

neural-networks-and-deep-learning-zh

Xiaohu Zhu

Published
with GitBook



null

目錄

1. [Introduction](#)
2. [第一章 使用神经网络识别手写数字](#)
3. [第二章 反向传播算法如何工作的？](#)
4. [第三章 改进神经网络的学习方式](#)
5. [第五章 深度神经网络为何很难训练](#)
6. [第六章 深度学习](#)

null

神经网络与深度学习

神经网络和深度学习是一本免费的在线书。本书会教会你：

- 神经网络，一种美妙的受生物学启发的编程范式，可以让计算机从观测数据中进行学习
- 深度学习，一个强有力的用于神经网络学习的众多技术的集合

神经网络和深度学习目前给出了在图像识别、语音识别和自然语言处理领域中很多问题的最好解决方案。本书将会教你在神经网络和深度学习背后的众多核心概念。

想了解本书选择的观点的更多细节，请看[这里](#)。或者直接跳到[第一章](#) 开始你们的旅程。

译者的话：

本书是 Michael Nielsen 的 Neural Networks and Deep Learning 的中译本。目前已经完成第二章、第五章和第六章的内容。后续会进行剩下章节的翻译。如果想要合作翻译，提供意见或者建议，给出翻译的笔误，都可以直接通过 xhzhu.nju@gmail.com 联系到我。

null

First Chapter

GitBook allows you to organize your book into chapters, each chapter is stored in a separate file like this one.

null

null

在上一章，我们看到了神经网络如何使用梯度下降算法来学习他们自身的权重和偏差。但是，这里还留下了一个问题：我们并没有讨论如何计算代价函数的梯度。这是很大的缺失！在本章，我们会解释计算这些梯度的快速算法，也就是反向传播。

反向传播算法最初在 1970 年代被发现，但是这个算法的重要性直到 David Rumelhart、Geoffrey Hinton 和 Ronald Williams 的 [1986年的论文](#) 中才被真正认可。这篇论文描述了对一些神经网络反向传播要比传统的方法更快，这使得使用神经网络来解决之前无法完成的问题变得可行。现在，反向传播算法已经是神经网络学习的重要组成部分了。

本章在全书的范围内要比其他章节包含更多的数学内容。如果你不是对数学特别感兴趣，那么可以跳过本章，将反向传播当成一个黑盒，忽略其中的细节。那么为何要研究这些细节呢？

答案当然是理解。反向传播的核心是对代价函数 C 关于 w （或者 b ）的偏导数 $\partial C / \partial w$ 的计算表示。该表示告诉我们在权重和偏差发生改变时，代价函数变化的快慢。尽管表达式会有点复杂，不过里面也包含一种美感，就是每个元素其实是拥有一种自然的直觉上的解释。所以反向传播不仅仅是一种学习的快速算法。实际上它还告诉我们一些细节的关于权重和偏差的改变影响整个网络行为方面的洞察。因此，这也是学习反向传播细节的重要价值所在。

如上面所说，如果你想要粗览本章，或者直接跳到下一章，都是可以的。剩下的内容即使你是把反向传播看做黑盒也是可以掌握的。当然，后面章节中也会有部分内容涉及本章的结论，所以会常常给出本章的参考。不过对这些知识点，就算你对推导的细节不太清楚你还是应该要理解主要的结论的。

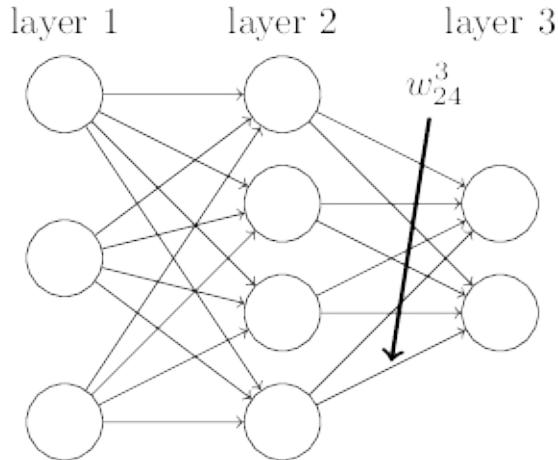
热身：神经网络中使用矩阵快速计算输出的观点

在讨论反向传播前，我们先熟悉一下基于矩阵的算法来计算网络的输出。事实上，我们在上一章的最后已经能够看到这个算法了，但是我在那里很快地略过了，所以现在让我们仔细讨论一下。特别地，这样能够用相似的场景帮助我们熟悉在反向传播中使用的矩阵表示。

我们首先给出网络中权重的清晰定义。我们使用 w_{jk}^l 表示从 $(l - 1)^{th}$ 层的 k^{th} 个神经元到 $(l)^{th}$ 层的 l^{th} 个神经元的链接上的权重。例如，下图给出了第二隐藏层的第四个神经元到第三隐藏层的第二个神经元的链接上的权重：

null

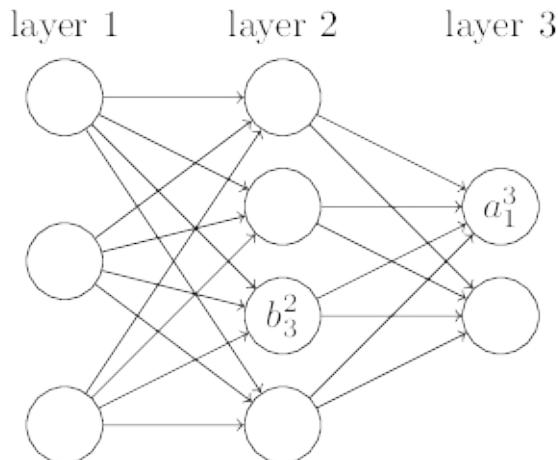
null



w_{jk}^l is the weight from the k^{th} neuron in the $(l - 1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

这样的表示粗看比较奇怪，需要花一点时间消化。但是，后面你会发现这样的表示会比较方便也很自然。奇怪的一点其实是下标 j 和 k 的顺序。你可能觉得反过来更加合理。但我接下来会告诉你为什么要这样做。

我们对网络偏差和激活值也会使用类似的表示。显式地，我们使用 b_j^l 表示在 l^{th} 层 j^{th} 个神经元的偏差，使用 a_j^l 表示 l^{th} 层 j^{th} 个神经元的激活值。下面的图清楚地解释了这样表示的含义：



有了这些表示， l^{th} 层的 j^{th} 个神经元的激活值 a_j^l 就和 l^{th} 层关联起来了（对比公式(4)和上一章的讨论）

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

其中求和是在 $(l - 1)^{\text{th}}$ 层的所有神经元上进行的。为了用矩阵的形式重写这个表达式，我们对每一层 l 都定义一个权重矩阵 w^l ，在 j^{th} 行第 k^{th} 列的元素是 w_{jk}^l 。类似的，对每一层 l ，定义一个偏差向量， b^l 。你已经猜到这些如何工作了——偏差向量的每个元素其实就是前面给出的 b_j^l ，每个元素对应于 l^{th} 层的每个神经元。最后，我们定义激活向量 a^l ，其元素是那些

null

null

激活值 a_j^l 。

最后我们需要引入向量化函数（如 σ ）来按照矩阵形式重写公式(23)。在上一章，我们其实已经碰到向量化了，其含义就是作用函数（如 σ ）到向量 v 中的每个元素。我们使用 $\sigma(v)$ 表示这种按元素进行的函数作用。所以， $\sigma(v)$ 的每个元素其实满足 $\sigma(v)_j = \sigma(v_j)$ 。给个例子，如果我们的作用函数是 $f(x) = x^2$ ，那么向量化的 f 的函数作用就起到下面的效果：

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

也就是说，向量化的 f 仅仅是对向量的每个元素进行了平方运算。

了解了这些表示，方程(23)就可以写成下面的这种美妙而简洁的向量形式了：

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

这个表达式给出了一种更加全局的思考每层的激活值和前一层的关联方式：我们仅仅用权重矩阵作用在激活值上，然后加上一个偏差向量，最后作用 σ 函数。

其实，这就是让我们使用之前的矩阵下标 w_{jk}^l 表示的初因。如果我们使用 j 来索引输入神经元， k 索引输出神经元，那么在方程(25)中我们需要将这里的矩阵换做其转置。这只是一个小小的困惑的改变，这会使得我们无法自然地讲出（思考）“应用权重矩阵到激活值上”这样的简单的表达。

这种全局的观点相比神经元层面的观点常常更加简明（没有更多的索引下标了！）其实可以看做是在保留清晰认识的前提下逃离下标困境的方法。在实践中，表达式同样很有用，因为大多数矩阵库提供了实现矩阵乘法、向量加法和向量化的快速方法。实际上，上一章的代码其实已经隐式使用了使用这种表达式来计算网络行为。

在使用方程(25)计算 a^l 时，我们计算了中间量 $z^l \equiv w^l a^{l-1} + b^l$ 。这个量其实是非常有用的：我们称 z^l 为 l 层的带权输入。在本章后面，我们会大量用到这个量。方程(25)有时候会写作 $a^l = \sigma(z^l)$ 。同样要指出的是 z_l 的每个元素是 $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$ ，其实 z_j^l 就是第 l 层第 j 个神经元的激活函数带权输入。

关于代价函数的两个假设

反向传播的目标是计算代价函数 C 分别关于 w 和 b 的偏导数 $\partial C / \partial w$ 和 $\partial C / \partial b$ 。为了让反向传播可行，我们需要做出关于代价函数的两个主要假设。在给出这两个假设之前，我们先看看具体的一个代价函数。我们会使用上一章使用的二次代价函数。按照上一节给出的表

null

null

示，二次代价函数有下列形式：

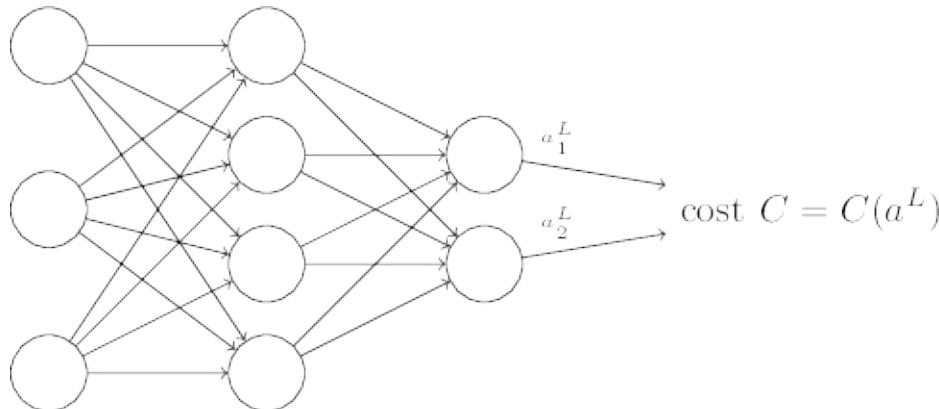
$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2, \quad (26)$$

其中 n 是训练样本的总数；求和是在所有的训练样本 x 上进行的； $y = y(x)$ 是对应的目标输出； L 表示网络的层数； $a^L = a^L(x)$ 是当输入是 x 时的网络输出的激活值向量。

好了，为了应用反向传播，我们需要对代价函数做出什么样的前提假设呢？第一个假设就是代价函数可以被写成一个在每个训练样本 x 上的代价函数 C_x 的均值 $C = \frac{1}{n} \sum_x C_x$ 。这是关于二次代价函数的例子，其中对每个独立的训练样本其代价是 $C_x = \frac{1}{2} \|y - a^L\|^2$ 。这个假设对书中提到的其他任何一个代价函数也都是必须满足的。

需要这个假设的原因是反向传播实际上是对一个独立的训练样本计算了 $\partial C_x / \partial w$ 和 $\partial C_x / \partial b$ 。然后我们通过在所有训练样本上进行平均化获得 $\partial C / \partial w$ 和 $\partial C / \partial b$ 。实际上，有了这个假设，我们会认为训练样本 x 已经被固定住了，丢掉了其下标，将代价函数 C_x 看做 C 。最终我们会把下标加上，现在为了简化表示其实没有这个必要。

第二个假设就是代价可以写成神经网络输出的函数：



例如，二次代价函数满足这个要求，因为对于一个单独的训练样本 x 其二次代价函数可以写作：

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

这是输出的激活值的函数。当然，这个代价函数同样还依赖于目标输出 y 。记住，输入的训练样本 x 是固定的，所以输出同样是一个固定的参数。所以说，并不是可以随意改变权重和偏差的，也就是说，这不是神经网络学习的对象。所以，将 C 看成仅有输出激活值 a^L 的函数才是合理的，而 y 仅仅是帮助定义函数的参数而已。

null

Hadamard 乘积

反向传播算法基于常规的线性代数运算——诸如向量加法，向量矩阵乘法等。但是有一个运算不大常见。特别地，假设 s 和 t 是两个同样维度的向量。那么我们使用 $s \odot t$ 来表示按元素的乘积。所以 $s \odot t$ 的元素就是 $(s \odot t)_j = s_j t_j$ 。给个例子，

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

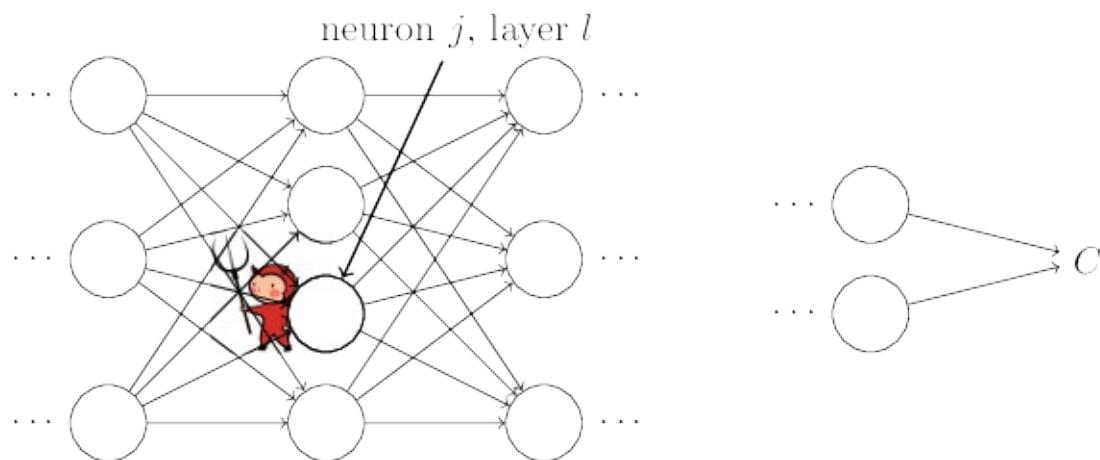
这种类型的按元素乘法有时候被称为 Hadamard 乘积或者 Schur 乘积。我们这里取前者。好的矩阵库通常会提供 Hadamard 乘积的快速实现，在实现反向传播的时候用起来很方便。

反向传播的四个基本方程

反向传播其实是对权重和偏差变化影响代价函数过程的理解。最终极的含义其实就是计算偏导数 $\partial C / \partial w_{jk}^l$ 和 $\partial C / \partial b_j^l$ 。但是为了计算这些值，我们首先引入一个中间量， δ_j^l ，这个我们称为在 l^{th} 层第 j^{th} 个神经元上的误差（error）。

反向传播将给出计算误差 δ_j^l 的流程，然后将其关联到计算 $\partial C / \partial w_{jk}^l$ 和 $\partial C / \partial b_j^l$ 上。

为了理解误差是如何定义的，假设在神经网络上有一个恶魔：



这个小精灵在 l 层的第 j^{th} 个神经元上。当输入进来时，精灵对神经元的操作进行搅局。他会增加很小的变化 Δz_j^l 在神经元的带权输入上，使得神经元输出由 $\sigma(z_j^l)$ 变成 $\sigma(z_j^l + \Delta z_j^l)$ 。
 $\frac{\partial C}{\partial z_j^l} \Delta z_j^l$ 。这个变化会向网络后面的层进行传播，最终导致整个代价函数产生 $\frac{\partial C}{\partial z_j^l} \Delta z_j^l$ 的改变。

现在，这个精灵变好了，试着帮助你来优化代价函数，他们试着找到可以让代价更小的 Δz_j^l

null

。假设 $\frac{\partial C}{\partial z_j^l}$ 有一个很大的值（或正或负）。那么这个精灵可以降低代价通过选择与 $\frac{\partial C}{\partial z_j^l}$ 相反符号的 Δz_j^l 。相反，如果 $\frac{\partial C}{\partial z_j^l}$ 接近 0，那么精灵并不能通过扰动带权输入 z_j^l 来改变太多代价函数。在小精灵看来，这时候神经元已经很接近最优了。

这里需要注意的是，只有在 Δz_j^l 很小的时候才能够满足。我们需要假设小精灵只能进行微小的调整。

所以这里有一种启发式的认识， $\frac{\partial C}{\partial z_j^l}$ 是神经元的误差的度量。

按照上面的描述，我们定义 l 层的第 j^{th} 个神经元上的误差 δ_j^l 为：

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

按照我们通常的惯例，我们使用 δ^l 表示关联于 l 层的误差向量。反向传播会提供给我们一种计算每层的 δ^l 的方法，然后将这些误差和最终我们需要的量 $\partial C / \partial w_{jk}^l$ 和 $\partial C / \partial b_j^l$ 联系起来。

你可能会想知道为何精灵在改变带权输入 z_j^l 。肯定想象精灵改变输出激活 a_j^l 更加自然，然后就使用 $\frac{\partial C}{\partial a_j^l}$ 作为度量误差的方法了。实际上，如果你这样做的话，其实和下面要讨论的差不同。但是看起来，前面的方法会让反向传播在代数运算上变得比较复杂。所以我们坚持使用 $\delta_j^l = \partial C / \partial z_j^l$ 作为误差的度量。

在分类问题中，误差有时候会用作分类的错误率。如果神经网络正确分类了 96.0% 的数字，那么其误差是 4.0%。很明显，这和我们上面提及的误差的差别非常大了。在实际应用中，区分这两种含义是非常容易的。

解决方案：反向传播基于四个基本方程。这些方程给我们一种计算误差和代价函数梯度的方法。我列出这四个方程。但是需要注意：你不需要一下子能够同时理解这些公式。因为过于庞大的期望可能会导致失望。实际上，反向传播方程内容很多，完全理解这些需要花费充分的时间和耐心，需要一步一步地深入理解。而好的消息是，这样的付出回报巨大。所以本节对这些内容的讨论仅仅是一个帮助你正确掌握这些公式的起步。

下面简要介绍我们的探讨这些公式的计划：首先给出这些公式的简短证明以解释他们的正确性；然后以伪代码的方式给出这些公式的算法形式，并展示这些伪代码如何转化成真实的可执行的 python 代码；在本章的最后，我们会发展处一个关于反向传播公式含义的直觉图景，以及人们如何能够从零开始发现这个规律。按照此法，我们会不断地提及这四个基本方程，随着你对这些方程理解的加深，他们会看起来更加舒服，甚至是美妙和自然的。

输出层误差的方程， δ^L ：每个元素定义如下：

null

null

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

这是一个非常自然的表达式。右式第一个项 $\partial C / \partial a_j^L$ 表示代价随着 j^{th} 输出激活值的变化而变化的速度。假如 C 不太依赖一个特定的输出神经元 j , 那么 δ_j^L 就会很小, 这也是我们想要的效果。右式第二项 $\sigma'(z_j^L)$ 刻画了在 z_j^L 处激活函数 σ 变化的速度。

注意到在 BP1 中的每个部分都是很好计算的。特别地, 我们在前向传播计算网络行为时已经计算过 z_j^L , 这仅仅需要一点点额外工作就可以计算 $\sigma'(z_j^L)$ 。当然 $\partial C / \partial a_j^L$ 依赖于代价函数的形式。然而, 给定了代价函数, 计算 $\partial C / \partial a_j^L$ 就没有什么大问题了。例如, 如果我们使用二次函数, 那么 $C = \frac{1}{2} \sum_j (y_j - a_j)^2$, 所以 $\partial C / \partial a_j^L = (a_j - y_j)$, 这其实很容易计算。

