

目錄

Introduction	1.1
使用神经网络识别手写数字	1.2
感知机	1.2.1
sigmoid神经元	1.2.2
神经网络的结构	1.2.3
用简单的网络结构解决手写数字识别	1.2.4
通过梯度下降法学习参数	1.2.5
实现我们的神经网络来分类数字	1.2.6
迈向深度学习	1.2.7
反向传播算法是如何工作的	1.3
热身：一个基于矩阵的快速计算神经网络输出的方法	1.3.1
关于损失函数的两个假设	1.3.2
Hadamard积， $s \odot t$	1.3.3
反向传播背后的四个基本等式	1.3.4
四个基本方程的证明（自选）	1.3.5
反向传播算法	1.3.6
反向传播算法代码	1.3.7
为什么说反向传播算法很高效	1.3.8
反向传播：整体描述	1.3.9
改进神经网络的学习方法	1.4
交叉熵代价函数	1.4.1
用交叉熵解决手写数字识别问题	1.4.2
交叉熵的意义是什么？它又是怎么来的？	1.4.3
Softmax	1.4.4
过拟合和正则化	1.4.5
正则化	1.4.5.1
为什么正则化能够降低过拟合	1.4.5.2
c3s6	1.4.6
神经网络能够计算任意函数的视觉证明	1.5
为什么深度神经网络的训练是困难的	1.6

「 Neural Networks and Deep Learning 」

中文翻译（进行中）

说明

本项目是[Neural Networks and Deep Learning](#)的中文翻译

后面的翻译都发表在了微信公众号 **HIT_SCIR**，请关注搜索，已连载完毕

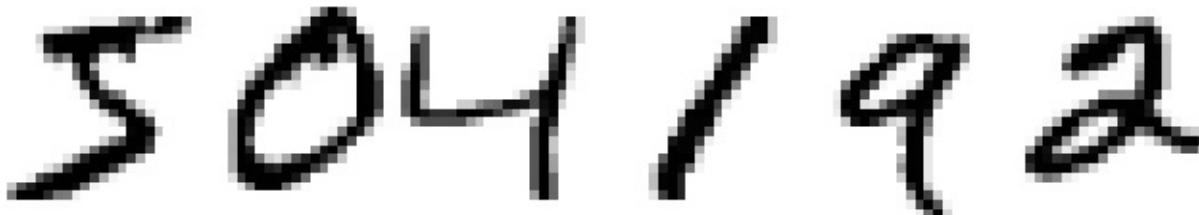
于 https://github.com/HIT-SCIR/Neural-Networks-and-Deep-Learning-zh_cn/issues/1 认领翻译和校对任务

Lisence

This work is licensed under a [Creative Commons Attribution-NonCommercial 3.0 Unported License](#).

使用神经网络识别手写数字

人类的视觉系统是世上的一个奇迹。考虑以下这串手写的数字：



大部分人都能轻易地识别出图上的数字是504192。这个看似简单的过程的背后，实际上很复杂。在我们大脑的每个脑半球中，有一个叫做初级视皮层（primary visual cortex）的部分，也被称作V1。它拥有1亿4千万个神经元，包含了上百亿的神经元连接。然而，人类的视觉系统不仅仅依赖于V1，还依赖于整套视皮层——V2、V3、V4和V5的协同工作，来实现复杂的图像处理任务。我们的大脑就像一台超级计算机，经过了上亿年的进化，才得以能够出色地理解视觉的世界。识别手写数字不是一个简单的任务，但是，人类极其擅长理解眼睛所看到的东西，并且几乎所有这些工作都是在无意识的情况下完成的，所以我们通常不会意识到我们的视觉系统解决了的任务有多么困难。

当你企图实现一个用来识别类似上图数字的计算机程序时，你就会逐渐意识到视觉模式识别的难度。一个本来对于人类看似很简单的事情，突然就变得极其困难。「数字9的上部有一个圆圈，右下部有一笔竖线」这种人类识别形状的直觉，在算法上却很难表示。当你试图定义明确的识别规则时，你会很快地被一大堆特例所困扰。这似乎毫无解决的希望。

神经网络（Neural Networks）使用一种不同的思路解决这个问题。它的思想是利用大量的手写数字，亦被称为训练样例，

0	4	1	9	2	1	3	1	4	3
5	3	6	1	7	2	8	6	9	4
0	9	1	1	2	4	3	2	7	3
8	6	9	0	5	6	0	7	6	1
8	7	9	3	9	8	5	9	3	3
0	7	4	9	8	0	9	4	1	4
4	6	0	4	5	6	1	0	0	1
7	1	6	3	0	2	1	1	7	9
0	2	6	7	8	3	9	0	4	6
7	4	6	8	0	7	8	3	1	5

从这些训练样例学习并建立一个系统。换一种说法，神经网络使用这些样例，从中能够自动地学习到识别手写数字的规则。而且，随着训练样例的增加，神经网络可以从中学习到更多信息，从而提高它的准确度。所以，尽管我上面只给出了100个训练数字，也许我们可以使用上千、上万、上亿的训练样例来构建一个更好的手写识别器。

在本章中，我们会实现一个神经网络的计算机程序，来学习并识别手写数字。虽然这个程序仅仅只有74行，并且没有使用任何特别的神经网络库，但是它可以在没有任何人工干预的情况下，达到超过百分之96的手写数字识别准确率。在之后的章节中，我们会进一步改善我们的方法，使之达到超过百分之99的准确率。实际上，目前最好的商用神经网络已经足够好到能被银行用来处理支票，以及被邮局用来识别地址。

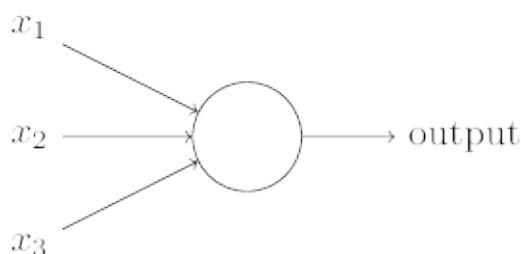
我们会专注于手写数字的识别这个问题，因为它基本上是学习神经网络最好的示范问题，之所以这么说是因为它戳中了几个痛点：它具有挑战性，识别手写数字不是一个简单的工程；同时它的难度也不是特别大，并不需要极其复杂的方法，或者大量的计算资源。而且它是实现更高级技术的基础，例如深度学习（*deep learning*）。所以，手写数字识别问题会贯穿本书。在本书的后部分，我们会讨论如何将这些想法应用到计算视觉的其它问题上，甚至语音处理、自然语言处理以及其它领域。

当然，本章的主旨如果只是实现一个程序来识别手写数字，那么本章的内容就会少很多！实际上，在这个过程中，我们会产生许多关于神经网络的关键思想，包括两种重要的人工神经元（感知机和sigmoid神经元），以及神经网络的标准学习算法，称为随机梯度下降（*stochastic gradient descent*）。在本书中，我关注于解释为什么问题能够被这样解决，以及为你建立关于神经网络的直觉。尽管相比于仅仅展示基础理论过程，这需要更长的篇幅来讨论，但是这对于你的深入理解是很有价值的。从中的收获，就是在章节的最后，我们就能够理解深度学习是什么，以及为什么它能够起作用。

感知机

什么是神经网络？在回答这个问题之前，我会先解释一种叫做感知机（perceptron）的人工神经元。感知机由科学家[Frank Rosenblatt](#)发明于1950至1960年代，他受到了来自[Warren McCulloch](#)和[Walter Pitts](#)的更早工作的启发。现如今，我们通常使用其它种类的人工神经元模型——在这本书里，以及在许多关于神经网络的最新工作里，主要使用的是一种叫做sigmoid神经元（sigmoid neuron）的神经元模型。我们会在稍后学习sigmoid神经元。为了理解sigmoid神经元，我们首先需要花一点时间来理解感知机，这是非常有价值的。

那么，感知机是怎么工作的呢？感知机的输入是几个二进制， x_1, x_2, \dots ，输出是一位单独的二进制：



本例中的感知机有三个输入， x_1, x_2, x_3 。通常，它可以有更多或者更少的输入。

[Rosenblatt](#)提出了一种计算输出的简单的规则。他引入了权重（weight）， w_1, w_2, \dots ，等实数来表示各个输入对于输出的重要程度。神经元的输出是0还是1，由加权和 $\sum_j w_j x_j$ 是否小于或者大于某一个阈值（threshold value）。和权重一样，阈值也是一个实数，同时它是神经元的一个参数。使用更严密的代数形式来表示：

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases} \quad (1)$$

这就是感知机的工作方式！

这是一个基础的数学模型。你可以这样理解感知机，它是一个通过给evidence赋予不同权重从而作出决策的机器。让我们来举一个例子，或许这不是一个真实的例子，但是它很容易理解，稍后我们会举一个更实际的例子。假设周末就要到了，你听说你所在的城市将会举办一个奶酪节。你非常喜欢奶酪，你正在犹豫是否要去参加这个节日。你可能需要权衡以下几个因素来作出决定：

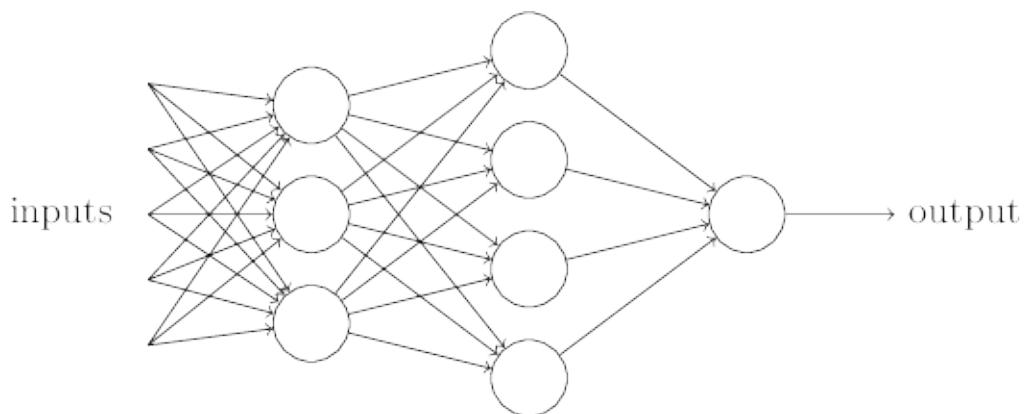
1. 天气好不好？
2. 你的男朋友或者女朋友会陪你去吗？
3. 坐公共交通方便去吗？（假设你自己没有车）

我们可以使用 x_1, x_2 和 x_3 这几个二进制变量来表示这三个因素。比如，如果天气很好，那么我们令 $x_1 = 1$ ，否则如果天气不好，那么 $x_1 = 0$ 。同样地，如果你的男朋友或者女朋友也想去，那么 $x_2 = 1$ ，否则 $x_2 = 0$ 。代表公共交通的 x_3 也用类似的方法来表示。

假设你真的超级喜欢奶酪，即便是你的男朋友或者女朋友的对此没兴趣并且交通也不方便的情况下，你也依然很想去。不过，你非常在意天气情况，如果天气不好的话你就不会去参加了。你可以用感知机来为这样的决策建立模型。一种方式是，你可以为天气赋予权重 $w_1 = 6$ ，为另外两个条件赋予权重 $w_2 = 2$ 和 $w_3 = 2$ 。相对来说值比较大的 w_1 表明了天气情况对你的决策影响更大，胜过了你的男朋友或者女朋友是否会陪你去，以及交通的便捷程度。最后，假设你选择5作为感知机的阈值。这样，这个感知机就构建起了一个决策模型，只要天气好就输出1，只要天气不好就输出0。你的男朋友或者女朋友是否会去，以及公共交通是否方便，都不会影响输出的结果。

通过调整权重和阈值的大小，我们可以得到不同的决策模型。例如，假设我们选择的阈值为3。那么此时，如果要让感知机做出你应该去参加这个节日的决策，就需要满足天气很好或者交通方便的同时你的男朋友或者女朋友也会陪你去。也就是说，这个决策模型与之前不同了。阈值的降低意味着你参加这个节日的意愿越强。

显然，感知机不能完全建模人类的决策系统！不过，这个例子阐明的是感知机如何赋予不同evidence权重来达到做出决策的目的。一个由感知机构成的复杂网络能够做出更加精细的决策，这貌似看上去是说得通的：



在这个网络中，第一列感知机——通常称为第一层感知机——通过赋予输入的evidence权重，做出三个非常简单的决策。第二层感知机呢？每一个第二层感知机通过赋予权重给来自第一层感知机的决策结果，来做出决策。通过这种方式，第二层感知机可以比第一层感知机做出更加复杂以及更高层次抽象的决策。第三层感知机能够做出更加复杂的决策。通过这种方式，一个多层网络感知机可以做出更加精细的决策。

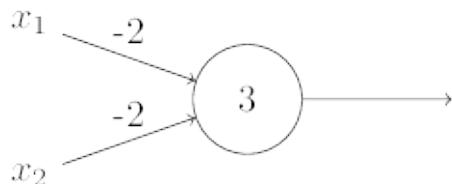
顺便提一句，当我之前定义感知机的时候，我说到感知机只有一个输出。在上面这个网络中，感知机似乎看上去有多个输出。实际上，它们仍然只有一个输出。多个输出箭头仅仅是用来方便表示它的输出被用作其它几个感知机的输入。如果画成一个输出分裂出几个箭头就太难看了。

让我们来简化一下对感知机的描述。 $\sum_j w_j x_j > \text{threshold}$ 的条件太笨重了，我们可以通过使用两个新记法来简化它。第一是使用点乘来代替 $\sum_j w_j x_j$ 。我们有 $w \cdot x \equiv \sum_j w_j x_j$ ，其中 w 和 x 都是向量，它们的元素分别代表了权重和输入。第二是将阈值移到不等号的另一侧，并使用偏移（bias）来代替阈值 threshold ， $b \equiv -\text{threshold}$ 。于是，感知机规则可以被重写为：

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \quad (2)$$

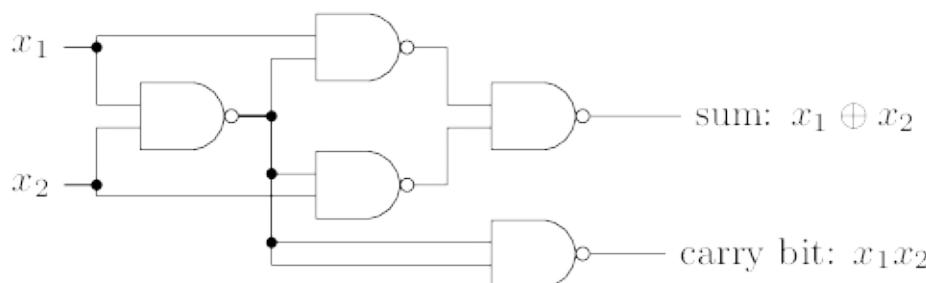
你可以将偏移（bias）理解为感知机为了得到输出为 1 的容易度的度量。如果从生物的角度来理解，偏移是使神经元被激活的容易度的度量。如果一个感知机的偏移非常大，那么这个感知机的输出很容易为 1，相反如果偏移非常小，那么输出 1 就很困难。显然，引入偏移的概念仅仅是一个很小的改动，稍后我们会对这个公式进行进一步的简化。在本书的剩余部分，我们将不再使用阈值，而是使用偏移（bias）。

我之前将感知机描述为通过权衡 evidence 做出决策的一种方法。感知机的另一种用途是计算初等逻辑函数，例如 AND、OR 和 NAND。例如，假如一个感知机有两个输入，每一个权重都是 -2 ，偏移为 3 ，如图：

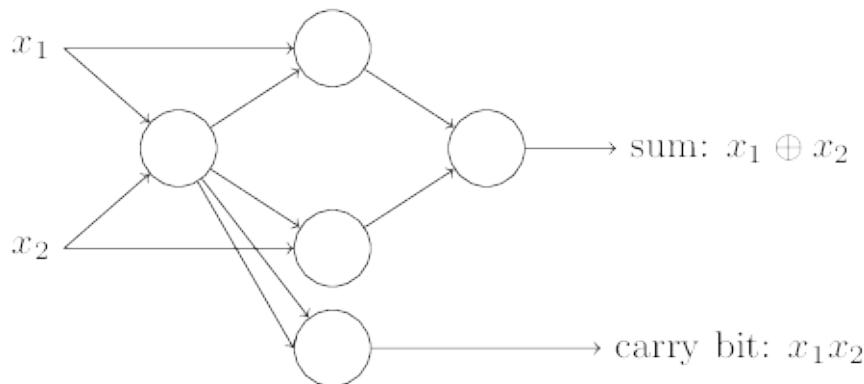


那么，对于输入 00 ，输出为 1 ，因为 $(-2) * 0 + (-2) * 0 + 3 = 3$ 的结果是正的。这里我用 * 符号显式写出了运算过程。类似地，对于输入 01 和 10 ，输出都是 1 。但是对于输入 11 ，输出为 0 ，因为 $(-2) * 1 + (-2) * 1 + 3 = -1$ 的结果是负的。从而，这个感知机就实现了 NAND 门！

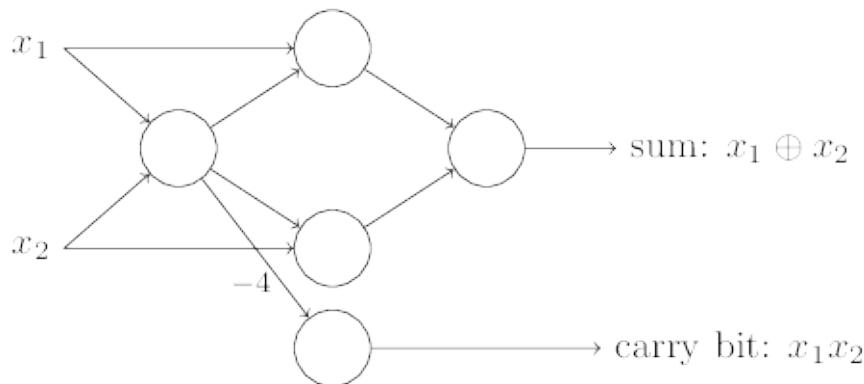
这个 NAND 的例子表明了我们可以使用感知机来计算初等逻辑函数。实际上，我们可以使用感知机网络计算任何逻辑函数，因为 NAND 门在计算上是通用的，这意味着我们可以使用 NAND 门构建任意逻辑运算。例如，我们可以使用 NAND 门构造一个求和两个比特 x_1 和 x_2 的电路。这需要计算 $x_1 \oplus x_2$ ，以及进位 $x_1 x_2$ ：



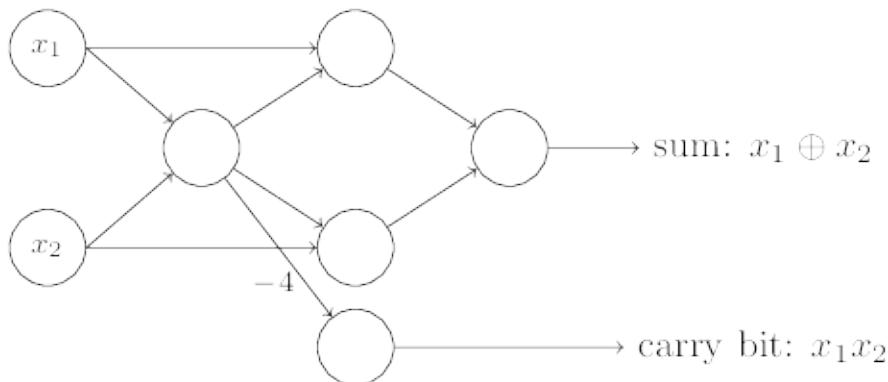
为了得到一个等价的感知机网络，我们可以将所有的 NAND 门替换成一个拥有两个输入，权重均为 -2 ，偏移为 3 的感知机。下图是替换后的结果。我将右下角的 NAND 门对应的感知机移动了一点位置，仅仅是为了方便画箭头。



值得注意的是，最左面的感知机的输出在作为最下面的感知机的输入时，被用到了两次。我之前定义感知机模型的时候，没有提到是否允许这种情况。实际上，这并没有什么影响。如果我们不允许这种情况发生，那么我们可以将这两条权重为 -2 的线合并成一个权重为 -4 的线。（如果你不能理解这点，你应该停下来试图证明这种替换是等价的。）通过这种变换，这个网络变成如下图的形式，所有未被标记的权重都为 -2 ，所有的偏移都为 3 ，以及一条被标出的边的权重为 -4 ：



到目前为止，我都将输入 x_1 和 x_2 当做变量画在了感知机网络的左边。实际上，大家更习惯使用另外一层感知机——输入层（input layer）——来为输入编码：



以下这个记号用来表示输入感知机，它有一个输出，没有输入，



但这并不意味着感知机可以没有输入。我们可以这样理解，假设存在一个没有输入的感知机，那么加权和 $\sum_j w_j x_j$ 恒为 0，所以如果 $b > 0$ ，那么感知机输出 1，如果 $b \leq 0$ ，那么感知机输出为 0。这意味着这个感知机仅仅输出一个固定的值，而非我们想要的值（在上面的例子中是 x_1 ）。所以，我们最好不要将输入感知机当做感知机，而是理解为一个特殊的单元，它能够输出我们想要的值 x_1, x_2, \dots 。

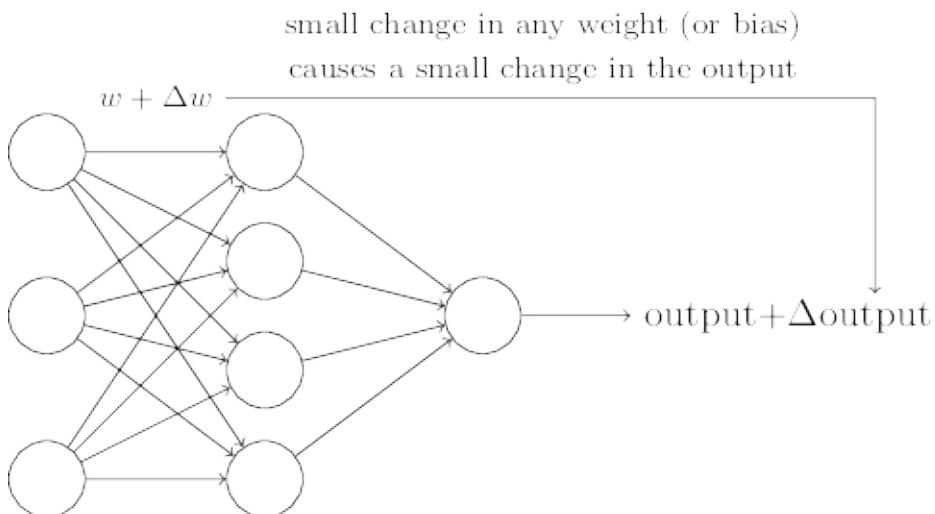
以上这个加法的例子展示了感知机网络如何模拟一个包含很多 NAND 门的电路。同时，由于 NAND 门对于计算是通用的，所以感知机对于计算也是通用的。

感知机的计算普遍性既让人感到安心，也让人感到失望。说它是安心的，因为感知机网络与任何其它的计算设备拥有同样强大的计算能力。说它是令人沮丧的，因为它看上去不过是一种新的 NAND 门，很难搞个大新闻。

不过，情况还是相对比较乐观的。我们可以设计学习算法 (learning algorithm) 使得能够自动地调整人工神经元网络的权重和偏移。这种调整能够对外部刺激作出响应，而不需要程序员的直接干预。这些学习算法使得我们能够用一种与传统逻辑门从根本上不同的方法使用人工神经元。我们不需要显式地排布 NAND 和其它逻辑门组成电路，取而代之的是，神经网络可以通过简单的学习来解决问题，通常这些问题用直接设计传统电路的方法来解决是非常困难的。

sigmoid神经元

学习算法是如此NICE的东西，但问题来了：我们该如何为神经网络量身设计一种学习算法呢？现在假设有一个由感知机构成的网络，我们想让这个网络学习如何去解决一些问题。举例来说，对于一个以手写数字的扫描图像的原始像素数据作为输入的网络，我们想要这个网络去学习权值（weights）和偏移（biases）以便最终正确地分类这些数字。为了说明学习算法如何才能有效，我们首先假设在网络的一些权值（或偏移）上做一个小的改变。我们期望的结果是，这些在权值上的小改变，将会为网络的输出结果带来相应的改变，且这种改变也必须是轻微的。我们在后面将会看到，满足这样的性质才能使学习变得可能。下面的图片反映了我们想要的结果（当然，图示的网络非常简单，它并不能被用来做手写数字识别）。

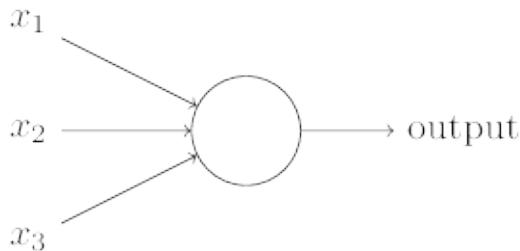


如果满足在权值（或偏移）上的小改变只会引起输出上的小幅变化这一性质，那么以此性质为基础，我们就可以改变权值和偏移来使得网络的表现越来越接近我们预期。例如，假设原始的网络会将一张写着「9」的手写数字图片错误分类为「8」。我们可以尝试找到一个正确的轻微改变权值和偏移的方法，来使得我们网络的输出更接近于正确答案——将该图片分类为「9」。重复这个过程，不断地修改权值和偏移并且产生越来越好的结果。这样我们的网络就开始学习起来了。

但问题在于，当我们的网络包含感知机时情况就与上述描述的不同了。事实上，轻微改变网络中任何一个感知机的权值或偏移有时甚至会导致感知机的输出完全翻转——比如说从0变为1。这个翻转行为可能以某种非常复杂的方式彻底改变网络中其余部分的行为。所以即使现在「9」被正确分类了，但网络在处理所有其他图片时的行为可能因一些难以控制的方式被彻底改变了。这导致我们逐步改变权值和偏移来使网络行为更加接近预期的学习方法变得很难实施。也许存在一些巧妙的方法来避免这个问题，但对于这种由感知机构成的网络，它的学习算法并不是显而易见的。

由此我们引入一种被称为S型(sigmoid)，通常我们更习惯使用它的英文称呼，所以本文的其他地方也将使用原始的英文)神经元的新型人工神经元来解决这个问题。sigmoid神经元与感知机有些相似，但做了一些修改使得我们在轻微改变其权值和偏移时只会引起小幅度的输出变化。这是使由sigmoid神经元构成的网络能够学习的关键因素。

好，接下来就开始介绍本节的主角了。我们将沿用感知机的方式来描述sigmoid神经元。



和感知机一样，sigmoid神经元同样有输入， x_1, x_2, \dots ，但不同的是，这些输入值不是只能取0或者1，而是可以取0到1间的任意浮点值。所以，举例来说，0.638...对于sigmoid神经元就是一个合法输入。同样，sigmoid神经元对每个输入也有相应的权值， w_1, w_2, \dots ，以及一个整体的偏移， b 。不过sigmoid神经元的输出不再是0或1，而是 $\sigma(w \cdot x + b)$ ，其中的 σ 被称为sigmoid函数（sigmoid function）¹，该函数定义如下：

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}. \quad (3)$$

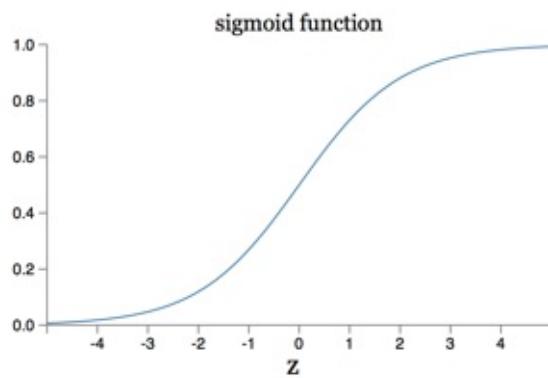
将上述内容总结一下，更加准确的定义，sigmoid神经元的输出是关于输入 x_1, x_2, \dots ，权值 w_1, w_2, \dots ，和偏移 b 的函数：

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}. \quad (4)$$

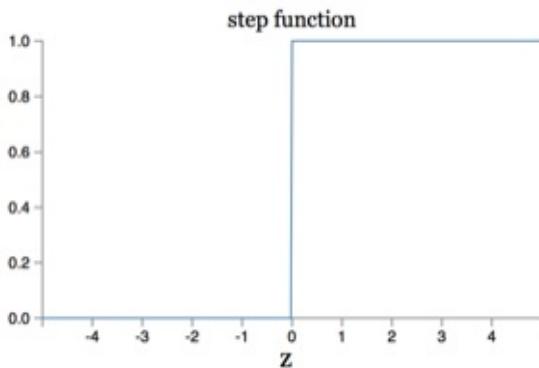
乍一看去，sigmoid神经元与感知机样子很不同。如果你对它不熟悉，sigmoid函数的代数形式看起来会有些晦涩难懂。但事实上，sigmoid神经元与感知机有非常多相似的地方。sigmoid函数的代数式更多地是展现了其技术细节，而不应是成为理解它的障碍。

为了理解sigmoid神经元与感知机模型的相似性，我们假设 $z \equiv w \cdot x + b$ 是一个很大的正数。这时 $e^{-z} \approx 0$ 且 $\sigma(z) \approx 1$ 。即是说，当 $z = w \cdot x + b$ 是一个很大的正数时，sigmoid神经元的输出接近于1，与感知机类似。另一方面，当 $z = w \cdot x + b$ 是一个绝对值很大的负数时， $e^{-z} \rightarrow \infty$ 且 $\sigma(z) \approx 0$ 。所以当 $z = w \cdot x + b$ 是一个绝对值很大的负数时，sigmoid神经元的行为与感知机同样很接近。只有当 $w \cdot x + b$ 是一个不太大的数时，其结果与感知机模型有较大的偏差。

我们不禁要问， σ 的代数式到底有何含义？我们该如何地理解它呢？事实上， σ 的确切形式并不是那么重要——对于我们理解问题，真正重要的是该函数画在坐标轴上的样子。下图表示了它的形状：



这个形状可以认为是下图所示阶梯函数（step function）的平滑版本：



如果把 σ 函数换成阶梯函数，那么sigmoid神经元就变成了一个感知机，这是因为此时它的输出只随着 $w \cdot x + b$ 的正负不同而仅在1或0这两个离散值上变化²。所以如前面所言，当使用 σ 函数时我们就得到了一个平滑的感知机。而且， σ 函数的平滑属性才是其关键，不用太在意它的具体代数形式。 σ 函数的平滑属性意味着当我们在权值和偏移上做出值为 Δw_j , Δb 的轻微改变时，神经元的输出也将只是轻微地变化 Δoutput 。事实上，由微积分的知识可知， Δoutput 近似于：

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b, \quad (5)$$

其中求和运算是将所有的权值 w_j 相加， $\partial \text{output}/\partial w_j$ 和 $\partial \text{output}/\partial b$ 表示分别求 output 对 w_j 和 b 的偏导。如果你看着偏微分不开心，不要惊慌，上面的公式虽然看起来有些复杂，但其实其中的偏微分非常简单（好消息~）： Δoutput 是关于权值和偏移的改变量 Δw_j 和 Δb 的线性函数（linear function）。这种线性属性，使得选择权值和偏移的轻微改变量并使输出按照预期发生小幅度变化成为易事。由上可知，sigmoid神经元不仅与感知机有很多相似的性质，同时也使描述「输出怎样随权值和偏移的改变而改变」这一问题变得简单。

如果真的只是 σ 的形状起作用而其具体代数形式没有什么用的话，为什么公式(3)要把 σ 表示为这种特定的形式？事实上，在书的后面部分我们也会偶尔提到一些在输出 $f(w \cdot x + b)$ 中使用其它激活函数（activation function） $f(\cdot)$ 的神经元。当我们使用其它不同的激活函数时主

要改变的是公式(5)中偏微分的具体值。在我们需要计算这些偏微分值之前，使用 σ 将会简化代数形式，因为指数函数在求微分时有着良好的性质。不管怎样， σ 在神经网络工作中是最常被用到的，也是本书中最频繁的激活函数。

我们该如何解释sigmoid神经元的输出呢？显然，感知机和sigmoid神经元一个巨大的不同在于，sigmoid神经元不仅仅只输出0或者1，而是0到1间任意的实数，比如0.173...，0.689...都是合法的输出。在一些例子，比如当我们想要把神经网络的输出值作为输入图片的像素点的平均灰度值时，这点很有用处。但有时这个性质也很讨厌。比如在我们想要网络输出关于「输入图片是9」与「输入图片不是9」的预测结果时，显然最简单的方式是如感知机那样输出0或者1。不过在实践中我们可以设置一个约定来解决这个问题，比如说，约定任何输出值大于等于0.5的为「输入图片是9」，而其他小于0.5的输出值表示「输入图片不是9」。当以后使用类似上面的一个约定时，我都会明确地说明，所以这并不会引起任何的困惑。

¹ 顺便提一句， σ 有时也被称作逻辑斯谛函数（logistic function），对应的这个新型神经元被称为逻辑斯谛神经元（logistic neurons）。记住这些术语很有用处，因为很多从事神经网络的人都会使用这些术语。不过本书中我们仍然使用sigmoid这一称呼。

² 事实上，当 $w \cdot x + b = 0$ 时感知机将输出0，但此时阶梯函数输出值为1。所以严格来讲，我们需要修改阶梯函数在0这个点的值。大家明白这点就好。

