \mid \phi)$ 通常是难以实现的。虽然如此，我们注意到如果 $P(\mathsf{x} \mid \phi)$ 的分布（对于相应的 a）足够陡峭的话，我们就可以用 $P(\mathsf{x} \mid \phi)$ 的最大值来估算以上积分。估算方法如下：

$$\phi^{*'} = \text{argmax}_\phi < \max_{\mathsf{a}} \log(P(\mathsf{x} \mid \phi)) > \tag{110}$$

跟之前一样，我们可以通过减小 a_i 或增大 ϕ 来增加概率的估算值（因为 $P(a_i)$ 在零值附近陡升）。因此我们要对特征向量 ϕ 加一个限制以防止这种情况发生。

最后，我们可以定义一种线性生成模型的能量函数，从而将原先的代价函数重新表述为：

$$\begin{aligned} E(\mathsf{x}, \mathsf{a} \mid \phi) &:= -\log\left(P(\mathsf{x} \mid \phi, \mathsf{a}) P(\mathsf{a})\right) \\ &= \sum_{j=1}^{m} \left|\left| \mathsf{x}^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \phi_i \right|\right|^2 + \lambda \sum_{i=1}^{k} S(a_i^{(j)}) \end{aligned}$$

其中 $\lambda = 2\sigma^2\beta$，并且关系不大的常量已被隐藏起来。因为最大化对数似然函数等同于最小化能量函数，我们就可以将原先的优化问题重新表述为：

$$\phi^*, \mathsf{a}^* = \text{argmin}_{\phi, \mathsf{a}} \sum_{j=1}^{m} \left|\left| \mathsf{x}^{(j)} - \sum_{i=1}^{k} a_i^{(j)} \phi_i \right|\right|^2 + \lambda \sum_{i=1}^{k} S(a_i^{(j)}) \tag{111}$$

使用概率理论来分析，我们可以发现，选择 L_1 惩罚和 $\log(1 + a_i^2)$ 惩罚作为函数 $S(\cdot)$，分别对应于使用了拉普拉斯概率 $P(a_i) \propto \exp(-\beta|a_i|)$ 和柯西先验概率 $P(a_i) \propto \frac{\beta}{1+a_i^2}$。

10.1.2 学习算法

使用稀疏编码算法学习基向量集的方法，是由两个独立的优化过程组合起来的。第一个是逐个使用训练样本 x 来优化系数 a_i，第二个是一次性处理多个样本对基向量 ϕ 进行优化。

如果使用 L_1 范式作为稀疏惩罚函数，对 $a_i^{(j)}$ 的学习过程就简化为求解由 L_1 范式正则化的最小二乘法问题，这个问题函数在域 $a_i^{(j)}$ 内为凸，已经有很多技术方法来解决这个问题（诸如

CVX 之类的凸优化软件可以用来解决 L_1 正则化的最小二乘法问题）。如果 $S(\cdot)$ 是可微的，比如是对数惩罚函数，则可以采用基于梯度算法的方法，如共轭梯度法。

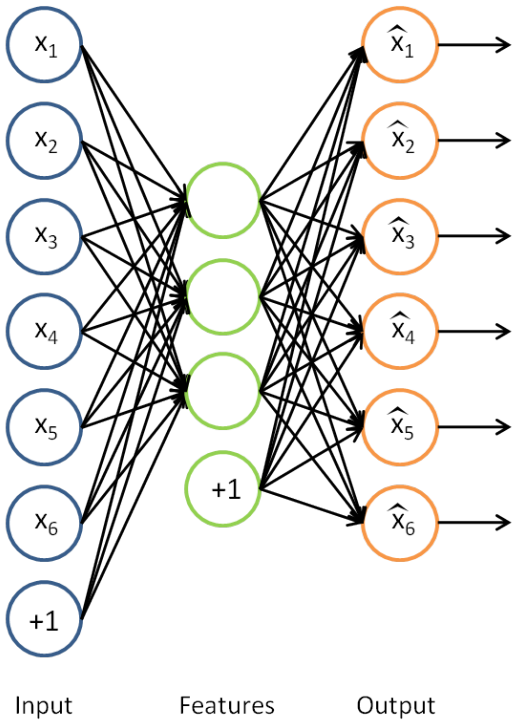
用 L_2 范式约束来学习基向量，同样可以简化为一个带有二次约束的最小二乘问题，其问题函数在域 ϕ 内也为凸。标准的凸优化软件（如 CVX）或其它迭代方法就可以用来求解 ϕ，虽然已经有了更有效的方法，比如求解拉格朗日对偶函数（Lagrange dual）。

根据前面的的描述，稀疏编码是有一个明显的局限性的，这就是即使已经学习得到一组基向量，如果为了对新的数据样本进行“编码”，我们必须再次执行优化过程来得到所需的系数。这个显著的“实时”消耗意味着，即使是在测试中，实现稀疏编码也需要高昂的计算成本，尤其是与典型的前馈结构算法相比。

10.2 稀疏编码自编码表达

10.2.1 稀疏编码

在稀疏自编码算法中，我们试着学习得到一组权重参数 W（以及相应的截距 b），通过这些参数可以使我们得到稀疏特征向量 $\sigma(Wx+b)$，这些特征向量对于重构输入样本非常有用。



稀疏编码可以看作是稀疏自编码方法的一个变形，该方法试图直接学习数据的特征集。利用与此特征集相应的基向量，将学习得到的特征集从特征空间转换到样本数据空间，这样我们就可以用学习得到的特征集重构样本数据。

确切地说，在稀疏编码算法中，有样本数据 x 供我们进行特征学习。特别是，学习一个用于表示样本数据的稀疏特征集 s, 和一个将特征集从特征空间转换到样本数据空间的基向量 A, 我们可以构建如下目标函数:

$$J(A,s) = \|As - x\|_2^2 + \lambda\|s\|_1$$

（$\|x\|_k$ 是 x 的 L_k 范数，等价于 $\left(\sum \left|x_i^k\right|\right)^{\frac{1}{k}}$。$L_2$ 范数即大家熟知的欧几里得范数，L_1 范数是向量元素的绝对值之和）

上式前第一部分是利用基向量将特征集重构为样本数据所产生的误差，第二部分为稀疏性惩罚项（sparsity penalty term），用于保证特征集的稀疏性。

但是，如目标函数所示，它的约束性并不强——按常数比例缩放 A 的同时再按这个常数的倒数缩放 s，结果不会改变误差大小，却会减少稀疏代价（表达式第二项）的值。因此，需要为 A 中每项 A_j 增加额外约束 $A_j^T A_j \leq 1$。问题变为：

$$\begin{aligned} \text{minimize} \quad & \|As - x\|_2^2 + \lambda\|s\|_1 \\ \text{s.t.} \quad & A_j^T A_j \leq 1 \; \forall j \end{aligned}$$

遗憾的是，因为目标函数并不是一个凸函数，所以不能用梯度方法解决这个优化问题。但是，在给定 A 的情况下，最小化 $J(A,s)$ 求解 s 是凸的。同理，给定 s 最小化 $J(A,s)$ 求解 A 也是凸的。这表明，可以通过交替固定 s 和 A 分别求解 A 和 s。实践表明，这一策略取得的效果非常好。