方程(BP1)对 δ^L 来说是个按部分构成的表达式。这是一个非常好的表达式, 但不是我们期望的用矩阵表示的形式。但是, 重写方程其实很简单,

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

这里 $\nabla_a C$ 被定义成一个向量, 其元素是偏导数 $\partial C / \partial a_j^L$ 。你可以将其看成 C 关于输出激活值的改变速度。方程(BP1)和方程(BP1a)的等价也是显而易见的, 所以现在开始, 我们会交替地使用这两个方程。举个例子, 在二次代价函数时, 我们有 $\nabla_a C = (a^L - y)$, 所以(BP1)的整个矩阵形式就变成

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

如你所见, 这个方程中的每个项都有一个很好的向量形式, 所以也可以很方便地使用像 Numpy 这样的矩阵库进行计算了。

使用下一层的误差 δ^{l+1} 来表示当前层的误差 δ_l : 特别地,

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

其中 $(w^{l+1})^T$ 是 $(l+1)^{th}$ 权重矩阵 w^{l+1} 的转置。这其实可以很直觉地看做是后在 l^{th} 层的输出的误差的反向传播, 给出了某种关于误差的度量方式。然后, 我们进行 Hadamard 乘积运算 $\odot \sigma'(z^l)$ 。这会让误差通过 l 层的激活函数反向传递回来并给出在第 l 层的带权输入的误差 δ 。

通过组合(BP1)和(BP2), 我们可以计算任何层的误差了。首先使用(BP1)计算 δ^l , 然后应用方程(BP2)来计算 δ^{l-1} , 然后不断作用(BP2), 一步一步地反向传播完整个网络。

代价函数关于网络中任意偏差的改变率 : 就是

null

null

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

这其实是，误差 δ_j^l 和偏导数值 $\partial C / \partial b_j^l$ 完全一致。这是很好的性质，因为(BP1)和(BP2)已经告诉我们如何计算 δ_j^l 。所以就可以将(BP3)简记为

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

其中 δ 和偏差 b 都是针对同一个神经元。

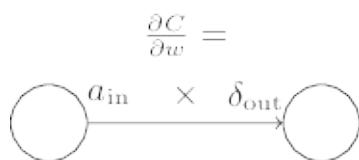
代价函数关于任何一个权重的改变率：特别地，

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

这告诉我们如何计算偏导数 $\partial C / \partial w_{jk}^l$ ，其中 δ^l a^{l-1} 这些量我们都已经知道如何计算了。方程也可以写成下面少下标的表示：

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

其中 a_{in} 是输出给 w 产生的神经元的输入和 δ_{out} 是来自 w 的神经元输出的误差。放大看看权重 w ，还有两个由这个链接相连的神经元，我们给出一幅图如下：



方程(32)的一个结论就是当激活值很小，梯度 $\partial C / \partial w$ 也会变得很小。这样，我们就说权重学习缓慢，表示在梯度下降的时候，这个权重不会改变太多。换言之，(BP4)的后果就是来自很低的激活值神经元的权重学习会非常缓慢。

这四个公式同样还有很多观察。让我们看看(BP1)中的项 $\sigma'(z_k^l)$ 。回忆一下上一章的 sigmoid 函数图像，当函数值接近 0 或者 1 的时候图像非常平。这就使得在这些位置的导数接近于 0。所以如果输出神经元处于或者低激活值或者高激活值时，最终层的权重学习缓慢。这样的情形，我们常常称输出神经元已经饱和了，并且，权重学习也会终止（或者学习非常缓慢）。类似的结果对于输出神经元的偏差也是成立的。

null

null

针对前面的层，我们也有类似的观点。特别地，注意在(BP2)中的项 $\sigma'(z^l)$ 。这表示 δ_j^l 很可能变小如果神经元已经接近饱和。这就导致任何输入进一个饱和的神经元的权重学习缓慢。

如果 $(w^{l+1})^T \delta^{l+1}$ 拥有足够的量能够补偿 $\sigma'(z_k^l)$ 的话，这里的推导就不能成立了。
但是我们上面是常见的情形。

总结一下，我们已经学习到权重学习缓慢如果输入神经元激活值很低，或者输出神经元已经饱和了（过高或者过低的激活值）。

这些观测其实也是不非常令人惊奇的。不过，他们帮助我们完善了关于神经网络学习的背后的思想模型。而且，我们可以将这种推断方式进行推广。四个基本方程也其实对任何的激活函数都是成立的（证明中也可以看到，其实推断本身不依赖于任何具体的代价函数）所以，我们可以使用这些方程来设计有特定属性的激活函数。我们这里给个例子，假设我们准备选择一个（non-sigmoid）的激活函数 σ 使得 σ' 总是正数。这会防止在原始的 sigmoid 神经元饱和时学习速度的下降的情况出现。在本书的后面，我们会见到这种类型的对激活函数的改变。时时回顾这四个方程可以帮助解释为何需要有这些尝试，以及尝试带来的影响。

Summary: the equations of backpropagation

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

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

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

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

问题

另一种反向传播方程的表示方式：我已经给出了使用了 Hadamard 乘积的反向传播的公式。如果你对这种特殊的乘积不熟悉，可能会有一些困惑。下面还有一种表示方式，那就是基于传统的矩阵乘法，某些读者可能会觉得很有启发。(1) 证明 (BP1) 可以写成

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

其中 $\Sigma'(z^L)$ 是一个方阵，其对角线的元素是 $\sigma'(z_j^L)$ ，其他的元素均是 0。注意，这个矩阵通过一般的矩阵乘法作用在 $\nabla_a C$ 上。(2) 证明(BP2) 可以写成

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

null

null

(3) 结合(1)和(2) 证明

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

对那些习惯于这种形式的矩阵乘法的读者, (BP1) (BP2) 应该更加容易理解。而我们坚持使用 Hadamard 乘积的原因在于其更快的数值实现。

四个基本方程的证明 (可选)

我们现在证明这四个方程。所有这些都是多元微积分的链式法则的推论。如果你熟悉链式法则, 那么我鼓励你在读之前自己证明一番。

练习

- 证明方程 (BP3) 和 (BP4)

这样我们就完成了反向传播四个基本公式的证明。证明本身看起来复杂。但是实际上就是细心地应用链式法则。我们可以将反向传播看成是一种系统性地应用多元微积分中的链式法则来计算代价函数的梯度的方式。这些就是反向传播理论上的内容——剩下的是实现细节。

反向传播算法

反向传播方程给出了一种计算代价函数梯度的方法。让我们显式地用算法描述出来：

1. 输入 x : 为输入层设置对应的激活值 a^l 。
2. 前向传播 : 对每个 $l = 2, 3, \dots, L$ 计算相应的 $z^l = w^l a^{l-1} + b^l$ 和 $a^l = \sigma(z^l)$
3. 输出层误差 δ^L : 计算向量 $\delta^L = \nabla_a C \odot \sigma'(z^L)$
4. 反向误差传播 : 对每个 $l = L-1, L-2, \dots, 2$, 计算 $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
$$\frac{C}{w_{jk}^l} = a_k^{l-1} \delta_j^l \quad \text{和} \quad \frac{\partial C}{\partial b_j^l} = \delta_j^l$$
5. 输出 : 代价函数的梯度由

看看这个算法, 你可以看到为何它被称作反向传播。我们从最后一层开始向后计算误差向量 δ^l 。这看起来有点奇怪, 为何要从后面开始。但是如果你认真思考反向传播的证明, 这种反向移动其实是代价函数是网络输出的函数的后果。为了理解代价随前面层的权重和偏差变化的规律, 我们需要重复作用链式法则, 反向地获得需要的表达式。

练习

- 使用单个修正的神经元的反向传播 假设我们改变一个前向传播网络中的单个神经元, 使

null

null

得那个神经元的输出是 $f(\sum_j w_j x_j + b)$ ，其中 f 是和 sigmoid 函数不同的某一函数。

我们如何调整反向传播算法？

- 线性神经元上的反向传播 假设我们将非线性神经元替换为 $\sigma(z) = z$ 。重写反向传播算法。

正如我们上面所讲的，反向传播算法对一个训练样本计算代价函数的梯度， $C = C_x$ 。在实践中，通常将反向传播算法和诸如随机梯度下降这样的学习算法进行组合使用，我们会对许多训练样本计算对应的梯度。特别地，给定一个大小为 m 的 minibatch，下面的算法应用一步梯度下降学习在这个 minibatch 上：

1. 输入训练样本的集合
2. 对每个训练样本 x ：设置对应的输入激活 $a^{x,1}$ ，并执行下面的步骤：
 - 前向传播：对每个 $l = 2, 3, \dots, L$ 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 $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$
 - 反向传播误差：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 $l = L-1, L-2, \dots, 2$ 根据 $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$ 和 $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$ 更新权重和偏差

当然，在实践中实现随机梯度下降，我们还需要一个产生训练样本 minibatch 的循环，还有就是训练次数的循环。这里我们先省略了。

代码

理解了抽象的反向传播的理论知识，我们现在就可以学习上一章中使用的实现反向传播的代码了。回想上一章的代码，需要研究的是在 `Network` 类中的 `update_mini_batch` 和 `backprop` 方法。这些方法的代码其实是我们上面的算法描述的直接翻版。特别地，`update_mini_batch` 方法通过计算当前 `mini_batch` 中的训练样本对 `Network` 的权重和偏差进行了更新：

```
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)]
```

null

null

主要工作其实是在 `delta_nabla_b, delta_nabla_w = self.backprop(x, y)` 这里完成的，调用了 `backprop` 方法计算出了偏导数， $\partial C_x / \partial b_j^l$ 和 $\partial C_x / \partial w_{jk}^l$ 。反向传播方法跟上一节的算法基本一致。这里只有个小小的差异——我们使用一个略微不同的方式来索引神经网络的层。这个改变其实是为了 Python 的特性——负值索引的使用能够让我们从列表的最后往前遍历，如 `l[-3]` 其实是列表中的倒数第三个元素。下面 `backprop` 的代码，使用了一些用来计算 σ 、导数 σ' 及代价函数的导数帮助函数。所以理解了这些，我们就完全可以掌握所有的代码了。如果某些东西让你困惑，你可能需要参考[代码的原始描述](#)

```
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))
```

null

null

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

问题

- 在 minibatch 上的反向传播的全矩阵方法 我们对于随机梯度下降的实现是对一个 minibatch 中的训练样本进行遍历。所以也可以更改反向传播算法使得它同时对一个 minibatch 中的所有样本进行梯度计算。这个想法其实就是我们可以用一个矩阵 $X = [x_1, x_2, \dots, x_m]$, 其中每列就是在 minibatch 中的向量, 而不是单个的输入向量, x 。我们通过乘权重矩阵, 加上对应的偏差进行前向传播, 在所有地方应用 sigmoid 函数。然后按照类似的过程进行反向传播。请显式写出这种方法下的伪代码。更改 network.py 来实现这个方案。这样做好处其实利用到了现代的线性代数库。所以, 这会比在 minibatch 上进行遍历要运行得更快 (在我的笔记本电脑上, 在 MNIST 分类问题上, 我相较于上一章的实现获得了 2 倍的速度提升)。在实际应用中, 所有靠谱的反向传播的库都是用了类似的基于矩阵或者变体的方式来实现的。

在哪种层面上, 反向传播是快速的算法?

为了回答这个问题, 首先考虑另一个计算梯度的方法。就当我们回到上世界50、60年代的神经网络研究。假设你是世界上首个考虑使用梯度下降方法学习的那位!为了让自己的想法可行, 就必须找出计算代价函数梯度的方法。想想自己学到的微积分, 决定试试看链式法则来计算梯度。但玩了一会后, 就发现代数式看起来非常复杂, 然后就退缩了。所以就试着找另外的方式。你决定仅仅把代价看做权重 C 的函数。你给这些权重 w_1, w_2, \dots 进行编号, 期望计算关于某个权值 w_j 关于 C 的导数。而一种近似的方法就是下面这种:

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

其中 $\epsilon > 0$ 是一个很小的正数, 而 e_j 是在第j个方向上的单位向量。换句话说, 我们可以通过计算 w_j 的两个接近相同的点的值来估计 $\partial C / \partial w_j$, 然后应用公式 (46)。同样方法也可以用来计算 $\partial C / \partial b$ 。

这个观点看起来非常有希望。概念上易懂, 容易实现, 使用几行代码就可以搞定。看起来, 这样的方法要比使用链式法则还要有效。

然后, 遗憾的是, 当你实现了之后, 运行起来这样的方法非常缓慢。为了理解原因, 假设我们有 1,000,000 权重。对每个不同的权重 w_j 我们需要计算 $C(w + \epsilon * e_j)$ 来计算 $\partial C / \partial w_j$ 。这意味着为了计算梯度, 我们需要计算代价函数 1,000,000 次, 需要 1,000,000 前向传播 (对每个样本)。我们同样需要计算 $C(w)$, 总共是 1,000,001 次。

null

null

反向传播聪明的地方就是它确保我们可以同时计算所有的偏导数 $\partial C / \partial w_j$ 使用一次前向传播，加上一次后向传播。粗略地说，后向传播的计算代价和前向的一样。*

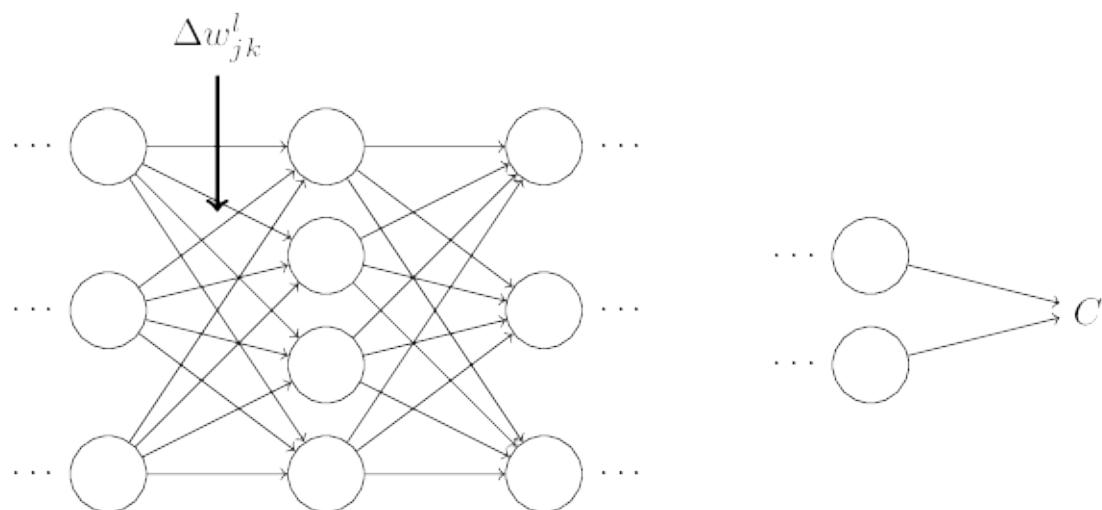
这个说法是合理的，但需要额外的说明来澄清这一事实。在前向传播过程中主要的计算代价消耗在权重矩阵的乘法上，而反向传播则是计算权重矩阵的转置矩阵。这些操作显然有着类似的计算代价。

所以最终的计算代价大概是两倍的前向传播计算大家。比起直接计算导数，显然 反向传播 有着更大的优势。所以即使 反向传播 看起来要比 (46) 更加复杂，但实际上要更快。

这个加速在1986年首次被众人接受，并直接导致神经网络可以处理的问题的扩展。这也导致了大量的研究者涌向了神经网络方向。当然，反向传播 并不是万能钥匙。在 1980 年代后期，人们尝试挑战极限，尤其是尝试使用反向传播来训练深度神经网络。本书后面，我们将看到现代计算机和一些聪明的新想法已经让 反向传播 成功地训练这样的深度神经网络。

反向传播：大视野

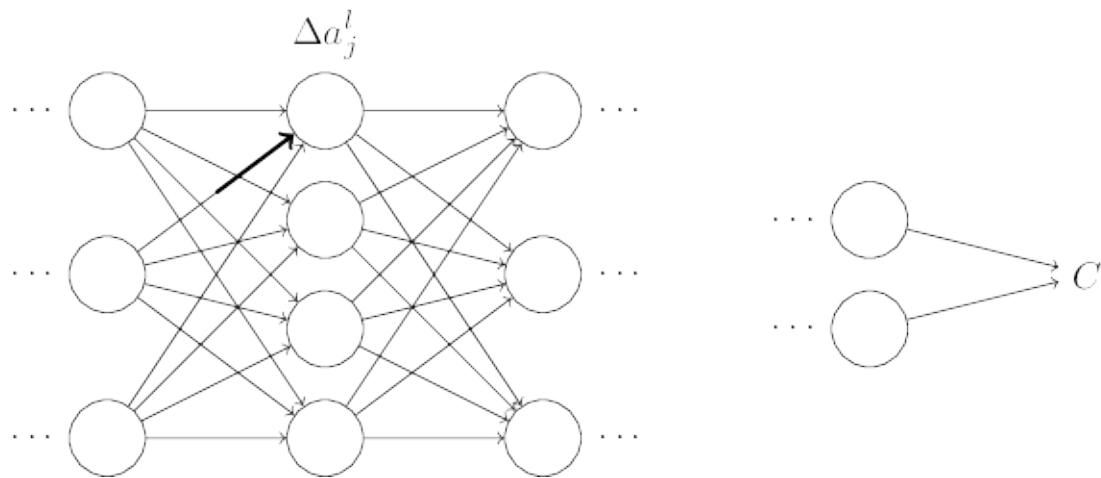
正如我所讲解的，反向传播 提出了两个神秘的问题。首先，这个算法真正在干什么？我们已经感受到从输出处的错误被反向传回的图景。但是我们能够更深入一些，构造出一种更加深刻的直觉来解释所有这些矩阵和向量乘法么？第二神秘点就是，某人为什么能发现这个 反向传播？跟着一个算法跑一遍甚至能够理解证明算法可以运行这是一回事。这并不真的意味着你理解了这个问题到一定程度，能够发现这个算法。是否有一个推理的思路可以指引我们发现 反向传播 算法？本节，我们来探讨一下这两个谜题。为了提升我们关于算法究竟做了什么的直觉，假设我们已经对 w_{jk}^l 做一点小小的变动 Δw_{jk}^l ：