练习

- sigmoid神经元模拟感知机（第一部分）

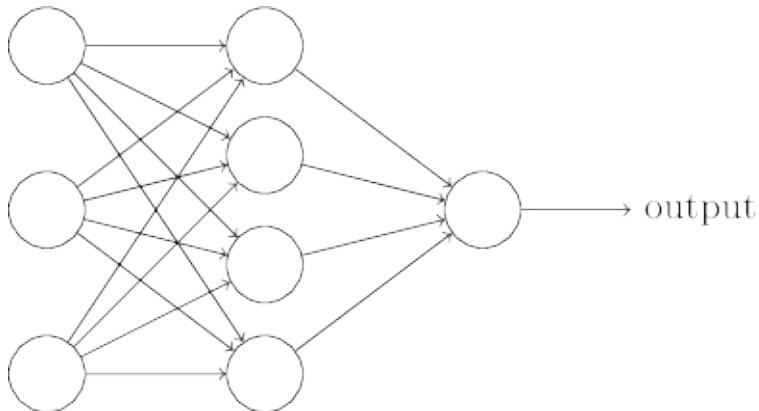
对一个由感知机构成的神经网络，假设将其中所有的权值和偏移都乘上一个正常数， $c > 0$ ，证明网络的行为并不会发生改变。

- sigmoid神经元模拟感知机（第二部分）

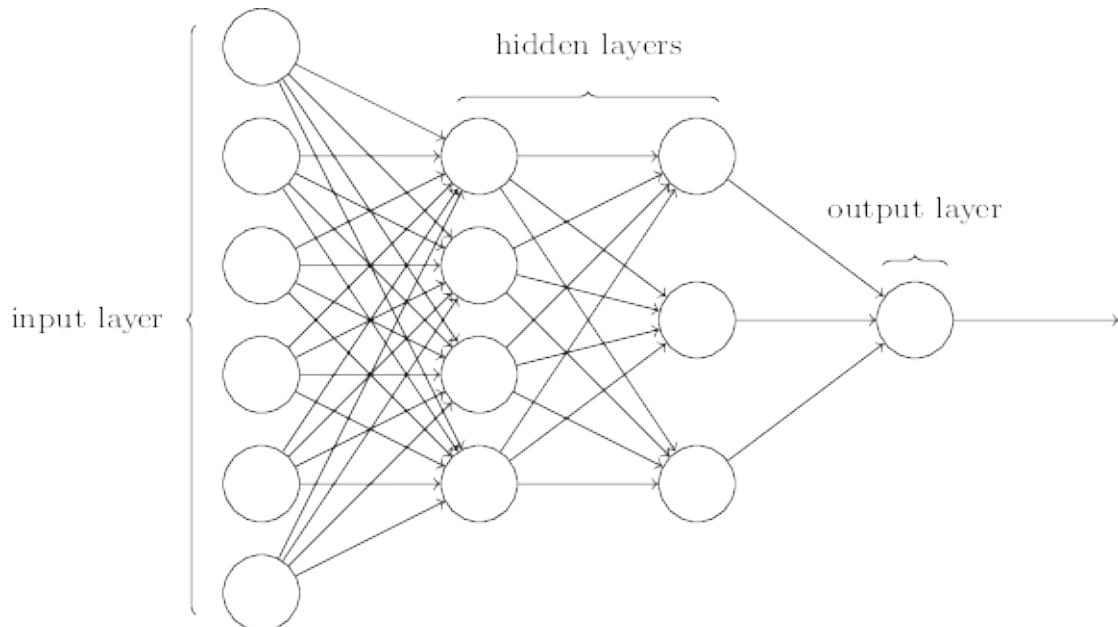
假设与上述问题相同的初始条件——由感知机构成的神经网络。假设感知机的所有输入都已经被选定。我们并不需要实际的值，只需要保证输入固定。假定对网络中任意感知机的输入 x 都满足 $w \cdot x + b \neq 0$ 。现在将网络中所有的感知机都替换为sigmoid神经元，然后将所有的权值和偏移都乘上一个正常数 $c > 0$ 。证明在极限情况即 $c \rightarrow \infty$ 下，这个由sigmoid神经元构成的网络与感知机构成的网络行为相同。同时想想当 $w \cdot x + b = 0$ 时为何不是如此？

神经网络的结构

下一章我将介绍一个能够良好分类手写数字的神经网络。在那之前，解释一些描述网络不同部分的术语是很有必要的。假设我们有如下网络：



如前所述，网络中最左边的一层被称作输入层，其中的神经元被称为输入神经元（**input neurons**）。最右边的一层是输出层（**output layer**），包含的神经元被称为输出神经元（**output neurons**）。在本例中，输出层只有一个神经元。网络中间的一层被称作隐层（**hidden layer**），因为它既不是输入层，也不是输出层。「隐」这个字听起来似乎有点神秘的感觉——当我第一次听到这个字时认为其必然包含着深刻的哲学或数学含义——然而除了「既非输入、又非输出」这个含义外，它真的没有别的意思了。上面的网络只有一个隐层，然而在一些网络中往往有多个隐层。比如下面图示的4层网络就有两个隐层：



由于历史原因，这些多层网络有时又被称为多层感知机（**multilayer perceptron**, MLP），这多少有些混淆，因为这些网络都是由**sigmoid**神经元构成的，而非感知机。在本书中我不准备使用MLP这个术语，因为我觉得这会带来困惑，在这里只想提醒读者这个名词的存在。

网络的输入输出层设计是比较直观的。比如说，假如我们尝试判断一张手写数字图片上面是否写着「9」。很自然地，我们将图片像素的灰度值作为网络的输入。假设图片是 64×64 的灰度图像，那么我们需要 $4,096 = 64 \times 64$ 个输入神经元，每个神经元接受规格化的0到1间的灰度值。输出层只需要包含一个神经元，当输出值小于0.5时说明「输入图片不是9」，否则表明「输入图片是9」。

相对于输入输出层设计的直观，隐层的设计就是一门艺术了。特别的，单纯地把一些简单规则结合到一起来作为隐层的设计是不对的。事实上，神经网络的研究者们已经总结了很多针对隐层的启发式设计规则，这些规则能够用来使网络变得符合预期。举例来说，一些启发式规则可以用来帮助我们在隐层数和训练网络所需的时间开销这二者间找到平衡。在书的后面部分我们就将遇到一些启发式设计规则。

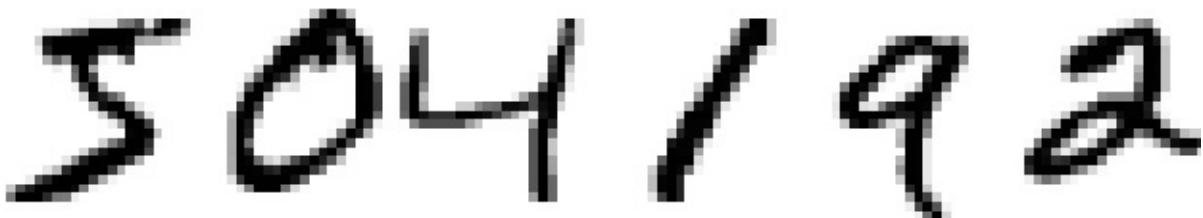
到目前为止，我们讨论的都是前馈神经网络（feedforward neural networks），即把上一层的输出作为下层输入的神经网络。这种网络是不存在环的——信息总是向前传播，从不反向回馈。如果我们要制造一个环，那么我们将会得到一个使 σ 函数输入依赖于其输出的网络。这很难去理解，所以我们并不允许存在这样的环路。

但是，我们也有一些存在回馈环路可能性的人工神经网络模型。这种模型被称为[递归神经网络](#)（recurrent neural networks）。该模型的关键在于，神经元在变为非激活态之前会在一段有限时间内均保持激活状态。这种激活状态可以激励其他的神经元，被激励的神经元在随后一段有限时间内也会保持激活状态。如此就会导致更多的神经元被激活，一段时间后我们将得到一个级联的神经元激活系统。在这个模型中环路并不会带来问题，因为神经元的输出只会在一段之间之后才影响到它的输入，它并非实时的。

递归神经网络比起前馈神经网络影响力小很多，一部分原因是递归神经网络的学习算法还不够强大，至少目前是如此。不过递归神经网络依然非常有吸引力。从思想上来看它要比前馈神经网络更接近我们大脑的工作方式。而且递归神经网络也可能解决一些重要的、前馈神经网络很难处理的问题。不过为控制篇幅，本书将主要关注更广泛应用的前馈神经网络。

用简单的神经网络识别手写数字

定义了神经网络之后，让我们回到手写数字识别的问题上来。我们可以把手写数字识别问题拆分为两个子问题。首先，我们要找到一种方法能够把一张包含若干数字的图像分割为若干小图片，其中每个小图像只包含一个数字。举个例子，我们想将下面的图像



分割为6张小图像



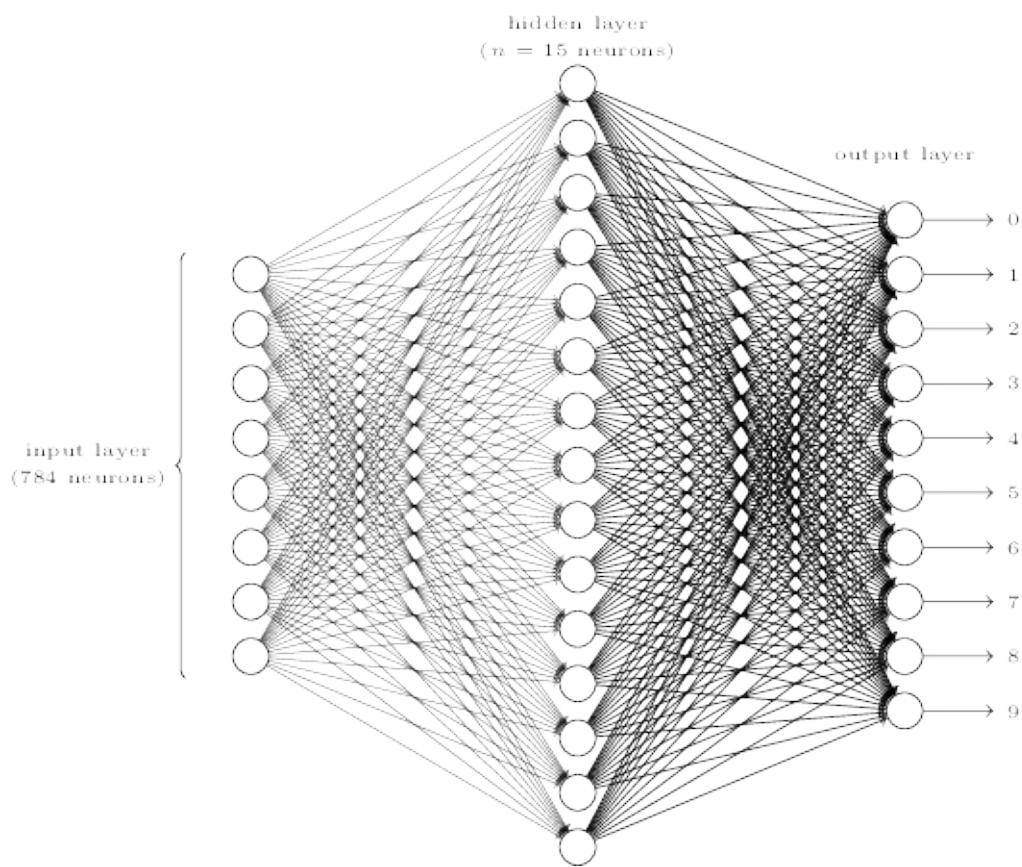
我们人类能够很容易地解决这个分割问题，但是让计算机去正确地分割图像是一件极具挑战的任务。当图像被分割之后，接下来的任务就是如何识别每个独立的手写数字。举个例子来说，我们想要程序去识别上述图像中的第一个数字，



结果是5。

我们将把精力集中在实现程序去解决第二个问题，即如何正确分类每个单独的手写数字。因为事实证明，只要你解决了数字分类的问题，分割问题相对来说不是那么困难。分割问题的解决方法有很多。一种方法是尝试不同的分割方式，用数字分类器对每一个切分片段打分。如果数字分类器对每一个片段的置信度都比较高，那么这个分割方式就能得到较高的分数；如果数字分类器在一或多个片段中出现问题，那么这种分割方式就会得到较低的分数。这种方法的思想是，如果分类器有问题，那么很可能是由于图像分割出错导致的。这种思想以及它的变种能够比较好地解决分割问题。因此，与其关心分割问题，我们不如把精力集中在设计一个神经网络来解决更有趣、更困难的问题，即手写数字的识别。

为了识别数字，我们将会使用一个三层神经网络：



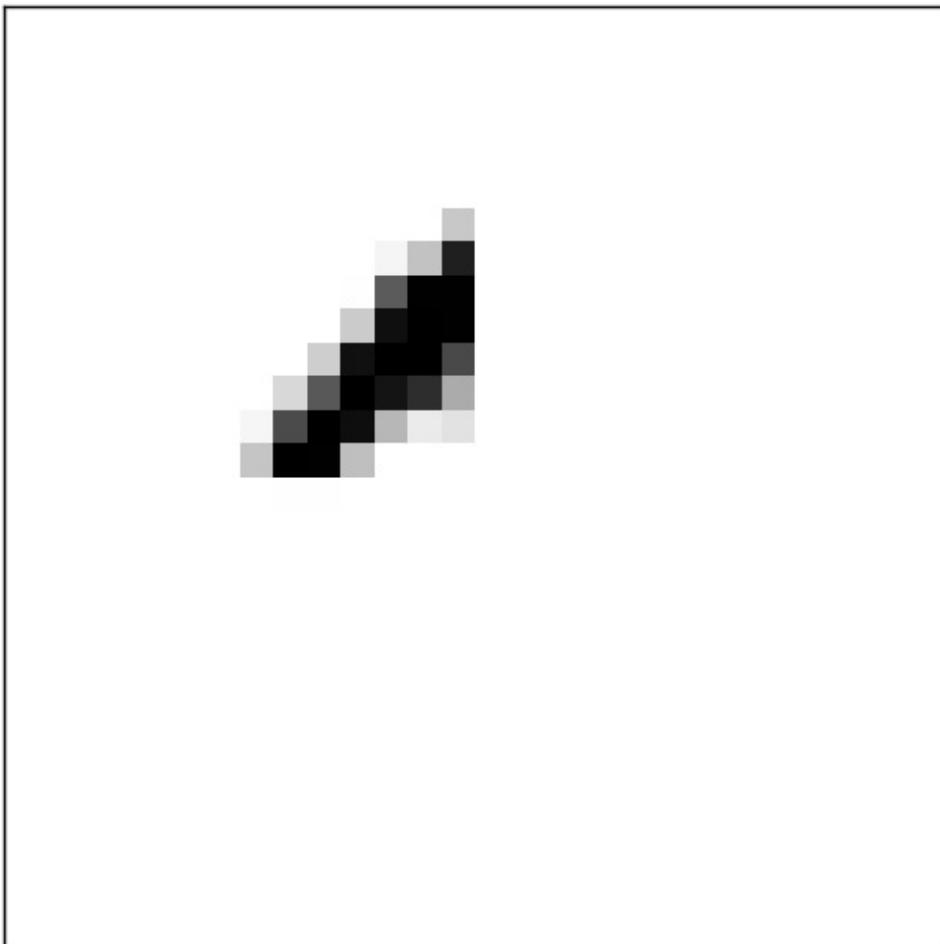
这个网络的输入层是对输入像素编码的神经元。正如我们在下一节将要讨论的，我们的训练数据是一堆 28×28 手写数字的位图，因此我们的输入层包含了 $784 = 28 \times 28$ 个神经元。为了方便起见，我在上图中没有完全画出784个输入神经元。输入的像素点是其灰度值，0.0代表白色，1.0代表黑色，中间值表示不同程度的灰度值。

网络的第二层是隐层。我们为隐层设置了 n 个神经元，我们会实验 n 的不同取值。在这个例子中我们只展现了一个规模较小的隐层，它仅包含了 $n = 15$ 个神经元。

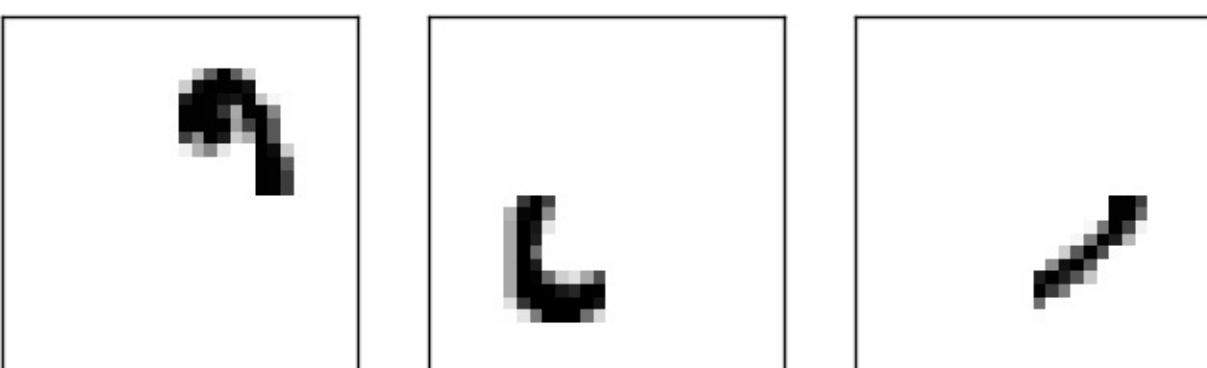
网络的输出层包含了10个神经元。如果第一个神经元被激活，例如输出 ≈ 1 ，然后我们可以推断出这个网络认为这个数字是0。如果第二层被激活那么我们可以推断出这个网络认为这个数字是1。以此类推。更精确一些的表述是，我们把输出层神经元依次标记为0到9，我们要找到哪一个神经元拥有最高的激活值。如果6号神经元有最高值，那么我们的神经网络预测输入数字是6。对其它的神经元也如此。

你可能会好奇为什么我们用10个输出神经元。毕竟我们的任务是能让神经网络告诉我们哪个数字 $(0, 1, 2, \dots, 9)$ 能和输入图片匹配。一个看起来更自然的方式就是使用4个输出神经元，把每一个当做一个二进制值，结果取决于它的输出更靠近0还是1。四个神经元足够编码这个问题了，因为 $2^4 = 16$ 大于10种可能的输入。为什么我们反而要用10个神经元呢？这样做难道效率不高吗？最终的判断是基于经验主义的：我们可以实验两种不同的网络设计，结果证明对于这个特定的问题而言，10个输出神经元的神经网络比4个的识别效果更好。但是令我们好奇的是为什么使用10个输出神经元的神经网络更有效呢。有没有什么启发性的思考能提前告诉我们用10个输出编码比使用4个输出编码更有好呢？

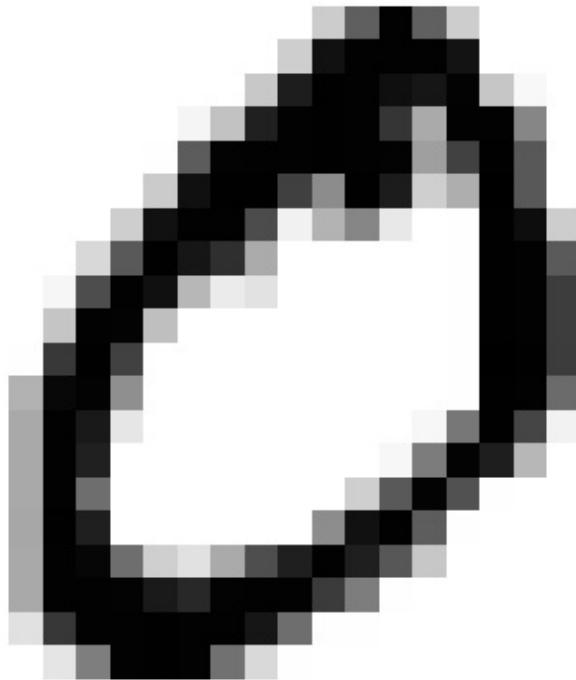
为了理解为什么我们这么做，我们需要从根本原理上理解神经网络究竟在做些什么。首先考虑有10个神经元的情况。我们首先考虑第一个输出神经元，它告诉我们一个数字是不是0。它能那么做是因为可以权衡从隐藏层来的信息。隐藏层的神经元在做什么呢？假设隐藏层的第一个神经元只是用于检测如下的图像是否存在：



为了达到这个目的，它通过对此图像对应部分的像素赋予较大权重，对其它部分赋予较小的权重。同理，我们可以假设第二，第三，第四个隐藏层的神经元是为检测下列图片是否存在：



正如你所猜到的那样，这四个图像拼到一起就组成了数字0：



如果所有这四个隐藏层的神经元被激活那么我们就可以推断出这个数字是0。当然，这不是我们推断出0的唯一方式——我们能通过很多其他合理的方式得到0（举个例子来说，通过上述图像的转换，或者稍微变形）。但至少在这个例子中我们可以推断出输入的数字是0。

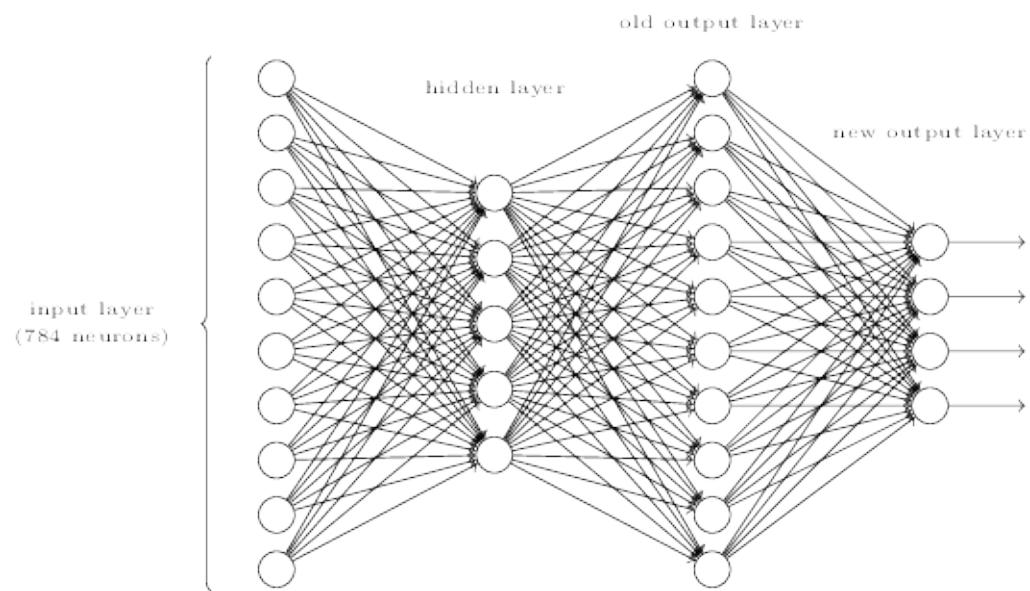
假设神经网络以上述方式运行，我们可以给出一个貌似合理的理由去解释为什么用10个输出而不是4个。如果我们有4个输出，那么第一个输出神经元将会尽力去判断数字的最高有效位是什么。把数字的最高有效位和数字的形状联系起来并不是一个简单的问题。很难想象出有什么恰当的历史原因，一个数字的形状要素会和一个数字的最高有效位有什么紧密联系。

上面我们说的只是一个启发性的思考。没有什么理由表明这个三层的神经网络必须按照我所描述的方式运行，即隐藏层是用来探测数字的组成形状。可能一个聪明的学习算法将会找到一些合适的权重能让我们仅仅用4个输出神经元就行。但是这个启发式的思考通常很有效，它会节省你大量时间去设计一个好的神经网络结构。

练习

- 通过在上述的三层神经网络加一个额外的一层就可以实现按位表示数字。额外的一层把原来的输出层转化为一个二进制表示，如下图所示。为新的输出层寻找一些合适的权重和偏移。假定原先的3层神经网络在第三层得到正确输出（即原来的输出层）的激活值至

少是0.99，得到错误的输出的激活值至多是0.01。



通过梯度下降法学习参数

我们已经设计出了我们的神经网络结构，现在的问题是，它要如何学习来识别数字呢？首先，我们需要一个待学习的数据集——即所谓的训练数据集。我们将会使用[MNIST数据集](#)，其中包含了成千上万个扫描手写数字图像以及它们正确的分类。MNIST数据集是由[NIST](#) (the United States' National Institute of Standards and Technology，美国国家标准与技术研究院) 收集了两个数据集后修改得到的子数据集构成，所以取名叫MNIST。下面是MNIST中的一些图像：



正如你所见，这些数字实际上就是我们在本章开始所提到的图像。当然，当我们测试我们的神经网络时，我们需要让它去识别训练集之外的数据。

MNIST数据集可以分为两个部分。第一个部分包含了60,000个用于训练的图像。这些图像是扫描了250个人的手写数字，他们之中一半是来自美国人口普查局的员工，另一半是高中生。这些图像是28乘28个像素点灰度图像。MNIST的第二个部分是10,000个用于测试的图像。同样，它们也是28乘28的灰度图像。我们将会用这些测试数据去评估我们的神经网络的识别效果。为了保证测试结果的准确性，这些测试数据是采样于另一个不同的250人的团体（尽管这些人也是美国人口普查局的员工和高中生）。这保证了我们的系统能够识别训练集之外的手写数字图像。

我们用 x 去定义训练输入。为了方便起见，我们将每一个训练输入 x 视为一个 $28 \times 28 = 784$ 维的向量。向量中的一个元素代表图像的一个像素点的灰度值。我们定义输出为 $y = y(x)$ ，每一个 y 是一个10维的向量。举例来说，对于一个数字6的训练图像 x ，我们期待的输出是 $y(x) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$ 。这里的 T 是转置符号，它能将一个行向量变为列向量。

现在我们需要一个算法能让我们找到合适的权重和偏置，从而对所有的训练输入为 x 的输出都近似于 $y(x)$ 。为了衡量我们当前取得的结果距离目标结果的好坏程度，我们定义一个代价函数：

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2. \quad (6)$$

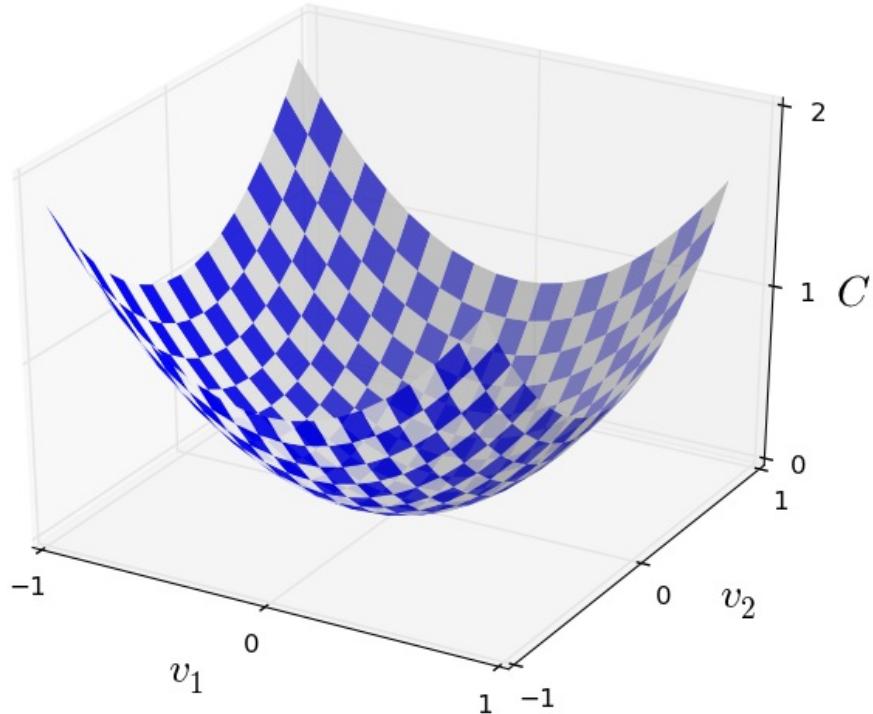
公式里面的 w 表示所有的权重的集合， b 表示所有的偏置， n 是训练数据的个数， a 代表 x 作为输入时输出层的向量，求和针对所有的训练输入 x 。显而易见，输出 a 取决于 x, w 和 b ，但是为了保持公式的简洁性，我们没有明确指出这种依赖性。符号 $\|v\|$ 是指求向量 v 的范数。这时，我们称 C 为二次代价函数；有时我们也称它为平方误差或者MSE。通过代价函数的形式我们可以得知 $C(w, b)$ 是非负的，因为加和的每一项都是非负的。另外，当所有的训练输入 x 的输出 a 基本都等于 $y(x)$ 时，代价函数 $C(w, b)$ 的值会变小，可能有 $C(w, b) \approx 0$ 。因此如果我们的学习算法可以找到合适的权重和偏置使得 $C(w, b) \approx 0$ ，那么这就是一个好的学习算法。反过来来说，如果 $C(w, b)$ 很大，那么就说明学习算法的效果很差——这意味着对于大量的输入，我们的结果 a 与正确结果 $y(x)$ 相差很大。因此，我们的训练算法的目标就是通过调整函数的权重和偏置来最小化代价函数 $C(w, b)$ 。换句话说，我们想寻找合适的权重和偏置让代价函数尽可能地小。我们将使用一个叫做梯度下降法 (*gradient descent*) 的算法来达到这个目的。

为什么要介绍平方代价 (quadratic cost) 呢？毕竟我们最初所感兴趣的内容不是对图像正确地分类么？为什么不增大正确输出的得分，而是去最小化一个像平方代价类似的间接评估呢？这么做是因为在神经网络中，被正确分类的图像的数量关于权重、偏置的函数并不是一个平滑的函数。大多数情况下，对权重和偏置做出的微小变动并不会影响被正确分类的图像的数量。这会导致我们很难去刻画如何去优化权重和偏置才能得到更好的结果。一个类似平方代价的平滑代价函数能够更好地指导我们如何去改变权重和偏置来达到更好的效果。这就是为何我们集中精力去最小化平方代价，只有通过这种方式我们才能让分类器更精确。

即使知道了我们需要选择一个平滑的代价函数，你可能仍然会好奇为什么我们选择方程 (6) 中的二次函数。这是一个拍脑袋的决定么？是不是我们选择另一个不同的代价函数将会得到完全不同的权重和偏置呢？这是一个合理的问题，稍后我们会再次提到这个代价函数并做一些修改。尽管如此，方程 (6) 中的平方代价函数能让我们更好地理解神经网络的基本原理，因此我们目前先用它。

再次回顾一下，我们训练神经网络的目的是寻找合适的权重和偏置来最小化平方代价函数 $C(w, b)$ 。这是一个明确定义了的问题，但是现在还存在很多让我们分散精力的部分——对权重 w 和偏置 b 合理的解释，隐藏在背后的 σ 函数，神经网络结构的选择，MNIST等等。实际上我们可以先忽略这些问题，把精力集中在最小化的问题上。现在让我们忘掉代价函数的具体形式，忘掉神经网络的结构，忘掉其它一切。现在想象我们仅仅是要去最小化一个给定的有很多变量的函数。我们即将要介绍一种可以解决最小化问题的技术，称作梯度下降法。然后我们再回到要在神经网络中最小化的函数上来。

假定我们要最小化某些函数， $C(v)$ 。它可能是任意的多元实值函数， $v = v_1, v_2, \dots$ 。注意我们将会用 v 去代表 w 和 b 以强调它可能是任意的函数——我们不会把问题局限在神经网络中。假定 $C(v)$ 有两个变量 v_1 和 v_2 ：



我们的目标就是找到 C 在何处取得全局最小值。当然，对于上图的函数，我们能通过肉眼就可以找到最小值。这是因为我所展示的函数太简单了！一个一般的函数 C ，可能是一个复杂的多元函数，通过肉眼通常找不到它的最小值。

一种解决方法就是用数值计算的方法去计算出它的最小值。我们可以计算出偏导，利用偏导去寻找函数 C 的极值点。运气好的话我们的函数 C 只有一个或者少数的几个变量。但是变量过多的话将是一个噩梦。在神经网络中我们通常会需要多得多得多的变量——最大的神经网络的代价函数包含了数亿个权重和偏置。根本没法使用数值计算的方法！

(我们只讨论了 C 只有两个变量的情况，如果说「嗨，如果函数有多于两个变量怎么办？」。对于这种情况我只能说很抱歉。请相信我把 C 看成一个二元函数是有助于我们的理解的。善于思考数学通常也包含善于利用多种直观的图片，学习什么时候用什么图片合适。)

所以，数值计算的方法没法完成这个任务了。幸运的是，有一个漂亮的类比预示着有一种算法能得到很好的效果。首先把我们的函数想象成一个山谷。我们想象有一个小球从山谷的斜坡滚落下来。我们的日常经验告诉我们这个球终会达到谷底。也许我们可以用这种思想来解决函数最小值的问题？我们随机地为小球选取一个起点，然后开始模拟小球滚落到谷底的运动过程。我们可以简单的通过计算 C 的导数（或者二阶导数）来模拟这个过程——这些导数将会告诉我们关于山谷「地形」的一切，从而告诉我们的小球该如何下落。

基于上述的想法，你可能会认为我们会写下牛顿运动定理，考虑摩擦力、重力等等。其实我们不打算真的去实现这个小球的类比——我们只想要一个算法去最小化 C ，而不是真的去模拟它真实的物理定律。小球的视角能激发我们的想象而不是束缚我们的思维。因此与其去考虑麻烦的物理定律，不如我们这样问自己：如果我们扮演一天的上帝，能够构造自己的物理定律，能够支配小球的下落方式，那么我们将会采取什么样的物理定律来让小球能够始终滑落到谷底呢？