但是，以上表达式带来了另一个难题：不能用简单的梯度方法来实现约束条件 $A_j^T A_j \leq 1 \; \forall j$。因此在实际问题中，此约束条件还不足以成为“权重衰变”（“weight decay"）项以保证 A 的每一项值够小。这样我们就得到一个新的目标函数：

$$J(A,s) = \|As - x\|_2^2 + \lambda\|s\|_1 + \gamma\|A\|_2^2$$

（注意上式中第三项，$\|A\|_2^2$ 等价于 $\sum_r \sum_c A_{rc}^2$，是 A 各项的平方和）

这一目标函数带来了最后一个问题，即 L_1 范数在 0 点处不可微影响了梯度方法的应用。尽管可以通过其他非梯度下降方法避开这一问题，但是本文通过使用近似值“平滑”L_1 范数的方法解决此难题。使用 $\sqrt{x^2 + \epsilon}$ 代替 $|x|$，对 L_1 范数进行平滑，其中 ϵ 是“平滑参数”（“smoothing parameter"）或者“稀疏参数”（“sparsity parameter"）（如果 ϵ 远大于 x，则 $x + \epsilon$ 的值由 ϵ 主导，其平方根近似于 ϵ）。在下文提及拓扑稀疏编码时，“平滑”会派上用场。

因此，最终的目标函数是：

$$J(A,s) = \|As - x\|_2^2 + \lambda\sqrt{s^2 + \epsilon} + \gamma\|A\|_2^2$$

(where $\sqrt{s^2 + \epsilon}$ is shorthand for $\sum_k \sqrt{s_k^2 + \epsilon}$)

该目标函数可以通过以下过程迭代优化：

1. 随机初始化 A

2. 重复以下步骤直至收敛：

(a) 根据上一步给定的 A，求解能够最小化 $J(A,s)$ 的 s

(b) 根据上一步得到的 s，求解能够最小化 $J(A,s)$ 的 A

观察修改后的目标函数 $J(A,s)$，给定 s 的条件下，目标函数可以简化为 $J(A;s)=\|As-x\|_2^2+\gamma\|A\|_2^2$（因为 s 的 L_1 范式不是 A 的函数，所以可以忽略）。简化后的目标函数是一个关于 A 的简单二次项式，因此对 A 求导是很容易的。这种求导的一种快捷方法是矩阵微积分（相关链接部分列出了跟矩阵演算有关的内容）。遗憾的是，在给定 A 的条件下，目标函数却不具备这样的求导方法，因此目标函数的最小化步骤只能用梯度下降或其他类似的最优化方法。

理论上，通过上述迭代方法求解目标函数的最优化问题最终得到的特征集（A 的基向量）与通过稀疏自编码学习得到的特征集是差不多的。但是实际上，为了获得更好的算法收敛性需要使用一些小技巧，后面的稀疏编码实践稀疏编码实践章节会详细介绍这些技巧。用梯度下降方法求解目标函数也略需技巧，另外使用矩阵演算或反向传播算法则有助于解决此类问题。

10.2.2 拓扑稀疏编码

通过稀疏编码，我们能够得到一组用于表示样本数据的特征集。不过，让我们来找些灵感，我们希望学习得到一组有某种“秩序”的特征集。举个例子，视觉特征，如前面所提到的，大脑皮层 V1 区神经元能够按特定的方向对边缘进行检测，同时，这些神经元（在生理上）被组织成超柱（hypercolumns），在超柱中，相邻神经元以相似的方向对边缘进行检测，一个神经元检测水平边缘，其相邻神经元检测到的边缘就稍微偏离水平方向，沿着超柱，神经元就可以检测到与水平方向相差更大的边缘了。

受该例子的启发，我们希望学习到的特征也具有这样“拓扑秩序”的性质。这对于我们要学习的特征意味着什么呢? 直观的讲，如果“相邻”的特征是“相似”的，就意味着如果某个特征被激活，那么与之相邻的特征也将随之被激活。

具体而言，假设我们（随意地）将特征组织成一个方阵。我们就希望矩阵中相邻的特征是相似的。实现这一点的方法是将相邻特征按经过平滑的 L1 范式惩罚进行分组，如果按 3×3 方阵分组，则用 $\sqrt{s_{1,1}^2+s_{1,2}^2+s_{1,3}^2+s_{2,1}^2+s_{2,2}^2+s_{3,2}^2+s_{3,1}^2+s_{3,2}^2+s_{3,3}^2+\epsilon}$ 代替 $\sqrt{s_{1,1}^2+\epsilon}$, 其分组通常是重合的，因此从第 1 行第 1 列开始的 3×3 区域是一个分组，从第 1 行第 2 列开始的 3×3 区域是另一个分组，以此类推。最终，这样的分组会形成环绕，就好像这个矩阵是个环形曲面，所以每个特征都以同样的次数进行了分组。

于是，将经过平滑的所有分组的 L1 惩罚值之和代替经过平滑的 L1 惩罚值，得到新的目标函数如下：

$$J(A,s)=\|As-x\|_2^2+\lambda\sum_{\text{all groups}g}\sqrt{\left(\sum_{\text{all}s\in g}s^2\right)+\epsilon}+\gamma\|A\|_2^2$$

实际上，“分组”可以通过“分组矩阵”V 完成，于是矩阵 V 的第 r 行标识了哪些特征被分到第 r 组中，即如果第 r 组包含特征 c 则 $V_{r,c}=1$。通过分组矩阵实现分组使得梯度的计算更加直观，使用此分组矩阵，目标函数被重写为：

$$J(A,s) = \|As - x\|_2^2 + \lambda \sum \sqrt{Vss^T + \epsilon} + \gamma \|A\|_2^2$$

(令 $D = \sqrt{Vss^T + \epsilon}$，$\sum \sqrt{Vss^T + \epsilon}$ 等价于 $\sum_r \sum_c D_{r,c}$

该目标函数能够使用之前部分提到的迭代方法进行求解。拓扑稀疏编码得到的特征与稀疏编码得到的类似，只是拓扑稀疏编码得到的特征是以某种方式有“秩序”排列的。

10.2.3 稀疏编码实践

如上所述，虽然稀疏编码背后的理论十分简单，但是要写出准确无误的实现代码并能快速又恰到好处地收敛到最优值，则需要一定的技巧。

回顾一下之前提到的简单迭代算法：

1. 随机初始化 A
2. 重复以下步骤直至收敛到最优值：
   (a) 根据上一步给定的 A，求解能够最小化 $J(A,s)$ 的 s
   (b) 根据上一步得到的 s，求解能够最小化 $J(A,s)$ 的 A

这样信手拈来地执行这个算法，结果并不会令人满意，即使确实得到了某些结果。以下是两种更快更优化的收敛技巧：

1. 将样本分批为“迷你块”
2. 良好的 s 初始值

将样本分批为“迷你块”

如果你一次性在大规模数据集（比如，有 10000 个 patch）上执行简单的迭代算法，你会发现每次迭代都要花很长时间，也因此这算法要花好长时间才能达到收敛结果。为了提高收敛速度，可以选择在迷你块上运行该算法。每次迭代的时候，不是在所有的 10000 个 patchs 上执行该算法，而是使用迷你块，即从 10000 个 patch 中随机选出 2000 个 patch，再在这个迷你块上执行这个算法。这样就可以做到一石二鸟——第一，提高了每次迭代的速度，因为现在每次迭代只在 2000 个 patch 上执行而不是 10000 个；第二，也是更重要的，它提高了收敛的速度（原因见 TODO）。

良好的 s 初始值

另一个能获得更快速更优化收敛的重要技巧是：在给定 A 的条件下，根据目标函数使用梯度下降（或其他方法）求解 s 之前找到良好的特征矩阵 s 的初始值。实际上，除非在优化 A 的最优值前已找到一个最佳矩阵 s，不然每次迭代过程中随机初始化 s 值会导致很差的收敛效果。下面给出一个初始化 s 的较好方法：

1. 令 $s \leftarrow W^T x$ (x 是迷你块中 patches 的矩阵表示)

2. s 中的每个特征（s 的每一列），除以其在 A 中对应基向量的范数。即，如果 $s_{r,c}$ 表示第 c 个样本的第 r 个特征，则 A_c 表示 A 中的第 c 个基向量，则令 $s_{r,c} \leftarrow \frac{s_{r,c}}{\|A_c\|}$.