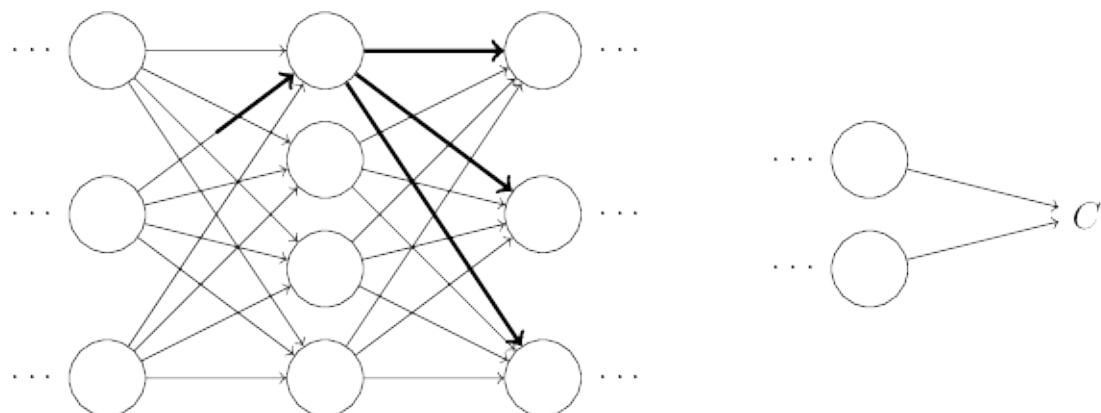
这个改变会导致在输出激活值上的相应改变：

null

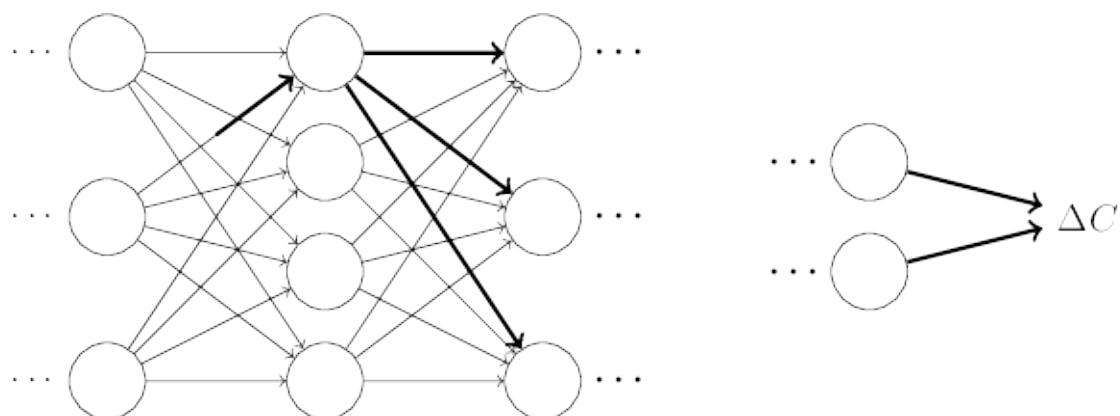
null



然后，会产生对下一层激活值的改变：



接着，这些改变都将影响到一个个下一层，到达输出层，最终影响代价函数：



所以代价函数 ΔC 改变和 Δw_{jk}^l 就按照下面的公式关联起来了：

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

这给出了一种可能的计算 $\frac{\partial C}{\partial w_{jk}^l}$ 的方法其实是细致地追踪一个 w_{jk}^l 的微小变化如何导致 C 中的变化值。如果我们可以做到这点，能够精确地使用易于计算的量来表达每种关系，那么我

null

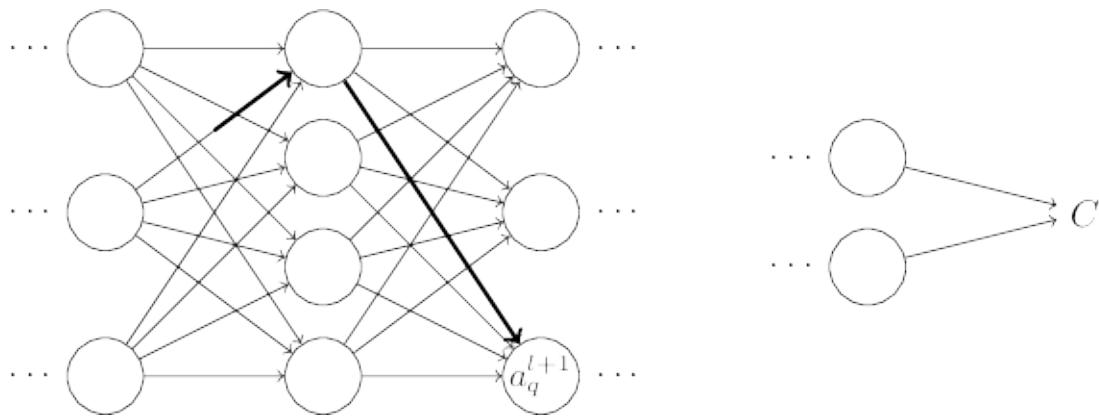
null

们就能够计算 $\frac{\partial C}{\partial w_{jk}^l}$ 了。

我们尝试一下这个方法。 Δw_{jk}^l 导致了在 l^{th} 层 j^{th} 神经元的激活值的变化 Δa_j^l 。这个变化由下面的公式给出：

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

Δa_j^l 的变化将会导致下一层的所有激活值的变化。我们聚焦到其中一个激活值上看看影响的情况，不妨设 a_q^{l+1} ，



实际上，这会导致下面的变化：

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

将其代入方程(48)，我们得到：

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

当然，这个变化又会去下一层的激活值。实际上，我们可以想象出一条从 w_{jk}^l 到 C 的路径，然后每个激活值的变化会导致下一层的激活值的变化，最终是输出层的代价的变化。假设激活值的序列如下 $a_j^l, a_q^{l+1}, \dots, a_n^{L-1}, a_m^L$ ，那么结果的表达式就是

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

null

null

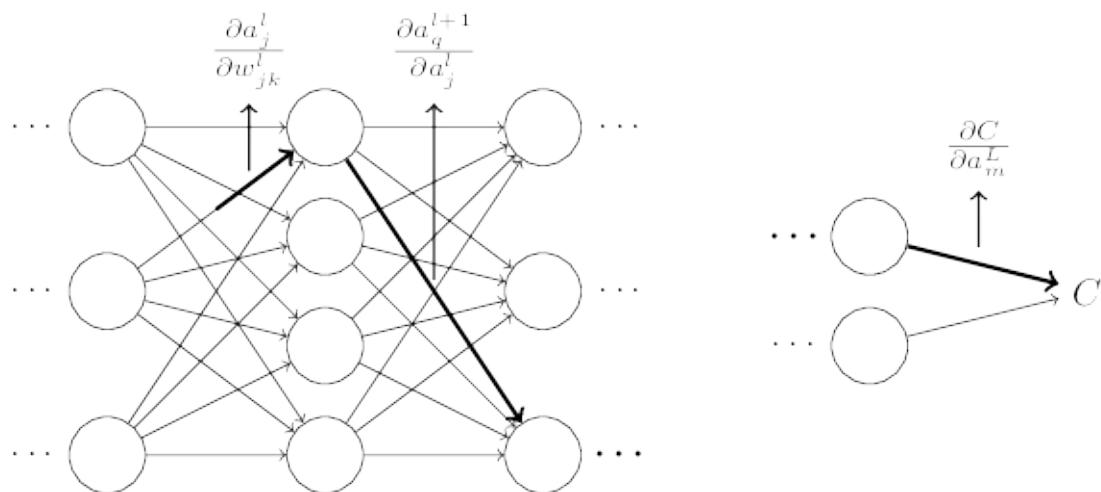
我们已经对每个经过的神经元设置了一个 $\partial a / \partial a$ 这种形式的项，还有输出层的 $\partial C / \partial a_m^L$ 。这表示除了 C 的改变由于网络中这条路径上激活值的变化。当然，整个网络中存在很多 w_{jk}^l 可以传播而影响代价函数的路径，这里我们就看其中一条。为了计算 C 的全部改变，我们就需要对所有可能的路径进行求和，即，

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

这里我们对路径中所有可能的中间神经元选择进行求和。对比 (47) 我们有

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

现在公式(53)看起来相当复杂。但是，这里其实有一个相当好的直觉上的解释。我们用这个公式计算 C 关于网络中一个权重的变化率。而这个公式告诉我们的只是：两个神经元之间的连接其实是关联与一个变化率因子，这仅仅是一个神经元的激活值相对于其他神经元的激活值的偏导数。从第一个权重到第一个神经元的变化率因子是 $\partial a_j^l / \partial w_{jk}^l$ 。路径的变化率因子其实就是这条路径上的众多因子的乘积。而整个的变化率 $\partial C / \partial w_{jk}^l$ 就是对于所有可能的从初始权重到最终输出的代价函数的路径的变化率因子的和。针对某一个路径，这个过程解释如下，



我们现在所给出的东西其实是一种启发式的观点，一种思考权重变化对网络行为影响的方式。让我们给出关于这个观点应用的一些流程建议。首先，你可以推导出公式(53)中所有单独的偏导数显式表达式。只是一些微积分的运算。完成这些后，你可以弄明白如何用矩阵运算写出对所有可能的情况的求和。这项工作会比较乏味，需要一些耐心，但不用太多的洞察。完成这些后，就可以尽可能地简化了，最后你发现，自己其实就是在做反向传播！所以你可以将反向传播想象成一种计算所有可能的路径变化率的求和的方式。或者，换句话说，反向传播就是一种巧妙地追踪权重（和偏差）微小变化的传播，抵达输出层影响代价函数的技术。

null

null

现在我不会继续深入下去。因为这项工作比较无聊。如果你想挑战一下，可以尝试与喜爱。即使你不去尝试，我也希望这种思维方式可以让你能够更好地理解反向传播。

那其他的一些神秘的特性呢——反向传播如何在一开始被发现的？实际上，如果你跟随我刚刚给出的观点，你其实是可以发现反向传播的一种证明的。不幸的是，证明会比本章前面介绍的证明更长和更加的复杂。那么，前面那个简短（却更加神秘）的证明如何被发现的？当你写出来所有关于长证明的细节后，你会发现其实里面包含了一些明显的可以进行改进的地方。然后你进行一些简化，得到稍微简短的证明，写下来。然后又能发现一些更加明显的简化。进过几次迭代证明改进后，你会发现最终的简单却看起来奇特的证明，因为你移除了很多构造的细节了！老实告诉你，其实最早的证明的出现也不是太过神秘的事情。因为那只是很多对简化证明的艰辛工作的积累。

null

null

当一个高尔夫球员刚开始学习打高尔夫时，他们通常会在挥杆的练习上花费大多数时间。慢慢地他们才会在基本的挥杆上通过变化发展其他的击球方式，学习低飞球、左曲球和右曲球。类似的，我们现在仍然聚焦在反向传播算法的理解上。这就是我们的“基本挥杆”——神经网络中大部分工作学习和研究的基础。本章，我会解释若干技术能够用来提升我们关于反向传播的初级的实现，最终改进网络学习的方式。

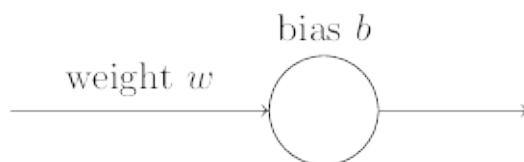
本章涉及的技术包括：更好的代价函数的选择——[交叉熵](#) 代价函数；四中规范化方法（L1 和 L2 规范化，dropout 和训练数据的人工扩展），这会让我们的网络在训练集之外的数据上更好地泛化；更好的[权重初始化方法](#)；还有[帮助选择好的超参数的启发式想法](#)。同样我也会再给出一些简要的[其他技术介绍](#)。这些讨论之间的独立性比较大，所有你们可以随自己的意愿挑着看。另外我还会在代码中实现这些技术，使用他们来提高在第一章中的分类问题上的性能。

当然，我们仅仅覆盖了大量已经在神经网络中研究发展出的技术的一点点内容。此处我们学习深度学习的观点是想要在一些已有的技术上入门的最佳策略其实是深入研究一小部分最重要那些的技术点。掌握了这些关键技术不仅仅对这些技术本身的理解很有用，而且会深化你对使用神经网络时会遇到哪些问题的理解。这会让你们做好在需要时快速掌握其他技术的充分准备。

交叉熵代价函数

我们大多数人觉得错了就很不爽。在开始学习弹奏钢琴不久后，我在一个听众前做了处女秀。我很紧张，开始时将八度音阶的曲段演奏得很低。我很困惑，因为不能继续演奏下去了，直到有个人指出了其中的错误。当时，我非常尴尬。不过，尽管不开心，我们却能够因为明显的犯错快速地学习到正确的东西。你应该相信下次我再演奏肯定会是正确的！相反，在我们的错误不是很好的定义的时候，学习的过程会变得更加缓慢。

理想地，我们希望和期待神经网络可以从错误中快速地学习。在实践中，这种情况经常出现么？为了回答这个问题，让我们看看一个小例子。这个例子包含一个只有一个输入的神经元：



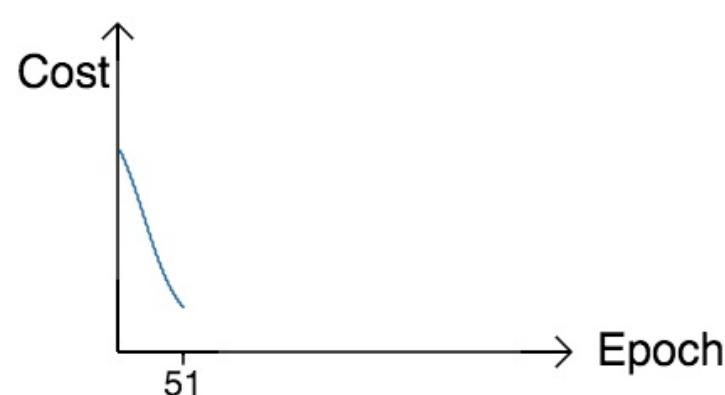
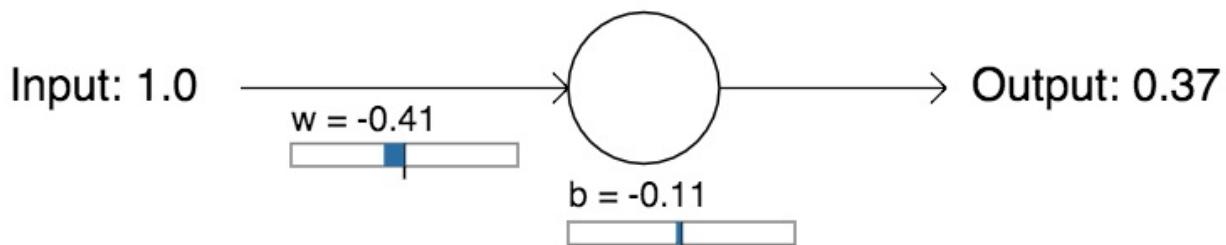
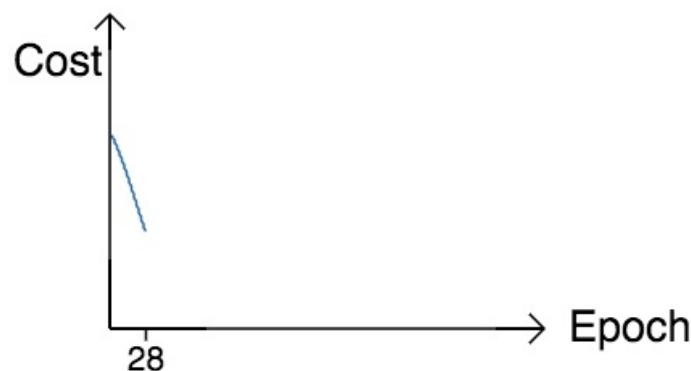
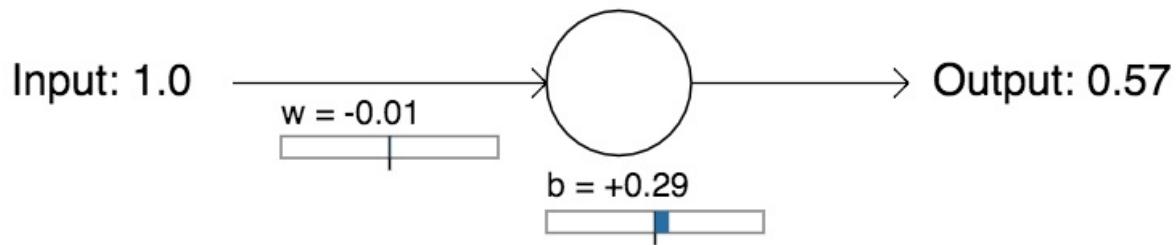
我们会训练这个神经元来做一件非常简单的事：让输入 1 转化为 0。当然，这很简单了，手工找到合适的权重和偏差就可以了，不需要什么学习算法。然而，看起来使用梯度下降的方式来学习权重和偏差是很有启发的。所以，我们来看看神经元如何学习。

为了让事情确定化，我会首先将权重和偏差初始化为 0.6 和 0.9。这些就是一般的开始学习的选择，并没有任何刻意的想法。一开始的神经元的输出是 0.82，所以这离我们的目标输出 0.0 还差得很远。点击右下角的“运行”按钮来看看神经元如何学习到让输出接近 0.0 的。注意

null

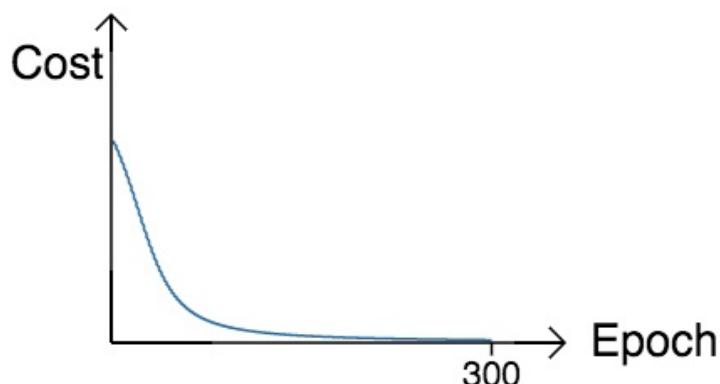
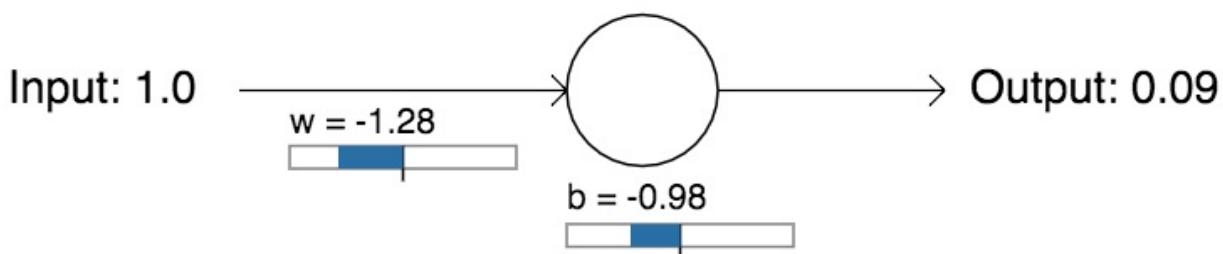
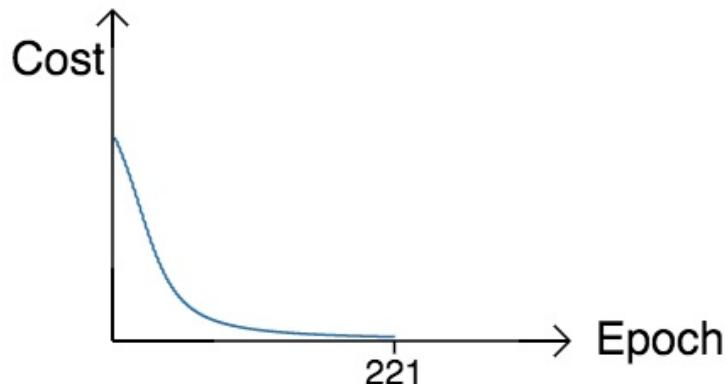
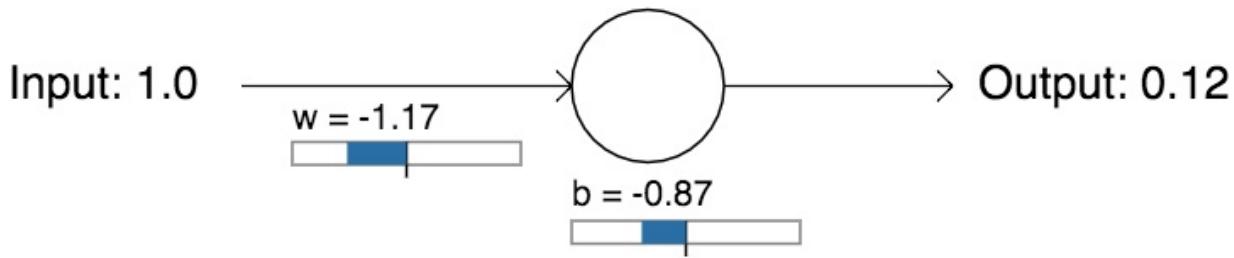
null