为了更精确地描述这个问题，让我们想象一下如果我们在 v_1 方向移动一个很小的量 Δv_1 并在 v_2 方向移动一个很小的量 Δv_2 将会发生什么呢。通过计算可以告诉我们 C 将会产生如下改变：

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2. \quad (7)$$

我们将要寻找一种方式去选择 Δv_1 和 Δv_2 使得 ΔC 为负，因为负值能够让小球下落。为了搞清楚如何选择，我们有必要定义 Δv ，它是描述 v 变化的向量， $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$ ， T 是转置符号。我们也定义 C 用来表示偏导的向量， $\left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T$ 。我们用 ∇C 来表示梯度向量：

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T. \quad (8)$$

待会我们将用 Δv 和 ∇C 来重写 ΔC 。在这之前我想先说明一些令人困惑的关于梯度的事情。当第一次碰到梯度的符号 ∇C 时，人们常会好奇为什么 ∇C 符号是这样。 ∇ 究竟代表什么？事实上你可以把梯度仅仅看做一个简单的数学记号——一个方便用来表示偏导的向量——这样我们就不必写两个符号了。这样来看， ∇ 仅仅是一面旗帜，它告诉你：「嗨， ∇C 是一个梯度向量」。也有很多其他的关于 ∇ 的解释（比如，作为一种微分符号），但我们不需要这种观点。

定义好上述符号之后，关于 ΔC 的表达式 (7) 能被重写成如下形式

$$\Delta C \approx \nabla C \cdot \Delta v. \quad (9)$$

这个表达式能很好地解释为什么 ∇C 被称作梯度向量： ∇C 和 C 中 v 的变化密切相关，只是我们把它称为梯度罢了。但是这个式子真正让我们兴奋的是，它让我们看到了如何选取 Δv 才能让 ΔC 为负数。假设我们选取

$$\Delta v = -\eta \nabla C, \quad (10)$$

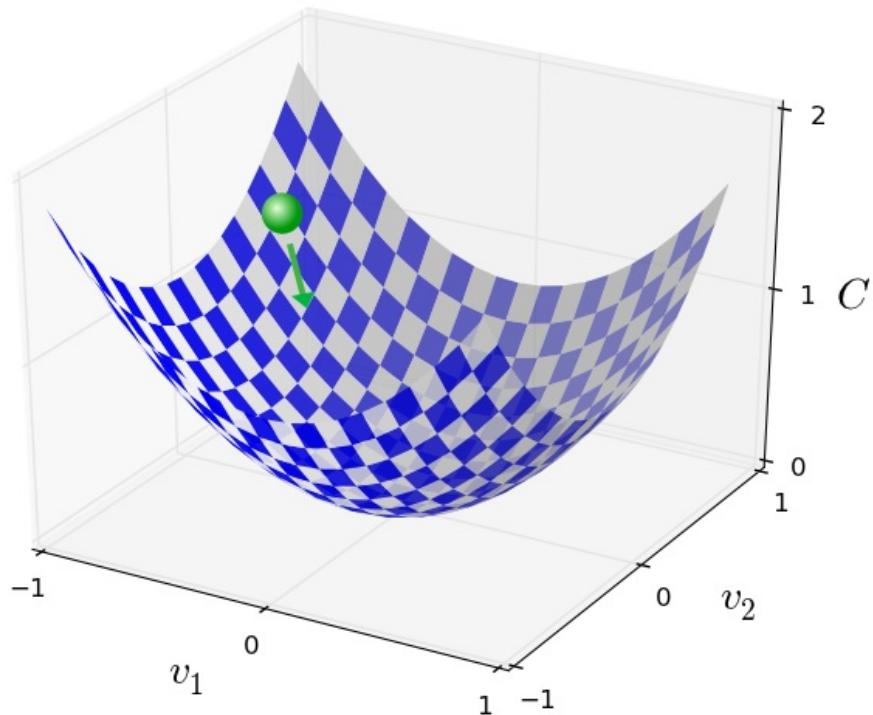
这里的 η 是个很小的正数（就是我们熟知的学习速率（learning rate））。等式 (9) 告诉我们 $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$ 。由于 $\|\nabla C\|^2 \geq 0$ ，这保证了 $\Delta C \leq 0$ ，例如，如果我们以等式 (10) 的方式去改变 v ，那么 C 将一直会降低，不会增加。（当然，要在

(9) 式的近似约束下)。这就是我们想要的特性！因此我们把 (10) 式看做梯度下降算法的「运动定律」。也就是说，我们用 (10) 式计算 Δv ，把小球位置 v 移动：

$$v \rightarrow v' = v - \eta \nabla C. \quad (11)$$

然后我们可以迭代地去更新。如果我们反复那么做，那么 C 会一直降低到我们想要寻找的全局最小值。

总结一下，梯度下降算法工作的方式就是重复计算梯度 ∇C ，然后沿着梯度的反方向运动，即沿着山谷的斜坡下降。就像下图一样：



需要注意这种梯度下降的规则不代表真实的物理运动。在真实世界里，小球有势能，势能让小球可以在山谷斜坡上滚动，甚至顷刻间滚向山顶。只有加上摩擦力的影响才能保证小球会下落到谷底。我们选择 Δv 的规则不一样，我们就好像在命令「立马下降！」。这仍然是一个寻找最小值的好办法！

为了使我们的梯度下降法能够正确地工作，我们需要选择足够小的学习速率 η 使得等式 (9) 能得到很好的近似。如果不那么做，我们将会以 $\Delta C > 0$ 结束，这显然不是一个很好的结果。与此同时，我们也不能把 η 设得过小，因为如果 η 过小，那么梯度下降算法就会变得异常缓慢。在真正的实现中， η 通常是变化的，从而方程 (9) 能保持很好地近似度同时保证算法不会太慢。我们稍后会看这是如何工作的。

我已经解释了具有两个变量的函数 C 的梯度下降法。但事实上当 C 是其他多元函数时也能很好地运行。我们假设 C 是一个有 m 个变量 v_1, \dots, v_m 的多元函数。我们对自变量做如下改变 $\Delta v = (\Delta v_1, \dots, \Delta v_m)^T$ ，那么 ΔC 将会变为

$$\Delta C \approx \nabla C \cdot \Delta v, \quad (12)$$

这里的 ∇C 是

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \dots, \frac{\partial C}{\partial v_m} \right)^T. \quad (13)$$

正如两个变量的例子一样，我们可以选取

$$\Delta v = -\eta \nabla C, \quad (14)$$

并且我们也会保证公式 (12) 中 ΔC 是负数。这使得我们能够随着梯度得到函数的最小值，即使 C 是任意的多元函数，我们也能重复运用下面的规则

$$v \rightarrow v' = v - \eta \nabla C. \quad (15)$$

你可以把这个更新规则看作梯度下降算法的定义。这给我们提供了一种方式去通过重复改变 v 来找到函数 C 的最小值。然而这种方式并不总是有效的——有几种情况能导致错误的发生，使得我们无法从梯度得到函数 C 的全局最小值，这种情况我们会在后面的章节中讨论。但在实践中，梯度下降法通常效果非常好，在神经网络中这是一种非常有效的方式去求代价函数的最小值，从而帮助神经网络学习。

事实上，有一种观念认为梯度下降法是求最小值的最优策略。我们假设努力去改变 Δv 来让 C 尽可能地减小，减小量为 $\Delta C \approx \nabla C \cdot \Delta v$ 。我们首先限制步长为固定值，即 $\|\Delta v\| = \epsilon$ ， $\epsilon > 0$ 。当步长固定时，我们要找到使得 C 减小最大的下降方向。可以证明，使得 $\nabla C \cdot \Delta v$ 取得最小值的 Δv 为 $\Delta v = -\eta \nabla C$ ，这里 $\eta = \epsilon / \|\nabla C\|$ 是由步长限制 $\|\Delta v\| = \epsilon$ 所决定的。因此，梯度下降法可以被视为一种通过在 C 下降最快的方向上做微小变化来使得 C 立即下降的方法。

练习

- 证明最后一段的断言。提示：利用柯西-斯瓦茨不等式。
- 我已经解释了当 C 是二元及其多元函数的情况。那如果 C 是一个单变量的函数呢？你能给出梯度下降法在一元函数的几何解释么？

人们已经研究出很多梯度下降法的变种，其中还包括一些去真正模拟真实物理运动的做法。这些模拟小球的变种有很多优点，但是也有一个主要的缺点：这些方法必须去计算 C 的二阶偏导，这可能代价非常大。为了理解为什么这种做法代价高，假设我们想求所有的二阶偏导

$\partial^2 C / \partial v_j \partial v_k$ 。如果我们有上百万的变量 v_j ，那我们必须要计算数万亿级别的二阶偏导！这的确会造成很大的计算代价。不过也有一些避免这些问题的技巧，寻找梯度下降方法的替代品也是个很活跃的研究领域。但在这本书中我们将主要用梯度下降法（包括变种）去训练我们的神经网络。

我们在神经网络中应用梯度下降法呢？方式就是利用梯度下降法去寻找权重 w_k 和偏置 b_l 能减小 (6) 式中的代价函数。为了弄清楚具体是如何工作的，我们将用权重和偏置代替变量 v_j 来重新描述梯度下降算法的更新规则。我们描述「位置」的元素是 w_k 和 b_l ，梯度向量 ∇C 对应 $\partial C / \partial w_k$ 和 $\partial C / \partial b_l$ 。申明了梯度下降中成分之后，我们得到

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \quad (16)$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}. \quad (17)$$

通过重复更新我们能「滚到山底」，从而有望能找到代价函数的最小值。换句话说，这个规则能用在神经网络的学习中。

应用梯度下降规则有很多挑战。我们将在下一章深入讨论。但是现在我想提及一个问题。为了理解问题是什么，我们先回顾 (6) 中的二次代价函数。注意这个代价函数有着如下的形式 $C = \frac{1}{n} \sum_x C_x$ ，也就是说，它是每个样本代价 $C_x \equiv \frac{\|y(x)-a\|^2}{2}$ 的平均值。事实上，为了计算梯度 ∇C ，我们需要为每个训练样本 x 单独地计算梯度值 ∇C_x ，然后求平均值，
 $\nabla C = \frac{1}{n} \sum_x \nabla C_x$ 。不幸的是，当训练输入过大时会花费很长时间，这样会使学习变得相当缓慢。

有种叫做随机梯度下降 (**SGD**) 的算法能够用来加速学习过程。想法就是通过随机选取小量输入样本来计算 ∇C_x ，进而可以计算 ∇C 。采取少量样本的平均值可以快速地得到梯度 ∇C ，这会加速梯度下降过程，进而加速学习过程。

更准确地说，**SGD** 是随机地选取小量的 m 个训练数据。我们将选取的这些训练数据标号 X_1, X_2, \dots, X_m ，并把它们称为一个 *mini-batch*。我们选取的 m 要足够大才能保证 ∇C_{X_j} 的平均值才能接近所有样本的平均值 ∇C_x ，也就是说，

$$\frac{\sum_{j=1}^m \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C, \quad (18)$$

其中，第二个求和是针对整个训练数据集的。交换两边我们可以得到

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^m \nabla C_{X_j}, \quad (19)$$

证实了我们可以通过计算随机选取的 *mini-batch* 的梯度来估计整体的梯度。

为了将这种想法更明确地联系到神经网络的学习中来，假设用 w_k 和 b_l 分别代表权重和偏置。SGD就是随机地选取大小为一个mini-batch的训练数据，然后去训练这些数据，

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \quad (20)$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}, \quad (21)$$

这个和就是当前mini-batch中每个 X_j 的梯度值。然后我们再随机选取另一个mini-batch去训练，直到我们用完了所有的训练数据，这就完成了一轮（epoch）迭代训练。这样我们就会重新开始下一轮迭代。

值得一提的是，关于调节代价函数和mini-batch去更新权重和偏置有很多不同的约定。在

(6) 式中，我们通过因子 $\frac{1}{n}$ 来缩放全部的代价函数。人们通常忽略 $\frac{1}{n}$ ，直接计算单独训练样本代价之和，而不是求平均值。这对我们不能提前知道训练数据集大小的情况下特别有效。这可能发生在有更多的训练数据是实时加入的情况下。同样，mini-batch的更新规则 (20) 和 (21) 有时也会舍弃 $\frac{1}{m}$ 。从概念上这也会有一点区别，因为它等价于重新调节学习速率 η 。但在对不同工作进行详细比较时，它是值得关注的。

我们能把SGD想象成一场民主选举：使用小规模的mini-batch计算梯度，比使用整个训练集计算梯度容易得多，就如开展一次民意调查比举行一次完整的选举容易得多。举个例子，在MNIST中有 60,000 个测试数据，我们选取mini-batch的大小为 $m = 10$ ，这样，计算梯度的过程就加速了 6,000 倍！当然，这个估计可能并不准确——仍然会存在统计波动——但也没必要准确：我们只关心在某个方向上移动可以减少 C ，这意味着我们没必要准确去计算梯度的精确值。事实上，SGD在神经网络的学习中被广泛使用的十分有效的技术，它也是本书中展开的大多数学习技术的基础。

练习

- 梯度下降一个比较极端的版本就是让mini-batch的大小变为1。这就是说，给定一个输入 x ，我们通过规则 $w_k \rightarrow w'_k = w_k - \eta \partial C_x / \partial w_k$ 和 $b_l \rightarrow b'_l = b_l - \eta \partial C_x / \partial b_l$ 更新权重和偏置。当我们选取另一个训练数据时也做同样的处理。其它的训练数据也做相同的处理。这种处理方式就是在线学习（online learning）或增量学习（incremental learning）。在线学习中，神经网络在一个时刻只学习一个训练数据（正如人类的做法）。说出online learning相比mini-batch为20的SGD的一个缺点和一个优点。

让我们讨论一个令刚刚接触梯度下降的人困惑的问题，来总结这部分的内容。在神经网络中，代价函数 C 是一个关于所有权重和偏置的多元函数，因此在某种意义上来说，就是在一高维空间定义了一个平面。有些人可能会那么想：「嗨，我必须要看见其它多出的维

度」。他们会开始疑惑：「我不能想象出四维空间，更不用说五维（或者五百万维）」。是不是他们缺少某种只有超级数学家才有的超能力？当然不是。即使大多数专业的数学家也不能想象出四维空间的样子。他们会用一些其它的方式来表示什么在发生。正如我们上面所做的那样，我们可以用代数（而不是几何）来表示 ΔC 如何变化才能让 C 减少。善于思考高维的人会在脑中做一些思想实验；我们的代数技巧是其中一个例子。这些方法可能不如观察三维那么直观，但我们熟悉这些方法之后会帮助我们思考更高维度。我不想在这里详细展开，如果你感兴趣，你可以读一下这篇关于数学家如何思考高维空间的[讨论](#)。我们讲的一些技术可能会有点复杂，但大多数内容还是比较直观的，任何人都能熟练掌握。

实现我们的神经网络来分类数字

好吧，现在让我们写一个学习怎么样识别手写数字的程序，使用随机梯度下降法和MNIST训练数据。我们需要做的第一件事情是获取MNIST数据。如果你是一个 git 用户，那么你能够通过克隆这本书的代码仓库获得数据，

```
git clone https://github.com/mnielsen/neural-networks-and-deep-learning.git
```

如果你不使用 git，那么你能够在[这里](#)下载数据和代码。

顺便说一下，当我在之前描述MNIST数据时，我说它分成了60,000个训练图像和10,000个测试图像。这是官方的MNIST的描述。实际上，我们将用稍微不同的方法对数据进行分割。我们将测试集保持原样，但是将60,000个图像的MNIST训练集分成两个部分：一部分50,000个图像，我们将用来训练我们的神经网络，和一个单独的10,000个图像的验证集（*validation set*）。在本章中我们不使用验证数据，但是在本书的后面我们将会发现它对于解决如何去设置神经网络中的超参数（*hyper-parameter*）——例如学习率等不是被我们的学习算法直接选择的参数——是很有用的。尽管验证数据集不是原始MNIST规范的一部分，然而许多人使用以这种方式使用MNIST，并且在神经网络中使用验证数据是很常见的。当我从现在起提到「MNIST训练数据」，我指的不是原始的60,000图像数据集，而是我们的50,000图像数据集1。

¹ 如前所述，MNIST数据集是基于NIST（美国国家标准与技术研究院）收集的两个数据集合。为了构建MNIST，NIST数据集合被Yann LeCun，Corinna Cortes和Christopher J. C. Burges拆分放入一个更方便的格式。更多细节请看[这个链接](#)。我的仓库中的数据集是在一种更容易在Python中加载和操纵MNIST数据的形式。我从蒙特利尔大学的LISA机器学习实验室获得了这个特殊格式的数据（[链接](#)）。

除了MNIST数据之外，我们还需要一个叫做Numpy的用于处理快速线性代数的Python库。如果你没有安装过Numpy，你能够在[这里](#)获得。

在给出一个完整的清单之前，让我解释一下神经网络代码的核心特征，如下。核心是一个 Network 类，我们用来表示一个神经网络。这是我们用来初始化一个 Network 对象的代码：

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                       for x, y in zip(sizes[:-1], sizes[1:])]
```

在这段代码中，列表 `sizes` 包含各层的神经元的数量。因此举个例子，如果我们想创建一个在第一层有2个神经元，第二层有3个神经元，最后层有1个神经元的 `network` 对象，我们应这样写代码：

```
net = Network([2, 3, 1])
```

`Network` 对象的偏差和权重都是被随机初始化的，使用 Numpy 的 `np.random.randn` 函数来生成均值为 0，标准差为 1 的高斯分布。随机初始化给了我们的随机梯度下降算法一个起点。在后面的章节中我们将会发现更好的初始化权重和偏差的方法，但是现在将采用随机初始化。注意 `Network` 初始化代码假设第一层神经元是一个输入层，并对这些神经元不设置任何偏差，因为偏差仅在之后的层中使用。

同样注意，偏差和权重以列表存储在 Numpy 矩阵中。因此例如 `net.weights[1]` 是一个存储着连接第二层和第三层神经元权重的 Numpy 矩阵。（不是第一层和第二层，因为 Python 列中的索引从 0 开始。）因此 `net.weights[1]` 相当冗长，让我们就这样表示矩阵 w 。矩阵中的 w_{jk} 是连接第二层的 k^{th} 神经元和第三层的 j^{th} 神经元的权重。这种 j 和 k 索引的顺序可能看着奇怪，当然交换 j 和 k 索引会更有意义？使用这种顺序的很大的优势是它意味着第三层神经元的激活向量是

$$a' = \sigma(wa + b). \quad (22)$$

这个等式有点复杂，所以让我们一块一块地进行理解。 a 是第二层神经元的激活向量。为了得到 a' ，我们用权重矩阵 w 乘以 a ，加上偏差向量 b ，我们然后对向量 $wa + b$ 中的每个元素应用函数 σ 。（这被叫做函数 σ 的向量化（vectorizing）。）很容易验证等式 (22) 给出了跟我们之前的计算一个 sigmoid 神经元的输出的等式 (4) 的结果相同。

练习

- 写出等式 (22) 的组成形式，并验证它跟我们之前的计算一个 sigmoid 神经元的输出的等式 (4) 的结果相同。

有了这些，很容易写从一个 `Network` 实例计算输出的代码。我们从定义 `sigmoid` 函数开始：

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

注意，当输入 z 是一个向量或者 Numpy 数组时，Numpy 自动的应用元素级的 `sigmoid` 函数，也就是向量化。

我们然后对 `Network` 类添加一个 `前馈` 方法，对于网络给定一个输入 a ，返回对应的输出²。这个方法所做的是对每一层应用等式 (22)：

2 假定输入 `a` 是 Numpy 的 n 维数组 ($n, 1$)，而不是向量 ($n,$)。这里， n 是输入到网络的数目。如果你试着用一个向量 ($n,$) 作为输入，将会得到奇怪的结果。虽然使用向量 ($n,$) 看上去好像是更自然的选择，但是使用 n 维数组 ($n, 1$) 使它特别容易的修改代码来立刻前馈多层输入，并且有的时候这很方便。

```
def feedforward(self, a):
    """Return the output of the network if "a" is input."""
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a)+b)
    return a
```

当然，我们想要我们的 `Network` 对象做的主要事情是学习。为此我们给它们一个实现随机梯度下降的 `SGD` 方法。下面是这部分的代码。在一些地方有一点神秘，我会在下面将它分解。

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
       test_data=None):
    """Train the neural network using mini-batch stochastic
    gradient descent. The "training_data" is a list of tuples
    "(x, y)" representing the training inputs and the desired
    outputs. The other non-optional parameters are
    self-explanatory. If "test_data" is provided then the
    network will be evaluated against the test data after each
    epoch, and partial progress printed out. This is useful for
    tracking progress, but slows things down substantially."""
    if test_data: n_test = len(test_data)
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test)
        else:
            print "Epoch {0} complete".format(j)
```

`training_data` 是一个代表着训练输入和对应的期望输出的元组 (x, y) 的列表。变量 `epochs` 和 `mini_batch_size` 是你期望的训练的迭代次数和取样时所用的 `mini-batch` 块的大小。`eta` 是学习率¹¹。如果可选参数 `test_data` 被提供，那么程序将在每次训练迭代之后评价网络，并输出我们的局部进展。这对于跟踪进展是有用的，但是大幅度降低速度。

代码如下工作。在每次迭代，它首先随机的将训练数据打乱，然后将它分成适当大小的迷你块。这是一个简单的从训练数据的随机采样方法。然后对于每一个 `mini_batch` 我们应用一次梯度下降。这是通过代码 `self.update_mini_batch(mini_batch, eta)` 做的，仅仅使

用 `mini_batch` 上的训练数据，根据单轮梯度下降更新网络的权重和偏差。这是 `update_mini_batch` 方法的代码：

```
def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch.
    The "mini_batch" is a list of tuples "(x, y)", and "eta"
    is the learning rate."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

大部分工作通过这行所做

```
delta_nabla_b, delta_nabla_w = self.backprop(x, y)
```

这行调用了一个叫做叫反向传播 (*backpropagation*) 的算法，这是一种快速计算代价函数的梯度的方法。因此 `update_mini_batch` 的工作仅仅是对 `mini_batch` 中的每一个训练样例计算梯度，然后适当的更新 `self.weights` 和 `self.biases`。

我现在不会展示 `self.backprop` 的代码。我们将在下章中学习反向传播是怎样工作的，包括 `self.backprop` 的代码。现在，就假设它可以如此工作，返回与训练样例 `x` 相关代价的适当梯度。

让我们看一下完整的程序，包括我之前忽略的文档注释。除了 `self.backprop`，程序是不需加以说明的。我们已经讨论过，所有重要的部分都在 `self.SGD` 和 `self.update_mini_batch` 中完成。`self.backprop` 方法利用一些额外的函数来帮助计算梯度，也就是说 `sigmoid_prime` 是计算 σ 函数的导数，而我不会在这里描述 `self.cost_derivative`。你能够通过查看代码或文档注释来获得这些的要点（以及细节）。我们将在下章详细的看一下它们。注意，虽然程序显得很长，但是大部分代码是用来使代码更容易理解的文档注释。实际上，程序只包含 74 行非空行、非注释代码。所有的代码可以在 GitHub [这里](#) 找到。

```
"""
network.py
\~~~~~

A module to implement the stochastic gradient descent learning
algorithm for a feedforward neural network. Gradients are calculated
using backpropagation. Note that I have focused on making the code
simple, easily readable, and easily modifiable. It is not optimized,
and omits many desirable features.
```

```

"""
#### Libraries
# Standard library
import random

# Third-party libraries
import numpy as np

class Network(object):

    def __init__(self, sizes):
        """The list ``sizes`` contains the number of neurons in the
        respective layers of the network. For example, if the list
        was [2, 3, 1] then it would be a three-layer network, with the
        first layer containing 2 neurons, the second layer 3 neurons,
        and the third layer 1 neuron. The biases and weights for the
        network are initialized randomly, using a Gaussian
        distribution with mean 0, and variance 1. Note that the first
        layer is assumed to be an input layer, and by convention we
        won't set any biases for those neurons, since biases are only
        ever used in computing the outputs from later layers."""
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                       for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent. The ``training_data`` is a list of tuples
        ``((x, y))`` representing the training inputs and the desired
        outputs. The other non-optional parameters are
        self-explanatory. If ``test_data`` is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out. This is useful for
        tracking progress, but slows things down substantially."""
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)

```

```

if test_data:
    print "Epoch {0}: {1} / {2}".format(
        j, self.evaluate(test_data), n_test)
else:
    print "Epoch {0} complete".format(j)

def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch.
    The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
    is the learning rate."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                   for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                  for b, nb in zip(self.biases, nabla_b)]

def backprop(self, x, y):
    """Return a tuple ``(nabla_b, nabla_w)`` representing the
    gradient for the cost function C_x. ``nabla_b`` and
    ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
    to ``self.biases`` and ``self.weights``."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = self.cost_derivative(activations[-1], y) * \
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    # Note that the variable l in the loop below is used a little
    # differently to the notation in Chapter 2 of the book. Here,
    # l = 1 means the last layer of neurons, l = 2 is the
    # second-last layer, and so on. It's a renumbering of the
    # scheme in the book, used here to take advantage of the fact
    # that Python can use negative indices in lists.
    for l in xrange(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp

```

```

        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-1-1].transpose())
    return (nabla_b, nabla_w)

def evaluate(self, test_data):
    """Return the number of test inputs for which the neural
    network outputs the correct result. Note that the neural
    network's output is assumed to be the index of whichever
    neuron in the final layer has the highest activation."""
    test_results = [(np.argmax(self.feedforward(x)), y)
                    for (x, y) in test_data]
    return sum(int(x == y) for (x, y) in test_results)

def cost_derivative(self, output_activations, y):
    """Return the vector of partial derivatives \partial C_x /
    \partial a for the output activations."""
    return (output_activations-y)

#### Miscellaneous functions
def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))

```

程序识别手写数字的效果如何？好吧，让我们先加载MNIST数据。我将用下面所描述的一小段辅助程序 `mnist_loader.py` 来完成。我们在一个Python shell中执行下面的命令，

```

>>> import mnist_loader
>>> training_data, validation_data, test_data =
... mnist_loader.load_data_wrapper()

```

当然，这也可以被做成一个单独的Python程序，但在Python shell执行最方便。

在加载完MNIST数据之后，我们将设置一个有30个隐层神经元的 `Network`。我们在导入如上所列的名字为 `network` 的Python程序后做，

```

>>> import network
>>> net = network.Network([784, 30, 10])

```

最后，我们将使用随机梯度下降来从MNIST `training_data` 学习超过30次迭代，迷你块大小为10，学习率 $\eta = 3.0$ ，

```

>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)

```

注意，当你阅读至此的时候正在运行代码，执行将会花费一些时间，对于一个普通的机器（截至2015年），它可能将会花费几分钟来运行。我建议你让它运行，继续阅读并定期的检查一下代码的输出。如果你赶时间，你可以通过减少迭代次数，减少隐层神经元次数或仅使用部分训练数据来提高速度。注意，这样产生的代码将会特别快：这些Python脚本的目的是帮助你理解神经网络是如何工作的，而不是高性能的代码！而且，当然，一旦我们已经训练一个网络，它能在几乎任何的计算平台上快速的运行。例如，一旦我们对于一个网络学会了一组好的权重集和偏置集，它能很容易的移植到web浏览器中以Javascript运行，或者如在移动设备上的本地应用。在任何情况下，这是一个神经网络训练运行时的部分输出文字记录。记录显示了在每轮训练之后神经网络能正确识别测试图像的数目。正如你所见到，在仅仅一次迭代后，达到了10,000中选中9,129个。而且数目还在持续增长，

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

经过训练的网络给我们一个一个约95%分类正确率为，在峰值时为95.42% ("Epoch 28")！作为第一次尝试，这是非常鼓舞人心的。然而我应该提醒你，如果你运行代码然后得到的结果不一定和我的完全一样，因为我们使用了（不同的）随机权重和偏置来初始化我们的网络。我采用了三次运行中的最优结果作为本章的结果。

让我们重新运行上面的实验，将隐层神经元数目改到100。正如前面的情况，如果你一边阅读一边运行代码，你应该被警告它将会花费相当长一段时间来执行（在我的机器上，这个实验每一轮训练迭代需要几十秒），因此当代码运行时，并行的继续阅读是很明智的。

```
>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

果然，它将结果提升至96.59%。至少在这种情况下，使用更多的隐层神经元帮助我们得到了更好的结果³。

³ 读者的反馈表明本实验在结果上有相当多的变化，而且一些训练运行给出的结果相当糟糕。使用第三章所介绍的技术将对于我们的网络在不同的训练执行上大大减少性能变化。

当然，为了获得这些正确率，我不得不对于训练的迭代次数，mini-batch大小和学习率 η 做了特殊的选择。正如我上面所提到的，这些在我们的神经网络中被称为超参数，以区别于通过我们的学习算法所学到的参数（权重和偏置）。如果我们较差的选择了超参数，我们会得到较差的结果。假设，例如我们选定学习率为 $\eta = 0.001$ ，

```
>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 0.001, test_data=test_data)
```

结果不太令人激励，

```
Epoch 0: 1139 / 10000
Epoch 1: 1136 / 10000
Epoch 2: 1135 / 10000
...
Epoch 27: 2101 / 10000
Epoch 28: 2123 / 10000
Epoch 29: 2142 / 10000
```

然而，你能够看到网络的性能随着时间的推移在缓慢的变好。这意味着着我们应该增大学习率，例如 $\eta = 0.01$ 。如果我们那样做了，我们会得到更好的结果，意味着我们应该再次增加学习率。（如果改变能够提高，试着做更多！）如果我们这样做几次，我们最终会得到一个像 $\eta = 1.0$ 的学习率（或者调整到 3.0 ），这跟我们之前的实验很接近。因此即使我们最初较差的选择了超参数，我们至少获得了足够的信息来帮助我们提升对于超参数的选择。

一般来说，调试一个神经网络是具有挑战性的。甚至有可能某种超参数的选择所产生的分类结果还不如随机分类。假定我们从之前成功的构建了 30 个隐层神经元的网络结构，但是学习率被改为 $\eta = 100.0$ 。

```
>>> net = network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 100.0, test_data=test_data)
```

在这点上，我们实际走的太远，学习率太高了：

```
Epoch 0: 1009 / 10000
Epoch 1: 1009 / 10000
Epoch 2: 1009 / 10000
Epoch 3: 1009 / 10000
...
Epoch 27: 982 / 10000
Epoch 28: 982 / 10000
Epoch 29: 982 / 10000
```

现在想象一下，我们第一次遇到这样的问题。当然，我们从我们之前的实验中知道正确的做法是减小学习率。但是如果我们在第一次遇到这样的问题，然而没有太多的输出来指导我们怎么做。我们可能不仅关心学习率，还要关心我们的神经网络中的其它每一个部分。我们可能想知道是否选择了让网络很难学习的初始化的权重和偏置？或者可能我们没有足够的训练数

据来获得有意义的学习？或者我们没有进行足够的迭代次数？或者可能对于这种神经网络的结构，学习识别手写数字是不可能的？可能学习率太低？或者可能学习率太高？当你第一次遇到某问题，你通常抱有不了把握。

从这得到的教训是调试一个神经网络是一个琐碎的工作，就像日常的编程一样，它是一门艺术。你需要学习调试的艺术来获得神经网络更好的结果。更普通的是，我们需要启发式方法来选择好的超参数和好的结构。所有关于这些的内容，我们都会在本书中进行讨论，包括我之前是怎样选择超参数的。

练习

- 尝试创建只有两层的神经网络，一个784个神经元的输入层和一个10个神经元的输出层，没有隐含层。用随机梯度下降法来训练这个网络。你能取得多高的分类精度？

早些时候，我跳过了MNIST数据时如何被加载的细节。它相当的简单。为了完整性，这是代码。被用于存储MNIST数据的数据结构在文档注释中被说明。这是简单明了的事情，由Numpy的 `ndarray` 对象构成的元组和列表（如果你不熟悉 `ndarrays`，将它们认为向量）：

```
"""
mnist_loader
\~~~~~

A library to load the MNIST image data. For details of the data
structures that are returned, see the doc strings for ``load_data``
and ``load_data_wrapper``. In practice, ``load_data_wrapper`` is the
function usually called by our neural network code.
"""

#### Libraries
# Standard library
import cPickle
import gzip

# Third-party libraries
import numpy as np

def load_data():
    """Return the MNIST data as a tuple containing the training data,
    the validation data, and the test data.

    The ``training_data`` is returned as a tuple with two entries.
    The first entry contains the actual training images. This is a
    numpy ndarray with 50,000 entries. Each entry is, in turn, a
    numpy ndarray with 784 values, representing the 28 * 28 = 784
    pixels in a single MNIST image.

    The second entry in the ``training_data`` tuple is a numpy ndarray
    containing 50,000 entries. Those entries are just the digit
    values (0...9) for the corresponding images contained in the first
    """