无疑，这样的初始化有助于算法的改进，因为上述的第一步希望找到满足 $Ws \approx x$ 的矩阵 s；第二步对 s 作规范化处理是为了保持较小的稀疏惩罚值。这也表明，只采用上述步骤的某一步而不是两步对 s 做初始化处理将严重影响算法性能。（TODO: 此链接将会对为什么这样的初始化能改进算法作出更详细的解释）

可运行算法

有了以上两种技巧，稀疏编码算法修改如下：

1. 随机初始化 A
2. 重复以下步骤直至收敛
   (a) 随机选取一个有 2000 个 patches 的迷你块
   (b) 如上所述，初始化 s
   (c) 根据上一步给定的 A，求解能够最小化 $J(A, s)$ 的 s
   (d) 根据上一步得到的 s，求解能够最小化 $J(A, s)$ 的 A

通过上述方法，可以相对快速的得到局部最优解。

10.3 Exercise:Sparse Coding

In this exercise, you will implement sparse coding(??) and topographic sparse coding(10.2.2) on black-and-white natural images.

In the file sparse_coding_exercise.zip we have provided some starter code. You should write your code at the places indicated “YOUR CODE HERE" in the files.

For this exercise, you will need to modify sparseCodingWeightCost.m, sparseCodingFeatureCost.m and sparseCodingExercise.m.

10.3.1 Dependencies

You will need:

- `computeNumericalGradient.m` from Exercise:Sparse Autoencoder (2.7)
- `display_network.m` from Exercise:Sparse Autoencoder (2.7)

If you have not completed the exercise listed above, we strongly suggest you complete it first.

10.3.2 Step 0: Initialization

In this step, we initialize some parameters used for the exercise.

10.3.3 Step 1: Sample patches

In this step, we sample some patches from the `IMAGES.mat` dataset comprising 10 black-and-white pre-whitened natural images.

10.3.4 Step 2: Implement and check sparse coding cost functions

In this step, you should implement the two sparse coding cost functions:

1. `sparseCodingWeightCost` in `sparseCodingWeightCost.m`, which is used for optimizing the weight cost given the features

2. `sparseCodingFeatureCost` in `sparseCodingFeatureCost.m`, which is used for optimizing the feature cost given the weights

Each of these functions should compute the appropriate cost and gradient. You may wish to implement the non-topographic version of sparseCodingFeatureCost first, ignoring the grouping matrix and assuming that none of the features are grouped. You can then extend this to the topographic version later. Alternatively, you may implement the topographic version directly - using the non-topographic version will then involve setting the grouping matrix to the identity matrix.

Once you have implemented these functions, you should check the gradients numerically.

Implementation tip - gradient checking the feature cost. One particular point to note is that when checking the gradient for the feature cost, `epsilon` should be set to a larger value, for instance `1e-2` (as has been done for you in the checking code provided), to ensure that checking the gradient numerically makes sense. This is necessary because as `epsilon` becomes smaller, the function `sqrt(x + epsilon)` becomes "sharper" and more "pointed", making the numerical gradient computed near 0 less and less accurate. To see this, consider what would happen if the numerical gradient was computed by using a point with x less than 0 and a point with x greater than 0 - the computed numerical slope would be wildly inaccurate.

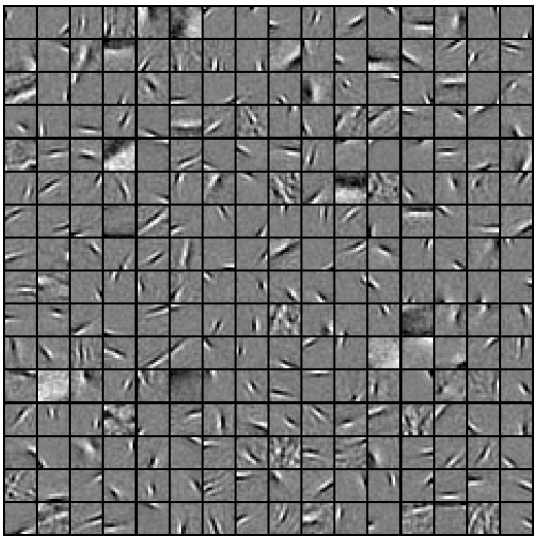
10.3.5 Step 3: Iterative optimization

In this step, you will iteratively optimize for the weights and features to learn a basis for the data, as described in the section on sparse coding(10.2). Mini-batching and initialization of the features s has already been done for you. However, you need to still need to fill in the analytic solution to the the optimization problem with respect to the weight matrix, given the feature matrix.

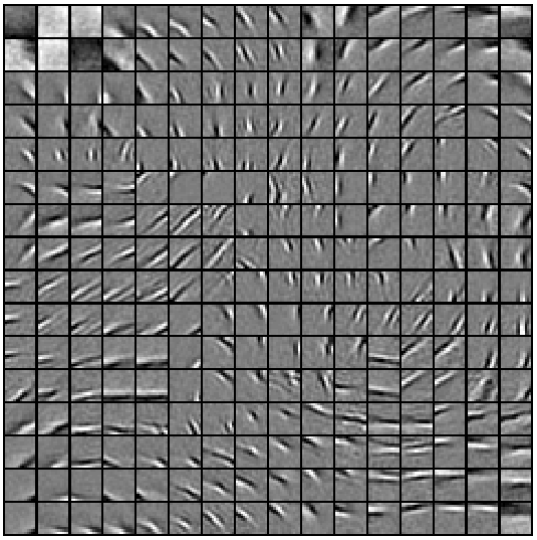
Once that is done, you should check that your solution is correct using the given checking code, which checks that the gradient at the point determined by your analytic solution is close to 0. Once your solution has been verified, comment out the checking code, and run the iterative

optimization code. 200 iterations should take less than 45 minutes to run, and by 100 iterations you should be able to see bases that look like edges, similar to those you learned in the sparse autoencoder exercise(2.7).

For the non-topographic case, these features will not be “ordered", and will look something like the following:



For the topographic case, the features will be “ordered topographically", and will look something like the following:



11 独立成分分析样式建模

11.1 独立成分分析

11.1.1 概述

试着回想一下，在介绍稀疏编码 (10) 算法中我们想为样本数据学习得到一个**超完备**基（over-complete basis）。具体来说，这意味着用稀疏编码学习得到的基向量之间不一定线性独立。尽管在某些情况下这已经满足需要，但有时我们仍然希望得到的是一组线性独立基。独立成分分析算法（ICA）正实现了这一点。而且，在 ICA 中，我们希望学习到的基不仅要线性独立，而且还是一组标准**正交**基。（一组标准正交基 $(\phi_1, \dots \phi_n)$ 需要满足条件：$\phi_i \cdot \phi_j = 0$（如果 $i \neq j$）或者 $\phi_i \cdot \phi_j = 1$（如果 $i = j$））

与稀疏编码算法类似，独立成分分析也有一个简单的数学形式。给定数据 x，我们希望学习得到一组基向量 – 以列向量形式构成的矩阵 W，其满足以下特点：首先，与稀疏编码一样，特征是**稀疏**的；其次，基是标准**正交**的（注意，在稀疏编码中，矩阵 A 用于将**特征** s 映射到**原始数据**，而在独立成分分析中，矩阵 W 工作的方向相反，是将**原始数据** x 映射到**特征**）。这样我们得到以下目标函数：

$$J(W) = \|Wx\|_1$$

由于 Wx 实际上是描述样本数据的特征，这个目标函数等价于在稀疏编码中特征 s 的稀疏惩罚项。加入标准正交性约束后，独立成分分析相当于求解如下优化问题：

$$\begin{aligned} \text{minimize} \quad & \|Wx\|_1 \\ \text{s.t.} \quad & WW^T = I \end{aligned}$$