到，这并不是一个已经录好的动画，你的浏览器实际上是正在进行梯度的计算，然后使用梯度更新来对权重和偏差进行更新，并且展示结果。设置学习率 $\eta = 0.15$ 进行学习一方面足够慢的让我们跟随学习的过程，另一方面也保证了学习的时间不会太久，几秒钟应该就足够了。代价函数是我们前面用到的二次函数， C 。这里我也会给出准确的形式，所以不需要翻到前面查看定义了。注意，你可以通过点击“Run”按钮执行训练若干次。



null

null



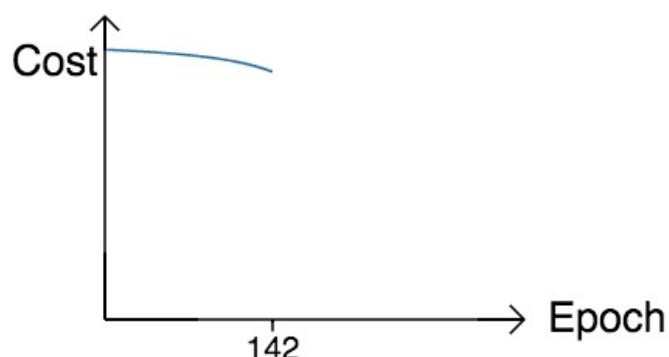
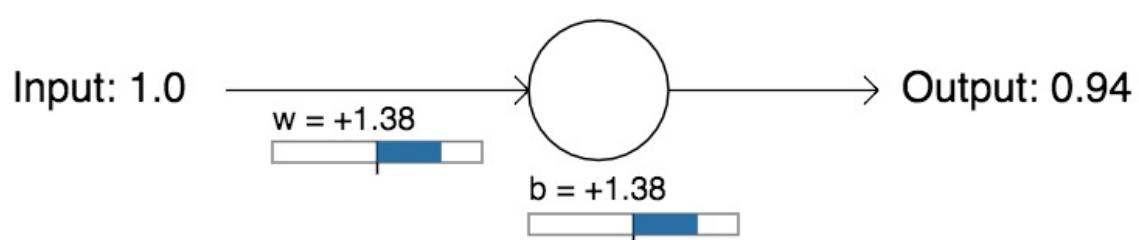
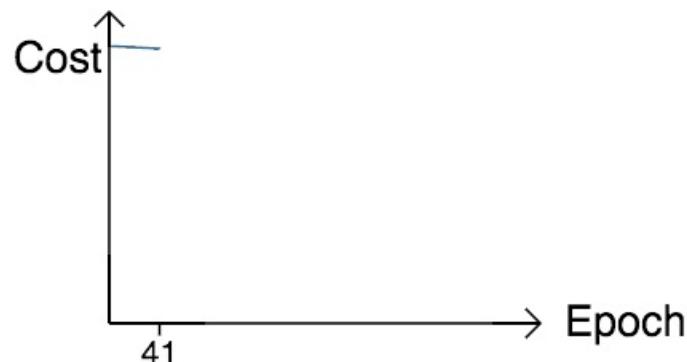
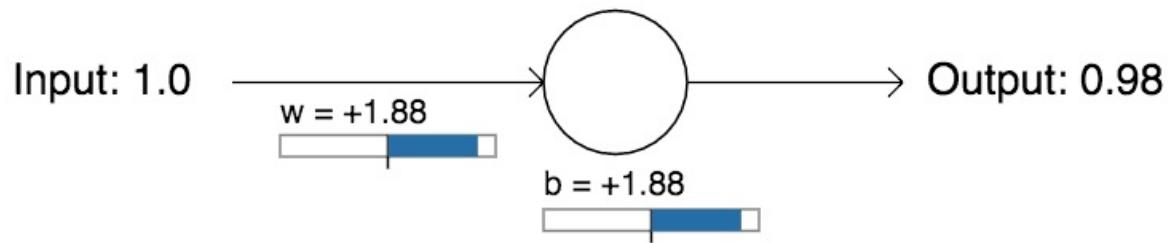
Run

我们这里是静态的例子，在原书中，使用的动态示例，所以为了更好的效果，请参考原书的[此处动态示例](#)。

正如你所见，神经元快速地学到了使得代价函数下降的权重和偏差，给出了最终的输出为 0.09。这虽然不是我们的目标输出 0.0，但是已经挺好了。假设我们现在将初始权重和偏差都设置为 2.0。此时初始输出为 0.98，这是和目标值的差距相当大的。现在看看神经元学习的过程。点击“Run”按钮：

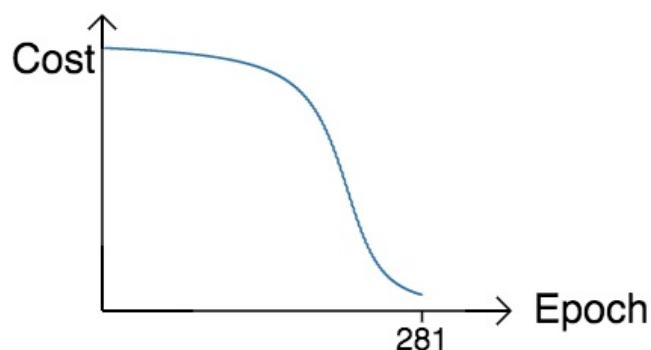
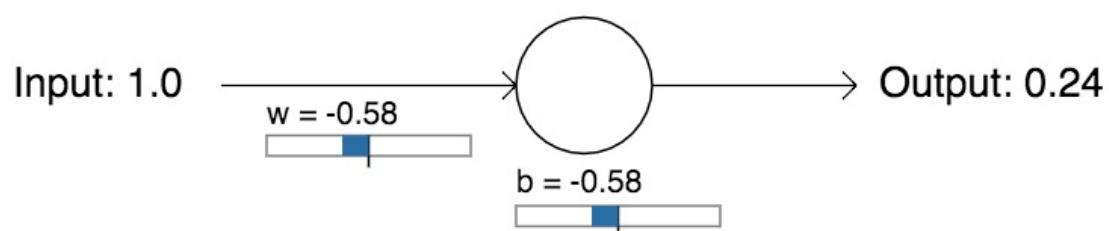
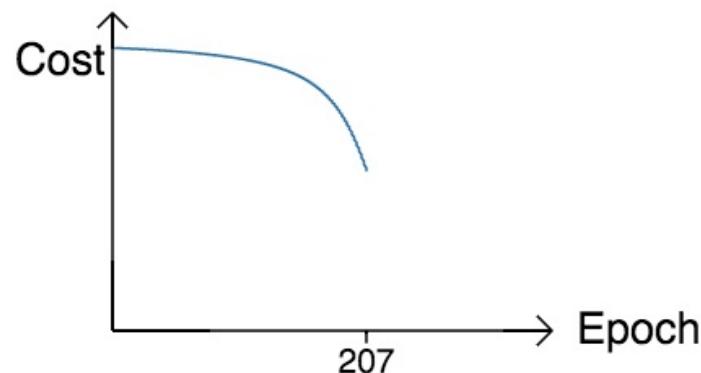
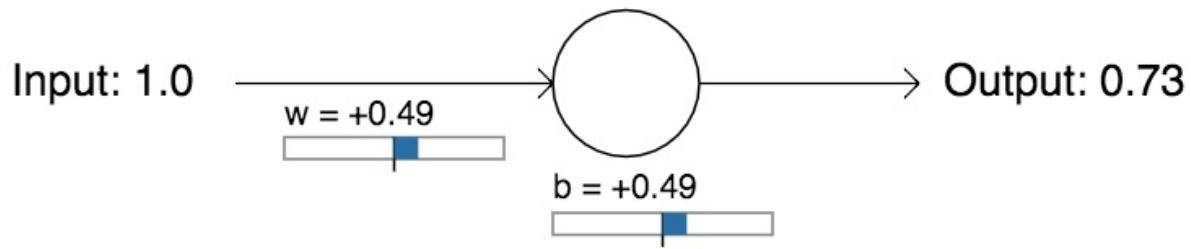
null

null



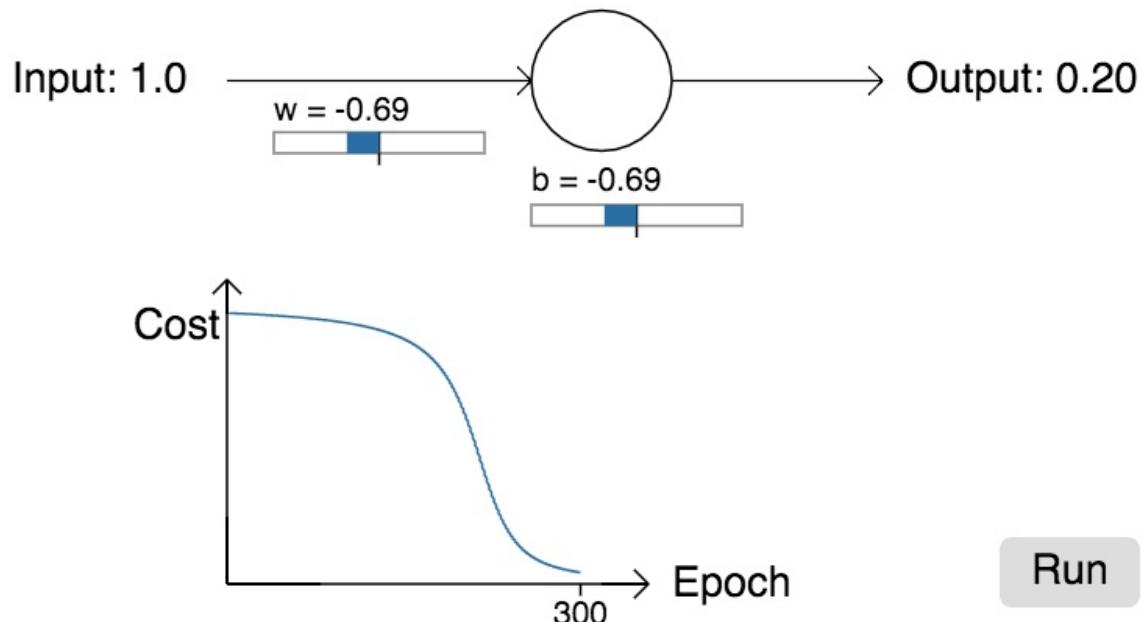
null

null



null

null



虽然这个例子使用的了同样的学习率 ($\eta = 0.15$)，我们可以看到刚开始的学习速度是比较缓慢的。对前 150 左右的学习次数，权重和偏差并没有发生太大的变化。随后学习速度加快，与上一个例子中类似了，神经网络的输出也迅速接近 0.0。

强烈建议参考[原书的此处动态示例](#)感受学习过程的差异。

这种行为看起来和人类学习行为差异很大。正如我在此节开头所说，我们通常是在犯错比较明显的时候学习的速度最快。但是我们已经看到了人工神经元在其犯错较大的情况下其实学习很有难度。而且，这种现象不仅仅是在这个小例子中出现，也会再更加一般的神经网络中出现。为何学习如此缓慢？我们能够找到缓解这种情况的方法么？

为了理解这个问题的源头，想想神经元是按照偏导数 ($\partial C / \partial w$ 和 $\partial C / \partial b$) 和学习率 (η) 的乘积来改变权重和偏差的。所以，我们在说“学习缓慢”时，实际上就是说这些偏导数很小。理解他们为何这么小就是我们面临的挑战。为了理解这些，让我们计算偏导数看看。我们一直在用的是二次代价函数，定义如下

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

其中 a 是神经元的输出，其中训练输入为 $x = 1$ ， $y = 0$ 则是目标输出。显式地使用权重和偏差来表达这个，我们有 $a = \sigma(z)$ ，其中 $z = wx + b$ 。使用链式法则来求偏导数就有：

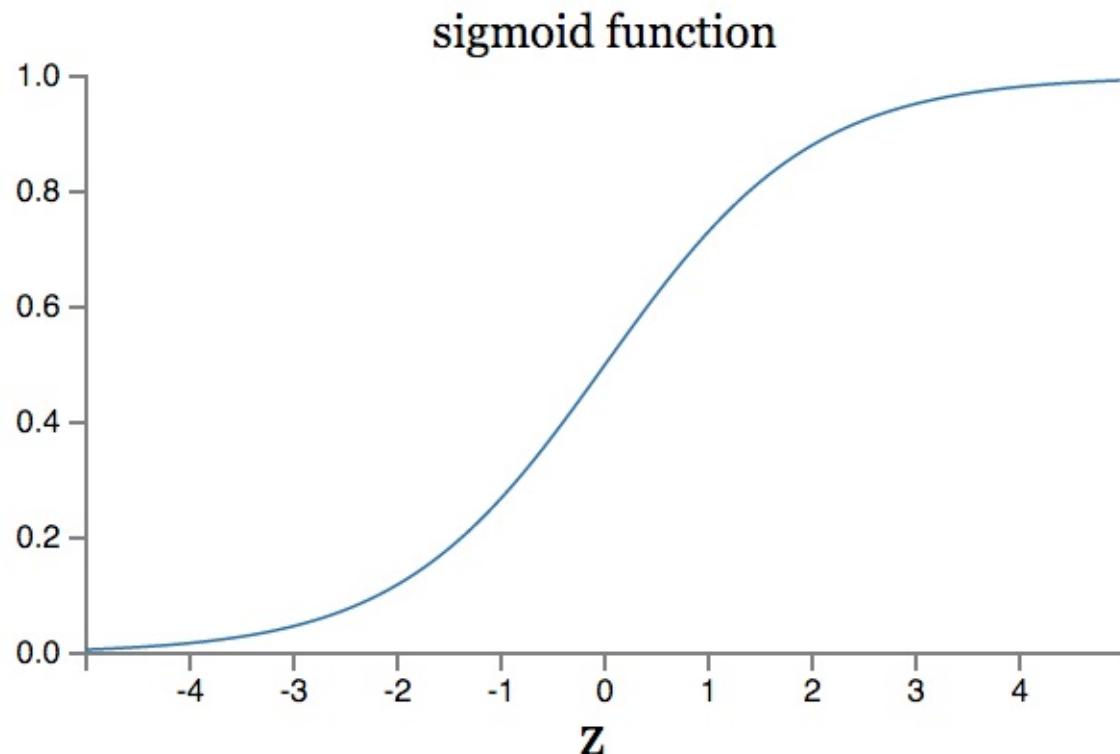
$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z) \quad (55)$$

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

null

null

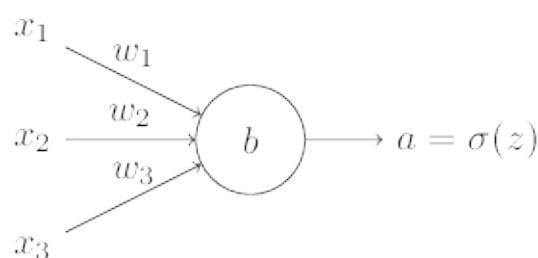
其中我已经将 $x = 1$ 和 $y = 0$ 代入了。为了理解这些表达式的行为，让我们仔细看 $\sigma'(z)$ 这一项。首先回忆一下 σ 函数图像：



我们可以从这幅图看出，当神经元的输出接近 1 的时候，曲线变得相当平，所以 $\sigma'(z)$ 就很小了。方程 (55) 和 (56) 也告诉我们 $\partial C / \partial w$ 和 $\partial C / \partial b$ 会非常小。这其实也是学习缓慢的原因所在。而且，我们后面也会提到，这种学习速度下降的原因实际上也是更加一般的神经网络学习缓慢的原因，并不仅仅是在这个特例中特有的。

引入交叉熵代价函数

那么我们如何解决这个问题呢？研究表明，我们可以通过使用交叉熵代价函数来替换二次代价函数。为了理解什么是交叉熵，我们稍微改变一下之前的简单例子。假设，我们现在要训练一个包含若干输入变量的神经元， x_1, x_2, \dots 对应的权重为 w_1, w_2, \dots 和偏差， b ：



神经元的输出就是 $a = \sigma(z)$ ，其中 $z = \sum_j w_j x_j + b$ 是输入的带权和。我们如下定义交叉熵代价函数：

null

null

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

其中 n 是训练数据的总数，对所有的训练数据 x 和对应的目标输出 y 进行求和。

公式(57)是否解决学习缓慢的问题并不明显。实际上，甚至将这个定义看做是代价函数也不是显而易见的！在解决学习缓慢前，我们来看看交叉熵为何能够解释成一个代价函数。

将交叉熵看做是代价函数有两点原因。第一，它使非负的， $C > 0$ 。可以看出：(a) 公式(57)的和式中所有独立的项都是非负的，因为对数函数的定义域是 $(0, 1)$ ；(b) 前面有一个负号。

第二，如果神经元实际的输出接近目标值。假设在这个例子中， $y = 0$ 而 $a \approx 0$ 。这是我们想到得到的结果。我们看到公式(57)中第一个项就消去了，因为 $y = 0$ ，而第二项实际上就是 $-\ln(1 - a) \approx 0$ 。反之， $y = 1$ 而 $a \approx 1$ 。所以在实际输出和目标输出之间的差距越小，最终的交叉熵的值就越低了。

综上所述，交叉熵是非负的，在神经元达到很好的正确率的时候会接近 0。这些其实就是我们想要的代价函数的特性。其实这些特性也是二次代价函数具备的。所以，交叉熵就是很好的选择了。但是交叉熵代价函数有一个比二次代价函数更好的特性就是它避免了学习速度下降的问题。为了弄清楚这个情况，我们来算算交叉熵函数关于权重的偏导数。我们将 $a = \sigma(z)$ 代入到公式(57)中应用两次链式法则，得到：

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

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

将结果合并一下，简化成：

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

根据 $\sigma(z) = 1/(1 + e^{-z})$ 的定义，和一些运算，我们可以得到 $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ 。后面在练习中会要求你计算这个，现在可以直接使用这个结果。我们看到 σ' 和 $\sigma(z)(1 - \sigma(z))$ 这两项在方程中直接约去了，所以最终形式就是：

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

null

null

这是一个优美的公式。它告诉我们权重学习的速度受到 $\sigma(z) - y$ ，也就是输出中的误差的控制。更大的误差，更快的学习速度。这是我们直觉上期待的结果。特别地，这个代价函数还避免了像在二次代价函数中类似公式中 $\sigma'(z)$ 导致的学习缓慢，见公式(55)。当我们使用交叉熵的时候， $\sigma'(z)$ 被约掉了，所以我们不再需要关心它是不是变得很小。这种约除就是交叉熵带来的特效。实际上，这也并不是非常奇迹的事情。我们在后面可以看到，交叉熵其实只是满足这种特性的一种选择罢了。

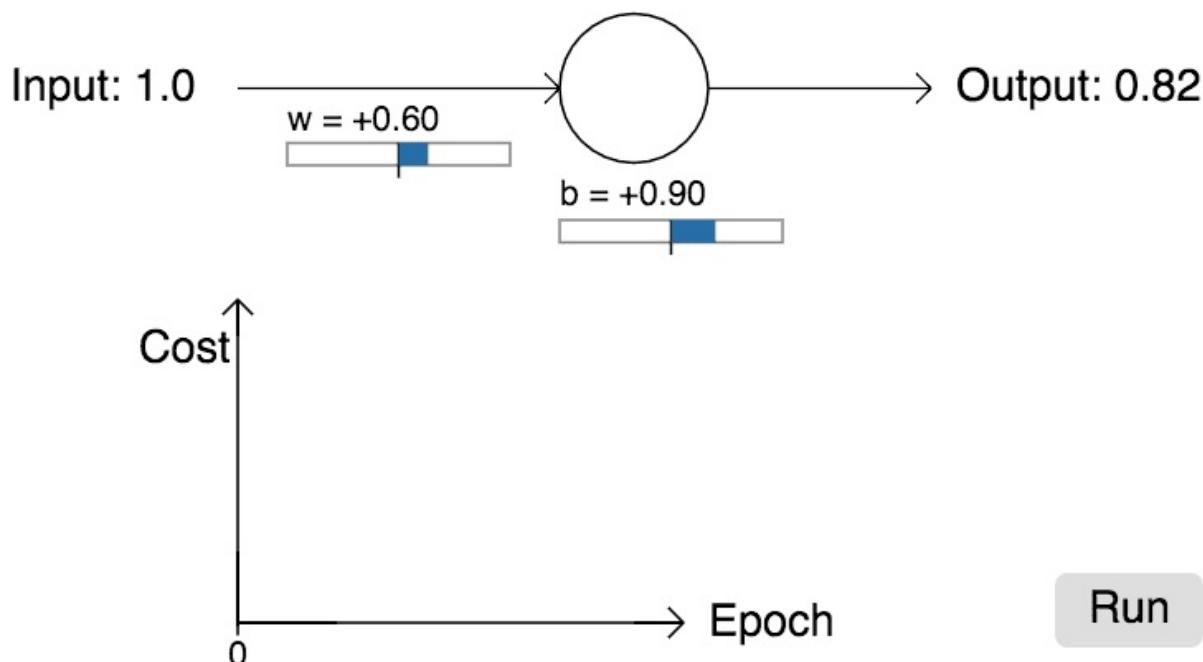
根据类似的方法，我们可以计算出关于偏差的偏导数。我这里不再给出详细的过程，你可以轻易验证得到