```

```

entry of the tuple.

The ``validation_data`` and ``test_data`` are similar, except
each contains only 10,000 images.

This is a nice data format, but for use in neural networks it's
helpful to modify the format of the ``training_data`` a little.
That's done in the wrapper function ``load_data_wrapper()``, see
below.
"""

f = gzip.open('../data/mnist.pkl.gz', 'rb')
training_data, validation_data, test_data = cPickle.load(f)
f.close()
return (training_data, validation_data, test_data)

def load_data_wrapper():
    """Return a tuple containing ``(training_data, validation_data,
    test_data)``. Based on ``load_data``, but the format is more
    convenient for use in our implementation of neural networks.

    In particular, ``training_data`` is a list containing 50,000
    2-tuples ``(x, y)``. ``x`` is a 784-dimensional numpy.ndarray
    containing the input image. ``y`` is a 10-dimensional
    numpy.ndarray representing the unit vector corresponding to the
    correct digit for ``x``.

    ``validation_data`` and ``test_data`` are lists containing 10,000
    2-tuples ``(x, y)``. In each case, ``x`` is a 784-dimensional
    numpy.ndarray containing the input image, and ``y`` is the
    corresponding classification, i.e., the digit values (integers)
    corresponding to ``x``.

    Obviously, this means we're using slightly different formats for
    the training data and the validation / test data. These formats
    turn out to be the most convenient for use in our neural network
    code."""
    tr_d, va_d, te_d = load_data()
    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
    training_results = [vectorized_result(y) for y in tr_d[1]]
    training_data = zip(training_inputs, training_results)
    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
    validation_data = zip(validation_inputs, va_d[1])
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = zip(test_inputs, te_d[1])
    return (training_data, validation_data, test_data)

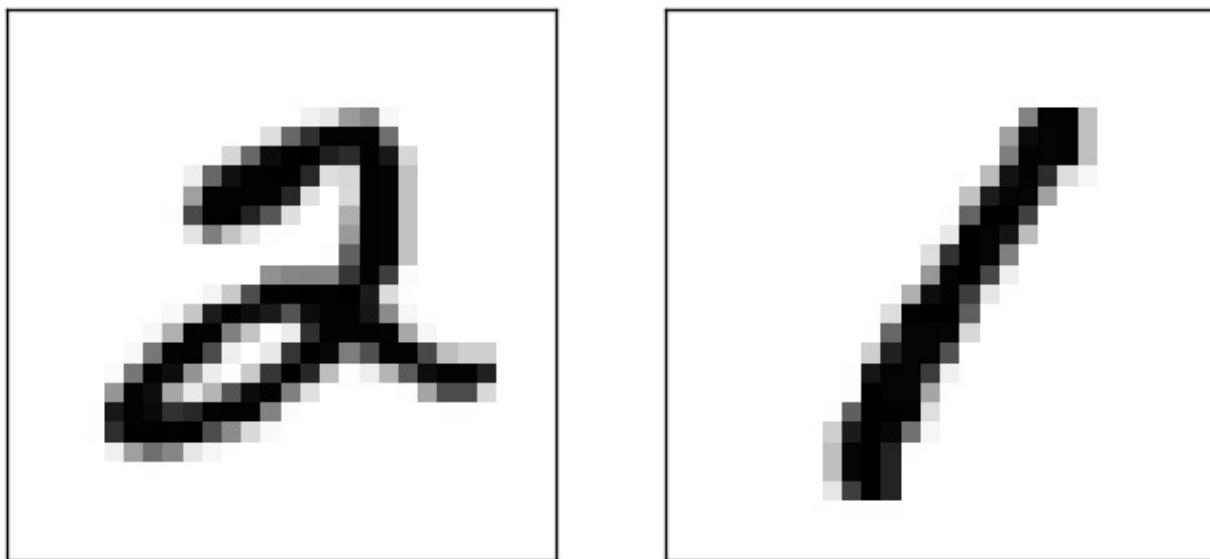
def vectorized_result(j):
    """Return a 10-dimensional unit vector with a 1.0 in the jth
    position and zeroes elsewhere. This is used to convert a digit
    (0...9) into a corresponding desired output from the neural
    network."""
    e = np.zeros((10, 1))
    e[j] = 1.0

```

```
return e
```

我之前说过我们的程序得到了很好的结果。这意味着什么？和什么比较很好？和一些简单的（非神经网络）**baseline**相比是非常有意义的，可以来理解什么样意味着表现好。当然，所有基准中最简单的是去随机的猜测数字，准确率大约是10%，我们做的比这好太多。

一个小的微不足道的**baseline**怎么样？让我们尝试一个极其简单的想法：我们来看看图片是如何的黑暗。例如一个2的图片显然比一个1的图片更黑，只是因为像下面示例中更多的像素点被涂黑：



这表明使用训练数据来对每个数字 $0, 1, 2, \dots, 9$ 计算平均暗度。当面对一个新的图像，我们计算这个图像的暗度是多少，然后再猜测它最近哪个数字的平均暗度。这是一个很简单的程序，很容易编写，因此我不明确的写出代码。如果你感兴趣，它在[GitHub仓库](#)。但是这是相比于随机猜测的一个大的提升，在10,000个测试图像中识别正确2,225个，也就是22.25%的准确率。

找到能够准确率达到20%到50%范围的想法并不困难。如果你努力一点，你能达到50%以上。但是使用已有的机器学习算法能帮助你达到更高的准确率。让我们尝试使用最出名的机器学习算法之一，支持向量机(*support vector machine*, SVM)。如果你不熟悉SVM，不用担心，我们不需要了解SVM具体是怎么工作的。我们而是使用一个叫做[scikit-learn](#)的Python库，它提供一个被称为[LIBSVM](#)的基于C的快速SVM库的简单的Python接口。

如果我们用默认的设置来运行scikit-learn的SVM分类器，那么它会在10,000测试图像中正确识别9,435。（[这里的代码](#)是可用的。）这是相比于我们的朴素的基于图片暗度的分类方法有着巨大的提升。实际上，这意味着SVM与我们的神经网络表现接近，只差了一点。在后面的章节中，我们将介绍新的技术来提升我们的神经网络使得它比SVM表现的好更多。

然而这并不是故事的结尾。在scikit-learn中对于SVM的默认设置的结果是10,000中的9,435。SVM有大量的可调参数，而且可以搜索到能够取得更高准确率的参数。我不会做这个探究，但是如果你想了解更多的话，请你留意[Andreas Mueller](#)的这篇[博客](#)。Mueller对SVM的一些参数进行优化，取得98.5%的准确率。换句话说，一个精心调参后的SVM仅仅对一个数字错误识别了70次。这个结果相当不错了！神经网络可以做得更好吗？

实际上，神经网络可以做的更好。目前，解决MNIST数字识别问题上，一个精心设计的神经网络能够比其它任何技术（包括SVM）取得更好的结果。当前（2013年）的最高纪录是10,000个中正确识别了9,979个。这个纪录是由[Li Wan](#), [Matthew Zeiler](#), [Sixin Zhang](#), [Yann LeCun](#)和[Rob Fergus](#)创造的。我们在本书的后面部分中会看到大多数他们所采用的技术。这个性能表现已经与人类的水平接近，甚至更好，因为相当多的MNIST图像对于人类来说是很困难有自信识别的，例如：



我相信你会同意这些是很难进行分类的！值得注意的是，在拥有像这样图像的MNIST数据集中，神经网络能够对于10,000个测试图像除了21个外都能正确分类。通常，我们认为像解决识别MNIST数字的复杂问题的程序需要一个复杂的算法。但尽管在Wan等人的论文中提及的神经网络和我们本章中所见到的算法有一些变化，但是也相当简单。所有复杂的事情都是可以从训练数据中自动学习的。在某种意义上，我们的结果和那些复杂论文中的结果表明了，在一些问题上：

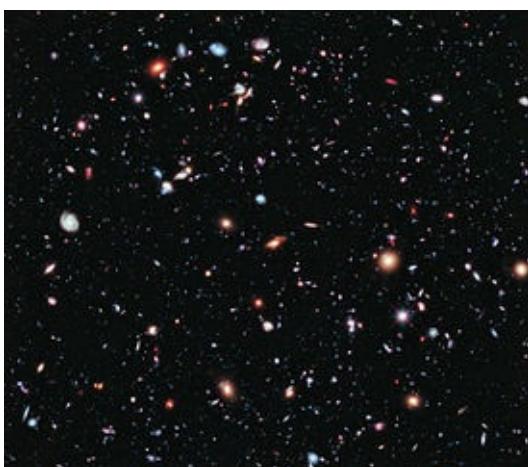
复杂算法 \leq 简单学习算法 + 好的训练数据

迈向深度学习

虽然我们的神经网络给出了令人印象深刻的表现，但是这个过程有一些神秘。网络中的权重和偏置是自动被发现的。这意味着我们不能对神经网络的运作过程有一个直观的解释。我们可以找到什么方法来理解我们的网络分类手写数字的原则吗？并且，在知道了这些原则后，我们能够帮助神经网络做得更好么？

为了更加清晰地解释这个问题，我们假设数十年后神经网络引发了人工智能（AI）。到那个时候，我们能明白这种智能网络的工作机制吗？可能的情况是，我们还是不能够理解它的原理，不能够理解其中的权重和偏置代表的意义，因为它们是自动学习得到的。在人工智能的早期研究阶段，人们希望在构建人工智能的努力过程中，也同时能够帮助我们理解智能背后的机制，以及人类大脑的运转方式。但最终的结果可能是我们既不能够理解大脑的机制，也不能够理解人工智能的机制。

为了解决这些问题，让我们回想一下我在这章开始对人工神经元作出的解释——它是一种对输入的evidence进行权衡的工具。假设我们想要确定一张图片里是否有人脸：



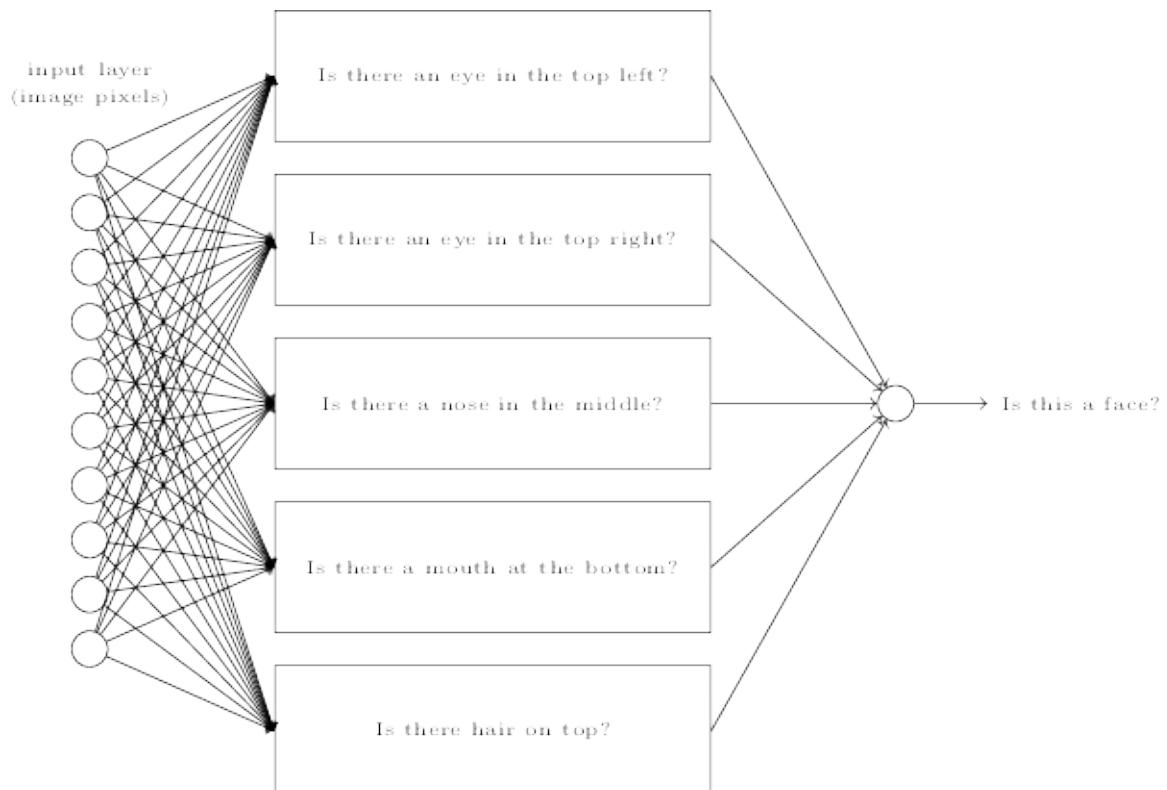
¹ 感谢：1.Ester Inbar 2.未知 3.NASA，ESA，G.Illingworth，D. Magee，P. Oesch（加利福尼亚大学，圣克鲁兹），R. Bouwens（莱顿大学）和HUDF09团队。点击图片获取更多信息。

我们可以用解决手写识别问题的方法来解决这个问题——网络的输入是图像中的像素点，网络的输出是一个单个的神经元用于表明「是的，这是一张脸」或「不，这不是一张脸」。

假设我们就采取了这个方法，但接下来我们先不去使用学习算法，而是去尝试人工设计一个网络，并为它选择合适的权重和偏置。我们要怎样做呢？首先暂时忘掉神经网络。一个想法是，我们可以将这个问题分解成一些子问题：图像的左上角有一个眼睛吗？右上角有一个眼睛吗？中间有一个鼻子吗？下面中央有一个嘴吗？上面有头发吗？诸如此类。

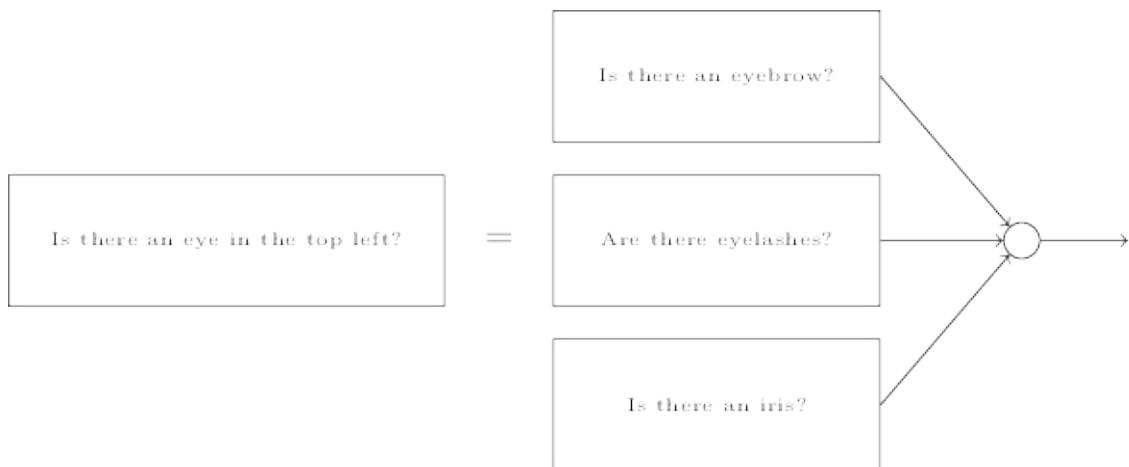
如果这些问题的回答是「是的」，或者甚至仅仅是「可能是」，那么我们可以作出结论这个图像可能是一张脸。相反地，如果大多数这些问题的答案是「不是」，那么这张图像可能不是一张脸。

当然，这仅仅是一个粗略的想法，而且它存在许多缺陷。也许有个人是秃头，没有头发。也许我们仅仅能看到部分脸，或者这张脸不是正面的，因此一些面部特征是模糊的。不过这个想法表明了如果我们能够使用神经网络来解决这些子问题，那么我们也许可以通过将这些解决子问题的网络结合起来，构成一个面部检测的神经网络。下图是一个可能的结构，子网络用矩阵来表示。注意，这不是一个人脸识别问题的现实的解决方法；而是用它来帮助我们构建起网络工作方式的直觉。下图是这个网络的结构：



我们还可以对子网络继续进行分解。假设我们考虑这个问题：「在左上角有一个眼睛吗？」这个问题可以被分解成这些子问题：「有一个眉毛吗？」，「有睫毛吗？」，「有虹膜吗？」等等。当然这些问题也应该包含关于位置的信息——「在左上角有眉毛，上面有虹膜

吗？」——但是为了方便起见，我们先不考虑位置的因素。所以，回答问题「左上角有一个眼睛吗？」的网络能够被分解成：



我们还可以对这些子问题的子问题继续进行分解，并通过多层网络传递得越来越远。最终，我们的子网络可以回答那些只包含若干个像素点的简单问题。举例来说，这些问题可能是询问图像中的几个像素是否构成非常简单的形状。这些问题就可以被那些与图像中原始像素点直接相连的单个神经元所回答。

最终的结果是，我们设计出了一个网络，它将一个非常复杂的问题——这张图像是否有一张人脸——分解成在单像素层面上就可回答的非常简单的问题。在前面的网络层，它回答关于输入图像非常简单明确的问题，在后面的网络层，它建立了一个更加复杂和抽象的层级结构。包含这种多层结构（两层或更多隐含层）的网络叫做深度神经网络（*deep neural network*）。

当然，我没有去解释应该如何将它递归地分解成子网络。在网络中通过人工来设置权重和偏置是不切实际的。取而代之的是，我们使用学习算法来让网络能够自动的从训练数据中学习权重和偏置。1980到1990年代的研究人员尝试了使用随机梯度下降和反向传播来训练深度网络。不幸的是，除了一些特殊的结构，他们并没有取得很好的效果。虽然网络能够学习，但是学习速度非常缓慢，不适合在实际中使用。

自2006年以来，人们发现了一系列技术能够帮助神经网络学习。这些深度学习技术基于了随机梯度下降和反向传播，并引进了新的想法。这些技术能够训练更深（更大）的网络——现在训练一个有5到10层隐层的网络都是很常见的。而且事实证明，在许多问题上，它们比那些仅有单个隐层网络的浅层神经网络表现的更加出色。当然，原因是深度网络能够对概念构建复杂的层次结构。这有点像传统编程语言使用模块化的设计和抽象的想法来创建复杂的计算机程序。将深度网络与浅层网络进行对比，有点像将一个能够进行函数调用的程序语言与一个不能进行函数调用的精简语言进行对比。抽象的概念在传统编程中和在神经网络中以不同形式存在，但对于它们来说都是非常重要的。

反向传播算法是如何工作的

在上一章中我们学习了神经网络是如何利用梯度下降算法来学习权重（weights）和偏置（biases）的。然而，在我们的解释中跳过了一个细节：我们没有讨论如何计算损失函数的梯度。这真是一个巨大的跳跃！在本章中我会介绍一个快速计算梯度的算法，就是广为人知的反向传播算法（backpropagation）。

反向传播算法最早于上世纪70年代被提出，但是直到1986年，由David Rumelhart, Geoffrey Hinton, 和Ronald Williams联合发表了一篇著名论文之后，人们才完全认识到这个算法的重要性。这篇论文介绍了几种神经网络，在这些网络的学习中，反向传播算法比之前提出的方法都要快。这使得以前用神经网络不可解的一些问题，现在可以通过神经网络来解决。今天，反向传播算法是神经网络学习过程中的关键（workhorse）所在。

比起其他章节，本章涉及到更多的数学内容。如果你对数学不那么感兴趣，你可以选择跳过本章，把反向传播当做一个黑盒，忽略其中的细节。为什么要花时间来学习那些细节呢？

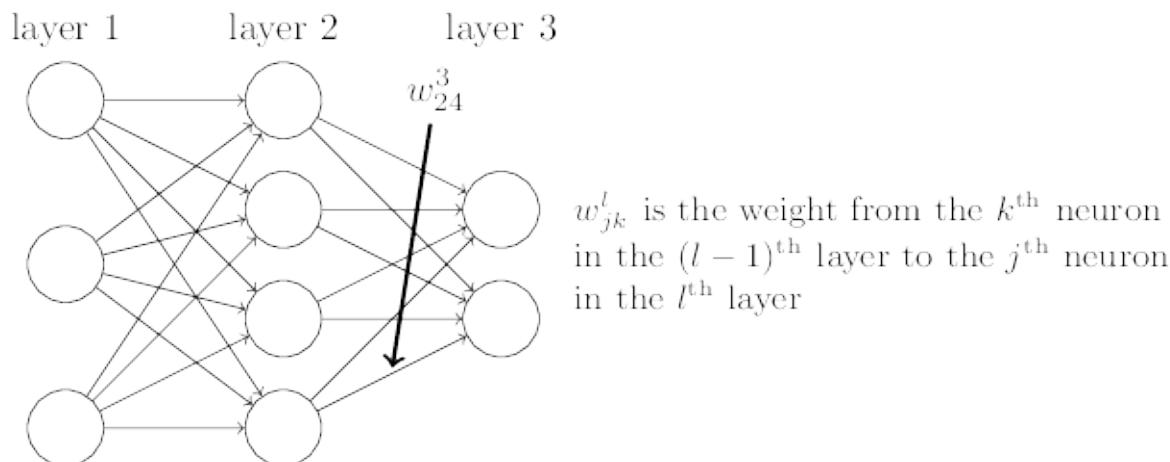
因为，如果你想要理解神经网络，你必须了解其中的细节。反向传播算法的核心是一个偏微分表达式 $\partial C / \partial w$ ，表示损失函数 C 对网络中的权重 w （或者偏置 b ）求偏导。这个式子告诉我们，当我们改变权重和偏置的时候，损失函数的值变化地有多快。尽管这个式子有点复杂，这个式子也是很漂亮的，它的每一个部分都有自然的，直觉上的解释。因此，反向传播不仅仅是一种快速的学习算法，它能够让我们详细深入地了解改变权重和偏置的值是如何改变整个网络的行为的。这是非常值得深入学习的。

尽管如此，你依然可以略读本章节，或者直接跳到下一章节。就算你把反向传播当做一个黑盒，本书的其它内容也是能够容易理解的。当然，在本书后续部分中我会提到本章的一些结论。但是，尽管在那些地方你不能跟上所有的推导，你依然能够理解主要的结论。

热身：一个基于矩阵的快速计算神经网络输出的方法

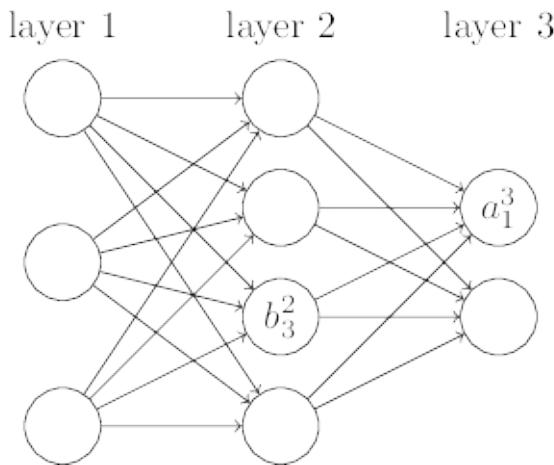
在讨论反向传播算法之前，我们先介绍一个基于矩阵的快速计算神经网络输出的方法来热热身。实际上在上一章的末尾我们经见过这个方法了，但是那时我说得很快，因此有必要详细回顾一下。另外在熟悉的上下文背景中，也能让你对反向传播会用到的符号记法感到习惯一些。

我们先介绍一种符号来表示网络中的权重参数，这种表示法不会引发歧义。我们用 w_{jk}^l 来表示从第 $l - 1$ 层的第 k 个神经元到第 l 层的第 j 个神经元的连接的权重。例如，下图展示了从第二层的第四个神经元到第三层的第二个神经元的连接的权重：



一开始你会觉得这种表示很麻烦，而且的确需要一些功夫来掌握它。但是经过一些努力，你就会发现这种表示简单而自然。这种表示的一个奇怪之处就是 j 和 k 下标的顺序。你可能会认为用 j 表示输入神经元，用 k 表示输出神经元更加合理。然而并不是这样的。在下面我会解释其中的原因。

我们用一种相似的记法来表示网络的偏置和激活值。确切地，我们用 b_j^l 表示第 l 层第 j 个神经元的偏置，用 a_j^l 表示第 l 层的第 j 个神经元的激活值。下图展示了这种记法的一个例子：



利用这些记法，第 l 层第 j 个神经元的激活值 a_j^l 通过下面的式子与第 $l - 1$ 层神经元的激活值联系起来（比较上一章节中的等式（4）和它附近的讨论内容）：

$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right), \quad (23)$$

这里的求和针对的是第 $l - 1$ 层的所有神经元 k 。为了将这个式子写成矩阵形式，我们为每一层 l 定义一个权重矩阵 w^l 。矩阵 w^l 的每一项就是连接到第 l 层神经元的权重，这就是说， w^l 中第 j 行第 k 列的元素就是 w_{jk}^l 。相似地，我们为每一层定义一个偏置向量 b^l 。你大概可以猜到这是如何工作的——偏置向量的每一项就是 b_j^l 的值，每一项对应第 l 层中的一个神经元。最后，我们定义一个激活向量 a^l ，其中的每一项就是激活值 a_j^l 。

最后一个我们需要改写成矩阵形式的想法就是把 σ 这样的函数向量化。我们在上一章中简单提到了向量化，这里面的思想就是我们想把一个函数比如 σ ，应用到一个向量 v 中的每一项。我们用一个直观的记号 $\sigma(v)$ 来表示这种作用在每一个元素上的函数操作。就是说， $\sigma(v)$ 的每一项就是 $\sigma(v)_j = \sigma(v_j)$ 。比如，如果我们有一个函数 $f(x) = x^2$ ， f 的向量化形式具有如下作用：

$$f \left(\begin{bmatrix} 2 \\ 3 \end{bmatrix} \right) = \begin{bmatrix} f(2) \\ f(3) \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \end{bmatrix}, \quad (24)$$

也就是说，向量化的函数 f 是对向量中的每一个元素求平方。

如果你记住了这些记法，那么等式（23）可以改写成下面简洁优美的矩阵形式：

$$a^l = \sigma(w^l a^{l-1} + b^l). \quad (25)$$

这个式子给我们对一层的激活值是如何与上一层中激活值产生关联的进行全局思考：我们仅仅将权重矩阵作用于上一层的激活值，然后加上偏置向量，最后用 σ 函数作用于这个结果，（就得到了本层的激活函数值）。相比于之前我们采用的神经元到神经元的视角，这种全局的视角通常更加简明和容易理解（涉及到更少的下标！）。你可以把它当做一种脱离下标魔

咒，同时还能对正在进行的事情保持精确的阐释。这种表达式在实际操作中也非常有用，因为大多数矩阵库提供了更快的方式来实现矩阵乘法，向量加法以及函数向量化操作。实际上，上一章节中的代码隐式地使用了这种表达式来计算网络的行为。

当使用等式 (25) 来计算 a^l 的时候，我们顺便也计算出了中间结果 $z^l \equiv w^l a^{l-1} + b^l$ 的值。这个数值非常有用，以至于值得我们专门给它一个名字：我们称 z^l 为对第 l 层神经元的加权输入 (weighted input)。在本章的后续部分中我们将大量地使用到加权输入 z^l 。等式

(25) 有时候会被写成与加权输入有关的形式，即 $a^l = \sigma(z^l)$ 。值得注意的是 z^l 由 $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$ 组成，这就是说， z_j^l 就是第 l 层第 j 个神经元激活函数的加权输入。

关于损失函数的两个假设

反向传播算法的目标是计算代价函数 C 对神经网络中出现的所有权重 w 和偏置 b 的偏导数 $\frac{\partial C}{\partial w}$ 和 $\frac{\partial C}{\partial b}$ 。为了使反向传播工作，我们需要对代价函数的结构做两个主要假设。在进行假设之前，在脑海中有一个代价函数的实例是很有用的。我们将会使用上一章讲到的平方代价函数作为例子。上一章的平方代价函数具有以下形式：

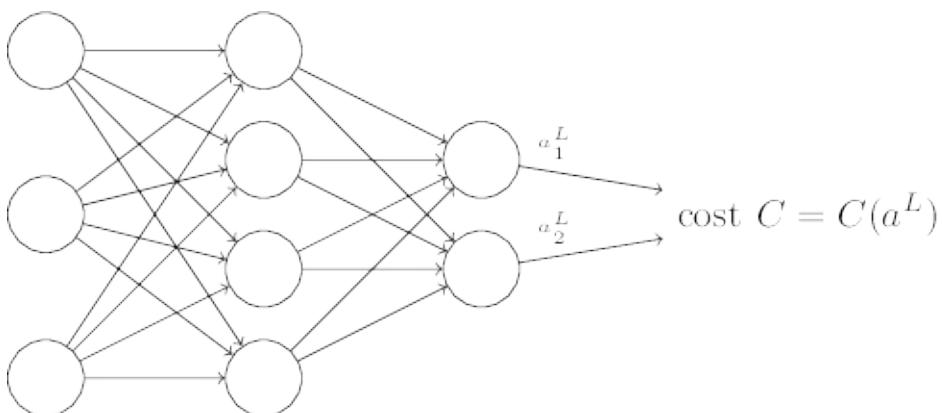
$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2, \quad (26)$$

其中 n 是训练样本总数；求和符号表示对每个独立训练样本 x 求和； $y = y(x)$ 是对应的希望输出； L 是神经网络层数； $a^L = a^L(x)$ 是输入为 x 时激活函数的输出向量。

那么，为了能够使用反向传播算法，我们需要对代价函数 C 进行怎样的假设呢？第一条假设是代价函数能够被写成 $C = \frac{1}{n} \sum_x C_x$ 的形式，其中 C_x 是每个独立训练样本 x 的代价函数。在代价函数为平方代价函数的情况下，一个训练样本的代价是 $C_x = \frac{1}{2} \|y - a^L\|^2$ 。该假设对于本书中涉及到的其它所有代价函数都成立。

我们需要上述假设的原因是，反向传播实际上是对单个训练数据计算偏导数 $\frac{\partial C_x}{\partial w}$ 和 $\frac{\partial C_x}{\partial b}$ 。然后通过对所有训练样本求平均值获得 $\frac{\partial C}{\partial w}$ 和 $\frac{\partial C}{\partial b}$ 。事实上，有了这个假设，我们可以认为训练样本 x 是固定的，然后把代价 C_x 去掉下标表示为 C 。最终我们会重新把 x 加回公式，但目前为了简便我们将它隐去。

我们对代价函数做的第二条假设是它可以写成关于神经网络输出结果的函数：



平方代价函数满足该要求，因为单一训练样本 x 的二次代价可以表示为：

$$C = \frac{1}{2} \|y - a^L\|^2 = \frac{1}{2} \sum_j (y_j - a_j^L)^2, \quad (27)$$

这是一个关于输出激活值的函数。显然，该代价函数也依赖于期望的输出 y ，所以你可能疑惑为什么我们不把代价视为关于 y 的函数。记住，输入的训练样本 x 是固定的，因此期望的输出 y 也是固定的。需要注意，我们不能通过改变权值或偏置来修改它，换句话说，它不是神经网络所学习的东西。所以把 C 视为只关于输出 a^L 的函数是有道理的。在该函数中 y 只是帮助定义函数的参数。

Hadamard积， $s \odot t$

反向传播算法是以常见线性代数操作为基础——诸如向量加法，向量与矩阵乘法等运算。但其中一个操作相对不是那么常用。具体来讲，假设 s 和 t 是两个有相同维数的向量。那么我们用 $s \odot t$ 来表示两个向量的对应元素(*elementwise*)相乘。因此 $s \odot t$ 的元素 $(s \odot t)_j = s_j t_j$ 。例如，

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}. \quad (28)$$

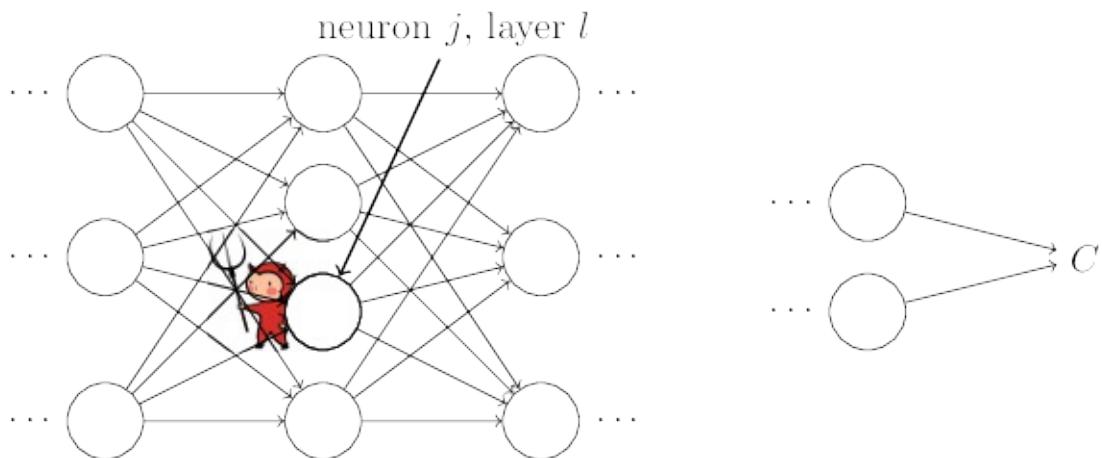
这种对应元素相乘有时被称为 **Hadamard积** (*Hadamard product*) 或 **Schur积** (*Schur product*)。我们将称它为 Hadamard积。优秀的矩阵库通常会提供 Hadamard积的快速实现，这在实现反向传播时将会有用。

反向传播背后的四个基本等式

反向传播(backpropagation)能够帮助解释网络的权重和偏置的改变是如何改变代价函数的。

归根结底，它的意思是指计算偏导数 $\partial C / \partial w_{jk}^l$ 和 $\partial C / \partial b_j^l$ 。但是为了计算这些偏导数，我们首先介绍一个中间量， δ_j^l ，我们管它叫做 l^{th} 层的 j^{th} 神经元的错误量(error)。反向传播会提供给我们一个用于计算错误量的流程，能够把 δ_j^l 和 $\partial C / \partial w_{jk}^l$ 、 $\partial C / \partial b_j^l$ 关联起来。