与深度学习中的通常情况一样，这个问题没有简单的解析解，而且更糟糕的是，由于标准正交性约束，使得用梯度下降方法来求解该问题变得更加困难 – 每次梯度下降迭代之后，必须将新的基映射回正交基空间中（以此保证正交性约束）。

实践中，在最优化目标函数的同时施加正交性约束（如下一节正交 ICA 11.1.2 中讲到的）是可行的，但是速度慢。在标准正交基是不可或缺的情况下，标准正交 ICA 的使用会受到一些限制。（哪些情况见：TODO ）

11.1.2 正交 ICA

标准正交 ICA 的目标函数是：

$$\begin{aligned} \text{minimize} \quad & \|Wx\|_1 \\ \text{s.t.} \quad & WW^T = I \end{aligned}$$

通过观察可知，约束 $WW^T = I$ 隐含着另外两个约束：

第一，因为要学习到一组标准正交基，所以基向量的个数必须小于输入数据的维度。具体来说，这意味着不能像通常在稀疏编码中所做的那样来学习得到超完备基（over-complete bases）。

第二，数据必须经过无正则 ZCA 白化（也即,ϵ 设为 0）。(为什么必须这样做? 见 TODO)

因此，在优化标准正交 ICA 目标函数之前，必须确保数据被白化过，并且学习的是一组**不完备**基（under-complete basis）。

然后，为了优化目标函数，我们可以使用梯度下降法，在梯度下降的每一步中增加投影步骤，以满足标准正交约束。过程如下：

重复以下步骤直到完成：

$W \leftarrow W - \alpha \nabla_W \|Wx\|_1$

$W \leftarrow \text{proj}_U W$, 其中 U 是满足 $WW^T = I$ 的矩阵空间

在实际中, 学习速率 α 是可变的, 使用一个线搜索算法来加速梯度. 投影步骤通过设置 $W \leftarrow (WW^T)^{-\frac{1}{2}}W$ 来完成, 这实际上可以看成就是 ZCA 白化 (TODO: 解释为什么这就象 ZCA 白化).

11.1.3 拓扑 ICA

与稀疏编码 (10.2) 算法类似，加上一个拓扑代价项，独立成分分析法可以修改成具有拓扑性质的算法。

11.2 Exercise:Independent Component Analysis

In this exercise, you will implement Independent Component Analysis(11.1) on color images from the STL-10 dataset.

In the file independent_component_analysis_exercise.zip we have provided some starter code. You should write your code at the places indicated "YOUR CODE HERE" in the files.

For this exercise, you will need to modify `OrthonormalICACost.m` and `ICAExercise.m`.

11.2.1 Dependencies

You will need:

- `computeNumericalGradient.m` from Exercise:Sparse Autoencoder(2.7)
- `displayColorNetwork.m` from Exercise:Learning color features with Sparse Autoencoders(8.2)

The following additional file is also required for this exercise:

- Sampled 8×8 patches from the STL-10 dataset (stl10_patches_100k.zip)

If you have not completed the exercises listed above, we strongly suggest you complete them first.

11.2.2 Step 0: Initialization

In this step, we initialize some parameters used for the exercise.

11.2.3 Step 1: Sample patches

In this step, we load and use a portion of the 8×8 patches from the STL-10 dataset (which you first saw in the exercise on linear decoders 8.2).

11.2.4 Step 2: ZCA whiten patches

In this step, we ZCA whiten the patches as required by orthonormal ICA.

11.2.5 Step 3: Implement and check ICA cost functions

In this step, you should implement the ICA cost function: `orthonormalICACost` in `orthonormalICA` which computes the cost and gradient for the orthonormal ICA objective. Note that the orthonormality constraint is not enforced in the cost function. It will be enforced by a projection in the gradient descent step, which you will have to complete in step 4.

When you have implemented the cost function, you should check the gradients numerically.

Hint - if you are having difficulties deriving the gradients, you may wish to consult the page on deriving gradients using the backpropagation idea (B.2).

11.2.6 Step 4: Optimization

In step 4, you will optimize for the orthonormal ICA objective using gradient descent with backtracking line search (the code for which has already been provided for you. For more details on the backtracking line search, you may wish to consult the appendix(11.2.7) of this exercise). The orthonormality constraint should be enforced with a projection, which you should fill in.

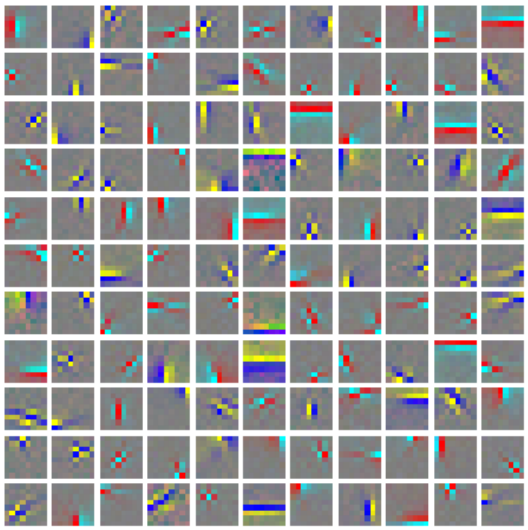
Once you have filled in the code for the projection, check that it is correct by using the verification code provided. Once you have verified that your projection is correct, comment out the verification code and run the optimization. 1000 iterations of gradient descent should take less than 15 minutes, and produce a basis which looks like the following:

It is comparatively difficult to optimize for the objective while enforcing the orthonormality constraint using gradient descent, and convergence can be slow. Hence, in situations where an orthonormal basis is not required, other faster methods of learning bases (such as sparse coding 10.2) may be preferable.

11.2.7 Appendix

Backtracking line search

The backtracking line search used in the exercise is based off that in Convex Optimization by Boyd and Vandenbergh(http://www.stanford.edu/~boyd/cvxbook/). In the backtracking line search, given a descent direction $\vec{u}$ (in this exercise we use $\vec{u} = -\nabla f(\vec{x})$), we want to find



a good step size t that gives us a steep descent. The general idea is to use a linear approximation (the first order Taylor approximation) to the function f at the current point $\vec{x}$, and to search for a step size t such that we can decrease the function's value by more than α times the decrease predicted by the linear approximation ($\alpha \in (0, 0.5)$. For more details, you may wish to consult the book.

However, it is not necessary to use the backtracking line search here. Gradient descent with a small step size, or backtracking to a step size so that the objective decreases is sufficient for this exercise.

A Using the MNIST Dataset

A.1 Introduction

The MNIST dataset is a dataset of handwritten digits, comprising 60 000 training examples and 10 000 test examples. The dataset can be downloaded from http://yann.lecun.com/exdb/mnist/.

A.2 Usage

The image and label data is stored in a binary format described on the website. For your convenience, we have provided two MATLAB helper functions for extracting the data. These functions are available at http://ufldl.stanford.edu/wiki/resources/mnistHelper.zip.