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

练习

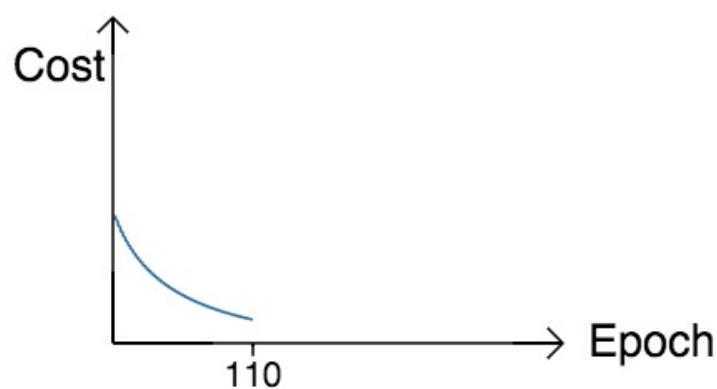
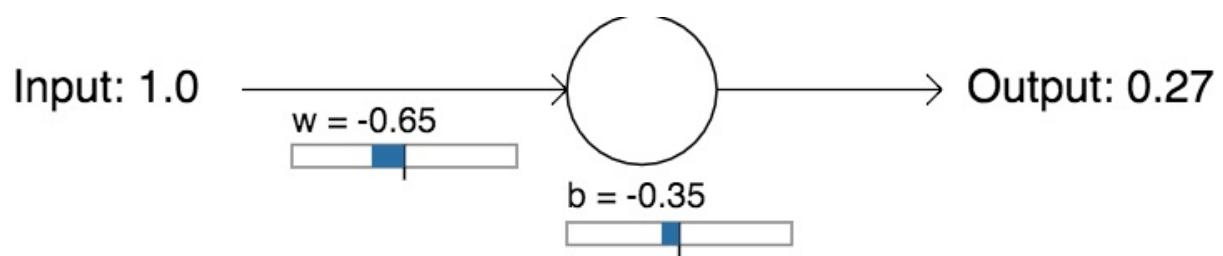
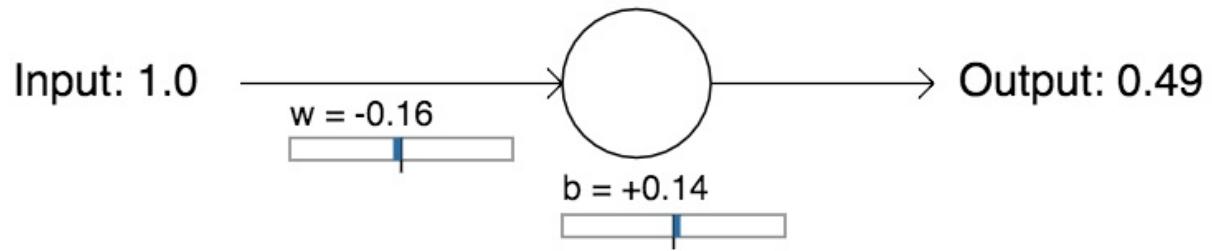
- 验证 $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ 。

让我们重回最原初的例子，来看看换成了交叉熵之后的学习过程。现在仍然按照前面的参数配置来初始化网络，开始权重为 0.6，而偏差为 0.9。点击“Run”按钮看看在换成交叉熵之后网络的学习情况：



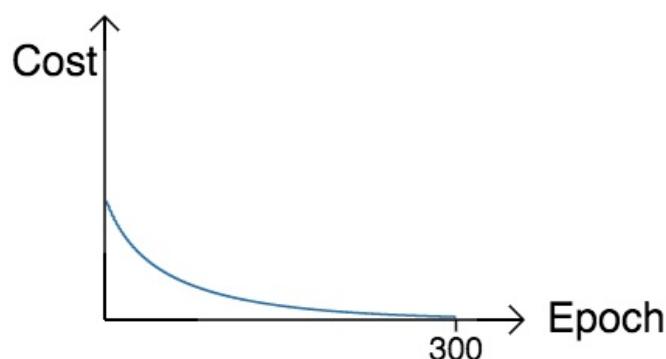
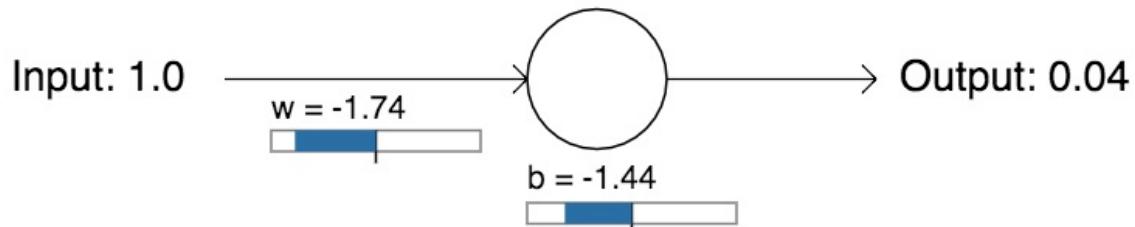
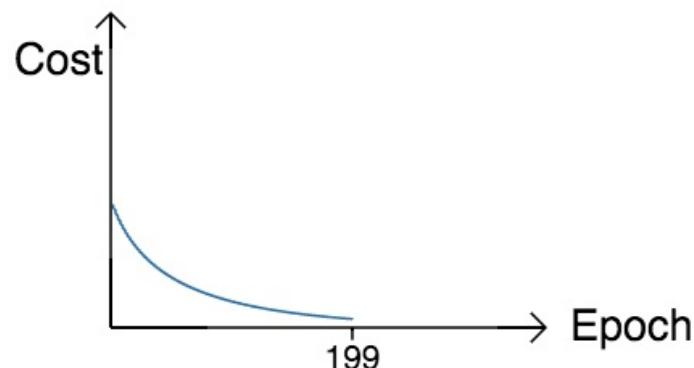
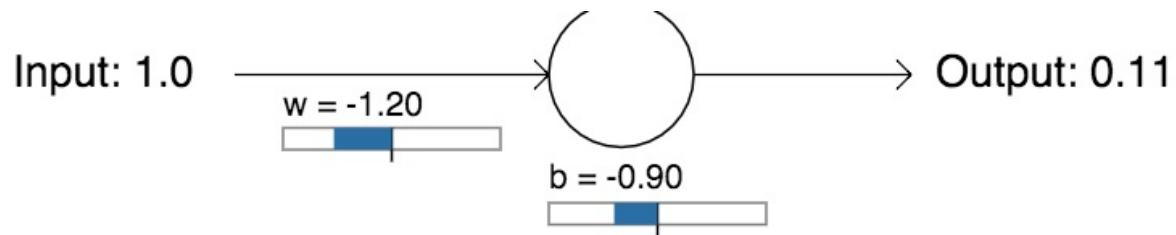
null

null



null

null

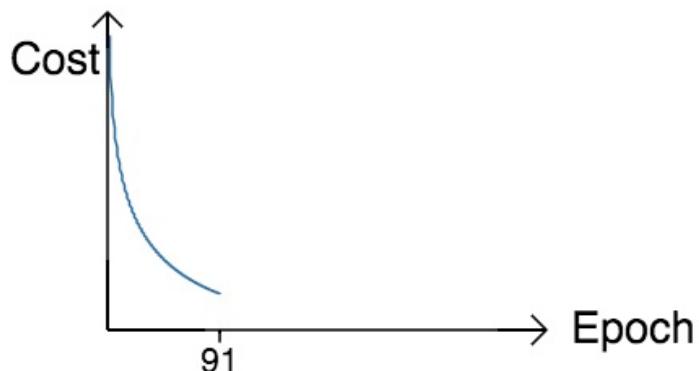
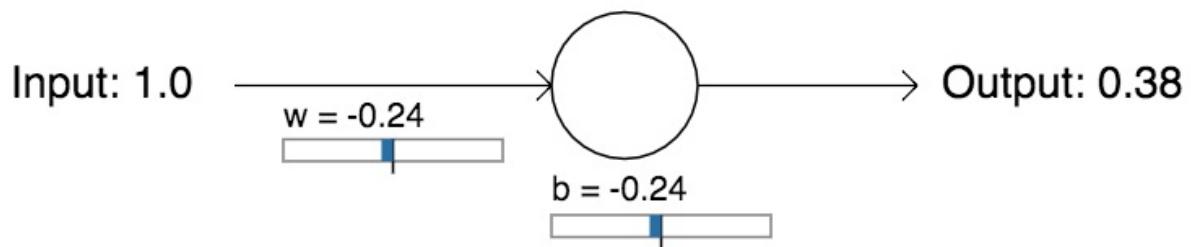
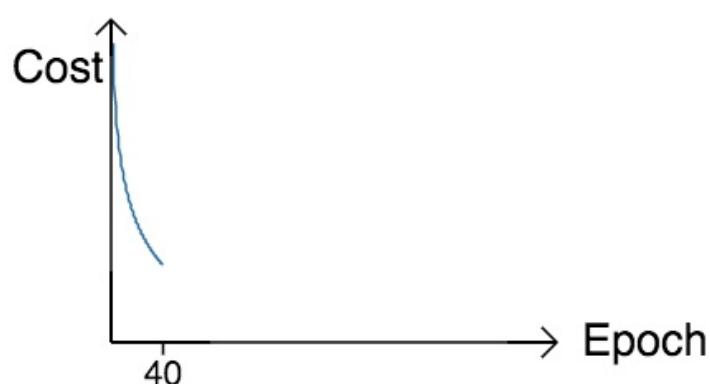
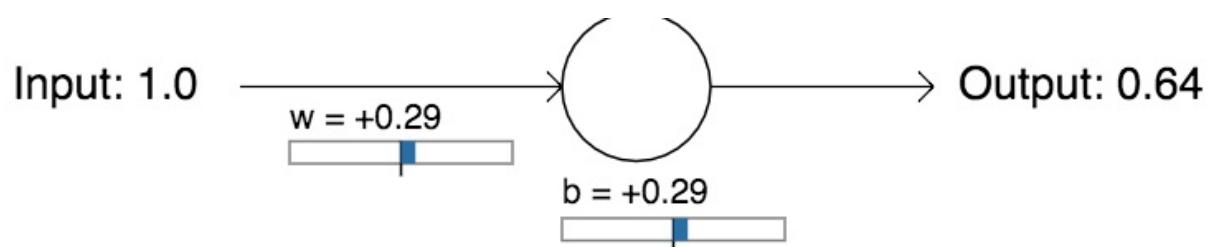
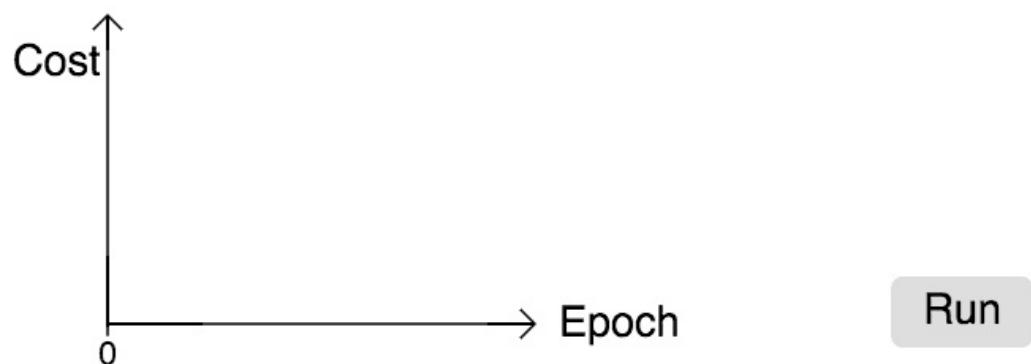
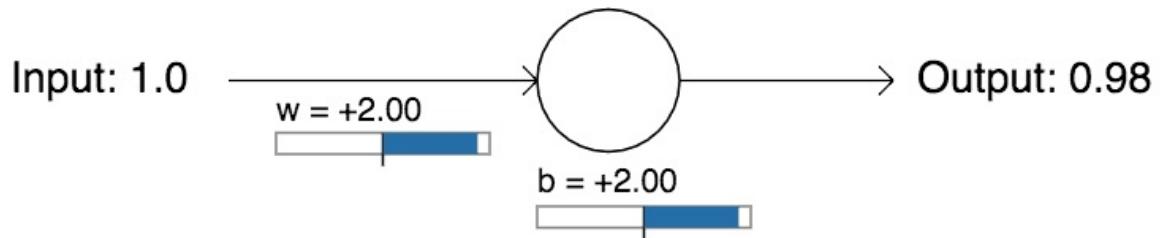


Run

毫不奇怪，在这个例子中，神经元学习得相当出色，跟之前差不多。现在我们再看看之前出问题的那个例子，权重和偏差都初始化为 2.0：

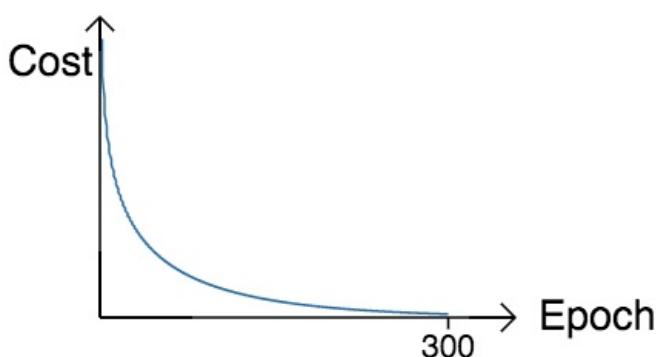
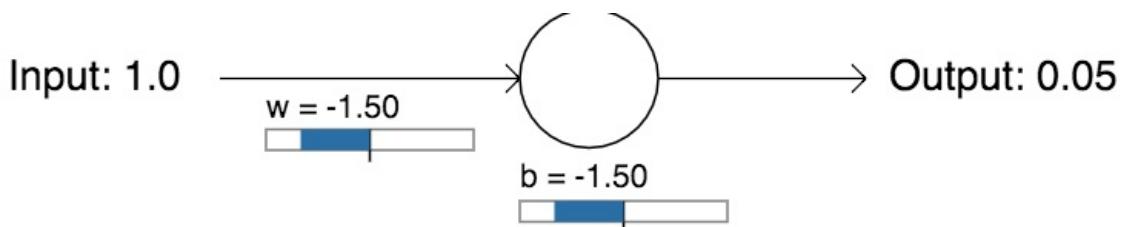
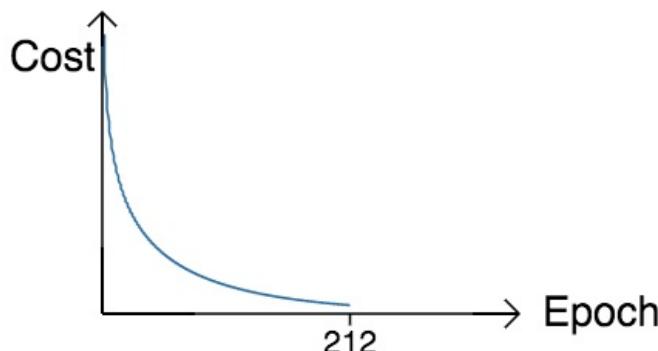
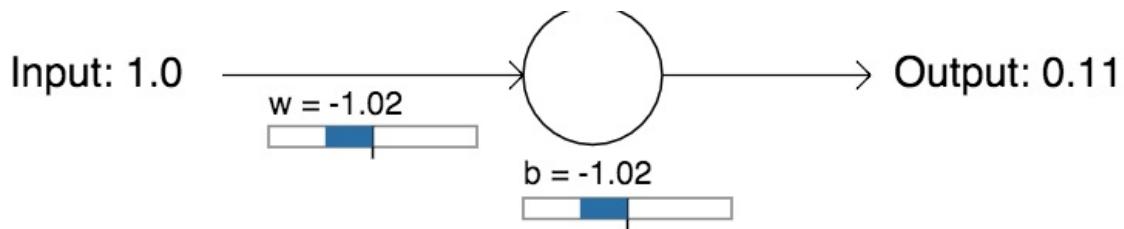
null

null



null

null



Run

成功了！这次神经元的学习速度相当快，跟我们预期的那样。如果你观测的足够仔细，你可以发现代价函数曲线要比二次代价函数训练前面部分要陡很多。正是交叉熵带来的快速下降的坡度让神经元在处于误差很大的情况下能够逃脱出学习缓慢的困境，这才是我们直觉上所期待的效果。

我们还没有提及关于学习率的讨论。刚开始使用二次代价函数的时候，我们使用了 $\eta = 0.15$ 。在新例子中，我们还应该使用同样的学习率么？实际上，根据不同的代价函数，我们不能够直接去套用同样的学习率。这好比苹果和橙子的比较。对于这两种代价函数，我只是通过简单的实验来找到一个能够让我们看清楚变化过程的学习率的值。尽管我不愿意提及，但如果你仍然好奇，这个例子中我使用了 $\eta = 0.005$ 。

你可能会反对说，上面学习率的改变使得上面的图失去了意义。谁会在意当学习率的选择是任意挑选的时候神经元学习的速度？！这样的反对其实没有抓住重点。上面的图例不是想讨论学习的绝对速度。而是想研究学习速度的变化情况。特别地，当我们使用二次代价函数时，学习在神经元犯了明显的错误的时候却比学习快接近真实值的时候缓慢；而使用交叉熵学习正是在神经元犯了明显错误的时候速度更快。这些现象并不是因为学习率的改变造成

null

null

的。

我们已经研究了一个神经元的交叉熵。不过，将其推广到多个神经元的多层神经网络上也是很简单的。特别地，假设 $y = y_1, y_2, \dots$ 是输出神经元上的目标值，而 a_1^L, a_2^L, \dots 是实际输出值。那么我们定义交叉熵如下

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

除了这里需要对所有输出神经元进行求和外，这个其实和我们早前的公式(57)一样的。这里不会给出一个推算的过程，但需要说明的是使用公式(63)确实会在很多的神经网络中避免学习的缓慢。如果你感兴趣，你可以尝试一下下面问题的推导。

那么我们应该在什么时候用交叉熵来替换二次代价函数？实际上，如果在输出神经元使用 sigmoid 激活函数时，交叉熵一般都是更好的选择。为什么？考虑一下我们初始化网络的时候通常使用某种随机方法。可能会发生这样的情况，这些初始选择会对某些训练输入误差相当明显——比如说，目标输出是 1，而实际值是 0，或者完全反过来。如果我们使用二次代价函数，那么这就会导致学习速度的下降。它并不会完全终止学习的过程，因为这些权重会持续从其他的样本中进行学习，但是显然这不是我们想要的效果。

练习

- 一个小问题是刚接触交叉熵时，很难一下子记住那些表达式对应的角色。又比如说，表达式的正确形式是 $-[y \ln a + (1 - y) \ln(1 - a)]$ 或者 $-[a \ln y + (1 - a) \ln(1 - y)]$ 。在 $y = 0$ 或者 1 的时候第二个表达式的结果怎样？这个问题会影响第一个表达式么？为什么？
- 在对单个神经元讨论中，我们指出如果对所有的训练数据有 $\sigma(z) \approx y$ ，交叉熵会很小。这个论断其实是和 y 只是等于 1 或者 0 有关。这在分类问题一般是可行的，但是对其他的问题（如回归问题） y 可以取 0 和 1 之间的中间值的。证明，交叉熵在 $\sigma(z) = y$ 时仍然是最小化的。此时交叉熵的表示是：

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

而其中 $-[y \ln y + (1 - y) \ln(1 - y)]$ 有时候被称为 二元熵

问题

- 多层多神经元网络 用上一章的定义符号，证明对二次代价函数，关于输出层的权重的偏导数为

null

null

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

项 $\sigma'(z_j^L)$ 会在一个输出神经元困在错误值时导致学习速度的下降。证明对于交叉熵代价函数，针对一个训练样本 x 的输出误差 δ^L 为

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

使用这个表达式来证明关于输出层的权重的偏导数为

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

这里 $\sigma'(z_j^L)$ 就消失了，所以交叉熵避免了学习缓慢的问题，不仅仅是在一个神经元上，而且在多层多元网络上都起作用。这个分析过程稍作变化对偏差也是一样的。如果你对这个还不确定，那么请仔细研读一下前面的分析。