为了理解错误量是如何定义的，想象一下在我们的神经网络中有一个恶魔：



这个恶魔位于 l^{th} 层的 j^{th} 神经元。当神经元的输入进入时，这个恶魔扰乱神经元的操作。它给神经元的加权输入添加了一点改变 Δz_j^l ，这就导致了神经元的输出变成了 $\sigma(z_j^l + \Delta z_j^l)$ ，而不是之前的 $\sigma(z_j^l)$ 。这个改变在后续的网络层中传播，最终使全部代价改变了 $\frac{\partial C}{\partial z_j^l} \Delta z_j^l$ 。

而今，这个恶魔变成了一个善良的恶魔，它试图帮助你改善代价，比如，它试图找到一个

Δz_j^l 能够让代价变小。假设 $\frac{\partial C}{\partial z_j^l}$ 是一个很大的值（或者为正或者为负）。然后这个善良的恶

魔可以通过选择一个和 $\frac{\partial C}{\partial z_j^l}$ 符号相反的 Δz_j^l 使得代价降低。相比之下，如果 $\frac{\partial C}{\partial z_j^l}$ 接近于 0，那么这个恶魔几乎不能通过扰乱加权输入 z_j^l 改善多少代价。在一定范围内这个善良的恶魔就可

以分辨出，这个神经元已经接近于最佳状态¹。至此，有了一种启发式的感觉： $\frac{\partial C}{\partial z_j^l}$ 可以用来衡量神经元里的错误量。

¹ 当然，这只是针对很小的 Δz_j^l 来说的。我们会做出假设来限制恶魔只能做出很小的变化。

受到这个故事的促动，我们定义 l 层第 j 个神经元的错误量 δ_j^l 为：

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}. \quad (29)$$

按照通常的习惯，我们使用 δ^l 来表示与 l 层相关联的错误量的向量。反向传播将会带给我们一个计算每一层 δ^l 的方法，然后把这些错误量联系到我们真正感兴趣的量： $\partial C / \partial w_{jk}^l$ 和 $\partial C / \partial b_j^l$ 。

你可能想知道为什么恶魔一直在更改加权输入 z_j^l 。的确，更加自然的想法是，这个恶魔更改的是输出激活量 a_j^l ，然后我们就可以使用 $\frac{\partial C}{\partial a_j^l}$ 来衡量错误。事实上如果你这么做，就会得到和之后的讨论十分相似的结果，但结果会使反向传播的代数形式变得复杂。所以我们会继续使用 $\delta_j^l = \frac{\partial C}{\partial z_j^l}$ 用于衡量错误²。

² 在例如MNIST的分类问题中，术语"错误量 (error)"通常用于表示分类的错误率 (failure rate)。例如，如果神经网络的数字分类准确率是96.0%，那么错误率就是4.0%。显然，它和我们的 δ 向量在概念上有一些不同。实际上，在上下文环境中你可以分辨出使用的是哪个意思。

进攻计划：反向传播基于四个基本等式。这些等式带给我们一个计算错误量 δ^l 以及代价函数的梯度的方法。下面我会说明这四个等式。尽管如此，我在这里要告诫各位一下：不要期望立刻吸收理解这四个等式。你的这种期望会带失望的。事实上，反向传播的这几个等式内容很多，理解它们需要一定的时间和耐心，随着你会逐渐深入的探索才会真正理解。所以本部分的讨论也只是一个开端，旨在帮助你在未来的道路上彻底理解这些等式。

我们将在本章的后续内容更加深入地探索这些等式，现在先预览一下这些方法：我将给出这些等式的简短证明，帮助解释为什么它们是正确的；我们将要以伪代码的算法形式重新阐述一下这些等式，然后再看一下这些伪代码是如何通过Python代码实现的；在本章的最后一部分，我们将会开发一个直观的图片用来解释反向传播的这些等式是什么意思、一个人如何从零开始发现这些等式。在这些过程中，我们会重复提到四个基本等式，这样你也会加深对这些等式的理解，它们会变得舒服，甚至有可能变得漂亮而且自然。

输出层中关于错误量 δ^L 的等式， δ^L 的构成为

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L). \quad (\text{BP1})$$

这是一种非常自然的表达。右侧的第一项 $\partial C / \partial a_j^L$ ，就是用于测量 j^{th} 输出激活代价改变有多快的函数。举个例子，如果 C 并不太依赖于某个特别的输出神经元 j ，那么 δ_j^L 就会很小，这是我们所期望的。右侧的第二项 $\sigma'(z_j^L)$ ，用于测量 z_j^L 处的激活函数 σ 改变有多快。

你应该注意到(BP1)中的每一项都是容易计算的。特别的，当计算神经网络的行为时就计算了 z_j^L ，而计算 $\sigma'(z_j^L)$ 也仅仅是一小部分额外的开销。当然了， $\partial C / \partial a_j^L$ 的确切形式依赖于代价函数的形式。然而，如果提供了代价函数，大家也应该知道计算 $\partial C / \partial a_j^L$ 也不会有什么困难。举个例子，如果我们使用平方代价函数，即 $C = \frac{1}{2} \sum_j (y_j - a_j)^2$ ，那么 $\partial C / \partial a_j^L = (a_j - y_j)$ ，这显然很容易计算。

等式(BP1)是 δ^L 的分量形式。它是一个完美的表达式，但并不是我们想要的基于矩阵的形式，那种矩阵形式可以很好的用于反向传播。然而，我们可以很容易把等式重写成基于矩阵的形式，就像：

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \quad (\text{BP1a})$$

其中， $\nabla_a C$ 是一个向量，它是由 $\partial C / \partial a_j^L$ 组成的。你可以把 $\nabla_a C$ 看做现对于输出激活的 C 的改变速率。很容易看出来等式(BP1)和(BP1a)是等价的，基于这个原因我们从现在开始将使用(BP1)交替地指代两个等式。举个例子，在使用平方代价函数的情况下我们有 $\nabla_a C = (a^L - y)$ ，所以完整的基于矩阵的(BP1)的形式变为：

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \quad (30)$$

就像你所看到的，表达式里的每一项都拥有一个漂亮的向量形式，并且很容易使用一个库来计算，比如 Numpy。

依据下一层错误量 δ^{l+1} 获取错误量 δ^l 的等式：

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \quad (\text{BP2})$$

其中， $(w^{l+1})^T$ 是 $(l+1)^{\text{th}}$ 层的权重矩阵 w^{l+1} 的转置。这个等式看着有些复杂，但是每一项都有很好的解释。假设我们知道 $l+1^{\text{th}}$ 层的错误量 δ^{l+1} 。当我们使用转置权值矩阵 $(w^{l+1})^T$ 的时候，我们可以凭借直觉认为将错误反向 (*backward*) 移动穿过网络，带给我们某种测量 l^{th} 层输出的错误量方法。然后我们使用 Hadamard 乘积 $\odot \sigma'(z^l)$ 。这就是将错误量反向移动穿过 l 层的激活函数，产生了 l 层的加权输入的错误量 δ^l 。

通过结合(BP2)和(BP1)我们可以计算网络中任意一层的错误量 δ^l 。我们开始使用(BP1)来计算 δ^L ，然后应用等式(BP2)来计算 δ^{L-1} ，然后再次应用等式(BP2)来计算 δ^{L-2} ，以此类推，反向通过网络中的所有路径。

网络的代价函数相对于偏置的改变速率的等式：

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \quad (\text{BP3})$$

也就是说，错误量 δ_j^l 完全等于改变速率 $\partial C / \partial b_j^l$ 。这是一个很好的消息，因为(BP1)和(BP2)已经告诉我们如何计算 δ_j^l 。我们把(BP3)重写成如下的简略形式：

$$\frac{\partial C}{\partial b} = \delta, \quad (31)$$

这可以理解成 δ 可以和偏置 b 在相同的神经元中被估计。

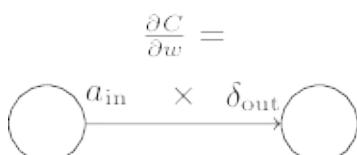
网络的代价函数相对于权重的改变速率的等式：

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \quad (\text{BP4})$$

这个等式告诉我们如何依据 δ^l 和 a^{l-1} 来计算偏导 $\partial C / \partial w_{jk}^l$ ，而这两个量我们已经知道如何计算了。这个等式可以重写成如下含有少量下标的形式：

$$\frac{\partial C}{\partial w} = a_{\text{in}} \delta_{\text{out}}, \quad (32)$$

可以这么理解， a_{in} 是神经元的激活量，输入到权重 w 中， δ_{out} 是神经元的错误量，从权重 w 输出。观察这个权重 w ，两个神经元通过这个权重连接起来，我们可以这样描画出来：



等式(32)的一个很好的结论是当激活量 a_{in} 很小的时候， $a_{\text{in}} \approx 0$ ，梯度项 $\partial C / \partial w$ 也将会趋近于很小。在这种情况下，我们说权重学习得很慢，也就是说在梯度下降的时候并没有改变很多。换而言之，等式(BP4)的一个结果就是从低激活量神经元里输出的权重会学习缓慢。

按照这样的思路，我们可以从(BP1)-(BP4)中获得另外一些见解。让我们从输出层开始看起。考虑(BP1)中的项 $\sigma'(z_j^L)$ 。回想一下上一章sigmoid函数的图像，当 $\sigma(z_j^L)$ 的值大约是0或1的时候 σ 函数的图像非常平缓。这时，我们将有 $\sigma'(z_j^L) \approx 0$ 。这告诉我们的时，如果输出神经元是低激活量(≈ 0)或高激活量(≈ 1)的时候，最后一层的权重将会学习缓慢。在这种情况下，我们通常说输出神经元已经饱和(saturated)，结果就是权重停止了学习(或者说是学习

缓慢)。输出层中的偏置也有类似的结论。

我们可以在早期的层中获得类似的见解。特别的，关注一下(BP2)中的 $\sigma'(z^l)$ 项。如果神经元接近饱和 δ_j^l 也可能变小。相应地，意味着任何一个输入到饱和神经元的权重都会学习缓慢³。

³ 这个结论不一定成立，因为当 $w^{l+1}^T \delta^{l+1}$ 的值非常大时，可以弥补比较小的 $\sigma'(z_j^l)$ 。我只是表达了一种笼统的趋势。

总结一下，我们获得的结论是：如果输入神经元是低激活量的，或者输出神经元已经饱和（高激活量或低激活量），那么权重就会学习得缓慢。

这些观察结果并不会令我们过于惊讶。并且它们仍然有助于我们理解在神经网络学习的过程中发生了什么。此外，我们可以把这样的推理应用在其它相关的地方。这四个基本的等式在使用任何激活函数的情况下都是成立的，不仅仅是标准的sigmoid函数(这是因为，就像我们待会看到的，证明的过程并没有使用*sigma*特别的属性)。所以我们可以利用这些等式来设计激活函数，使这些函数具有特殊目的的学习特性。给你举个例子吧，假设我们将要选择一个激活函数(非sigmoid) σ ，使得 σ' 总为正，并且始终不接近0。这就阻止了学习缓慢，而这是在原始sigmoid神经元饱和时会发生的。在本书的后面我们将看到一些例子，在这些例子中我们将会对激活函数做出这一类修改。把(BP1)-(BP4)记在心里，这将有助于解释为什么要尝试这样的修改，以及这会造成什么影响。

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad (\text{BP1})$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad (\text{BP2})$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (\text{BP3})$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad (\text{BP4})$$

问题

- 反向传播等式的另一种表示方法：我使用了Hadamard积给出了反向传播等式(BP1)和(BP2)。如果你不熟悉Hadamard积，这样的表示可能会让你感到不习惯。我们可以使用基于传统矩阵乘法的方法来替代这种记法，有些读者可能更喜欢这种记法。(1)将(BP1)重写为

$$\delta^L = \Sigma'(z^L) \nabla_a C, \quad (33)$$

其中 $\Sigma'(z^L)$ 是一个方阵，它的对角元素的值依次是 $\sigma'(z_j^L)$ ，非对角元素都是 0。注意到我们使用了传统的矩阵乘法将这个矩阵和 $\nabla_a C$ 相乘。(2) 将(BP2)重写为

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T \delta^{l+1}. \quad (34)$$

(3) 将(1)和(2)组合起来，得到

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T \dots \Sigma'(z^{L-1})(w^L)^T \Sigma'(z^L) \nabla_a C \quad (35)$$

对于习惯矩阵乘法的读者来说，这个等式可能比(BP1)和(BP2)更容易理解。我坚持使用(BP1)和(BP2)的原因是那样的记法在数值上能够更快地被实现。

四个基本方程的证明（自选）

现在我们将证明四个基本方程 (BP1) - (BP4)。这四个方程都是多元积分链式法则的结果。如果你对链式法则很熟悉，那我强烈建议你读之前自己推导一遍。

我们从方程 (BP1) 开始讲解，该方程给出了输出错误量的表达式 δ^L 。为证明这个方程，回想下定义

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}. \quad (36)$$

应用链式法则，我们可以把上述偏导数重新表达成带有输出激活的偏导数形式，

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L}, \quad (37)$$

这里的求和是对于输出层的所有神经元 k 而言的。当然，当 $k = j$ 时，输出激活 a_k^L 第 k^{th} 个神经元只依赖于对第 j^{th} 个神经元的输入权重 z_j^L 。并且当 $k \neq j$ 时， $\partial a_k^L / \partial z_j^L$ 项就没有了。因此我们可以简化之前的方程为

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}. \quad (38)$$

回想下， $a_j^L = \sigma(z_j^L)$ 右边第二项可以写为 $\sigma'(z_j^L)$ ，这样方程就变为

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L), \quad (39)$$

这就是以单元项形式表示的 (BP1)。

接下来我们再证 (BP2)，它给出了根据下一层的错误量 δ^{l+1} 计算 δ^l 的等式。为了证明该等式，我们先依据 $\delta_k^{l+1} = \partial C / \partial z_k^{l+1}$ 重新表达下等式 $\delta_j^l = \partial C / \partial z_j^l$ 。这里可以应用链式法则，

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \quad (40)$$

$$= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \quad (41)$$

$$= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}, \quad (42)$$

在最后一行，我们互换了下表达式右侧的两项，并取代了 δ_k^{l+1} 的定义。为了计算最后一行的第一项，注意

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}. \quad (43)$$

求导后，我们得到

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l). \quad (44)$$

代回 (42) 我们得到

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l). \quad (45)$$

这就是以单元项形式表示的 (BP2)。

我们要证明的最后两个方程 (BP3) 和 (BP4)。这些也遵循从链式法则，类似于前两个方程以上的证明的方式。我把他们给你作为练习。

练习

- 证明方程 (BP3) 和 (BP4)。

至此我们完成了反向传播的四个基本方程的证明。证明看似复杂，但其实就是仔细应用链式法则得到的结果。简而言之，我们可以把反向传播看作一步步地应用多元变量微积分的链式法则计算代价函数梯度的方法。这就是所有关于反向传播的内容，剩下的就是一些细节问题了。

反向传播算法

反向传播等式为我们提供了一个计算代价函数梯度的方法。下面让我们明确地写出该算法：

1. 输入 x : 计算输入层相应的激活函数值 a^1 。
2. 正向传播：对每个 $l = 2, 3, \dots, L$ ，计算 $z^l = w^l a^{l-1} + b^l$ 和 $a^l = \sigma(z^l)$ 。
3. 输出误差 δ^L ：计算向量 $\delta^L = \nabla_a C \odot \sigma'(z^L)$ 。
4. 将误差反向传播：对每个 $l = L-1, L-2, \dots, 2$ 计算

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad \text{和} \quad \frac{\partial C}{\partial b_j^l} = \delta_j^l$$
5. 输出：代价函数的梯度为

通过以上算法就能看出它为什么叫反向传播算法。我们从最后一层开始，反向计算错误向量 δ^l 。在神经网络中反向计算误差可能看起来比较奇怪。但如果回忆反向传播的证明过程，会发现反向传播的过程起因于代价函数是关于神经网络输出值的函数。为了了解代价函数是如何随着前面的权重和偏移改变的，我们必须不断重复应用链式法则，通过反向的计算得到有用的表达式。

练习

- 修改一个神经元后的反向传播

假设我们修改了正向传播网络中的一个神经元，使得该神经元的输出为

$f(\sum_j w_j x_j + b)$ ，其中 f 是一个非sigmoid 函数的函数。在这种情况下我们应该怎样修改反向传播算法？

- 线性神经元的反向传播

假设我们在整个神经网络中用 $\sigma(z) = z$ 代替常用的非线性方程 σ 。重新写出这种情况下反向传播算法。

正如我在上文中已经说过的，反向传播算法对每个训练样本 $C = C_x$ 计算代价函数的梯度。在实际情况中，经常将反向传播算法与诸如随机梯度下降的学习算法共同使用，在随机梯度下降算法中，我们需要计算一批训练样本的梯度。给定一小批(mini-batch) m 个训练样本，下面的算法给出了基于这些训练样本的梯度下降学习步骤：

1. 输入一组训练样本
2. 对每个训练样本 x ：设定相应的输入激活值 $a^{x,1}$ 并执行以下步骤：

- 正向传播：对于每个 $l = 2, 3, \dots, L$ ，计算 $z^{x,l} = w^l a^{x,l-1} + b^l$ 和 $a^{x,l} = \sigma(z^{x,l})$ 。
 - 输出误差 $\delta^{x,L}$ ：计算向量 $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$ 。
 - 将误差反向传播：对每个 $l = L-1, L-2, \dots, 2$ ，计算 $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$ 。
3. 梯度下降：对每个 $l = L, L-1, \dots, 2$ ，分别根据法则 $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$ 和 $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$ 更新权重和偏移。

当然，为了在实际应用时实现随机梯度下降，你还需要一个用于生成小批(mini-batches)训练样本的外部循环和一个用于逐步计算每一轮迭代的外部循环。为了简洁，这些都被省略了。

反向传播算法代码

在理论上理解了反向传播算法后，就可以理解上一章中用来实现反向传播算法的代码了。回忆一下第一章 Network 类中的 `update_mini_batch` 和 `backprop` 方法的代码。这些代码可以看做是上面算法描述的直接翻译。具体来说，`update_mini_batch` 方法通过计算梯度来为当前的小批次（mini_batch）更新 Network 的权重和偏置。

```
class Network(object):
    ...
    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The "mini_batch" is a list of tuples "(x, y)", and "eta"
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                      for b, nb in zip(self.biases, nabla_b)]
```

大部分工作是由 `delta_nabla_b, delta_nabla_w = self.backprop(x, y)` 这行代码完成的。它使用了 `backprop` 方法来计算偏导 $\frac{\partial C_x}{\partial b_j^l}$ 。`backprop` 方法基本上是按照上一节中描述的内容来实现的，但有一点不同：我们使用了一个稍微不同的方法来索引 layer。这个改动利用了 Python 中列表负索引特性的优势来从后向前索引一个列表。例如，`l[-3]` 代表列表 l 的倒数第三项。`backprop` 方法的代码如下所示，同时还有一些帮助方法用来计算 σ 函数、 σ' 的导数，以及代价函数的导数。你应该能够理解下面的代码了。但如果遇到了困难的话，可以参考第一章中对这段代码的描述。

```

class Network(object):
    ...
    def backprop(self, x, y):
        """Return a tuple "(nabla_b, nabla_w)" representing the
        gradient for the cost function C_x. "nabla_b" and
        "nabla_w" are layer-by-layer lists of numpy arrays, similar
        to "self.biases" and "self.weights"."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book. Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on. It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    ...
    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives \partial C_x /
        \partial a for the output activations."""
        return (output_activations-y)

    def sigmoid(z):
        """The sigmoid function."""
        return 1.0/(1.0+np.exp(-z))

    def sigmoid_prime(z):
        """Derivative of the sigmoid function."""
        return sigmoid(z)*(1-sigmoid(z))

```

问题

- 在一个批次（**mini-batch**）上应用完全基于矩阵的反向传播方法

在我们的随机梯度下降算法的实现中，我们需要依次遍历一个批次（mini-batch）中的训练样例。我们也可以修改反向传播算法，使得它可以同时为一个批次中的所有训练样例计算梯度。我们在输入时传入一个矩阵 $X = [x_1 \ x_2 \ \dots \ x_m]$ （而不是一个向量 x ），这个矩阵的列代表了这一个批次中的向量。前向传播时，每一个节点都将输入乘以权重矩阵、加上偏置矩阵并应用 `sigmoid` 函数来得到输出，反向传播时也用类似的方式计算。明确地写出这种反向传播方法，并修改 `network.py`，令其使用这种完全基于矩阵的方法进行计算。这种方式的优势在于它可以更好地利用现代线性函数库，并且比循环的方式运行得更快。（例如，在我的笔记本电脑上求解与上一章所讨论的问题相类似 的MNIST分类问题时，最高可以达到两倍的速度。）在实践中，所有正规的反向传播算法库都使用了这种完全基于矩阵的方法或其变种。

为什么说反向传播算法很高效？

为什么说反向传播算法很高效？要回答这个问题，让我们来考虑另一种计算梯度的方式。设想现在是神经网络研究的早期阶段，大概是在上世纪50年代或60年代左右，并且你是第一个想到使用梯度下降方法来进行训练的人！但是要实现这个想法，你需要一种计算代价函数梯度的方式。你回想了你目前关于演算的知识，决定试一下是否能用链式法则来计算梯度。但是琢磨了一会之后发现，代数计算看起来非常复杂，你也因此有些失落。所以你尝试着寻找另一种方法。你决定把代价单独当做权重的函数 $C = C(w)$ （我们一会再来讨论偏置）。将权重写作 w_1, w_2, \dots ，并且要对某个权重 w_j 计算 $\partial C / \partial w_j$ 。一个很明显的计算方式是使用近似：

$$\frac{\partial C}{\partial w_j} \approx \frac{C(w + \epsilon e_j) - C(w)}{\epsilon}, \quad (46)$$

其中 ϵ 是一个大于零的小正数。换句话说，我们可以通过计算两个差距很小的 w_j 的代价，然后利用等式(46)来估计 $\partial C / \partial w_j$ 。我们可以利用相同的思想来对偏置求偏导 $\partial C / \partial b$ 。

这种方式看起来很不错。它的概念很简单，实现起来也很简单，只需要几行代码就可以。当然了，他看起来要比使用链式法则来计算梯度靠谱多了！

然而遗憾的是，虽然这种方式看起来很美好，但当用代码实现之后就会发现，它实在是太慢了。要理解其中的原因的话，设想在我们的神经网络中有一百万个权重，对于每一个不同的权重 w_j ，为了计算 $\partial C / \partial w_j$ ，我们需要计算 $C(w + \epsilon e_j)$ 。这意味着为了计算梯度，我们需要计算一百万次代价函数，进而对于每一个训练样例，都需要在神经网络中前向传播一百万次。我们同样需要计算 $C(w)$ ，因此总计需要一百万零一次前向传播。

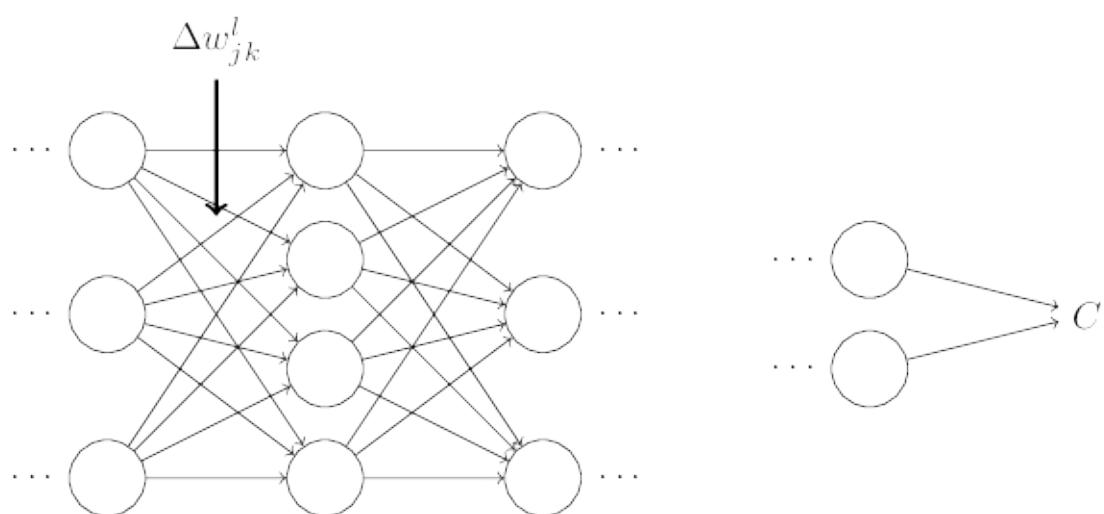
反向传播的优点在于它仅利用一次前向传播就可以同时计算出所有的偏导 $\partial C / \partial w_j$ ，随后也仅需要一次反向传播。大致来说，反向传播算法所需要的总计算量与两次前向传播的计算量基本相等（这应当是合理的，但若要下定论的话则需要更加细致的分析。合理的原因在于前向传播时主要的计算量在于权重矩阵的乘法计算，而反向传播时主要的计算量在于权重矩阵转置的乘法。很明显，它们的计算量差不多）。这与基于等式(46)的方法所需要的一百万零一次前向传播相比，虽然反向传播看起来更复杂一些，但它确实更快。

这种加速方式在1986年首次被人们所重视，极大地拓展了神经网络能够适用的范围，也导致了神经网络被大量的应用。当然了，反向传播算法也不是万能的。在80年代后期，人们终于触及到了性能瓶颈，在利用反向传播算法来训练深度神经网络（即具有很多隐含层的网络）时尤为明显。在本书后面的章节中我们将会看到现代计算机以及一些非常聪明的新想法是如何让反向传播能够用来训练深度神经网络的。

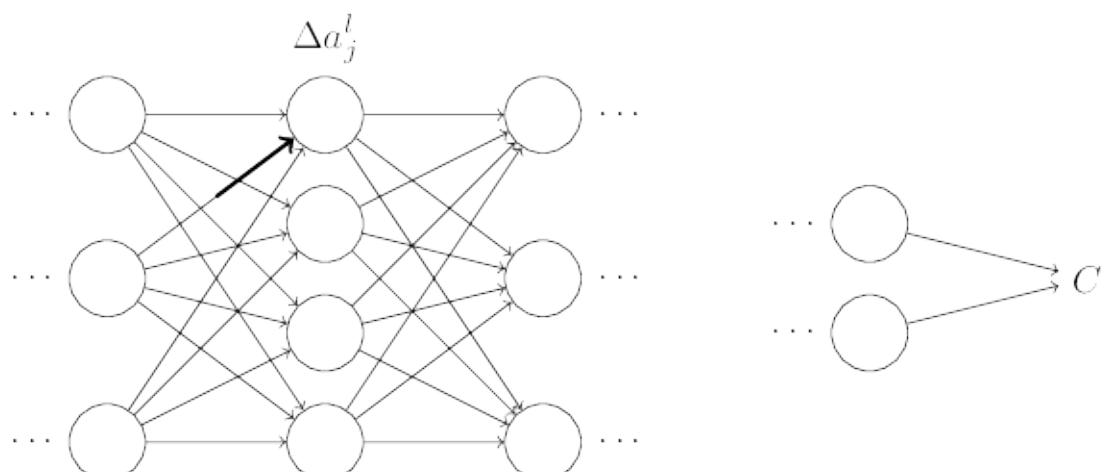
反向传播：整体描述

正如我之前所阐述的，反向传播涉及了两个谜题。第一个谜题是，这个算法究竟在做什么？我们之前的描述是将错误量从输出层反向传播。但是，我们是否能够更加深入，对这些矩阵、向量的乘法背后作出更加符合直觉的解释？第二个谜题是，人们一开始是如何发现反向传播算法的？按照算法流程一步步走下来，或者证明算法的正确性，这是一回事。但这并不代表你能够理解问题的本质从而能够从头发现这个算法。是否有一条合理的思维路线使你能够发现反向传播算法？在本节中，我会对这两个谜题作出解释。

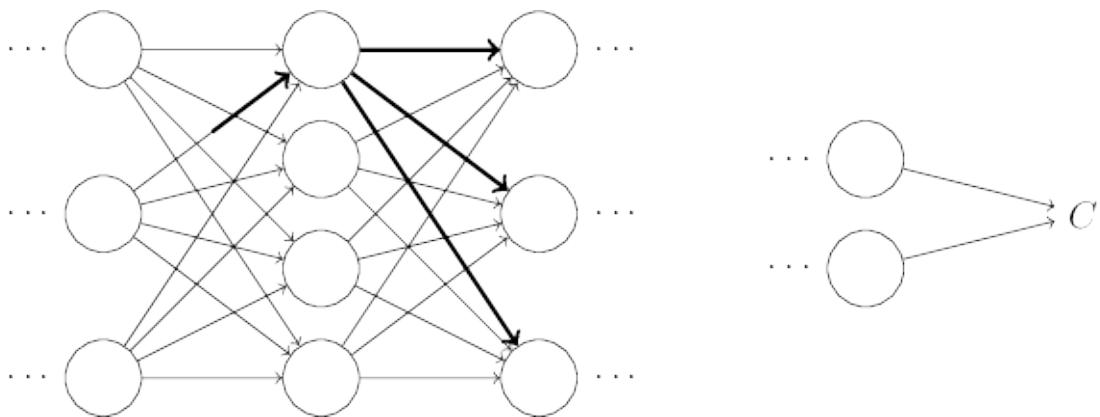
为了更好地构建的反向传播算法在做什么的直觉，让我们假设我们对网络中的某个权重 w_{jk}^l 做出了一个小的改变量 Δw_{jk}^l ：



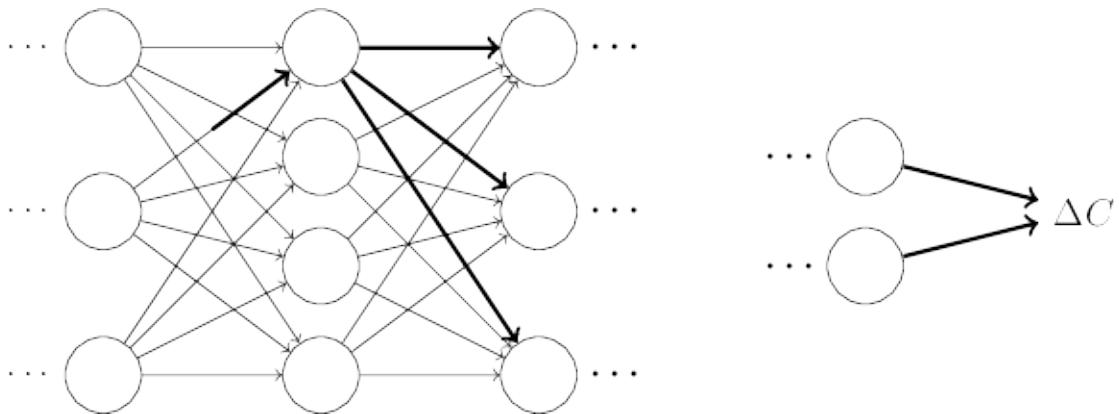
这个改变量会导致与其相关的神经元的输出激活值的改变：



以此类推，会引起下一层的所有激活值的改变：



这些改变会继续引起再下一层的改变、下下层...依次类推，直到最后一层，然后引起代价函数的改变：



代价函数的改变量 ΔC 与最初权重的改变量 Δw_{jk}^l 是有关的，关系是下面这个等式

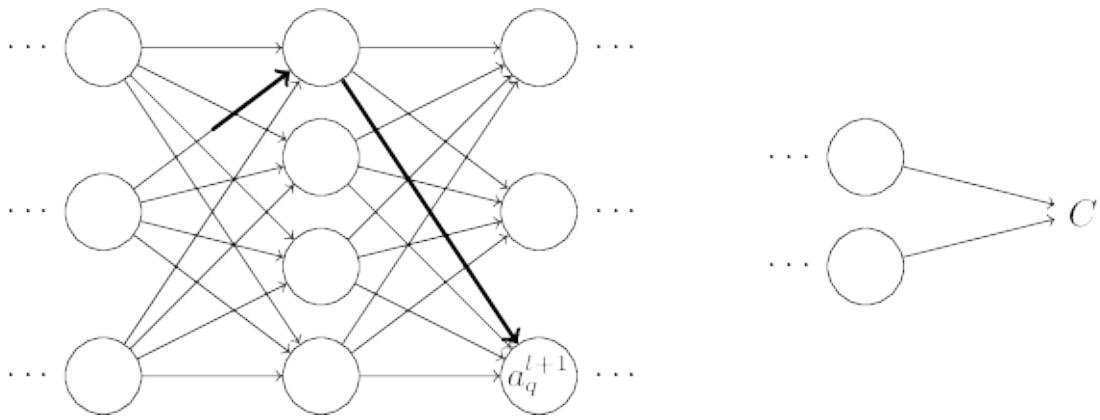
$$\Delta C \approx \frac{\partial C}{\partial w_{jk}^l} \Delta w_{jk}^l. \quad (47)$$

这表明了计算 $\frac{\partial C}{\partial w_{jk}^l}$ 的一种可能方法是，计算 w_{jk}^l 上的一个小改变量经过正向传播，对 C 引起了多大的改变量。如果我们能够通过小心翼翼的计算做到这点，那我们就可以计算出 $\partial C / \partial w_{jk}^l$ 。