As an example of how to use these functions, you can check the images and labels using the following code:

```
% Change the filenames if you've saved the files under different names
% On some platforms, the files might be saved as
```

```
% train-images.idx3-ubyte / train-labels.idx1-ubyte
images = loadMNISTImages('train-images-idx3-ubyte');
labels = loadMNISTLabels('train-labels-idx1-ubyte');

% We are using display_network from the autoencoder code
display_network(images(:,1:100)); % Show the first 100 images
disp(labels(1:10));
```

B Miscellaneous Topics

B.1 数据预处理

B.1.1 概要

数据预处理在众多深度学习算法中都起着重要作用，实际情况中，将数据做归一化和白化处理后，很多算法能够发挥最佳效果。然而除非对这些算法有丰富的使用经验，否则预处理的精确参数并非显而易见。在本页中，我们希望能够揭开预处理方法的神秘面纱，同时为预处理数据提供技巧（和标准流程）。

提示：当我们开始处理数据时，首先要做的事是观察数据并获知其特性。本部分将介绍一些通用的技术，在实际中应该针对具体数据选择合适的预处理技术。例如一种标准的预处理方法是对每一个数据点都减去它的均值（也被称为移除直流分量，局部均值消减，消减归一化），这一方法对诸如自然图像这类数据是有效的，但对非平稳的数据则不然。

B.1.2 数据归一化

数据预处理中，标准的第一步是数据归一化。虽然这里有一系列可行的方法，但是这一步通常是根据数据的具体情况而明确选择的。特征归一化常用的方法包含如下几种：

- 简单缩放
- 逐样本均值消减 (也称为移除直流分量)
- 特征标准化 (使数据集中所有特征都具有零均值和单位方差)

简单缩放

在简单缩放中，我们的目的是通过对数据的每一个维度的值进行重新调节（这些维度可能是相互独立的），使得最终的数据向量落在 [0,1] 或 [- 1,1] 的区间内（根据数据情况而定）。这对后续的处理十分重要，因为很多**默认**参数（如 PCA-白化中的 `epsilon`）都假定数据已被缩放到合理区间。

例子: 在处理自然图像时，我们获得的像素值在 [0,255] 区间中，常用的处理是将这些像素值除以 255，使它们缩放到 [0,1] 中.

逐样本均值消减

如果你的数据是**平稳的**（即数据每一个维度的统计都服从相同分布），那么你可以考虑在每个样本上减去数据的统计平均值 (逐样本计算)。

例子：对于图像，这种归一化可以移除图像的平均亮度值 (intensity)。很多情况下我们对图像的照度并不感兴趣，而更多地关注其内容，这时对每个数据点移除像素的均值是有意义的。注意：虽然该方法广泛地应用于图像，但在处理彩色图像时需要格外小心，具体来说，是因为不同色彩通道中的像素并不都存在平稳特性。

特征标准化

特征标准化指的是（独立地）使得数据的每一个维度具有零均值和单位方差。这是归一化中最常见的方法并被广泛地使用（例如，在使用支持向量机（SVM）时，特征标准化常被建议用作预处理的一部分）。在实际应用中，特征标准化的具体做法是：首先计算每一个维度上数据的均值（使用全体数据计算），之后在每一个维度上都减去该均值。下一步便是在数据的每一维度上除以该维度上数据的标准差。

例子: 处理音频数据时，常用 Mel 倒频系数 MFCCs 来表征数据。然而 MFCC 特征的第一个分量（表示直流分量）数值太大，常常会掩盖其他分量。这种情况下，为了平衡各个分量的影响，通常对特征的每个分量独立地使用标准化处理。

B.1.3 PCA/ZCA 白化

在做完简单的归一化后，白化通常会被用来作为接下来的预处理步骤，它会使我们的算法工作得更好。实际上许多深度学习算法都依赖于白化来获得好的特征。

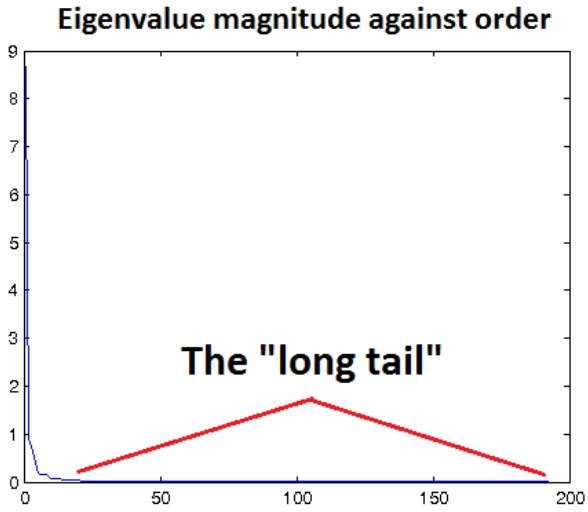
在进行 PCA/ZCA 白化时，首先使特征零均值化是很有必要的，这保证了 $\frac{1}{m}\sum_i x^{(i)} = 0$。特别地，这一步需要在计算协方差矩阵前完成。(唯一例外的情况是已经进行了逐样本均值消减，并且数据在各维度上或像素上是平稳的。)

接下来在 PCA/ZCA 白化 (4.2) 中我们需要选择合适的 `epsilon`（回忆一下，这是规则化项，对数据有低通滤波作用）。选取合适的 `epsilon` 值对特征学习起着很大作用，下面讨论在两种不同场合下如何选取 `epsilon`:

基于重构的模型

在基于重构的模型中 (包括自编码器, 稀疏编码, 受限 Boltzman 机(RBM), k-均值(K-Means))，经常倾向于选取合适的 `epsilon` 以使得白化达到低通滤波的效果。(译注：通常认为数据中的高频分量是噪声，低通滤波的作用就是尽可能抑制这些噪声，同时保留有用的信息。在 PCA 等方法中，假设数据的信息主要分布在方差较高的方向，方差较低的方向是噪声（即高频分量），因此后文中 `epsilon` 的选择与特征值有关）。一种检验 `epsilon` 是否合适的方法是用该值对数据进行 ZCA 白化，然后对白化前后的数据进行可视化。如果 `epsilon` 值过低，白化后的数据会显得噪声很大；相反，如果 `epsilon` 值过高，白化后的数据与原始数据相比就过于模糊。一种直观上得到 `epsilon` 大小的方法是以图形方式画出数据的特征值，如下图的例子所示，你可以看到一条 “长尾", 它对应于数据中的高频噪声部分。你需要选取合适的 `epsilon`，使其

能够在很大程度上过滤掉这条“长尾"，也就是说，选取的 epsilon 应大于大多数较小的、反映数据中噪声的特征值。



在基于重构的模型中，损失函数有一项是用于惩罚那些与原始输入数据差异较大的重构结果（译注：以自动编码机为例，要求输入数据经过编码和解码之后还能尽可能的还原输入数据）。如果 epsilon 太小，白化后的数据中就会包含很多噪声，而模型要拟合这些噪声，以达到很好的重构结果。因此，对于基于重构的模型来说，对原始数据进行低通滤波就显得非常重要。

提示：如果数据已被缩放到合理范围 (如 [0,1])，可以从 epsilon = 0.01 或 epsilon = 0.1 开始调节 epsilon。

基于正交化 ICA 的模型

基于正交化 ICA 的模型来说，保证输入数据尽可能地白化（即协方差矩阵为单位矩阵）非常重要。这是因为：这类模型需要对学习到的特征做正交化，以解除不同维度之间的相关性（详细内容请参考 ICA 11.1 一节）。因此在这种情况下，epsilon 要足够小（比如 epsilon = 1e - 6）。

提示：我们也可以在 PCA 白化过程中同时降低数据的维度。这是一个很好的主意，因为这样可以大大提升算法的速度（减少了运算量和参数数目）。确定要保留的主成分数目有一个经验法则：即所保留的成分的总方差达到总样本方差的 99% 以上。(详细内容请参考 PCA 4.1.6)

注意: 在使用分类框架时，我们应该只基于练集上的数据计算 PCA/ZCA 白化矩阵。需要保存以下两个参数留待测试集合使用：(a) 用于零均值化数据的平均值向量；(b) 白化矩阵。测试集需要采用这两组保存的参数来进行相同的预处理。

B.1.4 大图像

对于大图像，采用基于 PCA/ZCA 的白化方法是不切实际的，因为协方差矩阵太大。在这些情况下我们退而使用 1/f 白化方法（更多内容后续再讲）。

B.1.5 标准流程

在这一部分中，我们将介绍几种在一些数据集上有良好表现的预处理标准流程.