- 在输出层使用线性神经元时使用二次代价函数 假设我们有一个多层多神经元网络，最终输出层的神经元都是线性的，输出不再是 sigmoid 函数作用的结果，而是 $a_j^L = z_j^L$ 。证明如果我们使用二次代价函数，那么对单个训练样本 x 的输出误差就是

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

类似于前一个问题，使用这个表达式来证明关于输出层的权重和偏差的偏导数为

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

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

这表明如果输出神经元是线性的那么二次代价函数不再会导致学习速度下降的问题。在此情形下，二次代价函数就是一种合适的选择。

使用交叉熵来对 MNIST 数字进行分类

null

null

交叉熵很容易作为使用梯度下降和反向传播方式进行模型学习的一部分来实现。我们将会在下一章进行对前面的程序 `network.py` 的改进。新的程序写在 `network2.py` 中，不仅使用了交叉熵，还有本章中介绍的其他的技术。现在我们看看新的程序在进行 MNIST 数字分类问题上的表现。如在第一章中那样，我们会使用一个包含 30 个隐藏元的网络，而 minibatch 的大小也设置为 10。我们将学习率设置为 $\eta = 0.5$ 然后训练 30 回合。`network2.py` 的接口和 `network.py` 稍微不同，但是过程还是很清楚的。你可以使用如 `help(network2.Network.SGD)` 这样的命令来检查对应的文档。

```
>>> 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)
```

注意看下，`net.large_weight_initializer()` 命令使用和第一章介绍的同样的方式来进行权重和偏差的初始化。运行上面的代码我们得到了一个 95.49% 准确度的网络。这跟我们在第一章中使用二次代价函数得到的结果相当接近了，95.42%。

同样也来试试使用 100 个隐藏元的交叉熵及其他参数保持不变的情况。在这个情形下，我们获得了 96.82% 的准确度。相比第一章使用二次代价函数的结果 96.59%，这有一定的提升。这看起来是一个小小的变化，但是考虑到误差率已经从 3.41% 下降到 3.18% 了。我们已经消除了原误差的 $1/14$ 。这其实是可观的改进。

令人振奋的是交叉熵代价函数给了我们类似的或者更好的结果。然而，这些结果并没有能够确定性地证明交叉熵是更好的选择。原因在于我已经在选择诸如学习率，minibatch 大小等等这样的超参数上做出了一些努力。为了让这些提升更具说服力，我们需要进行对超参数进行深度的优化。当然，这些结果都挺好的，强化了我们早先关于交叉熵优于二次代价的理论论断。

这只是本章后面的内容和整本书剩余内容中的更为一般的模式的一部分。我们将给出一个新的技术，然后进行尝试，随后获得“提升的”结果。当然，看到这些进步是很好的。但是这些提升的解释一般来说都困难重重。在进行了大量工作来优化所有其他的超参数，使得提升真的具有说服力。工作量很大，需要大量的计算能力，我们也不会进行这样耗费的调查。相反，我采用上面进行的那些不正式的测试来达成目标。所以你要记住这样的测试缺少确定性的证明，需要注意那些使得论断失效的信号。

至此，我们已经花了大量篇幅介绍交叉熵。为何对一个只能给出一点点性能提升的技术上花费这么多的精力？后面，我们会看到其他的技术——规范化，会带来更大的提升效果。所以为何要这么细致地讨论交叉熵？部分原因在于交叉熵是一种广泛使用的代价函数，值得深入理解。但是更加重要的原因是神经元的饱和是神经网络中一个关键的问题，整本书都会不断回归到这个问题上。因此我现在深入讨论交叉熵就是因为这是一种开始理解神经元饱和和如何解决这个问题的很好的实验。

null

交叉熵的含义？源自哪里？

我们对于交叉熵的讨论聚焦在代数分析和代码实现。这虽然很有用，但是也留下了一个未能回答的更加宽泛的概念上的问题，如：交叉熵究竟表示什么？存在一些直觉上的思考交叉熵的方法么？我们如何想到这个概念？

让我们从最后一个问题开始回答：什么能够激发我们想到交叉熵？假设我们发现学习速度下降了，并理解其原因是因为在公式(55)(56)中的 $\sigma'(z)$ 那一项。在研究了这些公式后，我们可能就会想到选择一个不包含 $\sigma'(z)$ 的代价函数。所以，这时候对一个训练样本 x ，代价函数是 $C = C_x$ 就满足：

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

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

如果我们选择的代价函数满足这些条件，那么同样他们会拥有遇到明显误差时的学习速度越快这样的特性。这也能够解决学习速度下降的问题。实际上，从这些公式开始，现在就给你们看看若是跟随自身的数学嗅觉推导出交叉熵的形式是可能的。我们来推一下，由链式法则，我们有

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

使用 $\sigma'(z) = \sigma(z)(1 - \sigma(z)) = a(1 - a)$ ，上个等式就变成

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

对比等式(72)，我们有

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

对此方程关于 a 进行积分，得到

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

其中 constant 是积分常量。这是一个单独的训练样本 x 对代价函数的贡献。为了得到整个

null

null

的代价函数，我们需要对所有的训练样本进行平均，得到了

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

而这里的常量就是所有单独的常量的平均。所以我们看到方程(71)和(72)唯一确定了交叉熵的形式，并加上了一个常量的项。这个交叉熵并不是凭空产生的。而是一种我们以自然和简单的方法获得的结果。

那么交叉熵直觉含义又是什么？我们如何看待它？深入解释这一点会将我们带到一个不大愿意讨论的领域。然而，还是值得提一下，有一种源自信息论的解释交叉熵的标准方式。粗略地说，交叉熵是惊奇的度量（measure of surprise）。特别地，我们的神经元想要要计算函数 $x \rightarrow y = y(x)$ 。但是，它用函数 $x \rightarrow a = a(x)$ 进行了替换。假设我们将 a 想象成我们神经元估计 $y = 1$ 概率，而 $1 - a$ 则是 $y = 0$ 的概率。如果输出我们期望的结果，惊奇就会小一点；反之，惊奇就大一些。当然，我这里没有严格地给出“惊奇”到底意味着什么，所以看起来像在夸夸奇谈。但是实际上，在信息论中有一种准确的方式来定义惊奇究竟是什么。不过，我也不清楚在网络上，哪里有好的，短小的自包含对这个内容的讨论。但如果你要深入了解的话，Wikipedia 包含一个[简短的总结](#)，这会指引你正确地探索这个领域。而更加细节的内容，你们可以阅读[Cover and Thomas](#)的第五章涉及 Kraft 不等式的有关信息论的内容。

问题

- 我们已经深入讨论了使用二次代价函数的网络中在输出神经元饱和时候学习缓慢的问题，另一个可能会影响学习的因素就是在方程(61)中的 x_j 项。由于此项，当输入 x_j 接近 0 时，对应的权重 x_j 会学习得相当缓慢。解释为何不可以通过改变代价函数来消除 x_j 项的影响。

Softmax

本章，我们大多数情况会使用交叉熵来解决学习缓慢的问题。但是，我希望简要介绍一下基于 softmax 神经元层的解决这个问题的另一种观点。我们不会实际在剩下的章节中使用 softmax 层，所以你如果赶时间，就可以跳到下一个节了。不过，softmax 仍然有其重要价值，一方面它本身很有趣，另一方面，因为我们在第六章在对深度神经网络的讨论中使用 softmax 层。

softmax 的想法其实就是为神经网络定义一种新式的输出层。开始时和 sigmoid 层一样的，首先计算带权输入 $z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L$ 。不过，这里我们不会使用 sigmoid 函数来获得输出。而是，会应用一种叫做 softmax 函数在 z_j^L 上。根据这个函数，激活值 a_j^L 就是

null

null

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

其中，分母是对所有的输出神经元进行求和。

如果你不习惯这个函数，方程(78)可能看起来会比较难理解。因为对于使用这个函数的原因你不清楚。为了更好地理解这个公式，假设我们有一个包含四个输出神经元的神经网络，对应四个带权输入，表示为 $z_1^L, z_2^L, z_3^L, z_4^L$ 。下面的例子可以进行对应权值的调整，并给出对应的激活值的图像。可以通过固定其他值，来看看改变 z_4^L 的值会产生什么样的影响：



$$z_1^L = 2.5$$



$$z_2^L = -1$$



$$z_3^L = 3.2$$



$$z_4^L = 0.5$$



$$a_1^L = 0.315$$



$$a_2^L = 0.009$$



$$a_3^L = 0.633$$



$$a_4^L = 0.043$$



$$z_1^L = 2.5$$



$$z_2^L = -1$$



$$z_3^L = 3.2$$



$$z_4^L = 1.6$$



$$a_1^L = 0.290$$



$$a_2^L = 0.009$$



$$a_3^L = 0.584$$



$$a_4^L = 0.118$$

null

null



$$z_1^L = 2.5$$



$$z_2^L = -1$$



$$z_3^L = 3.2$$



$$z_4^L = 3.5$$



$$a_1^L = 0.174$$



$$a_2^L = 0.005$$



$$a_3^L = 0.349$$



$$a_4^L = 0.472$$

这里给出了三个选择的图像，建议去原网站体验

在我们增加 z_4^L 的时候，你可以看到对应激活值 a_4^L 的增加，而其他的激活值就在下降。类似地，如果你降低 z_4^L 那么 a_4^L 就随之下降。实际上，如果你仔细看，你会发现在两种情形下，其他激活值的整个改变恰好填补了 a_4^L 的变化的空白。原因很简单，根据定义，输出的激活值加起来正好为 1，使用公式(78)我们可以证明：

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

所以，如果 a_4^L 增加，那么其他输出激活值肯定会总共下降相同的量，来保证和式为 1。当然，类似的结论对其他的激活函数也需要满足。

方程(78)同样保证输出激活值都是正数，因为指数函数是正的。将这两点结合起来，我们看到 softmax 层的输出是一些相加为 1 正数的集合。换言之，softmax 层的输出可以被看做是一个概率分布。

这样的效果很令人满意。在很多问题中，将这些激活值作为网络对于某个输出正确的概率的估计非常方便。所以，比如在 MNIST 分类问题中，我们可以将 a_j^L 解释成网络估计正确数字分类为 j 的概率。

对比一下，如果输出层是 sigmoid 层，那么我们肯定不能假设激活值形成了一个概率分布。我不会证明这一点，但是源自 sigmoid 层的激活值是不能够形成一种概率分布的一般形式的。所以使用 sigmoid 输出层，我们没有关于输出的激活值的简单的解释。

null

练习

- 构造例子表明在使用 sigmoid 输出层的网络中输出激活值 a_j^L 的和并不会确保为 1。

我们现在开始体会到 softmax 函数的形式和行为特征了。来回顾一下：在公式(78)中的指数函数确保了所有的输出激活值是正数。然后分母的求和又保证了 softmax 的输出和为 1。所以这个特定的形式不再像之前那样难以理解了：反而是一种确保输出激活值形成一个概率分布的自然的方式。你可以将其想象成一种重新调节 z_j^L 的方法，然后将这个结果整合起来构成一个概率分布。

练习

- softmax** 的单调性 证明如果 $j = k$ 则 $\partial a_j^L / \partial z_k^L$ 为正，否则为负。结果是，增加 z_k^L 会提高对应的输出激活值 a_k^L 并降低其他所有输出激活值。我们已经在滑条示例中实验性地看到了这一点，这里需要你给出一个严格证明。
- softmax** 的非局部性 sigmoid 层的一个好处是输出 a_j^L 是对应带权输入 $a_j^L = \sigma(z_j^L)$ 的函数。解释为何对于 softmax 来说，并不是这样的情况：仍和特定的输出激活值 a_j^L 依赖所有的带权输入。

问题

- 逆转 softmax 层 假设我们有一个使用 softmax 输出层的神经网络，然后激活值 a_j^L 已知。证明对应带权输入的形式为 $z_j^L = \ln a_j^L + C$ ，其中 C 是独立于 j 的。

学习缓慢问题：我们现在已经对 softmax 神经元有了一定的认识。但是我们还没有看到 softmax 会怎么样解决学习缓慢问题。为了理解这点，先定义一个 log-likelihood 代价函数。我们使用 x 表示训练输入， y 表示对应的目标输出。然后关联这个训练输入样本的 log-likelihood 代价函数就是

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

所以，如果我们训练的是 MNIST 图像，输入为 7 的图像，那么对应的 log-likelihood 代价就是 $-\ln a_7^L$ 。看看这个直觉上的含义，想想当网络表现很好的时候，也就是确认输入为 7 的时候。这时，他会估计一个对应的概率 a_7^L 跟 1 非常接近，所以代价 $-\ln a_7^L$ 就会很小。反之，如果网络的表现糟糕时，概率 a_7^L 就变得很小，代价 $-\ln a_7^L$ 随之增大。所以 log-likelihood 代价函数也是满足我们期待的代价函数的条件的。

那关于学习缓慢问题呢？为了分析它，回想一下学习缓慢的关键就是量 $\partial C / \partial w_{jk}^L$ 和

null

$\partial C / \partial b_j^L$ 的变化情况。我不会显式地给出详细的推导——请你们自己去完成这个过程——但是会给出一些关键的步骤：

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

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

请注意这里的表示上的差异，这里的 y 和之前的目标输出值不同，是离散的向量表示，对应位的值为 1，而其他为 0。这些方程其实和我们前面对交叉熵得到的类似。就拿方程(82) 和 (67) 比较。尽管后者我对整个训练样本进行了平均，不过形式还是一致的。而且，正如前面的分析，这些表达式确保我们不会遇到学习缓慢的问题。实际上，将 softmax 输出层和 log-likelihood 组合对照 sigmoid 输出层和交叉熵的组合类比着看是非常有用的。

有了这样的相似性，你会使用哪一种呢？实际上，在很多应用场景中，这两种方式的效果都不错。本章剩下的内容，我们会使用 sigmoid 输出层和交叉熵的组合。后面，在第六章中，我们有时候会使用 softmax 输出层和 log-likelihood 的则和。切换的原因就是为了让我们的网络和某些在具有影响力的学术论文中的形式更为相似。作为一种更加通用的视角，softmax 加上 log-likelihood 的组合更加适用于那些需要将输出激活值解释为概率的场景。当然这不总是合理的，但是在诸如 MNIST 这种有着不重叠的分类问题上确实很有用。

问题

- 推导方程(81) 和 (82)
- softmax 这个名称从何处来？假设我们改变一下 softmax 函数，使得输出激活值定义如下

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

其中 c 是正的常量。注意 $c = 1$ 对应标准的 softmax 函数。但是如果我们使用不同的 c 得到不同的函数，其实最终的量的结果却和原来的 softmax 差不多。特别地，证明输出激活值也会形成一个概率分布。假设我们允许 c 足够大，比如说 $c \rightarrow \infty$ 。那么输出激活值 a_j^L 的极限值是什么？在解决了这个问题后，你应该能够理解 $c = 1$ 对应的函数是一个最大化函数的 softened 版本。这就是 softmax 的来源。

这让我联想到 EM 算法，对 k-Means 算法的一种推广。

null

null

- softmax 和 log-likelihood 的反向传播 上一章，我们推到了使用 sigmoid 层的反向传播算法。为了应用在 softmax 层的网络上，我们需要搞清楚最后一层上误差的表示 $\delta_j^L \equiv \partial C / \partial z_j^L$ 。证明形式如下：

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

使用这个表达式，我们可以应用反向传播在采用了 softmax 输出层和 log-likelihood 的网络上。

过匹配和规范化

诺贝尔奖得主美籍意大利裔物理学家恩里科·费米曾被问到他对一个同僚提出的尝试解决一个重要的未解决物理难题的数学模型。模型和实验非常匹配，但是费米却对其产生了怀疑。他问模型中需要设置的自由参数有多少个。答案是“4”。费米回答道：“我记得我的朋友约翰·冯·诺依曼过去常说，有四个参数，我可以模拟一头大象，而有五个参数，我还能让他卷鼻子。”

这里，其实是指拥有大量的自由参数的模型能够描述特别神奇的现象。即使这样的模型能够很好的拟合已有的数据，但并不表示是一个好模型。因为这可能只是因为模型中足够的自由度使得它可以描述几乎所有给定大小的数据集，不需要对现象的本质有创新的认知。所以发生这种情形时，模型对已有的数据会表现的很好，但是对新的数据很难泛化。对一个模型真正的测验就是它对没有见过的场景的预测能力。

费米和冯·诺依曼对有四个参数的模型就开始怀疑了。我们用来对 MNIST 数字分类的 30 个隐藏神经元神经网络拥有将近 24,000 个参数！当然很多。我们有 100 个隐藏元的网络拥有将近 80,000 个参数，而目前最先进的深度神经网络包含百万级或者十亿级的参数。我们应当信赖这些结果么？

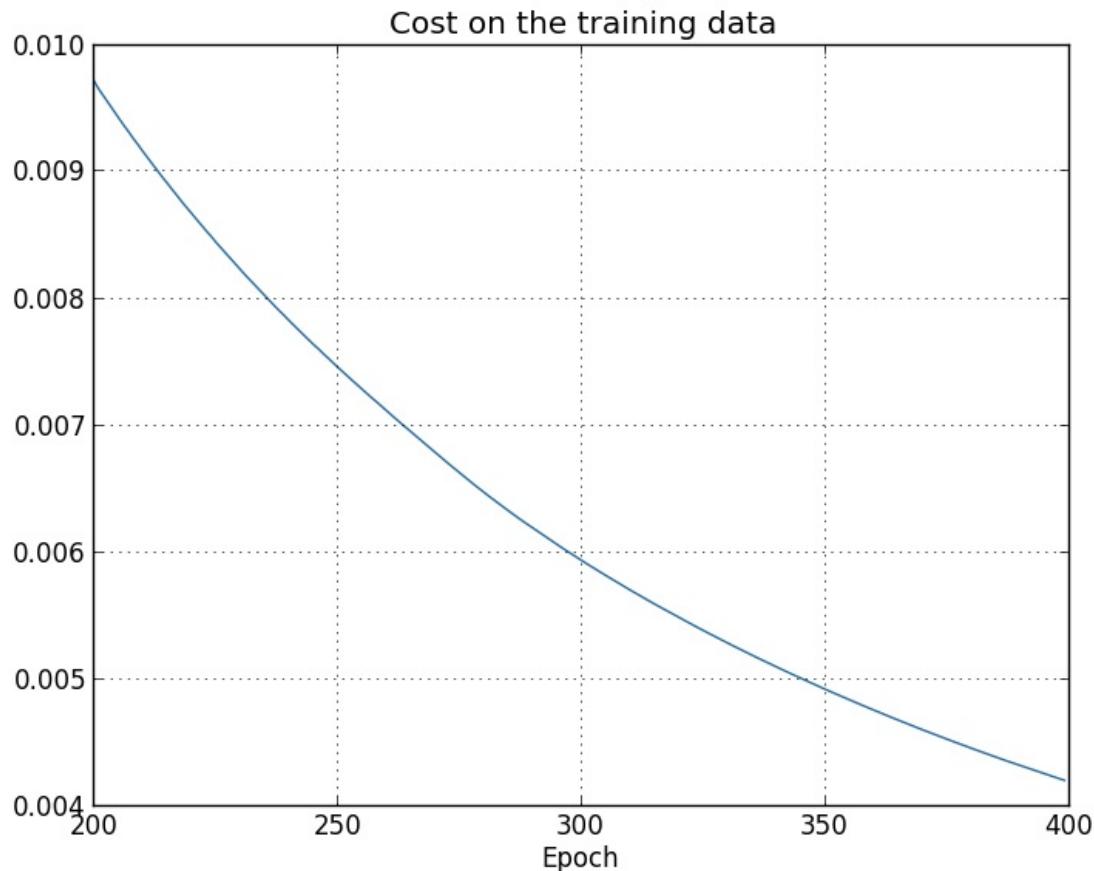
让我们将问题暴露出来，通过构造一个网络泛化能力很差的例子。我们的网络有 30 个隐藏神经元，共 23,860 个参数。但是我们不会使用所有 50,000 幅训练图像。相反，我们只使用前 1000 幅图像。使用这个受限的集合，会让泛化的问题突显。按照同样的方式，使用交叉熵代价函数，学习率设置为 $\eta = 0.5$ 而 minibatch 大小设置为 10。不过这里我们训练回合设置为 400，比前面的要多很多，因为我们只用了少量的训练样本。我们现在使用 network2 来研究代价函数改变的情况：

```
>>> 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)
```