让我们尝试来写一下计算过程。改变量 Δw_{jk}^l 对 l^{th} 层的第 j^{th} 个神经元的激活值带来了 Δa_j^l 的改变量。它的大小是

$$\Delta a_j^l \approx \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l. \quad (48)$$

激活值改变量 Δa_j^l 会引起下一层（第 $(l+1)^{\text{th}}$ 层）的所有激活值都改变。我们先关注其中的一个结点， a_q^{l+1} ，



它产生的改变量是：

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \Delta a_j^l. \quad (49)$$

将等式(48)带入其中，得到：

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l. \quad (50)$$

当然，改变量 Δa_q^{l+1} 会继续造成下一层的激活值的改变。实际上，我们可以想象一条从 w_{jk}^l 到 C 的路径，其中每一个结点的激活值的改变都会引起下一层的激活值的改变，最终引起输出层的代价的改变。如果这条路径是 $a_j^l, a_q^{l+1}, \dots, a_n^{L-1}, a_m^L$ ，那么最终的改变量是

$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \dots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l, \quad (51)$$

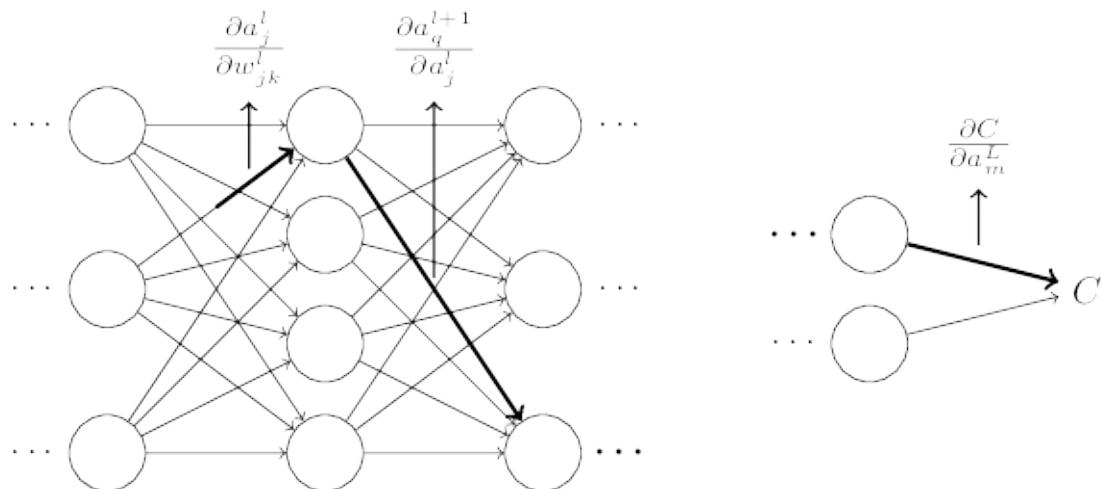
这样，我们使用了一系列 $\partial a / \partial a$ 的形式的项，对应了路径上的每一个结点，包括最终项 $\partial C / \partial a_m^L$ 。这就计算出了在神经网络的这条路径上，最初的改变量引起了 C 多大的改变。当然，由 w_{jk}^l 的改变量影响代价改变的路径选择是很多的，我们现在只考虑了其中的一条路径。为了计算 C 最终总共的改变量，很显然我们应该对所有可能的路径对其带来的改变量进行求和：

$$\Delta C \approx \sum_{mnp\dots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \dots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l, \quad (52)$$

其中，我们对每条路径中所有出现的神经元都进行求和。与等式(47)进行比较，我们得到：

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_{mnp\dots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \dots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l}. \quad (53)$$

等式(53)看上去很复杂。不过，它在直觉上很容易理解。我们计算出了 C 相对于网络中的一个权重的变化速率。这个等式告诉我们的的是，每一条连接两个神经元的边都可以对应一个变化速率，这个变化速率的大小是后一个神经元对前一个神经元的偏导。连接第一个权重和第一个神经元的边对应的变化速率是 $\frac{\partial a_j^l}{\partial w_{jk}^l}$ 。一条路径对应的变化速率恰好是路径上的变化速率的连乘。总变化速率 $\frac{\partial C}{\partial w_{jk}^l}$ 是从起始权重到最终代价上的所有可能的路径的变化速率的总和。用图片来说明这个过程，对于一条路径来说：



目前为止我所阐述的是一种启发式的观点，当你困惑于神经网络中的权重时，可以通过这种观点来思考。下面我会给你一些简要的思路使你可以更进一步的完善这个观点。首先，你可以显式地计算出等式53中的每一项的偏导表达式。这很容易做到，只需要一点计算量就行。做完之后，你可以尝试将所有的求和通过矩阵的形式来表示。这个过程会有一点无聊，并且需要一些毅力，但是不会特别的困难。随后，你可以尝试尽可能的将表达式简化，最终你会发现你得到了反向传播算法！所以，你可以将反向传播算法看作是一种对所有路径上的所有变化率进行求和的方法。或者用另外一种方式来说，反向传播算法是一种很聪明的方法，当小扰动沿着网络传播、到达输出并影响代价的过程中，它能够记录其对相应权重（和偏置）的影响量。

我的解释到此为止了。这可能有点复杂，并且需要仔细思考所有的细节。如果你乐于接受挑战，你可能会很享受这个过程。如果不是，我希望我的这些想法能够给你一些关于反向传播在做什么的启发。

那关于其它的谜题呢——反向传播最初是如何被发现的？实际上，如果你一路看下来我的阐述，你能够找到关于反向传播算法的一种证明。不幸的是，完整的证明实际上比我在本章的描述更长更复杂。那这个相对更简单（但更玄虚）的证明是如何发现的呢？当你把试图把完整的证明中的所有细节写出来时，你会发现有一些很显然能够被简化的形式。你做完这些简化，会得到一个简短一些的证明。然后你又会发现一些可以简化的內容。当你重复几次这个过程之后，你会得到本章中的这个简短的证明，但是它有点晦涩，因为其中所有复杂的结构都被简化掉了！我希望你能够相信我，完整的证明与本章中简短的证明没有什么本质区别。我只是对证明过程做了很多简化的工作。

改进神经网络的学习方式

当高尔夫运动员刚开始接触高尔夫时，他们通常会花费大量的时间来练习基本的挥杆。只有不断矫正自己的挥杆方式，才能学好其它的技能：切球，打左曲球，右曲球。同理，到目前为止我们把精力都放在了理解反向传播算法（backpropagation algorithm）上。这是我们的最基本的“挥杆方式”，它是大多数神经网络的基础。在本章中我将介绍一套技术能够用于改进朴素的反向传播算法，进而能改进神经网络的学习方式。

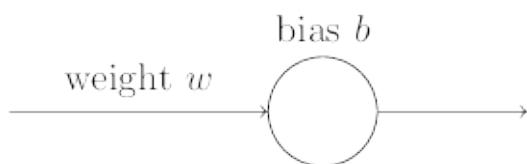
我们将要在本章介绍的技术包含：选取更好的代价函数，就是被称为交叉熵代价函数（**the cross-entropy cost function**）；四种正则化方法（L1和L2正则、dropout、训练数据的扩展），这能让我们的网络适应更广泛的数据；一种更好的初始化权重（weight）的方法；一系列更具启发式的方法用来帮助我们选择网络的超参数（hyper-parameters）。我也会简单介绍一下其它的技术。这些讨论的内容之间是独立的，因此你可以跳跃着阅读。我们也会用代码实现这些技术去改进我们第一章的手写数字识别的结果。

当然了，我们也只是涉及了神经网络众多技术的一部分。但是万变不离其宗，我们只需要掌握一些核心的技术就行。掌握这些核心的技术不仅仅是因为它们好用，更重要的是它们能够帮助你深入理解使用神经网络时可能遇到的问题，也能帮助你很快上手其它的技术。

交叉熵代价函数

大多数人都会对犯错感到不愉快。在我刚学钢琴不久时，我将要在观众面前进行我的第一场表演。我当时很紧张，把一个八度弹奏低了。我卡住了，直到别人指出我的错误后，我才得以继续弹奏。我当时非常尴尬。尽管犯错时很不愉快，但是我们能够从明显的错误中学到东西。你能猜到在我下次弹奏的时候会把这个八度弹对。相反，如果错误很不明显的话，我们的学习速度将会很慢。

理想情况下，我们希望神经网络能够快速地从错误中学习。这种想法现实么？为了回答这个问题，让我们看一个小例子。这个例子包含仅有一个输入的单个神经元：



我们将要训练这个神经元去做一些极其简单的事：输入1，输出0。当然，这是一个非常容易的任务，我们可以不利用任何学习算法，通过手算就能找到合适的权重（weight）和偏移（bias）。尽管如此，事实证明利用梯度下降法（gradient descent）能够帮助我们去学习权重和偏移。那么我们就来看一下这个神经元是如何学习的。

为了更明确，我将为权重选定初始值0.6，为偏移选定初始值0.9。这是学习算法开始时一般的初始选择，我没有用到什么特殊的方式来选取这些初始值。神经元的第一次输出为0.82，在到达我们的期望值0之前，神经元还需要很多轮学习迭代。点击右下角的"Run"，我们来看一下神经元是如何学习来让输出结果接近0.0的（译者注：观看交互式动画请前往原作网页）。注意这不是预先录制好的动画，你的浏览器能够真正地去计算梯度，然后用梯度值去更新权重和偏移，然后显示结果。学习率为 $\eta = 0.15$ ，事实证明这个学习率足够慢以至于我们能够很好地观察到发生了什么，同时它也足够快以至于我们能够获得在几秒内获得大量的学习。代价函数就是我们在第一节里面提到的均方误差函数（quadratic cost function），C。我将会在接下来给出代价函数的具体形式，这里没必要去深究它的定义。注意你能通过点击"Run"来多次运行这个动画。

[点击前往原作观看交互式动画](#)

正如你所见，神经元能够迅速地学习权重和偏移来降低代价函数，并最后给出大概0.09左右的输出。虽然这不是我们期待的输出，0.0，但这个结果已经足够好了。假设我们把权重和偏移的初始值都选为2.。在这种情况下，初始的输出是0.98，这是相当糟糕的结果。让我们看一下在这个例子中神经元是如何学习的。再次点击"Run"：

[点击前往原作观看交互式动画](#)

尽管在这个例子中用了相同的学习率 ($\eta = 0.15$)，但是我们能看到学习一开始时进行地很缓慢。事实上，在前 150 轮左右的迭代过程中，权重和偏移并没有改变太多。接下来学习过程和我们第一个例子很接近，神经元的输出迅速地接近 0.0。

当和人类的学习对比时，我们发现这种行为很奇怪。正如我在这一节开始所提到的那样，我们常常能够在错误很大的情况下能学习地更快。但是正如刚才所见，我们的人工神经元在错误很大的情况下学习遇到了很多问题。另外，事实证明这种行为不仅在这个简单的例子中出现，它也会在很多其他的神经网络结构中出现。为什么学习变慢了呢？我们能找到一种方法来避免这种情况么？

为了搞清问题的来源，我们来考虑一下神经元的学习方式：通过计算代价函数的偏导 $\partial C / \partial w$ 和 $\partial C / \partial b$ 来改变权重和偏移。那么我们说「学习速度很慢」其实上是在说偏导很小。那么问题就转换为理解为何偏导很小。为了解释这个问题，我们先来计算一下偏导。回忆一下，我们使用了均方代价函数，即等式 (6)：

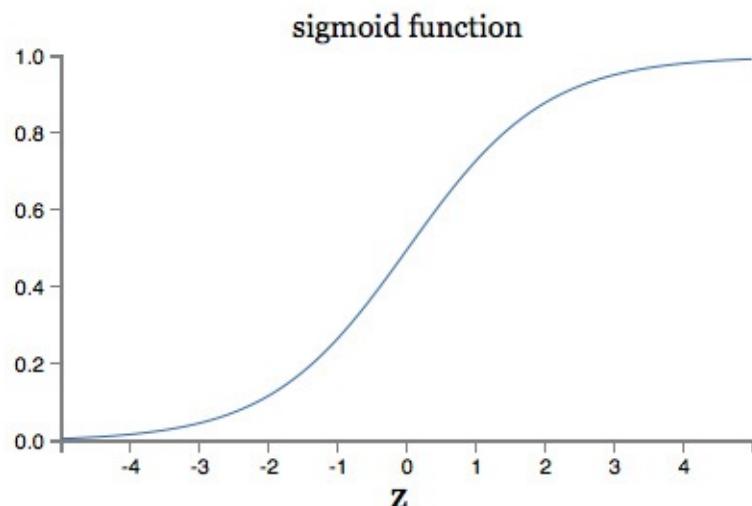
$$C = \frac{(y - a)^2}{2}, \quad (54)$$

这里 a 是输入 $x = 1$ 时神经元的输出， $y = 0$ 是我们期待的输出。下面我们用权重和偏移来重写这个式子。回忆一下 $a = \sigma(z)$ ，这里 $z = wx + b$ 。运用链式法则我们得到：

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z) \quad (55)$$

$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z) = a\sigma'(z), \quad (56)$$

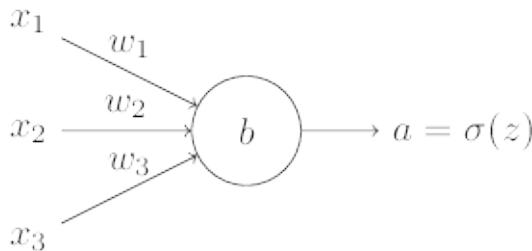
这里我已经替换了 $x = 1, y = 0$ 。为了理解这些表达式的行为，我们要对右面的 $\sigma'(z)$ 了解地更深入一点。回忆一下 σ 函数的形状。



我们能够从图像看出当神经元输出接近1时，曲线变得非常平坦，因此 $\sigma'(z)$ 就会变得非常小。等式(55)和等式(56)能告诉我们 $\partial C/\partial w$ 和 $\partial C/\partial b$ 会变得很小。这就是学习速度变慢的根源。另外，正如我们稍后所见到的那样，这种情况导致的速度下降不仅仅适应我们的示例神经元网络，它还适用于很多其他通用的神经元网络。

交叉熵代价函数简介

我们如何来避免这种减速呢？事实证明我们可以用不同的代价函数比如交叉熵(cross-entropy)代价函数来替代二次代价函数。为了理解交叉熵，我们暂时先不用管这个示例神经元模型。我们假设要训练一个拥有多个输入变量的神经元：输入 x_1, x_2, \dots ，权重 w_1, w_2, \dots ，偏移 b :



神经元的输出为 $a = \sigma(z)$ ，这里 $z = \sum_j w_j x_j + b$ 。我们定义这个神经元的交叉熵代价函数为：

$$C = -\frac{1}{n} \sum_z [y \ln a + (1 - y) \ln(1 - a)], \quad (57)$$

这里 n 是训练数据的个数，这个加和覆盖了所有的训练输入 x ， y 是期望输出。

仅从等式(57)我们看不出为何能解决速度下降的问题。事实上，老实讲，这个甚至看不出这个式子为何能称之为代价函数！在了解它能避免学习减速之前，我们还是有必要解读交叉熵为何能作为代价函数。

交叉熵有两个特性能够合理地解释为何它能作为代价函数。首先，它是非负的，也就是说， $C > 0$ 。为了说明这个，我们需要注意到：**(a)**等式(57)加和里的每一项都是负的，因为这些数是0到1之间的，它们的对数是负的；**(b)**整个式子的前面有一个负号。

其次，如果对于所有的训练输入 x ，这个神经元的实际输出值都能很接近我们期待的输出的话，那么交叉熵将会非常接近0。为了说明这个，假设有一些输入样例 x 得到的输出是 $y = 0$ ， $a \approx 0$ 。这些都是些比较好的输出。我们会发现等式(57)的第一项将会消掉，因为 $y = 0$ ，与此同时，第二项 $-\ln(1 - a) \approx 0$ 。同理，当 $y = 1$ 或 $a \approx 1$ 时也如此分析。那么如果我们的实际输出接近期望输出的话代价函数的分布就会很低。

总结一下，交叉熵是正的，并且当所有输入 x 的输出都能接近期望输出 y 的话，交叉熵的值将会接近0¹。这两个特征在直觉上我们都会觉得它适合做代价函数。事实上，我们的均方代价函数也同时满足这两个特征。这对于交叉熵来说是一个好消息。而且交叉熵有另一个均方代价函数不具备的特征，它能够避免学习速率降低的情况。为了理解这个，我们需要计算一下交叉熵关于权重的偏导。我们用 $a = \sigma(z)$ 代替等式(57)，并且运用链式法则，得到：

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n} \sum_z \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \frac{\partial \sigma}{\partial w_j} \quad (58)$$

$$= -\frac{1}{n} \sum_z \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \sigma'(z) x_j. \quad (59)$$

通分化简之后得到：

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_z \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))} (\sigma(z) - y). \quad (60)$$

利用sigmoid函数的定义， $\sigma(z) = 1/(1 + e^{-z})$ ，和一点代数知识我们就能得到 $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ 。在下面的练习题中我会让你证明这个结论，但是现在我们就默认接受它。可以看到 $\sigma'(z)$ 和 $\sigma(z)(1 - \sigma(z))$ 这一项在上式中消除了，它被简化成：

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y). \quad (61)$$

这是一个非常优美的表达式。它告诉我们权重的学习速率可以被 $\sigma(z) - y$ 控制，也就是被输出结果的误差所控制。误差越大我们的神经元学习速率越大。这正是我们直觉上所期待的那样。另外它能避免学习减速，这是 $\sigma'(z)$ 一项导致的。当我们使用交叉熵时， $\sigma'(z)$ 这一项会被抵消掉，因此我们不必担心它会变小。这种消除是交叉熵代价函数背后所带来的惊喜。实际上，这并不是一个惊喜。稍后我们会看到，我们特意选取了具有这种特性的函数。

同样，我们能够计算偏移的偏导。我在这里不详细介绍它了，你可以很容易证明：

$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_z (\sigma(z) - y). \quad (62)$$

同理，它也能够避免 $\sigma'(z)$ 这一项带来的学习减速。

¹ 为了证明这个结论，我需要假设 y 的输出只能为0或者1。这种情况特别在分类问题，或者在计算布尔函数时出现。如果你想知道如果我们不做这个假设时会发生什么，请查看本节最后的练习。

练习

- 证明 $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ 。

让我们回到之前的例子中来，一块研究一下如果我们使用交叉熵代价函数而不是均方误差会发生些什么。我们先从均方误差表现好的情况开始：权重为0.6，偏移为0.9。按下"Run"去观察一下我们使用交叉熵代价函数会发生什么：

[点击前往原作观看交互式动画](#)

不出所料，神经元在这种情况下和之前使用均方误差时一样好。那么现在我们就要看一下之前均方误差表现较差的情况（[点这里进行对比](#)）：权重和偏移都设置成2.0：

[点击前往原作观看交互式动画](#)

成功了！这次神经元学习速度很快。如果你细心观察你会发现代价函数曲线在初始的时候比使用均方误差时更陡峭。这意味着即使我们初始的条件很糟糕，交叉熵函数也能尽可能地降低学习速度减慢的可能性。

我并没有指明这些例子中用到的学习速率。在使用均方误差的时候，我选取 $\eta = 0.15$ 。那么我们应该在新的例子中用相同的学习速率么？事实上，代价函数发生改变之后我们不能很精确的定义什么是「相同」的学习速率。这就像对比苹果和橘子一样。对于这两种代价函数我都实验过一些不同的学习速率。如果你仍然好奇，那么事实是这样的：我在新的例子中选取 $\eta = 0.005$ 。

你可能会反对学习速率的改变，因为这会让上面的例子变得没有意义。如果我们随意选取学习速率那么谁还会在意神经元学习地有多快呢？这种反对偏离了重点。这个例子的重点不是在说学习速度的绝对值。它是在说明学习速度是如何变化的。当我们使用均方误差代价函数时，如果选取一个错的离谱的开始，那么学习速度会明显降低；而我们使用交叉熵时，这种情况下学习速度并没有降低。这根本不取决于我们的学习速率是如何设定的。

我们已经研究过交叉熵用于单个神经元的情况。事实上，这很容易推广到多层神经网络上。我们假设 $y = y_1, y_2, \dots$ 是我们期望的输出，例如，在神经元的最后一层， a_1^L, a_2^L, \dots 是真实的输出。那么我们可以定义交叉熵：

$$C = -\frac{1}{n} \sum_x \sum_j [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)]. \quad (63)$$

这其实和我们的等式 (57) 相同，只不过这里面的 \sum_j 是求所有神经元的输出。我不会再一次精确地求偏导了，但是容易看出用式表达式 (63) 也可以避免多层神经网络中学习速度下降的情况。如果你感兴趣，你可以在下面的问题中求一下偏导。

什么情况下我们要用交叉熵函数取代均方误差函数呢？事实上，如果输出神经元是sigmoid神经元的话，交叉熵都是更好的选择。为了理解这个，假定我们随机初始化权重和偏移。那么可能会发生这种情况，初始的选择会得到误差很大的输出，比如我们想得到0时，它却输出1

，或者相反的情况。如果我们用交叉熵代价函数，学习速度会明显降低。这种情况不会停止学习，因为权重还会通过其他训练数据学习，但这显然不是我们想要的。

练习

- 交叉熵带来的一个问题就是很难记住表达式中 y 和 a 的位置。我们很容易记不清正确的表达式是 $-[y \ln a + (1 - y) \ln(1 - a)]$ 还是 $-[a \ln y + (1 - a) \ln(1 - y)]$ 。当 $y = 0$ 或 1 时，如果使用了第二个表达式会发生什么呢？这个问题会发生在第一个表达式上吗？请说明你的理由。
- 在本节开始讨论单个神经元时，我曾声称如果所有的训练数据都有 $\sigma(z) \approx y$ ，那么交叉熵会变得非常小。这个假设依赖于 y 非 0 即 1 。这对于分类问题是正确的，但是对于其他问题（比如回归问题） y 的取值可能在 0 和 1 之间。证明当所有的训练数据 $\sigma(z) = y$ 时，交叉熵仍然是最小化的。当交叉熵有下面形式时：

$$C = -\frac{1}{n} \sum_x [y \ln y + (1 - y) \ln(1 - y)]. \quad (64)$$

$-[y \ln y + (1 - y) \ln(1 - y)]$ 的值有时被称作二进制熵（binary entropy）。

问题

- 多层神经网络

在上一章介绍这个概念的时候，我们利用均方误差代价函数得到输出层对权重求偏导有：

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n} \sum_x a_k^{L-1} (a_j^L - y_j) \sigma'(z_j^L). \quad (65)$$

其中 $\sigma'(z_j^L)$ 会导致当输出明显出错的时候学习速度下降。对于交叉熵，我们的输出误差 δ^L 对于每一个单个训练数据 x 有

$$\delta^L = a^L - y. \quad (66)$$

用这个表达式可以证明输出层对权重的偏导为

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n} \sum_x a_k^{L-1} (a_j^L - y_j) \sigma'(z_j^L). \quad (67)$$

这样 $\sigma'(z_j^L)$ 这一项就消掉了，因此交叉熵代价函数能够避免速度下降，这不仅仅对一个单个神经元成立，对于多层神经元也是成立的。简单变形一下也能得到偏移也具有相同的形式。

- 当我们的输出层是线性神经元（**linear neurons**）的时候使用均方误差

假设我们有一个多层神经网络。假设最后一层的所有神经元都是线性神经元（*linear neurons*）意味着我们不用sigmoid作为激活函数，输出仅仅是 $a_j^L = z_j^L$ 。如果我们用均方误差函数时，输出误差 δ^L 对于每个训练输入 x 为

$$\delta^L = a^L - y. \quad (68)$$

和我们之前的问题类似，利用这个表达式我们在输出层对权重和偏移求导有

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n} \sum_z a_k^{L-1} (a_j^L - y_j) \quad (69)$$

$$\frac{\partial C}{\partial b_j^L} = \frac{1}{n} \sum_z (a_j^L - y_j). \quad (70)$$

这就意味着如果输出神经元是线性神经元的话就不会产生速度下降的问题。

用交叉熵解决手写数字识别问题

我们可以很容易在程序中将交叉熵应用于梯度下降法（gradient descent）和反向传播算法（backpropagation）。在本章的后面我会改进之前的手写数字识别程序 `network.py`。新的程序取名 `network2.py`，它不仅仅用到了交叉熵，还用到了本章将要介绍的其他技术¹。但现在我们先看一下新程序解决手写数字问题的效果如何。和第一章的例子一样，我们的网络将用30个隐层神经元，mini-batch的大小选为10。设定学习速率为 $\eta = 0.52$ ，迭代次数选为30。`network2.py` 的接口和 `network.py` 稍有不同，但是仍然能够很清楚看到整个过程。