自然灰度图像

灰度图像具有平稳特性，我们通常在第一步对每个数据样本分别做均值消减（即减去直流分量），然后采用 PCA/ZCA 白化处理，其中的 `epsilon` 要足够大以达到低通滤波的效果。

彩色图像

对于彩色图像，色彩通道间并不存在平稳特性。因此我们通常首先对数据进行特征缩放（使像素值位于 [0,1] 区间），然后使用足够大的 `epsilon` 来做 PCA/ZCA。注意在进行 PCA 变换前需要对特征进行分量均值归零化。

音频 (MFCC/**频谱图**)

对于音频数据 (MFCC 和频谱图)，每一维度的取值范围（方差）不同。例如 MFCC 的第一分量是直流分量，通常其幅度远大于其他分量，尤其当特征中包含时域导数 (temporal derivatives) 时（这是音频处理中的常用方法）更是如此。因此，对这类数据的预处理通常从简单的数据标准化开始（即使得数据的每一维度均值为零、方差为 1），然后进行 PCA/ZCA 白化（使用合适的 `epsilon`）。

MNIST **手写数字**

MNIST 数据集的像素值在 [0,255] 区间中。我们首先将其缩放到 [0,1] 区间。实际上，进行逐样本均值消去也有助于特征学习。注：也可选择以对 MNIST 进行 PCA/ZCA 白化，但这在实践中不常用。

B.2 用反向传导思想求导

B.2.1 简介

在反向传导算法 (2.2) 一节中，我们介绍了在稀疏自编码器中用反向传导算法来求梯度的方法。事实证明，反向传导算法与矩阵运算相结合的方法，对于计算复杂矩阵函数（从矩阵到实数的函数，或用符号表示为：从 $\mathbb{R}^{r\times c}\to\mathbb{R}$）的梯度是十分强大和直观的。

首先，我们回顾一下反向传导的思想，为了更适合我们的目的，将其稍作修改呈现于下:

1. 对第 n_l 层（最后一层）中的每一个输出单元 i，令

$$\delta_i^{(n_l)}=\frac{\partial}{\partial z_i^{(n_l)}}\ J(z^{(n_l)})$$

   其中 $J(z)$ 是我们的“目标函数”（稍后解释）。

2. 对 $l=n_l-1,n_l-2,n_l-3,\ldots,2$,

   对第 l 层中的每个节点 i, 令

$$\delta_i^{(l)}=\left(\sum_{j=1}^{s_{l+1}}W_{ji}^{(l)}\delta_j^{(l+1)}\right)\bullet\frac{\partial}{\partial z_i^{(l)}}f^{(l)}(z_i^{(l)})$$

3. 计算我们要的偏导数

$$\nabla_{W^{(l)}} J(W,b;x,y) = \delta^{(l+1)}(a^{(l)})^T, \tag{112}$$

$$\tag{113}$$

符号扼要重述:

- l 是神经网络的层数
- n_l 第 l 层神经元的个数
- $W_{ji}^{(l)}$ 是 l 层第 i 个节点到第 $(l+1)$ 层第 j 个节点的权重
- $z_i^{(l)}$ 是第 l 层第 i 个单元的输入
- $a_i^{(l)}$ 是第 l 层第 i 个节点的激励
- $A \bullet B$ 是矩阵的 Hadamard 积或逐个元素乘积，对 $r \times c$ 矩阵 A 和 B ，它们的乘积是 $r \times c$ 矩阵 $C = A \bullet B$ ，即 $C_{r,c} = A_{r,c} \cdot B_{r,c}$
- $f(l)$ 是第 l 层中各单元的激励函数

假设我们有一个函数 F，F 以矩阵 X 为参数生成一个实数。我们希望用反向传导思想计算 F 关于 X 的梯度，即 $\nabla_X F$。大致思路是将函数 F 看成一个多层神经网络，并使用反向传导思想求梯度。

为了实现这个想法，我们取目标函数为 $J(z)$，当计算最后一层神经元的输出时，会产生值 $F(X)$ 。对于中间层，我们将选择激励函数 $f^{(l)}$。

稍后我们会看到，使用这种方法，我们可以很容易计算出对于输入 X 以及网络中任意一个权重的导数。

B.2.2 示例

为了阐述如何使用反向传导思想计算关于输入的导数，我们要在示例 1, 示例 2 中用稀疏编码 (10.2) 章节中的两个函数。在示例 3 中，我们使用独立成分分析 (11.1) 一节中的一个函数来说明使用此思想计算关于权重的偏导的方法，以及在这种特殊情况下，如何处理相互捆绑或重复的权重。

示例 1：稀疏编码中权重矩阵的目标函数

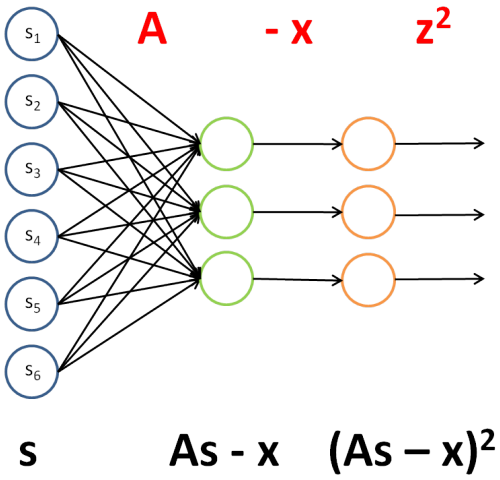
回顾一下稀疏编码 (10.2)，当给定特征矩阵 s 时，权重矩阵 A 的目标函数为:

$$F(A;s) = \|As - x\|_2^2 + \gamma\|A\|_2^2$$

我们希望求 F 对于 A 的梯度，即 $\nabla_A F(A)$ 。因为目标函数是两个含 A 的式子之和，所以它的梯度是每个式子的梯度之和。第二项的梯度很容易求，因此我们只考虑第一项的梯度。

第一项, $\|As-x\|_2^2$, 可以看成一个用 s 做输入的神经网络的实例，通过四步进行计算，文字以及图形描述如下：

1. 把 A 作为第一层到第二层的权重。
2. 将第二层的激励减 x，第二层使用了单位激励函数。
3. 通过单位权重将结果不变地传到第三层。在第三层使用平方函数作为激励函数。
4. 将第三层的所有激励相加



该网络的权重和激励函数如下表所示：

层	权重	激励函数 f
1	A	$f(z_i)=z_i$ （单位函数）
2	I （单位向量）	$f(z_i)=z_i-x_i$
3	N/A	$f(z_i)=z_i^2$

为了使 $J(z(3))=F(x)$，我们可令 $J(z^{(3)})=\sum_k J(z_k^{(3)})$ 。
一旦我们将 F 看成神经网络，梯度 $\nabla_X F$ 就很容易求了——使用反向传导得到：

层	激励函数的导数 f'	Delta	该层输入 z
3	$f'(z_i)=2z_i$	$f'(z_i)=2z_i$	$As-x$
2	$f'(z_i)=1$	$\left(I^T\delta^{(3)}\right)\bullet 1$	As
1	$f'(z_i)=1$	$\left(A^T\delta^{(2)}\right)\bullet 1$	s