null

null

使用上面的结果，我们可以画出代价函数变化的情况：

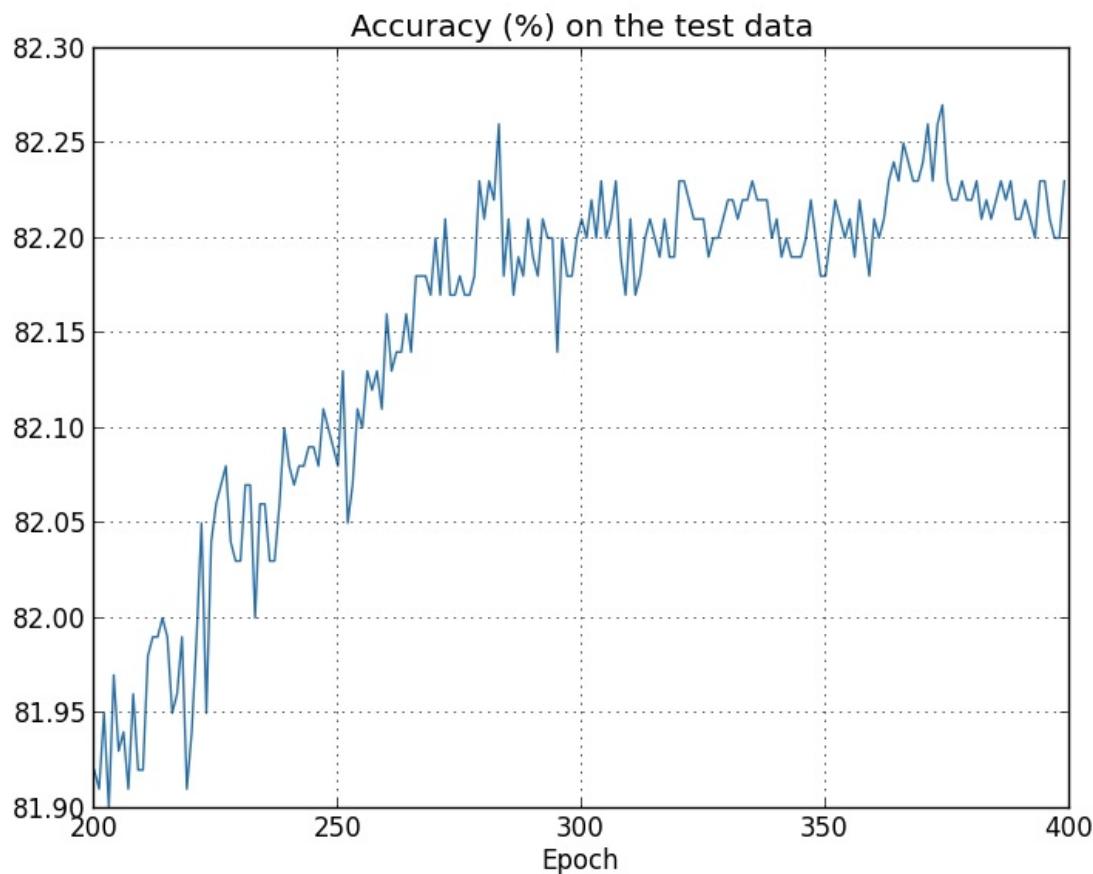


这看起来令人振奋，因为代价函数有一个光滑的下降，跟我们预期一致。注意，我只是展示了 200 到 399 回合的情况。这给出了很好的近距离理解训练后期的情况，这也是出现有趣现象的地方。

让我们看看分类准确度在测试集上的表现：

null

null

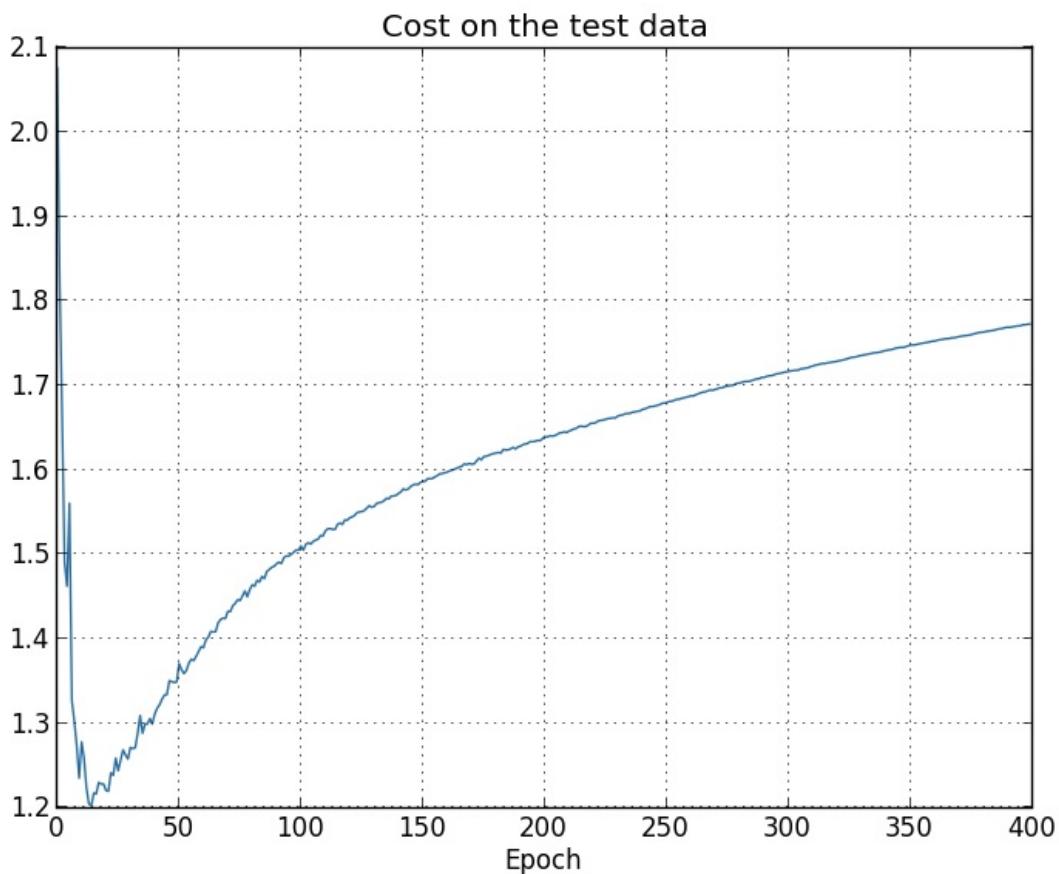


这里我还是聚焦到了后面的过程。在前 200 回合（图中没有显示）准确度提升到了 82%。然后学习逐渐变缓。最终，在 280 回合左右分类准确度就停止了增长。后面的回合，仅仅看到了在 280 回合准确度周围随机的震荡。将这幅图和前面的图进行对比，和训练数据相关的代价函数持续平滑下降。如果我们只看哪个代价，我们会发现模型的表现变得“更好”。但是测试准确度展示了提升只是一种假象。就像费米不大喜欢的那个模型一样，我们的网络在 280 回合后就不在能够繁华到测试数据上。所以这种学习不大有用。也可以说网络在 280 后就过匹配（或者过度训练）了。

你可能想知道这里的问题是不是由于我们看的是训练数据的代价，而对比的却是测试数据上的分类准确度导致的。换言之，可能我们这里在进行苹果和橙子的对比。如果我们比较训练数据上的代价和测试数据上的代价，会发生什么，我们是在比较类似的度量么？或者可能我们可以比较在两个数据集上的分类准确度啊？实际上，不管我们使用什么度量的方式尽管，细节会变化，但本质上都是一样的。让我们来看看测试数据集上的代价变化情况：

null

null

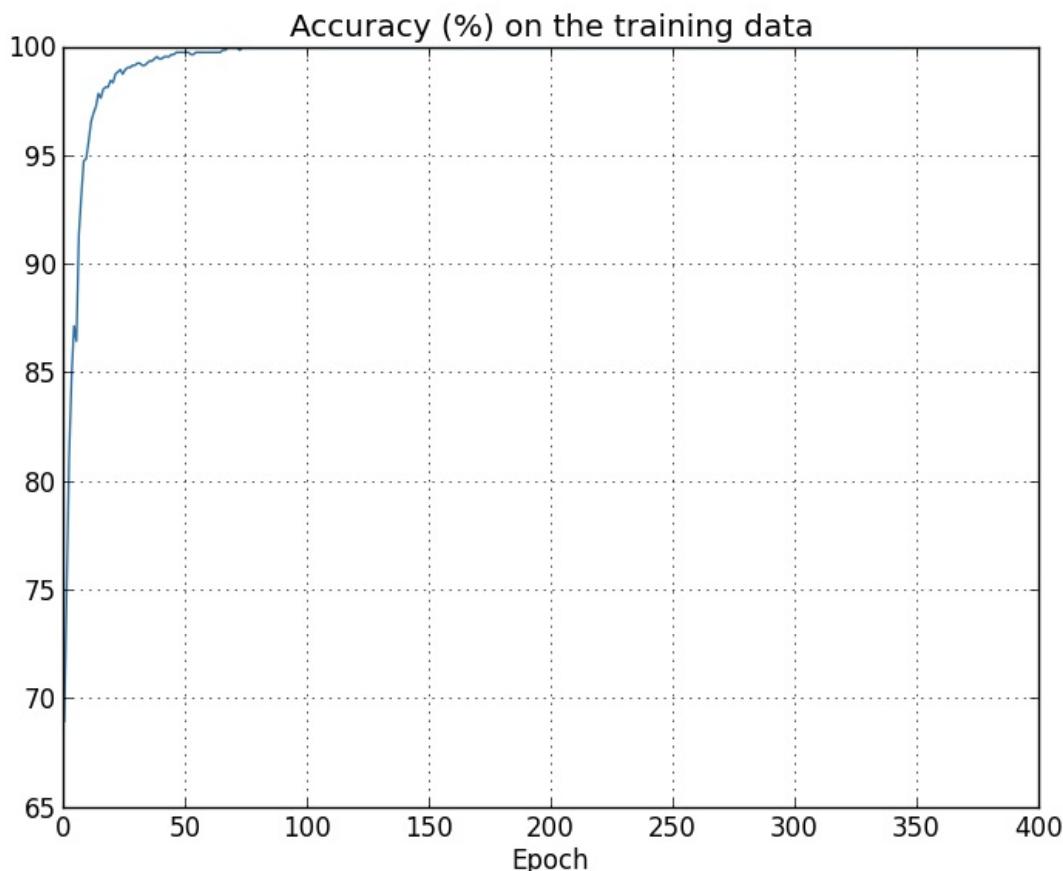


我们可以看到测试集上的代价在 15 回合前一直在提升，随后越来越差，尽管训练数据机上的代价表现是越来越好的。这其实是另一种模型过匹配的标志。尽管，这里带来了关于我们应当将 15 还是 280 回合当作是过匹配占主导的时间点的困扰。从一个实践角度，我们真的关心的是提升测试数据集上的分类准确度，而测试集合上的代价不过是分类准确度的一个反应。所以更加合理的选择就是将 280 看成是过匹配开始占统治地位的时间点。

另一个过匹配的信号在训练数据上的分类准确度上也能看出来：

null

null



准确度一直在提升接近 100%。也就是说，我们的网络能够正确地对所有 1000 幅图像进行分类！而在同时，我们的测试准确度仅仅能够达到 82.27%。所以我们的网络实际上在学习训练数据集的特例，而不是能够一般地进行识别。我们的网络几乎是在单纯记忆训练集合，而没有对数字本质进行理解能够泛化到测试数据集上。

过匹配是神经网络的一个主要问题。这在现代网络中特别正常，因为网络权重和偏差数量巨大。为了高效地训练，我们需要一种检测过匹配是不是发生的技术，这样我们不会过度训练。并且我们也想要找到一些技术来降低过匹配的影响。

检测过匹配的明显方法是使用上面的方法——跟踪测试数据集合上的准确度随训练变化情况。如果我们看到测试数据上的准确度不再提升，那么我们就停止训练。当然，严格地说，这其实并非是过匹配的一个必要现象，因为测试集和训练集上的准确度可能会同时停止提升。当然，采用这样的方式是可以阻止过匹配的。

实际上，我们会使用这种方式变形来试验。记得之前我们载入 MNIST 数据的时候有：

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

到现在我们一直在使用 `training_data` 和 `test_data`，没有用过 `validation_data`。`validation_data` 中包含了 10,000 幅数字图像，这些图像和训练数据集中的 50,000 幅图像以及测试数据集中的 10,000 幅都不相同。我们会使用 `validation_data`

null

null

来防止过匹配。我们会使用和上面应用在 `test_data` 的策略。我们每个回合都计算在 `validation_data` 上的分类准确度。一旦分类准确度已经饱和，就停止训练。这个策略被称为提前停止（Early stopping）。当然，实际应用中，我们不会立即知道什么时候准确度会饱和。相反，我们会一直训练知道我们确信准确度已经饱和。

这里需要一些判定标准来确定什么时候停止。在我前面的图中，将 280 回合看成是饱和的地方。可能这有点太悲观了。因为神经网络有时候会训练过程中处在一个平原期，然后又开始提升。如果在 400 回合后，性能又开始提升（也许只是一些少量提升），那我也不会诧异。所以，在提前停止中采取一点激进的策略也是可以的。

为何要使用 `validation_data` 来替代 `test_data` 防止过匹配问题？实际上，这是一个更为一般的策略的一部分，这个一般的策略就是使用 `validation_data` 来衡量不同的超参数（如训练回合，学习率，最好的网络架构等等）的选择的效果。我们使用这样方法来找到超参数的合适值。因此，尽管到现在我并没有提及这点，但其实本书前面已经稍微介绍了一些超参数选择的方法。

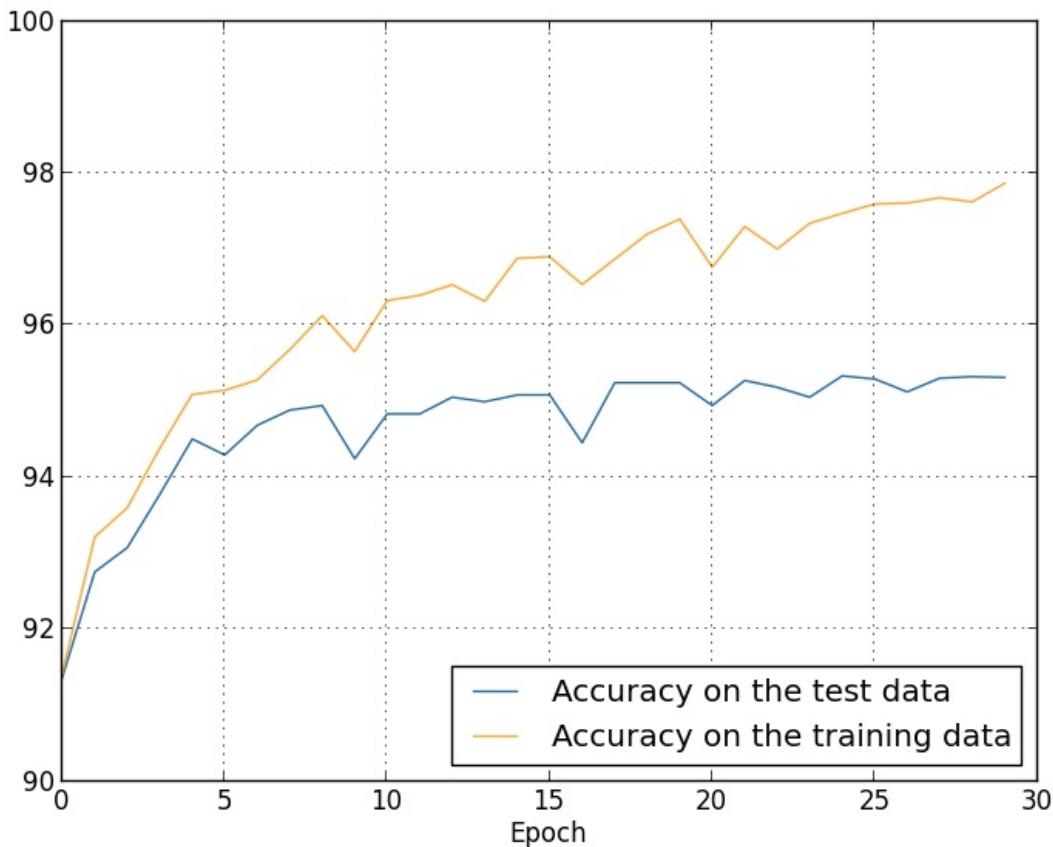
当然，这对于我们前面关于 `validation_data` 取代 `test_data` 来防止过匹配的原因仍旧没有回答。实际上，有一个更加一般的问题，就是为何用 `validation_data` 取代 `test_data` 来设置更好的超参数？为了理解这点，想想当设置超参数时，我们想要尝试许多不同的超参数选择。如果我们设置超参数是基于 `test_data` 的话，可能最终我们就会得到过匹配于 `test_data` 的超参数。也就是说，我们可能会找到那些符合 `test_data` 特点的超参数，但是网络的性能并不能够泛化到其他数据集合上。我们借助 `validation_data` 来克服这个问题。然后一旦获得了想要的超参数，最终我们就使用 `test_data` 进行准确度测量。这给了我们在 `test_data` 上结果是一个网络泛化能力真正的度量方式的信心。换言之，你可以将验证集看成是一种特殊的训练数据集能够帮助我们学习好的超参数。这种寻找好的超参数的观点有时候被称为 **hold out** 方法，因为 `validation_data` 是从训练集中保留出来的一部分。

在实际应用中，甚至在衡量了测试集上的性能后，我们可能也会改变想法并去尝试另外的方法——也许是一种不同的网络架构——这将会引入寻找新的超参数的过程。如果我们这样做，难道不会产生过匹配于 `test_data` 的困境么？我们是不是需要一种潜在无限大的数据集的回归，这样才能够确信模型能够泛化？去除这样的疑惑其实是一个深刻而困难的问题。但是对实际应用的目标，我们不会担心太多。相反，我们会继续采用基于 `training_data`, `validation_data`, and `test_data` 的基本 **hold out** 方法。

我们已经研究了在使用 1,000 幅训练图像时的过匹配问题。那么如果我们使用所有的训练数据会发生什么？我们会保留所有其他的参数都一样（30 个隐藏元，学习率 0.5，mini-batch 规模为 10），但是训练回合为 30 次。下图展示了分类准确度在训练和测试集上的变化情况。注意我们使用的测试数据，而不是验证集合，为了让结果看起来和前面的图更方便比较。

null

null



如你所见，测试集和训练集上的准确度相比我们使用 1,000 个训练数据时相差更小。特别地，在训练数据上的最佳的分类准确度 97.86% 只比测试集上的 95.33% 准确度高一点点。而之前的例子中，这个差距是 17.73%！过匹配仍然发生了，但是已经减轻了不少。我们的网络从训练数据上更好地泛化到了测试数据上。一般来说，最好的降低过匹配的方式之一就是增加训练样本的量。有了足够的训练数据，就算是一个规模非常大的网络也不大容易过匹配。不幸的是，训练数据其实是很困难或者很昂贵的资源，所以这不是一种太切实际的选择。

规范化

增加训练样本的数量是一种减轻过匹配的方法。还有其他的一些方法能够减轻过匹配的程度么？一种可行的方式就是降低网络的规模。然而，大的网络拥有一种比小网络更强的潜力，所以这里存在一种应用冗余性的选项。

幸运的是，还有其他的技术能够缓解过匹配，即使我们只有一个固定的网络和固定的训练集合。这种技术就是规范化。本节，我会给出一种最为常用的规范化手段——有时候被称为权重下降（weight decay）或者 **L2** 规范化。**L2** 规范化的想法是增加一个额外的项到代价函数上，这个项叫做 规范化 项。下面是规范化交叉熵：