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
...     mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data,
...     monitor_evaluation_accuracy=True)
```

注意，这里顺便提一下 `net.large_weight_initializer()` 这条命令是用来初始化权重和偏移的。我们需要运行一下这条命令，因为在这章的后面我会改变我们网络初始化权重的默认方式。运行完这些命令显示的正确率为95.49个百分点。这非常接近我们第一章用均方误差函数的结果95.42个百分点。

接下来我们选取隐藏层神经元个数为100，代价函数为交叉熵函数，其它的参数保持不变。在这种情况下，我们得到的准确率为96.82个百分点。这比我们第一章的结果96.59有了提升。你可能觉得只是提升了一点，但是考虑到误差率从3.41个百分点降低到3.18个百分点。这也就是说，我们缩减了十四分之一的原始误差。这的确是个不错的提升。

这对交叉熵来说是个好消息，因为和使用均方误差相比，它能得到相似或者更好的结果。原因是花了一些时间去寻找一些比较好的超参数，比如学习速率，mini-batch的大小等等。这些改进也鼓舞我们要做一个彻底的工作去优化超参数。这些实验结果也验证了我们选择使用交叉熵的理论优势。

顺便说一句，这也是我们在这章或者在这本书中采用的模式。我们采用了一些新的技术，然后做实验验证它，然后我们「改进了」结果。我们当然乐意看到结果得到了改进。但是带来改进的原因通常是令人困扰的。只有我们花费大量时间去优化所有的超参数才能得到令人信服的结果。这样做的工作量非常大，通常我们不会做那么详尽的研究。相反，我们会做一些类似于上面的非正式的测试。不过你仍要记得这种测试的证据并不充分，要对参数失效的迹象要保持警觉。

到目前为止，我们花费了大量篇幅去讨论交叉熵。它只对我们手写数字识别有一点提升，为何要那么大费周章地去介绍它呢？在这章的后面我们会看到其他技术——尤其是正则化，能够大大改进我们的结果。那么为什么要如此关注交叉熵呢？一部分原因是交叉熵是广泛使用的代价函数，因此它值得我们深入了解。但是更重要的理由是神经元饱和在神经元网络中是一个重要的问题，它会贯穿整本书。我花费那么长篇幅讨论交叉熵是因为它是一个很好的实验，能让我们对神经元饱和有了初步的认知。

1 代码托管在[Github](#)

2 在第一章中，我们使用了平方代价函数和 $\eta = 3.0$ 的学习率。正如之前所讨论的，当代价函数改变后，我们不能精确地定义什么是「相同的」学习率。对于对比的两种代价函数，在其它超参数相同的情况下，我尝试了进行实验找到能够产生相似表现的学习率。有一种非常粗浅的想法找到交叉熵代价函数和平方代价函数两者的学习率的联系。正如我们之前所看到的，平方代价函数的梯度表达式中多一项 $\sigma' = \sigma(1 - \sigma)$ 。如果我们对 σ 算一下均值，我们得到 $\int_0^1 d\sigma \sigma(1 - \sigma) = 1/6$ 。我们可以（大致上）得到，当学习率相同时，平方代价函数的速度会平均慢6倍。这表明一个可行的方法是将平方代价函数的学习率除以6。当然，这远远不是一个严谨的推断，别太当真了。但是你也可以将其视为一种有用的初始化方法。

交叉熵的意义是什么？它又是怎么来的？

我们之前对交叉熵的讨论集中在代数分析和实际实现。这些工作看起来是足够了，但也留下一些待回答的更宽泛的概念问题，比如：交叉熵的意义是什么？有没有直观方式去思考交叉熵？还有，我们怎么才能在一开始的时候就想到交叉熵？

我们从最后一个问题入手：什么会促使我们在第一时间想到交叉熵？假设我们发现了之前描述过的学习减缓问题，并且明白根源是公式 (55) 和公式 (56) 中的 $\sigma'(z)$ 项。在仔细观察了这两个公式之后，我们可能会猜想——是否可以通过选择一个代价函数使得 $\sigma'(z)$ 项消失。这样的话，一个单一训练样本 x 的代价 $C = C_x$ 就会满足：

$$\frac{\partial C}{\partial w_j} = x_j(a - y) \quad (71)$$

$$\frac{\partial C}{\partial b} = (a - y). \quad (72)$$

如果我们能选择某个代价函数使得这个等式成立，那么它们将会直接使得如下直觉成立：一开始的错误越大，神经元学习得越快。同时它们也消除了学习减缓的问题。实际上，如果我们从这些等式着手，凭借我们的数学嗅觉，就能够推导出交叉熵的公式。注意由链式法则，我们有：

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a} \sigma'(z). \quad (73)$$

运用 $\sigma'(z) = \sigma(z)(1 - \sigma(z)) = a(1 - a)$ 最后一个等式变成：

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a} a(1 - a). \quad (74)$$

和等式 (72) 对比，我们得到：

$$\frac{\partial C}{\partial a} = \frac{a - y}{a(1 - a)}. \quad (75)$$

将这个表达式对 a 求积分有：

$$C = -[y \ln a + (1 - y) \ln(1 - a)] + \text{constant}, \quad (76)$$

常数部分为某个值。这是单个训练样本 x 对代价的贡献。想得到完整的代价函数，我们必须在所有样本上平均一下，得到：

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)] + \text{constant}, \quad (77)$$

其中，常数部分是每个训练样本各自常数的平均值。因而我们可以看出，公式 (71) 和 (72) 唯一确定了交叉熵的形式，以及一个整体的常数项。交叉熵不是奇迹般凭空产生的，而是我们能够以一种简单自然的方式发现的。

那交叉熵的直观意义是什么？我们又该如何理解它呢？深入地解释这个问题会扯得很远，我就不细说了。但值得一提的是，在信息论领域是有一种标准方式来解释交叉熵的。大致说来，想法就是：交叉熵是对惊讶的测度。特别地，我们的神经元尝试去计算函数 $x \rightarrow y = y(x)$ 。但是，取而代之的是，它计算函数 $x \rightarrow a = a(x)$ 。假设我们把 a 当作 y 为 1 时神经元估计的概率， $1 - a$ 是 y 的正确值为 0 时估计的概率。然后，交叉熵衡量的是我们在了解 y 的真实值时的平均「惊讶」程度。当输出是我们期望的值，我们得到低程度的惊讶；当输出不是我们期望的，我们得到高程度的惊讶。当然，我还没准确说明「惊讶」是什么意思，所以这个措辞听起来很空洞。但事实上是有一种精确的信息理论方法来阐述惊讶所表达的意思的。不幸的是，我并不知晓网络上是否能够找到有关该主题的出色、简短、内自含的讨论。但是如果你想深究下去，维基百科上有一个能让你正确入门的 [简要概述](#)。细节部分可通过研读有关 Kraft 不等式的材料来补充，这些材料在 [Cover and Thomas](#) 所写的有关信息论的书籍的第五章中可以找到。

问题

- 我们已经详尽地讨论了当我们使用平方代价来训练的神经网络时，会产生输出神经元饱和、学习速率下降的问题。另一个会妨碍学习的因素是等式 (61) 中 x_j 项。因为该项的存在，当输入 x_j 接近于 0 时，对应的权重 w_j 会学习得很慢。解释一下，为什么我们不能通过选择一个好的代价函数来消除 x_j 项。

Softmax

在本章中我们主要使用交叉熵代价函数来解决学习速度衰退的问题。不过，我想首先简要的介绍一下解决这个问题的另一种方法，这种方法是基于神经元中所谓的 softmax 层。我们并不打算在本章余下的部分使用 softmax 层，所以，如果你很心急，你可以跳过本节直接进入下一节的阅读。然而，softmax 还是很值得理解一下，一方面因为 softmax 在本质上很有趣，另一方面因为我们将要在第六章使用 softmax 层，那时候我们要讨论深度神经网络。

softmax 的主要思想是为我们的神经网络定义一种新的输出层。跟之前的 sigmoid 层一样，它也是对加权输入 $\mathbf{z}_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L$ 进行计算。不一样的地方是，在获取输出结果的时候我们并不使用 sigmoid 函数。取而代之，在 softmax 层中我们使用 softmax 函数应用到 \mathbf{z}_j^L 。根据这个函数，第 j 个输出神经元的激活值 a_j^L 是

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}, \quad (78)$$

其中，分母是对所有的输出神经元求和。

如果你对 softmax 函数不熟悉，那么公式 (78) 可能看上去很难理解。同时也不能理解为什么我们想要用这个函数以及这个函数如何帮助我们解决学习速度衰退的问题。为了更好的理解公式 (78)，我们假设有一个包含 4 个输出神经元的神经网络，同时对应 4 个带权的输入，用 $\mathbf{z}_1^L, \mathbf{z}_2^L, \mathbf{z}_3^L, \mathbf{z}_4^L$ 表示。下面展示的是可调节的滑块，能够通过调整滑块来展示可能的带权输入以及相对应的输出激活值。我们可以通过逐步增加 \mathbf{z}_4^L 来开始我们的探索：

[点击前往原作观看交互式动画](#)

当你增加 \mathbf{z}_4^L 的时候，你会发现其对应的输出激活值 a_4^L 也会增加。同样地，如果减少 \mathbf{z}_4^L 那么 a_4^L 也会减少。实际上，如果你仔细观察，可以发现在 a_4^L 增加或者减少的两种情况下，其它 3 个输出激活值得总体改变，补偿了 a_4^L 的改变。原因是所有的输出激活值加起来必须为 1。我们可以使用公式 (78) 以及一点代数知识证明：

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1. \quad (79)$$

所以，如果 a_4^L 增加，那么其它的输出激活值肯定会总共下降相同的量，来保证 4 个激活值得和仍然为 1。当然，这个结论也对其它的激活值成立。

公式 (78) 也表明所有的输出激活值是正的，因为指数函数肯定是正的。结合这点以及上一段的结论，我们可以知道，从 softmax 层得到的输出是一系列相加和为 1 的正数。换言之，从 softmax 层得到的输出可以看做是一个概率分布。

softmax 层的输出是一个概率分布的这个结论是很有价值的。在许多问题中，我们可以很方便地将输出激活值 a_j^L 看作是神经网络认为结果是 j 的概率。比如，在 MNIST 分类问题中，我们可以将 a_j^L 看作是神经网络认为这个数字是 j 的概率估计。

对比一下，如果输出层是 sigmoid 层，我们并不能假设这些激活值能够作为一个概率分布。我不会证明这个结论，但这个结论应该是合理的。所以如果使用 sigmoid 输出层，我们不能使用 softmax 的结论去解读。

¹ 在描述 softmax 的时候，我们会频繁使用上一章介绍过的记号表示。你可能需要重新回顾一下上一章来加强对这些记号的印象。

练习

- 构造例子说明在使用 sigmoid 输出层的网络中，输出激活值 a_j^L 的和并不一定为 1。

我们现在开始体会到 softmax 函数及其 softmax 层的表现。回顾一下：公式 (78) 中的指数函数保证了所有的输出激活值都是正的。同时在公式 (78) 中的分母求和保证了 softmax 输出和为 1。现在这个式子应该不再像开始时候那么难以理解了，它用了一种很自然的方式来确保输出的激活值形成一个概率分布。你可以认为 softmax 对 z_j^L 进行了重新调节、挤压收敛，使得它们形成了一个概率分布。

练习

- softmax** 的单调性 - 证明如果 $j = k$ ，那么 $\partial a_j^L / \partial z_k^L$ 是正的，如果 $j \neq k$ ，那么 $\partial a_j^L / \partial z_k^L$ 是负的。如果我们增加 z_j^L 能够保证增加相应的输出激活值 a_j^L ，同时会减少其它所有的输出激活值，我们已经通过滑块清楚地看到这个结论，但是现在需要一个严格的证明。
- softmax** 的非局部性 - sigmoid 层的一个好处是输出 a_j^L 是其对应输入的一个函数 $a_j^L = \sigma(z_j^L)$ 。解释一下为什么对于 softmax 层并不是这样的情况：任何一个输出激活值 a_j^L 依赖于所有的输入。

问题

- 反转 softmax 层 - 假设我们有一个带有 softmax 输出层的神经网络，同时已知激活值 a_j^L 。证明对应的带权输入的形式为 $z_j^L = \ln a_j^L + C$ ，其中常数 C 是不依赖于 j 的。

学习速度衰退的问题：我们现在已经对神经元的 softmax 层有了一定的认识。但是我们还没有看到 softmax 层如何解决学习速率衰退的问题。为了理解这点，我们先要定义 *log-likelihood* 代价函数。我们用 x 表示输入网络的训练数据，用 y 表示相应的期待输出。然后对应这个输入的 log-likelihood 代价是

$$C \equiv -\ln a_y^L. \quad (80)$$

举例来说，如果我们的训练数据是 MNIST 图像，输入是数字 7 对应的一个图像，那么 log-likelihood 代价是 $-\ln a_7^L$ 。我们从直觉上来看一下其中的含义：当网络工作得很好时，也就是说对于输入为 7 非常有自信。在这种情况下，它估计一个相应的概率 a_7^L 会非常接近 1，所以代价 $-\ln a_7^L$ 就会很小。反之，如果网络工作得不是那么好，那么概率 a_7^L 会变小，同时代价 $-\ln a_7^L$ 会变大。所以 log-likelihood 代价函数满足我们对代价函数表现形式的预期。

那么关于学习速度衰退的问题呢？为了分析它，回忆一下学习速度衰退的关键因素是数量 $\partial C / \partial w_{jk}^L$ 和 $\partial C / \partial b_j^L$ 的表现情况。我不会完整地推导整个过程——请你们自己完成推导过程——通过一些推导，你可以证明²

$$\frac{\partial C}{\partial b_j^L} = a_j^L - y_j \quad (81)$$

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1} (a_j^L - y_j) \quad (82)$$

我们得到了前面从交叉熵的分析中得到的类似的等式。比如等式 (82) 和等式 (67)。尽管后者中我们对整个训练样本数据进行了平均，但是它们的确是同样的公式。同时，正如之前分析的那样，这些表达式确保了我们不会遇到学习速度衰退的问题。事实上，我们可以将 softmax 输出层与 log-likelihood 代价函数的搭配，类比成 sigmoid 输出层与交叉熵代价函数的搭配。

考虑到这样的相似性，你是应该选择 sigmoid 输出层搭配交叉熵还是选择 softmax 输出层搭配 log-likelihood 呢？实际上，在许多情景下，两种方法的效果都不错，在本章的剩余部分，我们会使用 sigmoid 输出层搭配交叉熵。后续在第 6 章中，我们有时会使用 softmax 输出层搭配 log-likelihood 代价函数。切换的原因就是让我们后面的网络和一些有影响力的学术论文中的网络更加接近。通常来说，在任何你想要把输出激活值解读成概率的时候，softmax 加上 log-likelihood 总是不错的选择，至少对于没有交集的分类问题（例如 MNIST）来说是这样的。

²

2 注意这里我对 y 的使用和之前不太一样。之前的 y 用来表示网络期望的输出，例如，如果输入的图片是7则期望的输出是7。但在下面的式子中， y 代表一个与7相关的向量，意思是这个向量的第7位是1，其余位都是0。

问题

- 推导等式 (81) 和 (82)。
- 「softmax」这个名字来源于哪里？- 假设我们改变一下 softmax 函数，使得输出激活值如下

$$a_j^L = \frac{e^{cz_j^L}}{\sum_k e^{cz_k^L}}, \quad (83)$$

其中 c 是一个正常数。注意 $c = 1$ 对应到标准的 softmax 函数。但是如果我们使用不同的 c ，我们会得到不同的函数，尽管如此，最后得到的结果也和 softmax 很相似。证明改变 c 以后，也会像通常的 softmax 函数一样形成一个概率分布。假设我们允许 c 非常大，比如 $c \rightarrow \infty$ ，那么输出激活值 a_j^L 的极限是什么？在解决了这个问题后，你应该能理解 $c = 1$ 时的函数是一个最大化函数的「softened」版本。这就是术语「softmax」的来源。

- 包含 softmax 和 log-likelihood 的反向传播 - 在上一章，我们推导了包含 sigmoid 层的网络的反向传播算法。为了把这个算法用到包含 softmax 层的网络中，我们需要算出最后一层的误差表达式 $\delta_j^L \equiv \partial C / \partial z_j^L$ 。证明表达式如下：

$$\delta_j^L = a_j^L - y_j. \quad (84)$$

使用这个表达式以后，我们就可以在包含 softmax 输出层和 log-likelihood 代价函数的网络中应用反向传播算法了。

过拟合和正则化

当某个数学模型被一些物理学家应用到了一个悬而未决的重要的物理问题上时，诺贝尔物理学奖获得者恩里科·费米（Enrico Fermi）被问及对于此举的观点。虽然模型与实验结果契合的很好，但费米还是对此有所怀疑。他问道「在这个模型中有多少个可以自由设置的参数」，他得到的答案是「4个」。费米回复说¹「我的朋友约翰尼·冯诺依曼（Johnny von Neumann）常说，他用四个参数可以模拟一头大象，用五个可以使它摇动鼻子。」

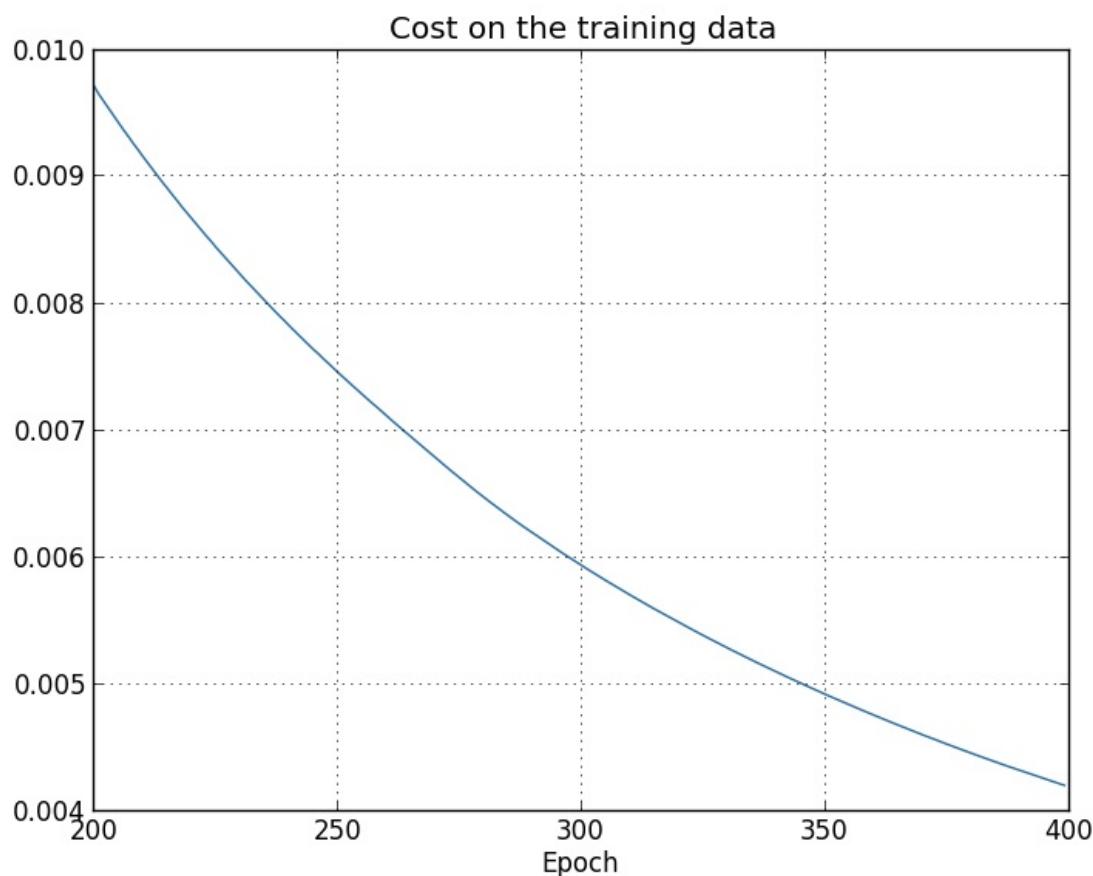
费米的这个观点是在表明：有大量自由参数的模型能够描述一个足够宽泛的现象。即使这样的模型与现有的数据吻合得很好，这也不能说它是一个好的模型。这仅仅只能说明，有足够的自由度的模型基本上可以描述任何给定大小的数据集，但是它并没有真正地洞察到现象背后的本质。这就会导致该模型在现有的数据上表现得很好，但是却不能普及到新的情况上。判断一个模型真正好坏的方法，是看其对未知情况的预测能力。

费米和冯诺伊曼对持有四个参数的模型都抱有怀疑的态度。然而我们分类 MNIST 数字所用的 30 个隐藏神经元网络就有将近 24000 个特征！这个数量非常庞大。100 个隐藏神经元网络有接近 80000 个特征，最先进的深度神经网络或许包含数百万，甚至数亿计的特征。我们应该相信这些结果吗？

下面让我们来举个例子来突出问题的严重性。我们会制造出一种情况，使我们的网络不能够很好地适应新数据。假设我们使用有 30 个隐藏神经元网络，其中含有 23860 个特征。但是不用全部的 50000 个 MINST 图像做训练，而仅仅使用前 1000 个。用较小的训练集能够使突出泛化的问题。此外，我们依然用交叉熵代价函数来训练模型，学习率为 $\eta = 0.5$ ，mini-batch 的大小选为 10。然而，与之前不同的是，我们将要迭代 400 次，由于使用了较少的训练实例，所以就要较多的训练次数。让我们用 `network2` 看一下代价函数的变化趋势：

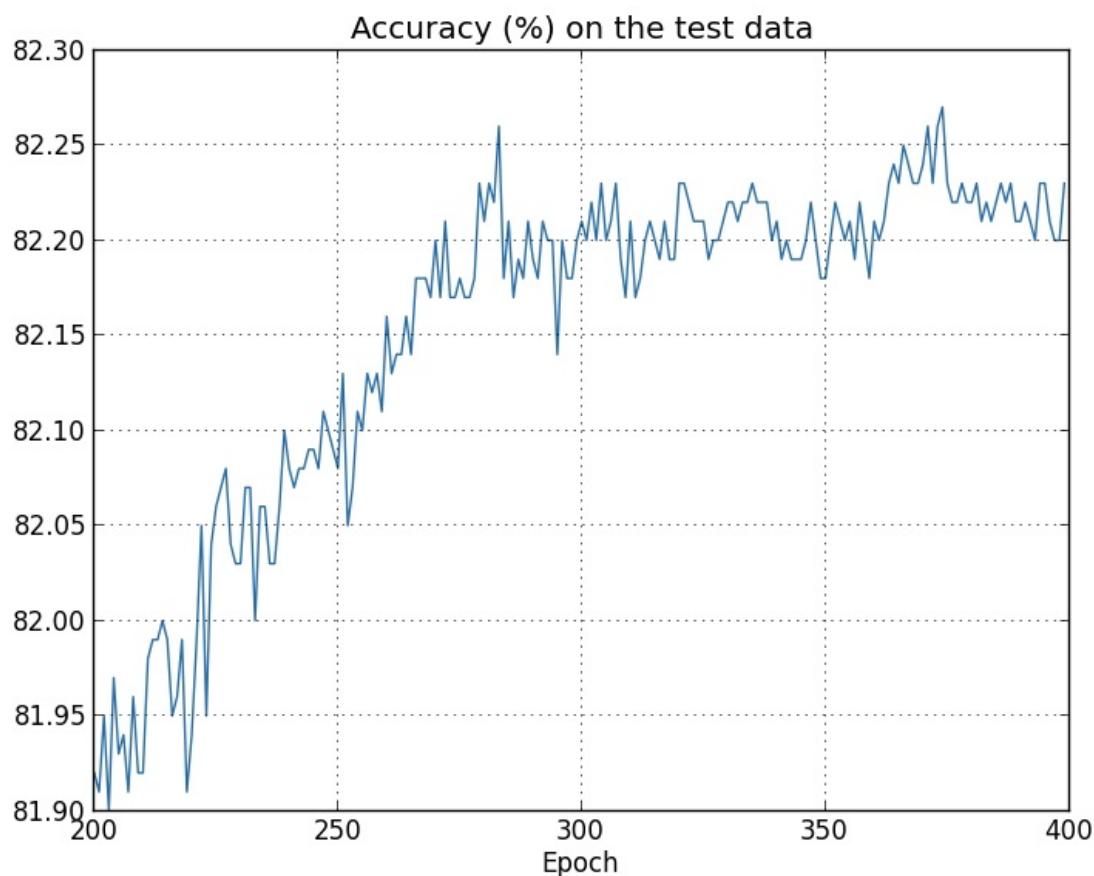
```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
...     mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data[:1000], 400, 10, 0.5, evaluation_data=test_data,
...     monitor_evaluation_accuracy=True, monitor_training_cost=True)
```

根据结果，我们能绘制网络学习成本的变化趋势²：



正如我们期望的那样，代价在平滑地下降。请注意，图中仅仅展示了第200次迭代到第399次迭代的阶段。我们能够从此判断出之后阶段的学习趋势。我们稍后会看到，后面的阶段是真正有趣的地方所在。

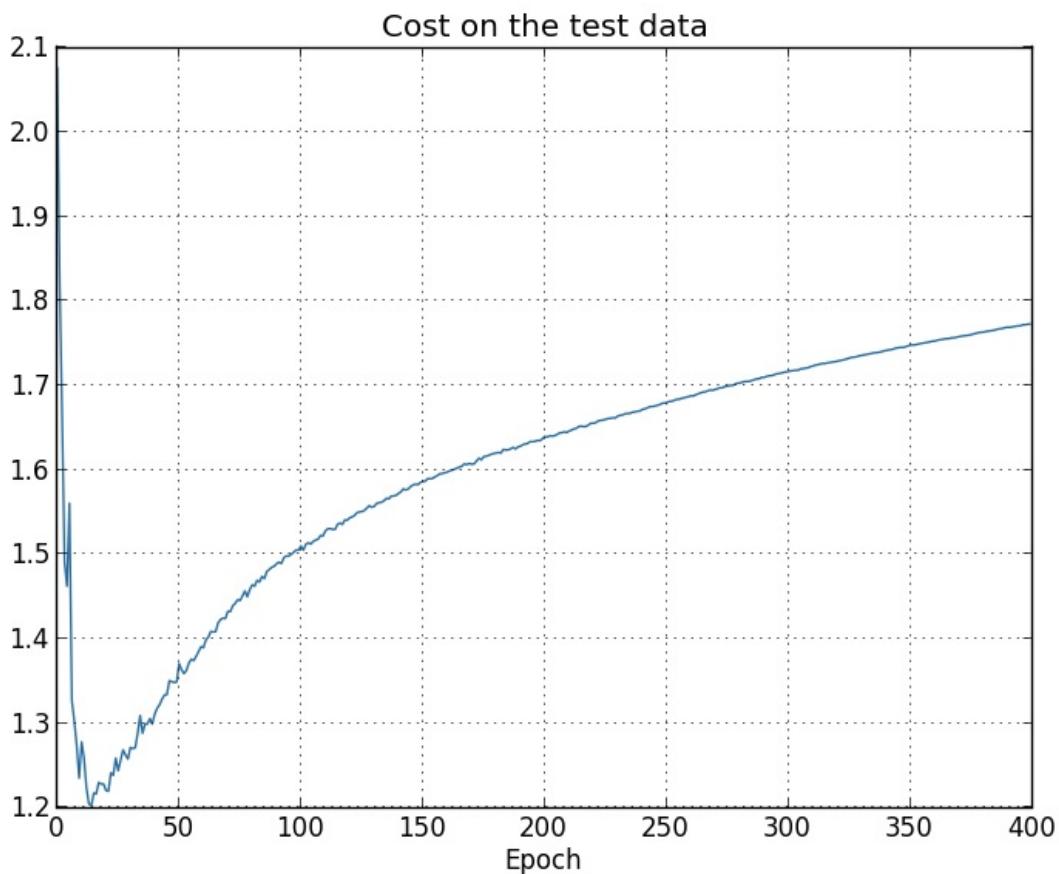
现在让我们来看看测试数据上的分类准确率随着时间是怎样变化的：



同样，我选取了图表的一部分。在前200步（未显示）准确率上升到接近百分之82。然后学习的效果就逐渐放缓。最后，在280步附近，分类准确率几乎停止改善。之后的学习仅仅在280步就达到的准确率附近有一些小的随机波动。与上一个图表对比，我们会发现，训练数据的代价函数值是持续下降的。如果我们只关注代价函数，模型似乎一直在“改进”。但测试精度结果表明：此时的改进只是一种错觉。正如费米所不喜欢的模型那样，在280步之后，我们网络的学习不能够再很好地推广到测试数据上。所以此时的学习是无用的。在280步之后，我们称此时的网络是过拟合（*overfitting*）或过训练（*overtraining*）的。

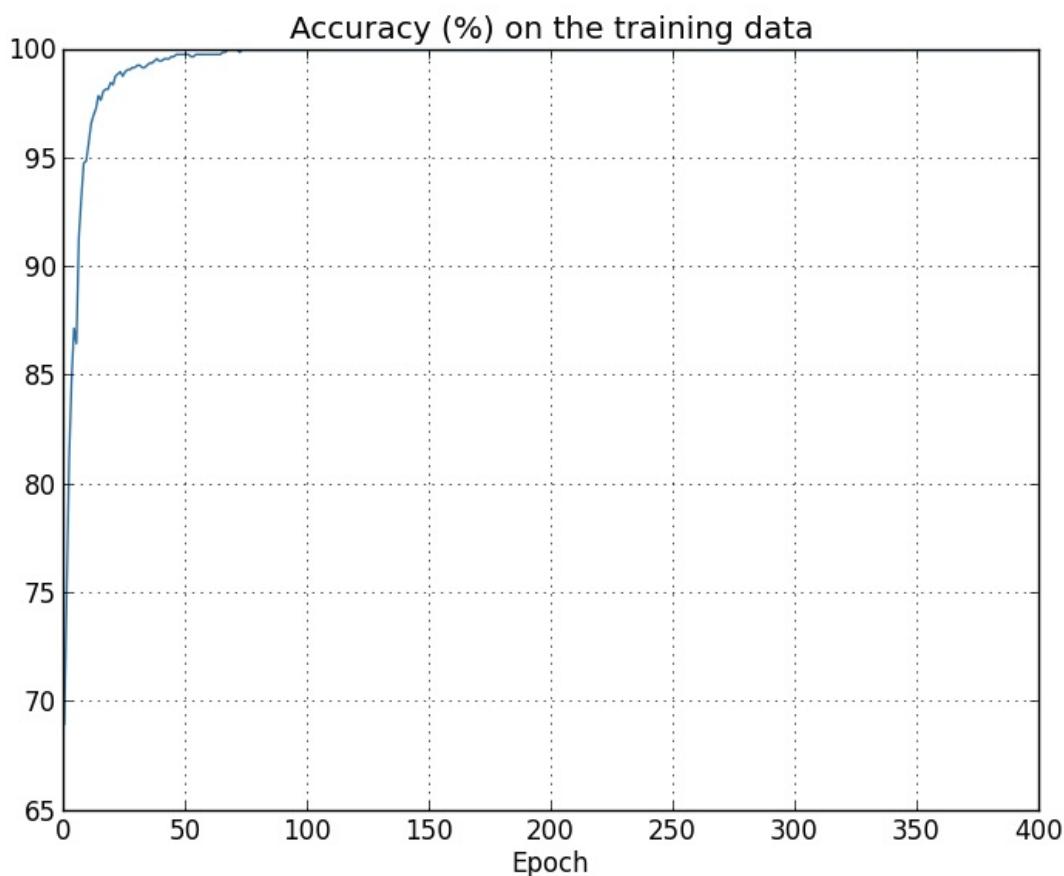
你也许在猜测问题是否出在我关注着训练数据的代价函数，却期待着测试数据的分类准确率，这两个指标完全是风马牛不相及。如果我们选择比较训练数据的代价和测试数据的代价会发生什么呢？这两个同类的指标总可以比较吧？或者我们比较训练数据和测试数据的分类

精度又会怎样呢？事实上，无论我们选取什么指标作比较，本质上都会产生同样的结果。不信的话，让我们来看看测试数据的代价变化：



从图中可以看到，测试数据的代价在15步前一直在降低，之后它开始变大，然而训练数据的代价是在持续降低的。这是另外一个能够表明我们的模型是过拟合的迹象。而此时，我们遇到了一个难题，到底哪一个才是发生过拟合的关键点，15步还是280步？从实际应用的角度来看，我们真正关心的是提高测试数据的分类精度，而测试数据的代价只不过是分类精度的附属品。因此，在我们的神经网络中，把280步视为模型开始过拟合的转折点。

在训练数据的分类精度中也可以看到过拟合的迹象：



准确率一直上升到百分之100。也就是说，网络能正确分类所有1000个训练图像！而与此同时，测试准确率仅为百分之82.27。所以我们的网络只是在学习训练集的特性，而不能完全识别普通的数字。就好像网络仅仅是在记忆训练集，而没有真正理解了数字能够推广到测试集上。

过拟合是神经网络中一个主要的问题。尤其是在含有大量权重和偏差参数的现代网络中。为了更有效地训练，我们需要一种能够检测过拟合发生时间的方法，这样就不会发生过度训练。此外，我们也需要一种能减少过拟合影响的技术。

检测过拟合一个显而易见的方法就是如上面提到的，跟踪网络训练过程中测试数据的准确率。如果测试数据的精度不再提高，就应该停止训练。当然，严格地说，这也不一定就是过拟合的迹象，也许需要同时检测到测试数据和训练数据的精度都不再提高时才行。当然，这个策略是能够避免过拟合的。

事实上，我们将采用这种策略的一个变种。回想一下，我们曾载入了三个MNIST数据集：

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
```

到目前为止，我们一直在使用 `training_data` 和 `test_data`，忽视了 `validation_data`。`validation_data` 包含 10000 张与 50000 张 `training_data` 和 1000 张 `test_data` 图像不同的 MNIST 数字图像。我们将使用 `validation_data` 而不是 `test_data` 来预防过拟合。为了做到这一点，将使用与上面 `test_data` 相同的方法。也就是说，在每一步训练之后，计算 `validation_data` 的分类精度。一旦 `validation_data` 的分类精度达到饱和，就停止训练。这种策略叫做提前终止（*early stopping*）。当然在实践中，我们并不能立即知道什么时候准确度已经饱和。取而代之，我们在确信精度已经饱和之前会一直训练³。

为什么要用 `validation_data` 而不是 `test_data` 来防止过拟合呢？通过 `validation_data` 来选择不同的超参数（例如，训练步数、学习率、最佳网络结构、等等）是一个普遍的策略。我们通过这样的评估来计算和设置合适的超参数值。虽然我到现在才提到超参数，但其实我在之前就已经提到怎样选择超参数了。（后面会介绍更多）

当然，上面的解释不能回答为什么我用 `validation_data` 而不是 `test_data` 来防止过拟合。实际上，更一般的问题是为什么用 `validation_data` 而不是 `test_data` 来设置超参数？当我们设置超参数时，有很多种不同的选择。如果基于 `test_data` 的评估结果设置超参数，有可能我们的网络最后是对 `test_data` 过拟合。也就是说，我们或许只是找到了适合 `test_data` 具体特征的超参数，网络的性能不能推广到其它的数据集。通过 `validation_data` 来设置超参数能够避免这种情况的发生。然后，一旦我们得到了想要的超参数，就用 `test_data` 做最后的精度评估。这让我们相信 `test_data` 的精度能够真正提现网络的泛化能力。换句话说，你能把 `validation_data` 视为帮助我们学习合适超参数的一种训练数据。由于 `validation_data` 和 `test_data` 是完全分离开的，所以这种找到优秀超参数的方法被称为分离法（*hold out method*）。

如果我们在对 `test_data` 进行了性能评估后，我们突然改变主意想要尝试选择不同方法——例如另一种网络结构——其中需要也涉及一批新的超参数设置。如果我们这样做了，不也有可能导致模型对 `test_data` 过拟合吗？我们需要一个潜在的无限数据集来使我们的模型更有泛化能力吗？这是个很深刻很棘手的问题。不过，出于实践目的，我们不必有过多担心。仅仅用上面提到的，基于 `training_data`、`validation_data` 和 `test_data` 的分离法来解决就行。

目前为止，我们已经看过了只用 1000 张训练图片的过拟合状况。而当用全部的 50000 图片训练集时又会发生什么呢？保持相同的参数（30 个隐藏神经元，学习率为 0.5，mini-batch 的大小为 10），使用全部 50000 张图片，迭代 30 次。下面是显示训练数据和测试数据分类精度的图表。注意我在这里用了测试数据，而不是验证数据，是为了与前面的图片做更直接的比较。

目前为止，我们已经看过了只用1000张训练图片的过拟合状况。而当用全部的50000图片训练集时又会发生什么呢？保持相同的参数（30个隐藏神经元，学习率为0.5，mini-batch的大小为10），使用全部50000张图片，迭代30次。下面是显示训练数据和测试数据分类精度的图表。注意我在这里用了测试数据，而不是验证数据，是为了与前面的图片做更直接的比较。



正如你看到的，相比使用1000张训练实例，使用50000张训练实例的情况下，测试数据和训练数据的准确率更加接近。特别地，训练数据上最高的分类精度百分之97.86仅仅比测试数据的百分之95.33高出1.53个百分点。而之前有百分之17.73的差距！虽然过拟合仍然存在，但已经大大降低了。我们的网络能从训练数据更好地泛化到测试数据。一般来说，增加训练数据的数量是降低过拟合的最好方法之一。即便拥有足够的训练数据，要让一个非常庞大的网络过拟合也是比较困难的。不幸的是，训练数据的获取成本太高，因此这通常不是一个现实的选择。

¹引用来自Freeman Dyson的一篇精彩的文章。他曾提出了有缺陷的模型，四参数大象的例子可以在[这里](#)找到。

²本图和接下来四幅图都是通过运行`overfitting.py`产生的。

³它需要一些判断来决定何时停止。在前面的图表中，我挑选280步作为精度饱和点。这有可能是过于悲观。神经网络在训练的过程中有时会停滞一段时间，然后才会接着改善。如果在400步之后网络还能够继续学习，我也不惊讶，不过，就算有任何进一步的提高，幅度可能

正则化

避免过拟合的方法之一是增加训练数据数量。那么，还有没有别的方法能让我们避免过拟合呢？一种可能的方法是减小网络的规模。然而，我们并不情愿减小规模，因为大型网络比小型网络有更大的潜力。

幸好，哪怕使用固定的网络和固定的训练数据，我们还有别的方法来避免过拟合。这就是所谓的正则化（regularization）技术。在这一节我将描述一种最常用的正则化技术——权重衰减（weight decay）或叫 L2 正则（L2 regularization）。L2 正则的思想是，在代价函数中加入一个额外的正则化项。这是正则化之后的交叉熵：

$$C = -\frac{1}{n} \sum_{x_j} [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)] + \frac{\lambda}{2n} \sum_w w^2. \quad (85)$$

第一项是常规的交叉熵表达式。但我们加入了第二项，也就是网络中所有权值的平方和。它由参数 $\lambda/2n$ 进行调整，其中 $\lambda > 0$ 被称为正则化参数（regularization parameter）， n 是我们训练集的大小。我稍后会讨论应该如何选择 λ 。另外请注意正则化项不包括偏移。我稍后也会提到这点。

当然，我们也可以对其他的代价函数进行正则化，例如平方代价。正则化的方法与上面类似：

$$C = \frac{1}{2n} \sum_x \|y - a^L\|^2 + \frac{\lambda}{2n} \sum_w w^2. \quad (86)$$

在两种情况中，我们都能够把正则化的代价函数写成：

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2, \quad (87)$$

其中 C_0 是原本的、没有正则化的代价函数。

直观来说，正则化的作用是让网络偏好学习更小的权值，而在其它的方面保持不变。选择较大的权值只有一种情况，那就是它们能显著地改进代价函数的第一部分。换句话说，正则化可以视作一种能够折中考虑小权值和最小化原来代价函数的方法。两个要素的相对重要性由 λ 的值决定：当 λ 较小时，我们偏好最小化原本的代价函数，而 λ 较大时我们偏好更小的权值。

为什么这种折中能够减少过拟合，其中的原因也太不明显了！但事实是它的确能够减少过拟合。我们将在下一部分阐述为何这样的折中能够减少过拟合。先让我们通过一个例子展示正则化确实能够减小过拟合。

为了构建这样一个例子，我们首先要解决如何把随机梯度下降学习算法应用于正则化的神经网络中。我们尤其需要知道如何对网络中所有的权值和偏移计算偏导数 $\partial C / \partial w$ 和 $\partial C / \partial b$ 。对等式 (87) 求偏导可得：

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w \quad (88)$$

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}. \quad (89)$$

正如上一章所述，其中 $\partial C_0 / \partial w$ 和 $\partial C_0 / \partial b$ 可由反向传播计算。于是我们发现计算正则化代价函数的梯度相当简单：只要照常使用反向传播，并把 $\frac{\lambda}{n} w$ 加到所有权值项的偏导数中。偏移的偏导数保持不变，所以偏移的梯度下降学习的规则保持常规的不变：

$$b \rightarrow b - \eta \frac{\partial C_0}{\partial b}. \quad (90)$$

而对于权值的学习规则变为：

$$w \rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \quad (91)$$

$$= \left(1 - \frac{\eta \lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}. \quad (92)$$

其它部分与常规的梯度下降学习规则完全一样，不一样的地方是我们以 $1 - \frac{\eta \lambda}{n}$ 调整权值 w 。这种调整有时也被称作权重衰减（*weight decay*），因为它减小了权重。一眼看去权值将被不停地减小直到为 0。但实际上并不是这样的，因为如果可以减小未正则化的代价函数的话，式中的另外一项可能会让权值增加。

好的，这就是梯度下降实现的方法。那么随机梯度下降呢？和未正则化的随机梯度下降一样，我们首先在包含 m 个训练样例的 mini-batch 数据中进行平均以估计 $\partial C_0 / \partial w$ 的值。因此对于随机梯度下降法而言正则化的学习方法就变成了（参考等式 (20)）：

$$w \rightarrow \left(1 - \frac{\eta \lambda}{n}\right) w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w}, \quad (93)$$