因此

$$\nabla_X F = A^T I^T 2(As - x) \tag{114}$$

$$= A^T 2(As - x) \tag{115}$$

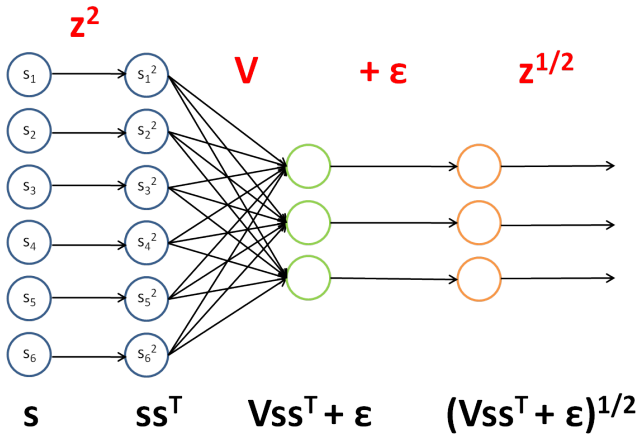
示例 2：稀疏编码中的平滑地形 L1 稀疏罚函数

回顾稀疏编码 (10.2) 一节中对 s 的平滑地形 L1 稀疏罚函数：

$$\sum \sqrt{Vss^T + \epsilon}$$

其中 V 是分组矩阵，s 是特征矩阵，ϵ 是一个常数。

我们希望求得 $\nabla_s \sum \sqrt{Vss^T + \epsilon}$。像上面那样，我们把这一项看做一个神经网络的实例：



该网络的权重和激励函数如下表所示：

层	权重	激励函数 f
1	I	$f(z_i) = z_i^2$
2	V	$f(z_i) = z_i$
3	I	$f(z_i) = z_i + \epsilon$
4	N/A	$f(z_i) = z_i^{\frac{1}{2}}$

为使 $J(z(4)) = F(x)$，我们可令 $J(z^{(4)}) = \sum_k J(z_k^{(4)})$。

一旦我们把 F 看做一个神经网络，梯度 $\nabla_X F$ 变得很容易计算——使用反向传导得到：

因此

$$\nabla_X F = I^T V^T I^T \frac{1}{2}(Vss^T + \epsilon)^{-\frac{1}{2}} \bullet 2s \tag{116}$$

$$= V^T \frac{1}{2}(Vss^T + \epsilon)^{-\frac{1}{2}} \bullet 2s \tag{117}$$

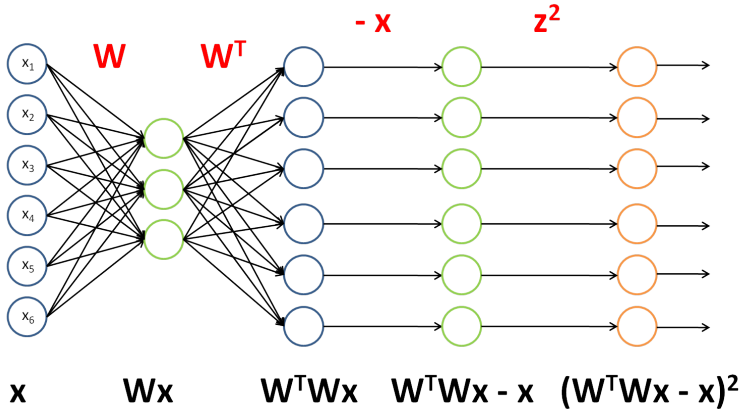
$$= V^T (Vss^T + \epsilon)^{-\frac{1}{2}} \bullet s \tag{118}$$

层	激励函数的导数 f'	Delta	该层输入 z
4	$f'(z_i) = \frac{1}{2} z_i^{-\frac{1}{2}}$	$f'(z_i) = \frac{1}{2} z_i^{-\frac{1}{2}}$	$(Vss^T + \epsilon)$
3	$f'(z_i) = 1$	$\left(I^T \delta^{(4)}\right) \bullet 1$	Vss^T
2	$f'(z_i) = 1$	$\left(V^T \delta^{(3)}\right) \bullet 1$	ss^T
1	$f'(z_i) = 2z_i$	$\left(I^T \delta^{(2)}\right) \bullet 2s$	s

示例 3：ICA 重建代价

回顾独立成分分析 (ICA 11.1) 一节重建代价一项：$\|W^T W x - x\|_2^2$，其中 W 是权重矩阵，x 是输入。

我们希望计算 $\nabla_W \|W^T W x - x\|_2^2$ – 对于权重矩阵的导数，而不是像前两例中对于输入的导数。不过我们仍然用类似的方法处理，把该项看做一个神经网络的实例：



该网络的权重和激励函数如下表所示：

层	权重	激励函数 f
1	W	$f(z_i) = z_i$
2	W^T	$f(z_i) = z_i$
3	I	$f(z_i) = z_i - x_i$
4	N/A	$f(z_i) = z_i^2$

为使 $J(z(4)) = F(x)$，我们可令 $J(z^{(4)}) = \sum_k J(z_k^{(4)})$ 。

既然我们可将 F 看做神经网络，我们就能计算出梯度 $\nabla_W F$ 。然而，我们现在面临的难题是 W 在网络中出现了两次。幸运的是，可以证明如果 W 在网络中出现多次，那么对于 W 的梯度是对网络中每个 W 实例的梯度的简单相加（你需要自己给出对这一事实的严格证明来说服自己）。知道这一点后，我们将首先计算 delta：

为计算对于 W 的梯度，首先计算对网络中每个 W 实例的梯度。

层	激励函数的导数 f'	Delta	该层输入 z
4	$f'(z_i) = 2z_i$	$f'(z_i) = 2z_i$	$(W^TWx - x)$
3	$f'(z_i) = 1$	$\left(I^T\delta^{(4)}\right) \bullet 1$	W^TWx
2	$f'(z_i) = 1$	$\left((W^T)^T\delta^{(3)}\right) \bullet 1$	Wx
1	$f'(z_i) = 1$	$\left(W^T\delta^{(2)}\right) \bullet 1$	x

对于 W^T:

$$\nabla_{W^T} F = \delta^{(3)} a^{(2)T} \tag{119}$$

$$= 2(W^TWx - x)(Wx)^T \tag{120}$$

对于 W:

$$\nabla_W F = \delta^{(2)} a^{(1)T} \tag{121}$$

$$= (W^T)(2(W^TWx - x))x^T \tag{122}$$

最后进行求和，得到对于 W 的最终梯度，注意我们需要对 W^T 梯度进行转置，来得到关于 W 的梯度（原谅我在这里稍稍滥用了符号）：

$$\nabla_W F = \nabla_W F + (\nabla_{W^T} F)^T \tag{123}$$

$$= (W^T)(2(W^TWx - x))x^T + 2(Wx)(W^TWx - x)^T \tag{124}$$

C Miscellaneous

C.1 MATLAB Modules

Sparse autoencoder | sparseae_exercise.zip

- checkNumericalGradient.m - Makes sure that computeNumericalGradient is implmented correctly
- computeNumericalGradient.m - Computes numerical gradient of a function (to be filled in)
- display_network.m - Visualizes images or filters for autoencoders as a grid
- initializeParameters.m - Initializes parameters for sparse autoencoder randomly
- sampleIMAGES.m - Samples 8×8 patches from an image matrix (to be filled in)

- sparseAutoencoderCost.m - Calculates cost and gradient of cost function of sparse autoencoder
- train.m - Framework for training and testing sparse autoencoder

Using the MNIST Dataset | mnistHelper.zip

- loadMNISTImages.m - Returns a matrix containing raw MNIST images
- loadMNISTLabels.m - Returns a matrix containing MNIST labels

PCA and Whitening | pca_exercise.zip

- display_network.m - Visualizes images or filters for autoencoders as a grid
- pca_gen.m - Framework for whitening exercise
- sampleIMAGESRAW.m - Returns 8×8 raw unwhitened patches

Softmax Regression | softmax_exercise.zip

- checkNumericalGradient.m - Makes sure that computeNumericalGradient is implmented correctly
- display_network.m - Visualizes images or filters for autoencoders as a grid
- loadMNISTImages.m - Returns a matrix containing raw MNIST images
- loadMNISTLabels.m - Returns a matrix containing MNIST labels
- softmaxCost.m - Computes cost and gradient of cost function of softmax
- softmaxTrain.m - Trains a softmax model with the given parameters
- train.m - Framework for this exercise

C.2 Style Guide

File / Function Names

Functions and file names should be alphanumeric, with the first letter of the first word in lowercase, and the first letter in the remaining words in uppercase. E.g.