其中的求和是对 mini-batch 中的所有训练样例进行的， C_x 是每个样例对应的未正则化的代价。这与通常的随机梯度下降方法一致，除了权重衰减变量 $1 - \frac{\eta \lambda}{n}$ 。最后，为了表述完整，我要说明对于偏移的正则化学习规则。毫无疑问，那就是与未正则化的情况（如等式 (21)）完全一样，

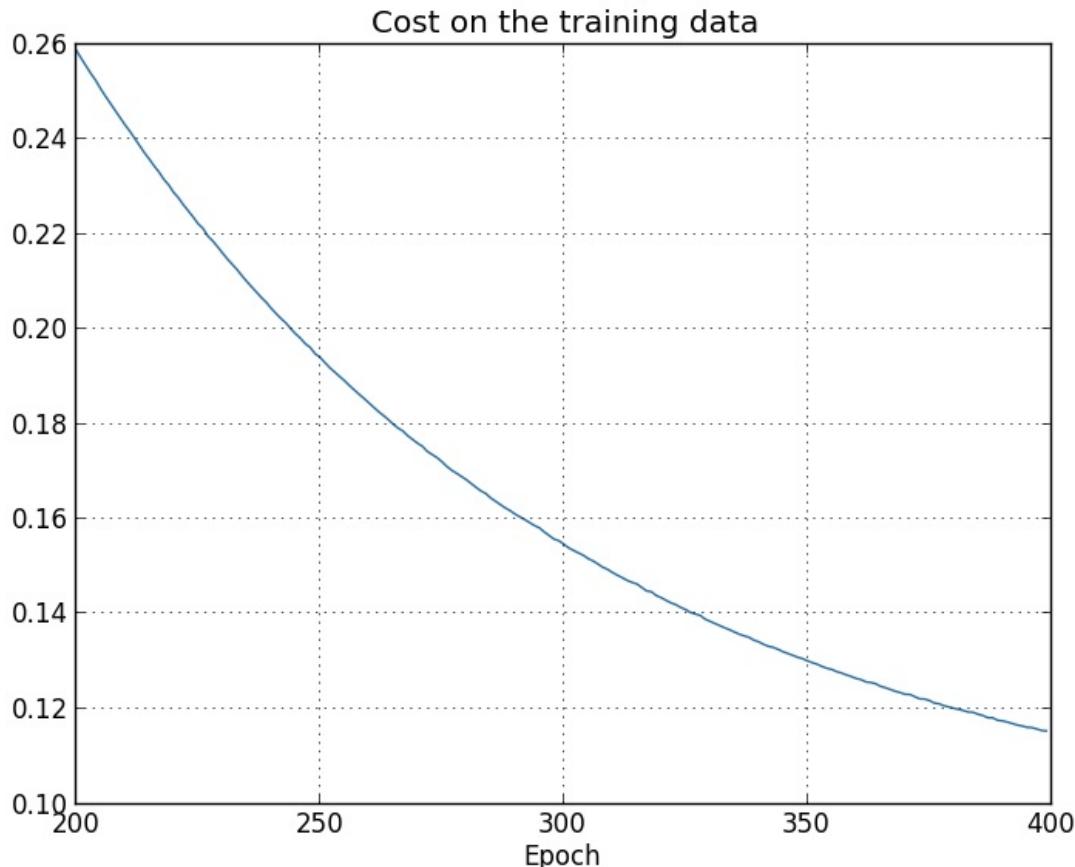
$$b \rightarrow b - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial b}, \quad (94)$$

其中的求和是对 mini-batch 中的所有训练样例 x 进行的。

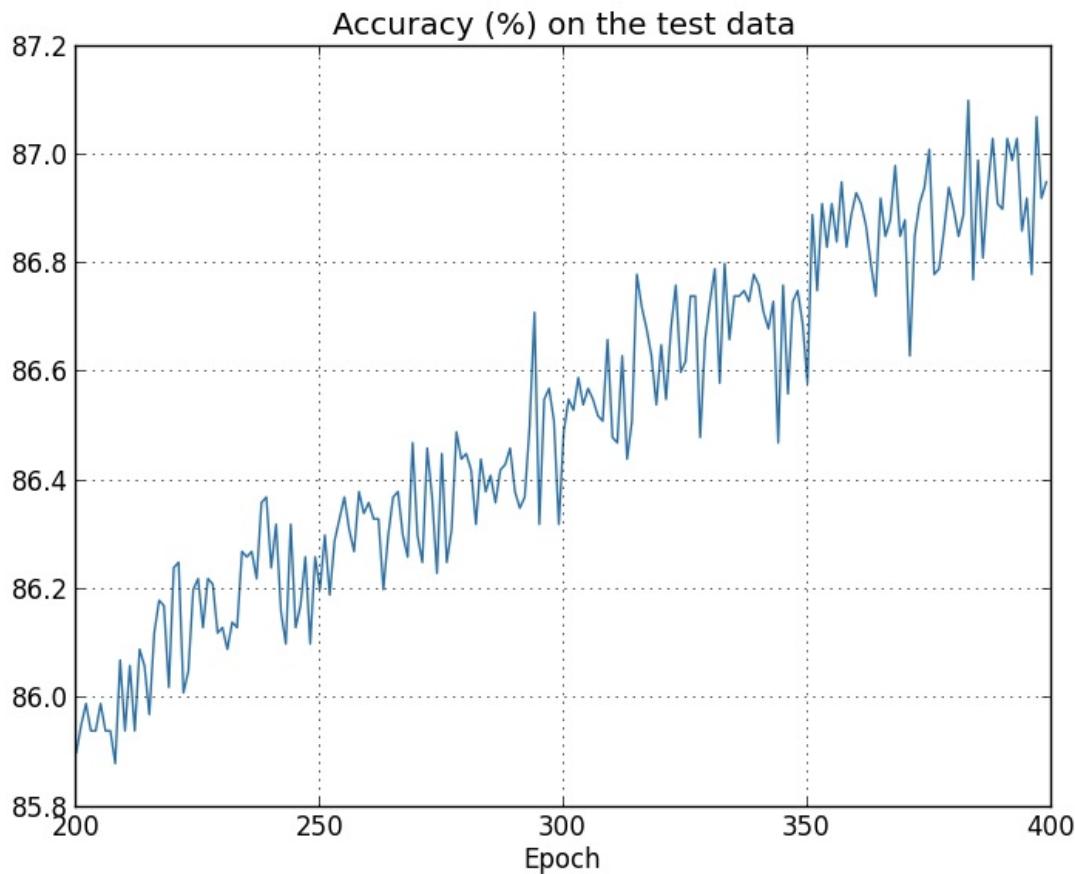
让我们看看正则化如何改变了我们的神经网络的表现。我们将使用一个神经网络来进行验证，其包含 30 个隐藏神经元，mini-batch 大小为 10，学习率为 0.5，并以交叉熵作为代价函数。然而，这次我们设置正则化参数 $\lambda = 0.1$ 。注意在代码中，我们使用变量名 `lmbda`，因为 `lambda` 是 Python 的保留字，其含义与此无关。我也再次使用了 `test_data` 而非 `validation_data`。严格来说，我们应该使用 `validation_data`，详细原因我们在此前已经解释过。不过我决定用 `test_data` 因为它能让结果与我们此前未正则化的结果进行更直观的比较。对代码稍作改动你即可使用 `validation_data`，并且你将发现得到的结果很相似。

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
...     mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10](), cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data[:1000](), 400, 10, 0.5,
...     evaluation_data=test_data, lmbda = 0.1,
...     monitor_evaluation_cost=True, monitor_evaluation_accuracy=True,
...     monitor_training_cost=True, monitor_training_accuracy=True)
```

代价函数一直都在下降，几乎与此前在未正则化的情形一样¹：



但这次它在 `test_data` 上的准确率在 400 次迭代中保持提升：



显然，应用正则化抑制了过拟合。同时，准确率也显著提升了：分类准确率峰值为 87.1，更高于未正则化例子中的峰值 82.27。实际上，我们几乎确定，继续进行迭代可以得到更好的结果。从经验来看，正则化让我们的网络生成得更好，并有效地减弱了过拟合效应。

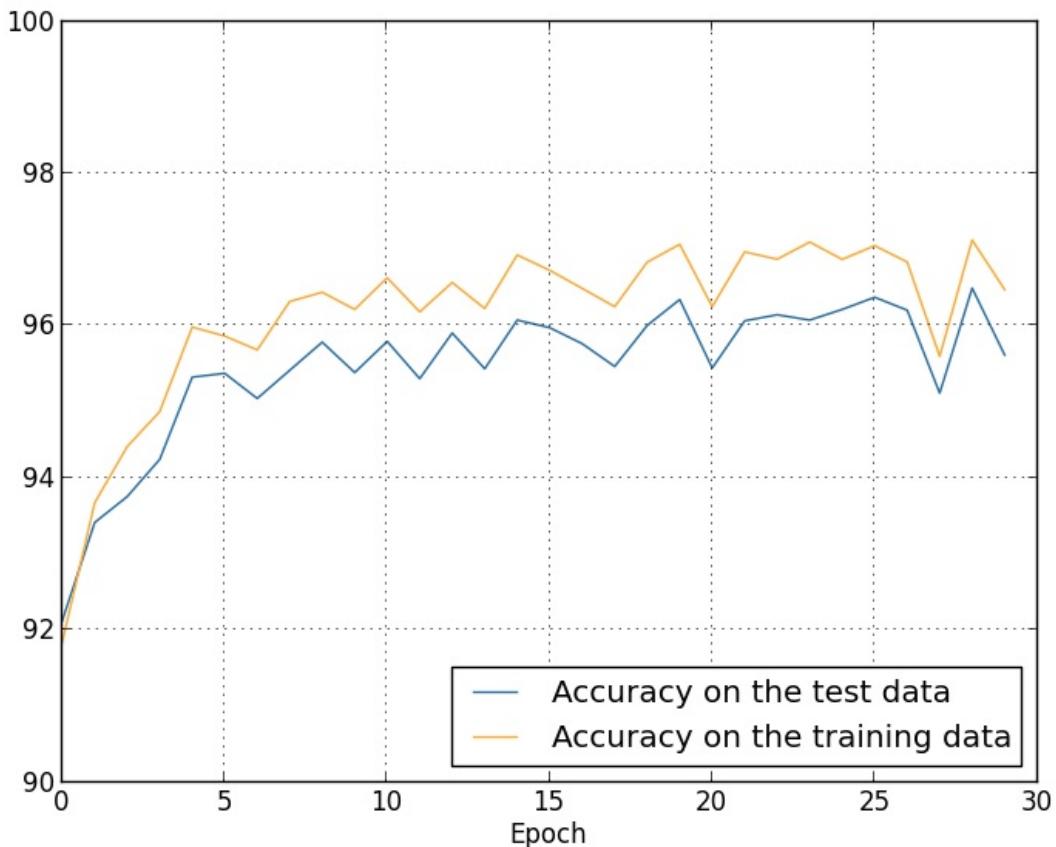
如果我们从仅有 1000 张训练图片迁移到有着 50000 张图片的环境，结果又会怎么样呢？当然，我们已经看到过拟合对于 50000 张图片来说不再是个大问题了。那么正则化能在这上面提供更多帮助吗？让我们把超参数保持跟以前的一样：30 次迭代，学习率为 0.5，mini-batch 数目为 10。然而，我们需要调整正则化参数。原因是，训练数据集的大小已经从

$n = 1000$ 变成了 $n = 50000$ ，而这就改变了权重衰减因数 $1 - \frac{\eta\lambda}{n}$ 。如果我们继续使用 $\lambda = 0.1$ 那么权重衰减将会大大减小，而正则化的效果也会相应减弱。我们调整 $\lambda = 5.0$ 以进行补偿。

好的，让我们训练我们的网络，并在此之前重新初始化权值：

```
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5,
... evaluation_data=test_data, lmbda = 5.0,
... monitor_evaluation_accuracy=True, monitor_training_accuracy=True)
```

我们得到结果如图：



这里有很多好消息。首先，我们在测试数据上的分类准确率提升了，从未正则化时的百分之 95.49 到百分之 96.49。这是一个重大的提升。第二，我们能看到训练数据的结果与测试数据之间的间隙相比之前也大大缩小了，在一个百分点之下。虽然这仍然是一个明显的间隙，但我们显然已经在减弱过拟合上做出了大量的提升。

最后，让我们看看当我们使用 100 个隐藏神经元并设置 $\lambda = 5.0$ 时测试分类准确率。我将不会对这里的过拟合进行详细的分析，这只是纯粹为了有趣，只为了看看我们使用新的技巧——交叉熵代价函数和 L2 正则化——能让我们得到多高的准确率。

```
>>> net = network2.Network([784, 100, 10](), \
... cost=network2.CrossEntropyCost)
>>> net.large_weight_initializer()
>>> net.SGD(training_data, 30, 10, 0.5, lambda=5.0, \
... evaluation_data=validation_data, \
... monitor_evaluation_accuracy=True)
```

最终结果是，分类准确率在验证数据上达到了 97.92。相对于 30 个隐藏神经元的情形，这是一个飞跃。事实上，只要再训练更多些，保持 $\eta = 0.1$ 和 $\lambda = 5.0$ ，增加到 60 次迭代，我们能够突破百分之 98 的壁垒，在验证数据上达到百分之 98.04 的准确率。对于这 152 行代码来说是不错的结果了！

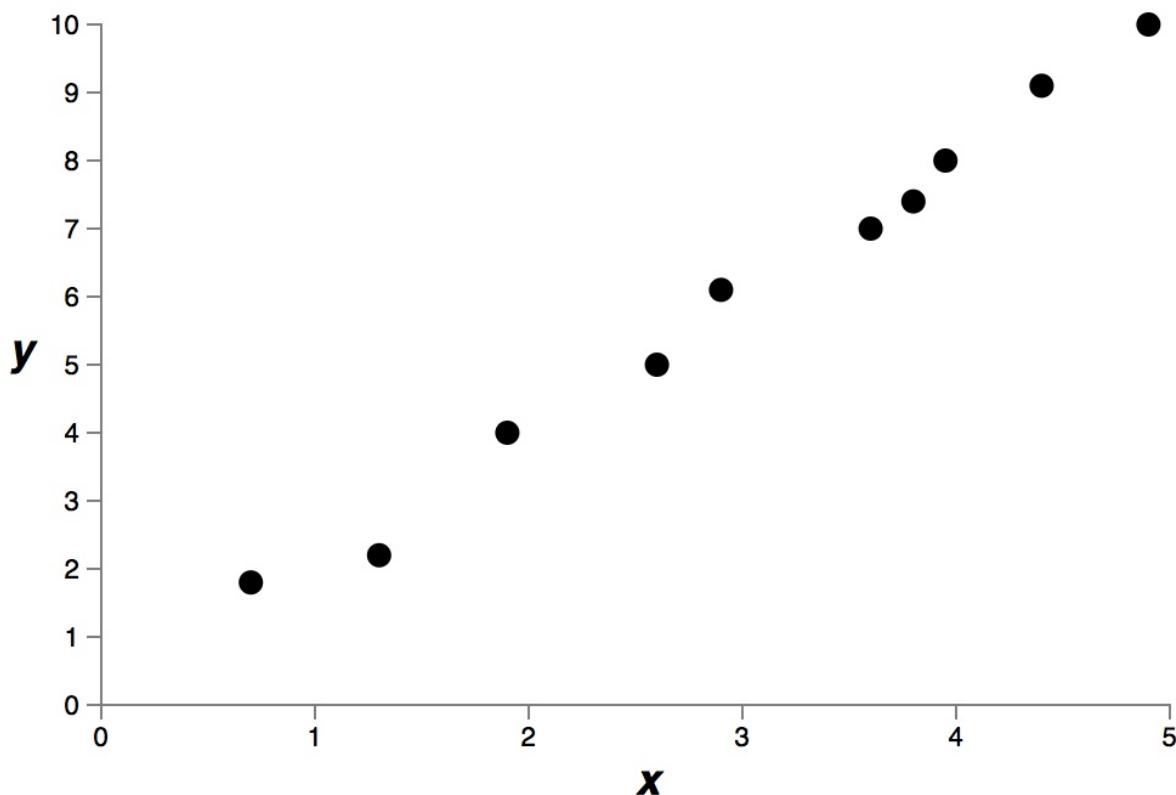
我已经描述了正则化作为一种减弱过拟合并提升分类准确率的方法。事实上，这并不是它唯一的好处。经验表明，当在多次运行我们的 MNIST 网络，并使用不同的（随机）权值初始化时，我发现未正则化的那些偶尔会被「卡住」，似乎陷入了代价函数的局部最优中。结果就是每次运行可能产生相当不同的结果。相反，正则化的那些的每次运行可以提供更容易复现的结果。

为什么会这样呢？启发式地来说，如果代价函数没有正则化，那么权重向量的长度倾向于增长，而其它的都不变。随着时间推移，权重向量将会变得非常大。这可能导致权重向量被限制得或多或少指向同一个方向，因为当长度过长时，梯度下降只能带来很小的变化。我相信这一现象令我们的学习算法难于恰当地探索权重空间，因而难以给代价函数找到一个好的极小值。

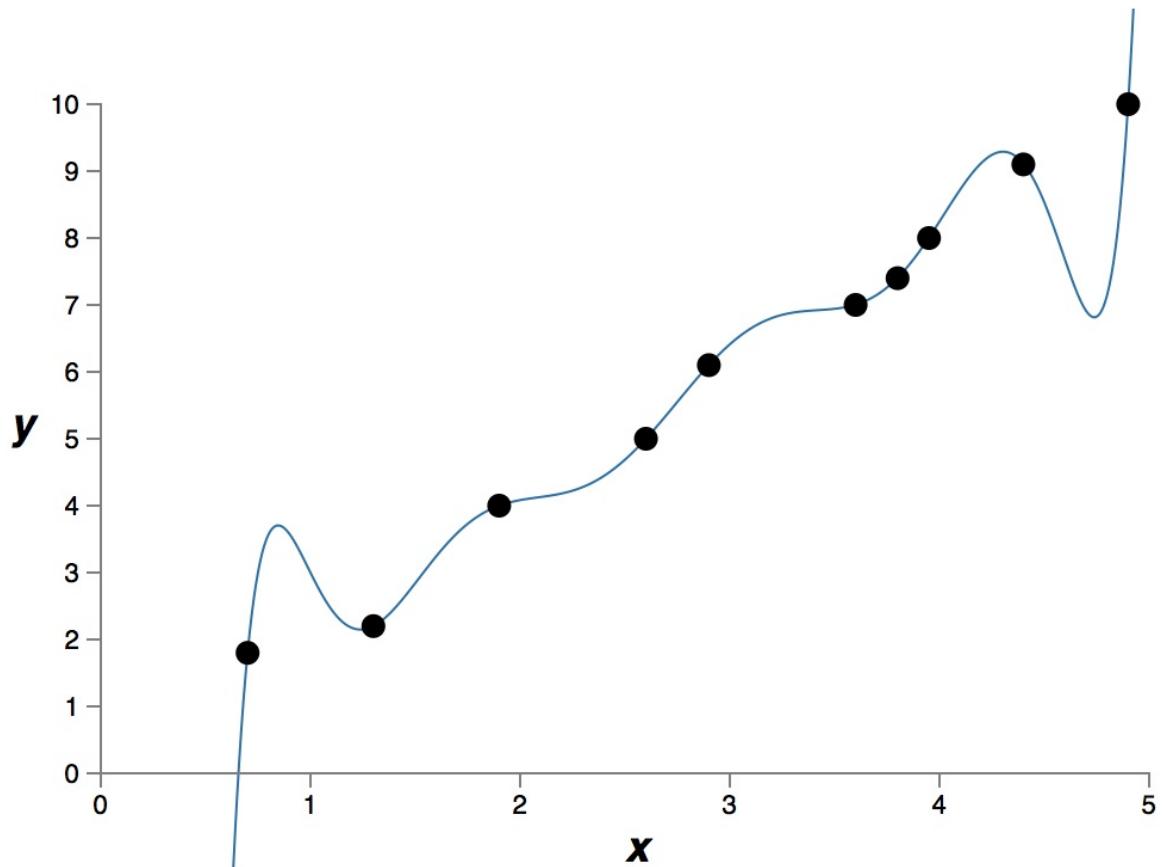
¹ 这张图和下两张图的结果是由程序 [overfitting.py](#) 产生的。

为什么正则化能够降低过拟合

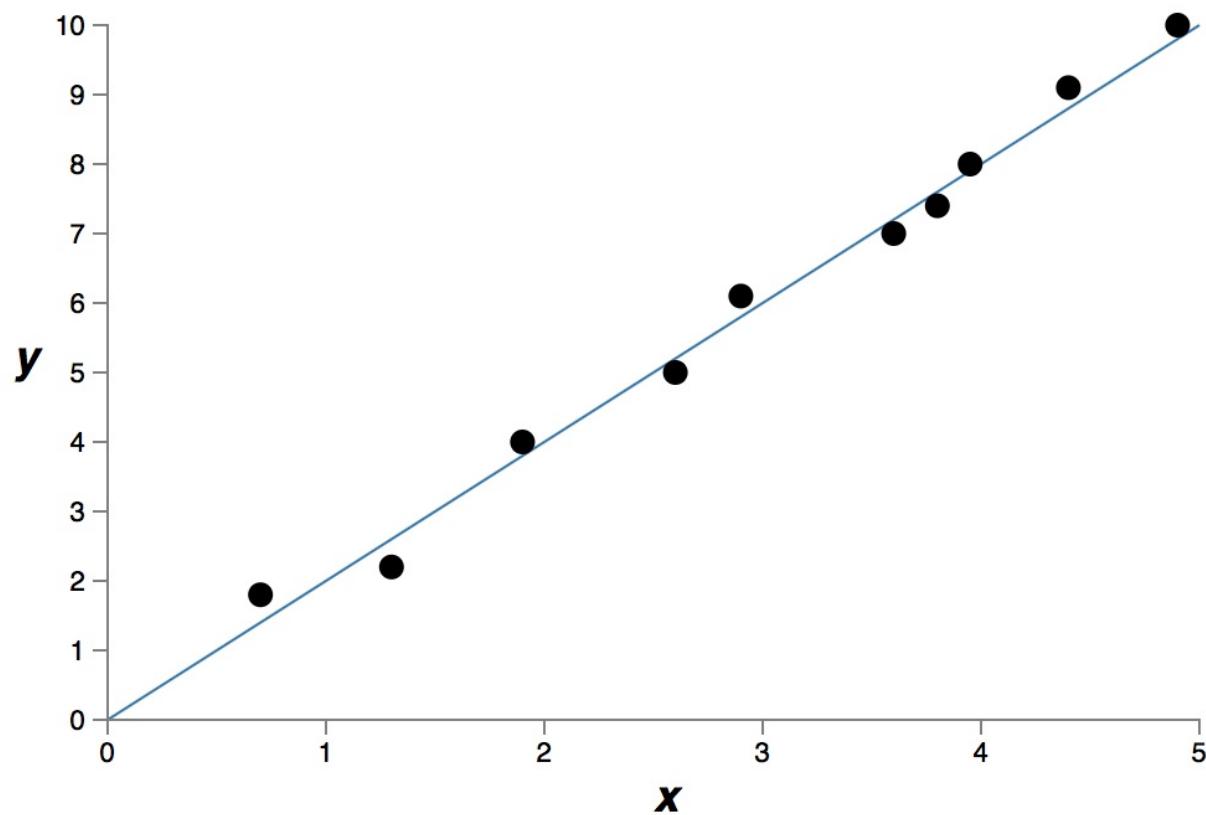
我们通过实验发现正则化能帮助减少过拟合。这是令人高兴的事，然而不幸的是，我们没有明显的证据证明为什么正则化可以起到这个效果！一个大家经常说起的解释是：在某种程度上，越小的权重复杂度越低，因此能够更简单且更有效地描绘数据，所以我们倾向于选择这样的权重。尽管这是个很简短的解释，却也包含了一些疑点。让我们来更加仔细地探讨一下这个解释。假设我们要对一个简单的数据集建立模型：



这个数据是现实世界某个问题提取得到的 x 和 y 。我们的目标是构建一个模型，得到基于 x 的能预测 y 的函数。我们可以尝试使用神经网络来构建模型，但是我将使用更简单的方法：我将把 y 建模为关于 x 的多项式。我们将使用这种方法来代替神经网络，因为多项式模型十分透明。一旦我们理解了多项式的情况，我们就可以把它迁移到神经网络上。现在，在上图中有十个点，这意味着我们可以找到一个 9 次多项式 $y = a_0 x^9 + a_1 x^8 + \dots + a_9$ 来精确拟合这些数据。这是该多项式的图像：



它精确拟合了数据。但是我们也可以使用线性模型 $y = 2x$ 来得到一个较好的拟合：



这两个哪个才是更好的模型呢？哪个更贴近真实情况？另外哪个模型能更好地泛化该问题的其它数据呢？

这些问题很难回答。事实上，如果没有足够多关于现实情况的信息时，我们很难回答任何一个问题。但是让我们考虑两种可能性：(1) 9次多项式事实上真正描述了现实情况，因此这个模型可以完美泛化；(2) 正确的模型是 $y = 2x$ ，但是实验中有一些因为测量误差引入的额外的噪声，也因此这个模型没有完美拟合。

确定这两种可能（可能还有第三种可能存在）哪一个正确的是不能先验的。逻辑上讲，任何一个都可能是正确的。并且这两者的差异并非微不足道。我们承认如果只关注提供的数据集，这两个模型只有很细微的差别。但是如果我们要在比我们之前展示的任何一张图都多得多的 x 上，预测 y 的值，那么两个模型的预测结果将有巨大的差异，9次多项式模型将主要由 x^9 项决定，而线性模型依旧还是线性。

有一种看法是，在科学上，除非迫不得已，我们都应该用更简单的解释。当我们找到一个看起来能解释很多数据点的简单的模型的时候我们会忍不住大喊「找到啦！」。毕竟一个简单的解释的出现似乎不可能仅仅是因为巧合，我们猜测这个模型一定表达了关于这个现象的一些潜在真理。在我们的情况中，模型 $y = 2x + \text{noise}$ 看起来要比

$y = a_0 x^9 + a_1 x^8 + \dots$ 更简单。这种简单的模型的意外出现令人吃惊，我们也因此猜测 $y = 2x + \text{noise}$ 表达了一些潜在的真理。基于这种观点，9次模型真的只是学习到了一些局部噪声的影响。因此当9次模型完美拟合了这一特定数据集的时候，这个模型不能很好泛化到其它数据集上，所以包含噪声的线性模型在预测中会有更好的表现。

我们来看看这种观点对神经网络来说意味着什么。设想我们的网络大部分都有较小的权重，正如在正则化网络中常出现的那样。小权重意味着网络的行为不会因为我们随意更改了一些输入而改变太多。这使得它不容易学习到数据中局部噪声。可以把它想象成一种能使孤立的数据不会过多影响网络输出的方法，相反地，一个正则化的网络会学习去响应一些经常出现在整个训练集中的实例。与之相对的是，如果输入有一些小的变化，一个拥有大权重的网络会大幅改变其行为来响应变化。因此一个未正则化的网络可以利用大权重来学习得到训练集中包含了大量噪声信息的复杂模型。概括来说，正则化网络能够限制在对训练数据中常见数据构建出相对简单的模型，并且对训练数据中的各种各样的噪声有较好的抵抗能力。所以我们希望它能使我们的网络真正学习到问题中的现象的本质，并且能更好的进行泛化。

按照这种说法，你可能会对这种更倾向简单模型的想法感到紧张。人们有时把这种想法称作「奥卡姆剃刀」，并且就好像它是科学原理一样，热情地应用它。然而，它并不是一个普遍成立的科学原理。并不存在一个先验的符合逻辑的理由倾向于简单的模型，而不是复杂的模型。实际上，有时候更复杂的模型反而是正确的。

让我介绍两个正确结果是复杂模型的例子吧。在 1940 年代物理学家马塞尔施恩 (Marcel Schein) 宣布发现了一个新的自然粒子。他工作所在的通用电气公司欣喜若狂并广泛地宣传了这一发现。但是物理学家汉斯贝特 (Hans Bethe) 却怀疑这一发现。贝特拜访了施恩，并且查看了新粒子的轨迹图表。施恩向贝特一张一张地展示，但是贝特在每一张图表上都发现

了一些问题，这些问题暗示着数据应该被丢弃。最后，施恩向贝特展示了一张看起来不错的图表。贝特说它可能只是一个统计学上的巧合。施恩说「是的，但是这种统计学巧合的几率，即便是按照你自己的公式，也只有五分之一。」贝特说「但是我们已经看过了五个图表。」最后，施恩说道「但是在我的图表上，每一个较好的图表，你都用不同的理论来解释，然而我有一个假设可以解释所有的图表，就是它们是新粒子。」贝特回应道「你我的学说的唯一区别在于你的是错误的而我的都是正确的。你简单的解释是错的，而我复杂的解释是正确的。」随后的研究证实了大自然是赞同贝特的学说的，之后也没有什么施恩的粒子了¹。

另一个例子是，1859年天文学家勒维耶（Urbain Le Verrier）发现水星轨道没有按照牛顿的引力理论，形成应有的形状。它跟牛顿的理论有一个很小很小的偏差，一些当时被接受的解释是，牛顿的理论或多或少是正确的，但是需要一些小小的调整。1916年，爱因斯坦表明这一偏差可以很好地通过他的广义相对论来解释，这一理论从根本上不同于牛顿引力理论，并且基于更复杂的数学。尽管有额外的复杂性，但我们今天已经接受了爱因斯坦的解释，而牛顿的引力理论，即便是调整过的形式，也是错误的。这某种程度上是因为我们现在知道了爱因斯坦的理论解释了许多牛顿的理论难以解释的现象。此外，更令人印象深刻的是，爱因斯坦的理论准确的预测了一些牛顿的理论完全没有预测的现象。但这些令人印象深刻的优点在早期并不是显而易见的。如果一个人仅仅是以朴素这一理由来判断，那么更好的理论就会是某种调整后的牛顿理论。

这些故事有三个意义。第一，判断两个解释哪个才是真正的「简单」是一个非常微妙的事情。第二，即便我们能做出这样的判断，简单是一个必须非常谨慎使用的指标。第三，真正测试一个模型的不是简单与否，更重要在于它在预测新的情况时表现如何。

谨慎来说，经验表明正则化的神经网络通常要比未正则化的网络泛化能力更好。因此本书的剩余部分我们将频繁地使用正则化。我举出上面的故事仅仅是为了帮助解释为什么还没有人研究出一个完全令人信服的理论来解释为什么正则化会帮助网络泛化。事实上，研究人员仍然在研究正则化的不同方法，对比哪种效果更好，并且尝试去解释为什么不同的方法有更好或更差的效果。所以你可以看到正则化是作为一种「杂牌军」存在的。虽然它经常有帮助，但我们并没有一套令人满意的系统理解为什么它有帮助，我们有的仅仅是是没有科学依据的经验法则。

这存在一个更深层的问题，一个科学的核心问题。就是我们怎么去泛化这一问题。正则化可以给我们一个计算魔法棒来帮助我们的网络更好的泛化，但是它并没有给我们一个原则性的解释泛化是如何工作的，也没有告诉我们最好的方法是什么。

这尤其令人烦恼，因为在日常生活中，我们人类有很好的泛化现象的能力。给一个孩子看几张大象的图片，他就能很快地学习并辨认出其它大象。当然，他们偶尔也会犯错，也许会无法区分一个犀牛和一个大象，但是总体来说这个过程非常的准确。因此我们有一个系统——人的大脑——拥有大量的自由变量。对这个系统展示一张或几张训练图片之后，这个系统就可以学习并泛化其它的图片。我们的大脑在某种程度上，正则化得非常好！我们是怎么做到的？目前我们还不知道。我预计未来几年人工神经网络领域将开发出更强大的正则化技术，这些技术能使神经网络能更好地泛化，即使数据集非常小。

事实上，我们的网络已先天地泛化得很好。一个具有 100 个隐层神经元的网络有近 80,000 个参数。而我们的训练数据中只有 50,000 个图像。这就好比试图将一个 80,000 次多项式拟合为 50,000 个数据点。按理来说，我们的网络应该退化得非常严重。然而，正如我们所见，这样一个网络事实上泛化得非常好。为什么会是这样？这不太好理解。据推测²说「多层网络中的梯度下降学习的过程中有一个『自我正则化』效应」。这是非常意外的好处，但是这也某种程度上让人不安，因为我们不知道它是怎么工作的。与此同时，我们将采用一些更务实的方法并尽可能地应用正则化。我们的神经网络将会因此变得更好。

让我用一个之前没有解释到的细节作为这一部分的总结：事实上 L2 正则化没有约束偏置（biases）。当然，通过修改正则化过程来正则化偏置会很容易。但根据经验，这样做往往不能较明显地改变结果。所以在一定程度上，是否正则化偏置仅仅是一个习惯问题。然而值得注意的是，有一个较大的偏置并不会使得神经元对它的输入像有大权重那样敏感。所以我们不用担心较大的偏置会使我们的网络学习训练数据中的噪声。同时，允许大规模的偏置使我们的网络在性能上更为灵活——特别是较大的偏置使得神经元更容易饱和，这通常是我们期望的。由于这些原因，我们通常不对偏置做正则化。

¹ 这个故事来自于物理学家 Richard Feynman 和历史学家 Charles Weiner 的一次访谈中。

² Gradient-Based Learning Applied to Document Recognition, Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998)