Variable Names

Variable names should follow the same convention as the style guide.

C.3 Useful Links

Matlab Guide

Writing Fast MATLAB Code (by Pascal Getreuer)

Matrix Calculus Reference

The Matrix Cookbook

Notes on Convolutional Neural Networks

D 中文版本说明

D.1 中英文词汇对照

表 2: 中英文词汇对照

英文	中文
neural networks	神经网络
activation function	激活函数
hyperbolic tangent	双曲正切函数
bias units	偏置项
activation	激活值
forward propagation	前向传播
feedforward neural network	前馈神经网络 (参照 Mitchell 的《机器学习》的翻译)
Backpropagation Algorithm	反向传播算法
(batch) gradient descent	（批量）梯度下降法
(overall) cost function	（整体）代价函数
squared-error	方差
average sum-of-squares error	均方差
regularization term	规则化项
weight decay	权重衰减
bias terms	偏置项

续下页 ...

英文	中文
Bayesian regularization method	贝叶斯规则化方法
Gaussian prior	高斯先验概率
MAP	极大后验估计
maximum likelihood estimation	极大似然估计
activation function	激活函数
tanh function	双曲正切函数
non-convex function	非凸函数
hidden (layer) units	隐藏层单元
symmetry breaking	对称失效
learning rate	学习速率
forward pass	前向传导
hypothesis	假设值
error term	残差
weighted average	加权平均值
feedforward pass	前馈传导
Hadamard product	阿达马乘积
forward propagation	前向传播
off-by-one error	缺位错误
bias term	偏置项
numerically checking	数值检验
numerical roundoff errors	数值舍入误差
significant digits	有效数字
unrolling	组合扩展
learning rate	学习速率
Hessian matrix	Hessian 矩阵

续下页 ...

英文	中文
Newton's method	牛顿法
conjugate gradient	共轭梯度
step-size	步长值
自编码算法 Autoencoders	
稀疏性 Sparsity	
神经网络 neural networks	
监督学习 supervised learning	
无监督学习 unsupervised learning	
隐藏神经元 hidden units	
像素灰度值 the pixel intensity value	
独立同分布 IID	
主元分析 PCA	
激活 active	
抑制 inactive	
激活函数 activation function	
激活度 activation	
平均活跃度 the average activation	
稀疏性参数 sparsity parameter	
惩罚因子 penalty term	
相对熵 KL divergence	
伯努利随机变量 Bernoulli random variable	
总体代价函数 overall cost function	
后向传播 backpropagation	
前向传播 forward pass	

续下页 ...

英文	中文
梯度下降 gradient descent	
目标函数 the objective	
梯度验证方法 the derivative checking method	
可视化 Visualizing	
自编码器 Autoencoder	
隐藏单元 hidden unit	
非线性特征 non-linear feature	
激励 activate	
平凡解 trivial answer	
范数约束 norm constrained	
稀疏自编码器 sparse autoencoder	
有界范数 norm bounded	
输入域 input domains	
逻辑回归 Logistic Regression	
批量梯度上升法 batch gradient ascent	
截距 intercept term	
对数似然函数 the log likelihood	
导函数 derivative	
梯度 gradient	
向量化 vectorization	
正向传播 forward propagation	
反向传播 backpropagation	
训练样本 training examples	
激活函数 activation function	

续下页 ...

英文	中文
稀疏自编码网络 sparse autoencoder	
稀疏惩罚 sparsity penalty	
平均激活率 average firing rate	
Principal Components Analysis 主成份分析	
whitening 白化	
intensity 亮度	
mean 平均值	
variance 方差	
covariance matrix 协方差矩阵	
basis 基	
magnitude 幅值	
stationarity 平稳性	
normalization 归一化	
eigenvector 特征向量	
eigenvalue 特征值	
白化 whitening	
冗余 redundant	
方差 variance	
平滑 smoothing	
降维 dimensionality reduction	
正则化 regularization	
反射矩阵 reflection matrix	
去相关 decorrelation	
Softmax 回归 Softmax Regression	

续下页 ...

英文	中文
有监督学习 supervised learning	
无监督学习 unsupervised learning	
深度学习 deep learning	
logistic 回归 logistic regression	
截距项 intercept term	
二元分类 binary classification	
类型标记 class labels	
估值函数/估计值 hypothesis	
代价函数 cost function	
多元分类 multi-class classification	
权重衰减 weight decay	
自我学习/自学习 self-taught learning	
无监督特征学习 unsupervised feature learning	
自编码器 autoencoder	
白化 whitening	
激活量 activation	
稀疏自编码器 sparse autoencoder	
半监督学习 semi-supervised learning	
自我学习 self-taught learning	
深层网络 deep networks	
微调 fine-tune	
稀疏自编码器 sparse autoencoder	
梯度下降 gradient descent	
非监督特征学习 unsupervised feature learning	

续下页 ...

英文	中文
pre-training	预训练
深度网络 Deep Networks	
深度神经网络 deep neural networks	
非线性变换 non-linear transformation	
激活函数 activation function	
简洁地表达 represent compactly	
“部分 -整体”的分解 part-whole decompositions	
目标的部件 parts of objects	
高度非凸的优化问题 highly non-convex optimization problem	
共轭梯度 conjugate gradient	
梯度的弥散 diffusion of gradients	
逐层贪婪训练方法 Greedy layer-wise training	
自动编码器 autoencoder	
微调 fine-tuned	
自学习方法 self-taught learning	
栈式自编码神经网络（可以考虑翻译为“多层自动编码机”或“多层自动编码神经网络”）Stacked autoencoder	
微调 Fine tuning	
反向传播算法 Backpropagation Algorithm	
前馈传递 feedforward pass	
激活值（可以考虑翻译为“激励响应”或“响应”）activation	

续下页 . . .

英文	中文

线性解码器 Linear Decoders
稀疏自编码 Sparse Autoencoder
输入层 input layer
隐含层 hidden layer
输出层 output layer
神经元 neuron
神经网络 neural network
自编码器 autoencoder
激励函数 activation function
鲁棒 robust
S 型激励函数 sigmoid activation function
tanh 激励函数 tanh function
线性激励函数 linear activation function
恒等激励函数 identity activation function
隐单元 hidden unit
权重 weight
偏差项 error term
反向传播算法 backpropagation
全联通网络 Full Connected Networks
稀疏编码 Sparse Autoencoder
前向输送 Feedforward
反向传播 Backpropagation
部分联通网络 Locally Connected Networks
连接区域 Contiguous Groups
视觉皮层 Visual Cortex
卷积 Convolution
固有特征 Stationary
池化 Pool
稀疏编码 Sparse Coding 无监督学习 unsupervised method 超完备基 over-complete bases 主成分分析 PCA 稀疏性 sparsity 退化 degeneracy 代价函数 cost function 重构项 reconstruction

英文	中文

D.2 翻译人员

表 3: 翻译人员

章节	翻译
2.1	孙逊（sunpaofu@foxmail.com），林锋（xlfg@yeah.net），刘鸿鹏飞（just.dark@foxmail.com），许利杰（csxulijie@gmail.com）
2.2	王方(fangkey@gmail.com)，林锋(xlfg@yeah.net)，许利杰(csxulijie@gmail.com)
2.3	袁晓丹（shadowwalker1991@gmail.com），王方（fangkey@gmail.com），林锋（xlfg@yeah.net），许利杰（csxulijie@gmail.com）
2.4	周韬(ztsailing@gmail.com)，葛燕儒(yrgehi@gmail.com)，林锋(xlfg@yeah.net)，余凯（kai.yu.cool@gmail.com）

索引

sigmoid, 2

tanh, 2

前向传播, 4
双曲正切函数, 2
反向传导, 5

激活函数, 2

神经元, 2
神经网络, 2