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

null

null

其中第一个项就是常规的交叉熵的表达式。第二个现在加入到就是所有权重的平方的和。然后使用一个因子 $\lambda/2n$ 进行量化调整，其中 $\lambda > 0$ 可以成为规范化参数，而 n 就是训练集合的大小。我们会在后面讨论 λ 的选择策略。需要注意的是，规范化项里面并不包含偏差。这点我们后面也会在讲述。

当然，对其他的代价函数也可以进行规范化，例如二次代价函数。类似的规范化的形式如下：

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

两者都可以写成这样：

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

其中 C_0 是原始的代价函数。

直觉地看，规范化的效果是让网络倾向于学习小一点的权重，其他的东西都一样的。大的权重只有能够给出代价函数第一项足够的提升时才被允许。换言之，规范化可以当做一种寻找小的权重和最小化原始的代价函数之间的折中。这两部分之前相对的重要性就由 λ 的值来控制了： λ 越小，就偏向于最小化原始代价函数，反之，倾向于小的权重。

现在，对于这样的折中为何能够减轻过匹配还不是很清楚！但是，实际表现表明了这点。我们会在下一节来回答这个问题。但现在，我们来看看一个规范化的确减轻过匹配的例子。

为了构造这个例子，我们首先需要弄清楚如何将随机梯度下降算法应用在一个规范化的神经网络上。特别地，我们需要知道如何计算偏导数 $\partial C / \partial w$ 和 $\partial C / \partial b$ 。对公式(87)进行求偏导数得：

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

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

$\partial C_0 / \partial w$ 和 $\partial C_0 / \partial b$ 可以通过反向传播进行计算，和上一章中的那样。所以我们看到其实计算规范化的代价函数的梯度是很简单的：仅仅需要反向传播，然后加上 $\frac{\lambda}{n} w$ 得到所有权重的偏导数。而偏差的偏导数就不要变化，所以梯度下降学习规则不会发生变化：

null

null

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

权重的学习规则就变成：

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

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

这其实和通常的梯度下降学习规则相同，除了乘了 $1 - \frac{\eta \lambda}{n}$ 因子。这里就是权重下降的来源。粗看，这样会导致权重会不断下降到 0。但是实际不是这样的，因为如果在原始代价函数中造成下降的话其他的项可能会让权重增加。

好的，这就是梯度下降工作的原理。那么随机梯度下降呢？正如在没有规范化的随机梯度下降中，我们可以通过平均 minibatch 中 m 个训练样本来估计 $\partial C_0 / \partial w$ 。因此，为了随机梯度下降的规范化学习规则就变成（参考方程(20)）

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

其中后面一项是对 minibatch 中的训练样本 x 进行求和，而 C_x 是对每个训练样本的（无规范化的）代价。这其实和之前通常的随机梯度下降的规则是一样的，除了有一个权重下降的因子 $1 - \frac{\eta \lambda}{n}$ 。最后，为了完备性，我给出偏差的规范化学习规则。这当然是和我们之前的非规范化的情形一致了（参考公式(32)）

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

这里也是对 minibatch 中的训练样本 x 进行求和的。

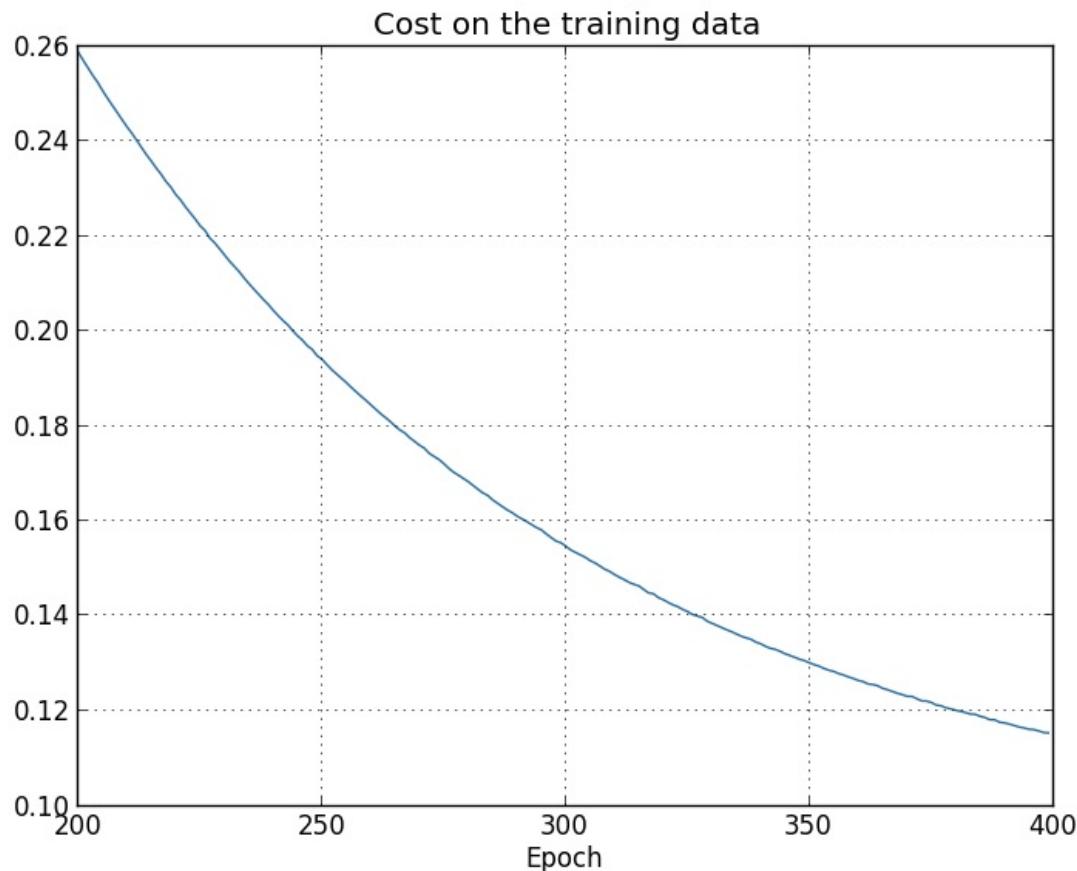
让我们看看规范化给网络带来的性能提升吧。这里还会使用有 30 个隐藏神经元、minibatch 为 10，学习率为 0.5，使用交叉熵的神经网络。然而，这次我们会使用规范化参数为 $\lambda = 0.1$ 。注意在代码中，我们使用的变量名字为 `lmbda`，这是因为在 Python 中 `lambda` 是关键字，尤其特定的作用。我也会使用 `test_data`，而不是 `validation_data`。不过严格地讲，我们应当使用 `validation_data` 的，因为前面已经讲过了。这里我这样做，是因为这会让结果和非规范化结果对比起来效果更加直接。你可以轻松地调整为 `validation_data`，你会发现有相似的结果。

null

null

```
>>> 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)
```

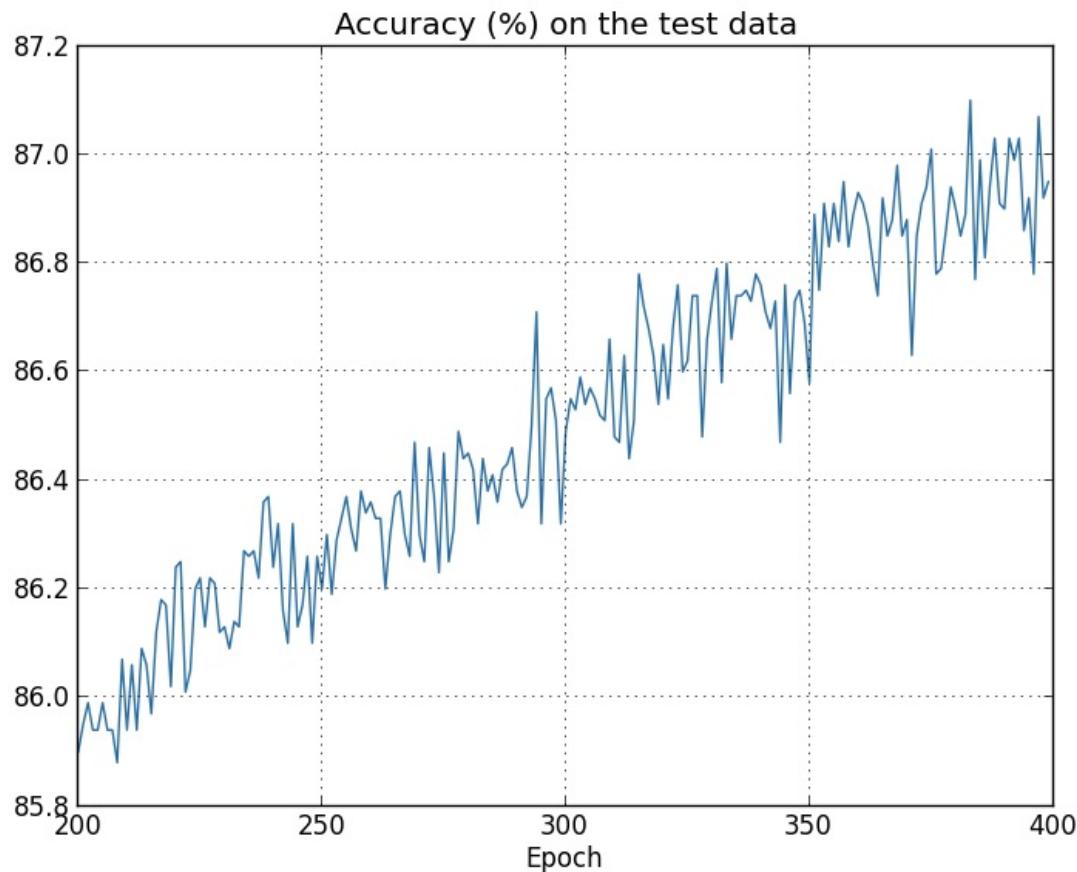
训练集上的代价函数持续下降，和前面无规范化的情况一样的规律：



但是这里测试集上的准确度是随着回合次数持续增加的：

null

null



显然，规范化的使用能够解决过匹配的问题。而且，准确度相当搞了，最高处达到了 87.1%，相较于之前的 82.27%。因此，我们几乎可以确信持续训练会有更加好的结果。实验起来，规范化让网络具有更好的泛化能力，显著地减轻了过匹配的效果。

如果我们换成全部的训练数据进行训练呢？当然，我们之前已经看到过匹配在大规模的数据上其实不是那么明显了。那规范化能不能起到相应的作用呢？保持超参数和之前一样。不过我们这里需要改变规范化参数。原因在于训练数据的大小已经从 $n = 1,000$ 改成了 $n = 50,000$ ，这个会改变权重下降因子 $1 - \frac{\eta\lambda}{n}$ 。如果我们持续使用 $\lambda = 0.1$ 就会产生很小的权重下降，因此就将规范化的效果降低很多。我们通过将 $\lambda = 5.0$ 来补偿这种下降。

好了，来训练网络，重新初始化权重：

```
>>> 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：

null

null



这个结果很不错。第一，我们在测试集上的分类准确度在使用规范化后有了提升，从 95.49% 到 96.49%。这是个很大的进步。第二，我们可以看到在训练数据和测试数据上的结果之间的差距也更小了。这仍然是一个大的差距，不过我们已经显著得到了本质上的降低过匹配的进步。

最后，我们看看在使用 100 个隐藏元和规范化参数为 $\lambda = 5.0$ 相应的测试分类准确度。我不会给出详细分析，就为了有趣，来看看我们使用一些技巧（交叉熵函数和 L2 规范化）能够达到多高的准确度。

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

最终在验证集上的准确度达到了 97.92%。这是比 30 个隐藏元的较大飞跃。实际上，稍微改变一点，60 回合 $\eta = 0.1$ 和 $\lambda = 5.0$ 。我们就突破了 98%，达到了 98.04% 的分类准确度。对于 152 行代码这个效果还真不错！

我们讨论了作为一种减轻过匹配和提高分类准确度的方式的规范化技术。实际上，这不是仅有好处。实践表明，在使用不同的（随机）权重初始化进行多次 MNIST 网络训练的时候，我发现无规范化的网络会偶然被限制住，明显困在了代价函数的局部最优值处。结果就是不同的运行会给出相差很大的结果。对比看来，规范化的网络能够提供更容易复制的结果。

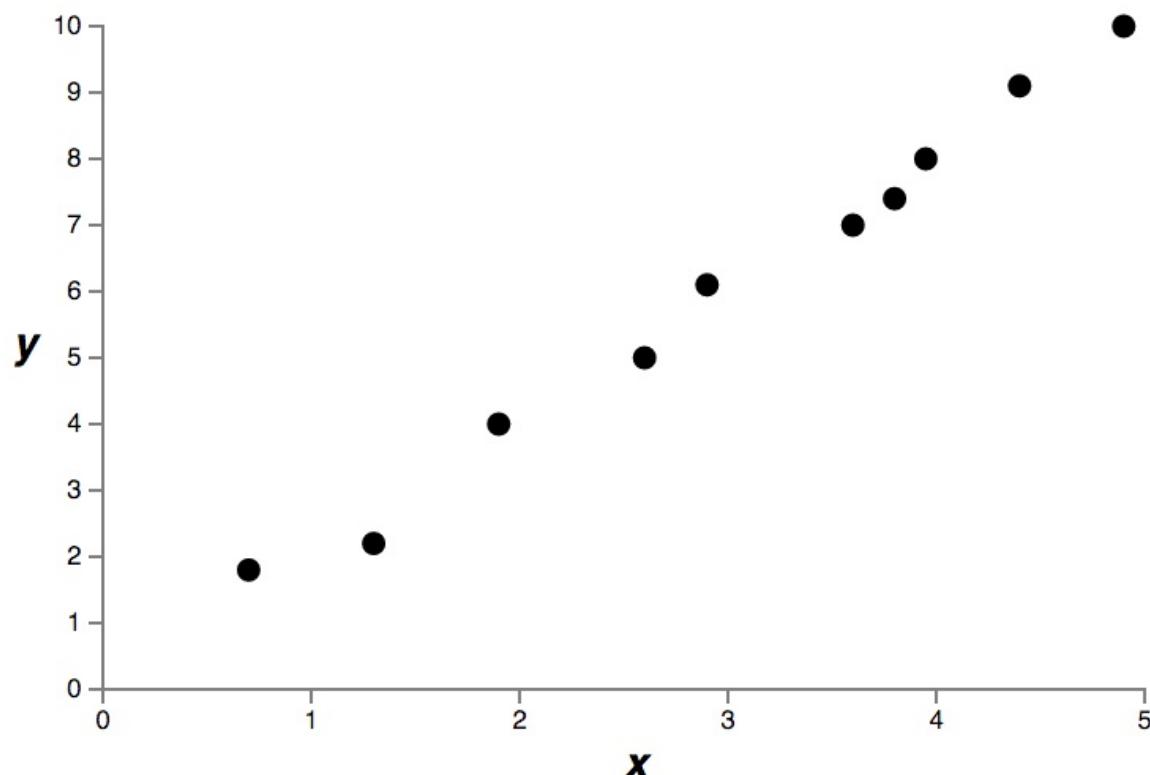
null

null

为何会这样子？从经验上看，如果代价函数是无规范化的，那么权重向量的长度可能会增长，而其他的东西都保持一样。随着时间的推移，这个会导致权重向量变得非常大。所以会使得权重向困在差不多方向上，因为由于梯度下降的改变当长度很大的时候仅仅会在那个方向发生微小的变化。我相信这个现象让学习算法更难有效地探索权重空间，最终导致很难找到代价函数的最优值。

为何规范化可以帮助减轻过匹配

我们已经看到了规范化在实践中能够减少过匹配了。这是令人振奋的，不过，这背后的原因还不得而知！通常的说法是：小的权重在某种程度上，意味着更低的复杂性，也就给出了一种更简单却更强大的数据解释，因此应该优先选择。这虽然很简短，不过暗藏了一些可能看起来会令人困惑的因素。让我们将这个解释细化，认真地研究一下。现在给一个简单的数据集，我们为其建立模型：



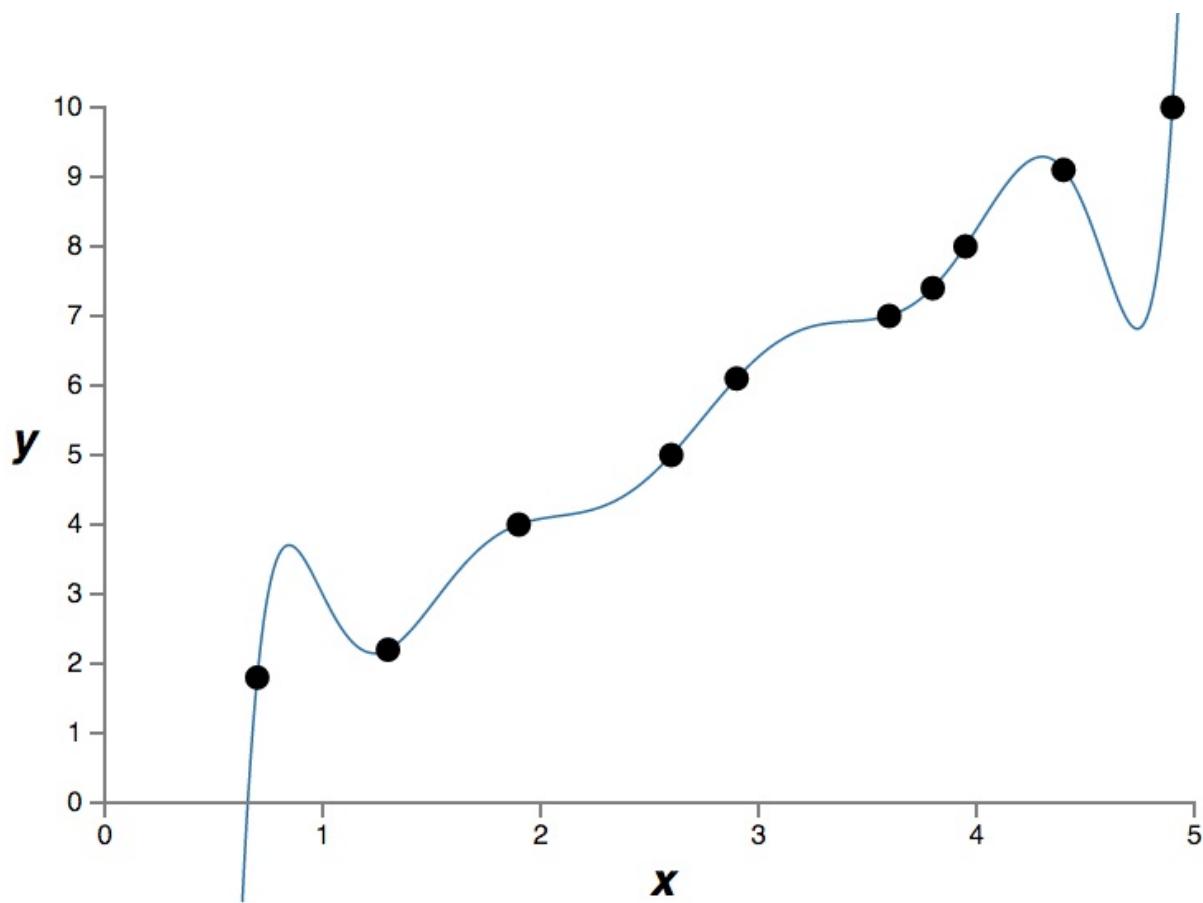
这里我们其实在研究某种真实的现象， x 和 y 表示真实的数据。我们的目标是训练一个模型来预测 y 关于 x 的函数。我们可以使用神经网络来构建这个模型，但是我们先来个简单的：用一个多项式来拟合数据。这样做的原因其实是多项式相比神经网络能够让事情变得更加清楚。一旦我们理解了多项式的场景，对于神经网络可以如法炮制。现在，图中有十个点，我们就可以找到唯一的 9 阶多项式 $y = a_0 x^9 + a_1 x^8 + \dots + a_9$ 来完全拟合数据。下面是多项式的图像：

I won't show the coefficients explicitly, although they are easy to find using a routine such as Numpy's `polyfit`. You can view the exact form of the polynomial in the [source code for the graph](#) if you're curious. It's the function `p(x)` defined starting on line 14 of

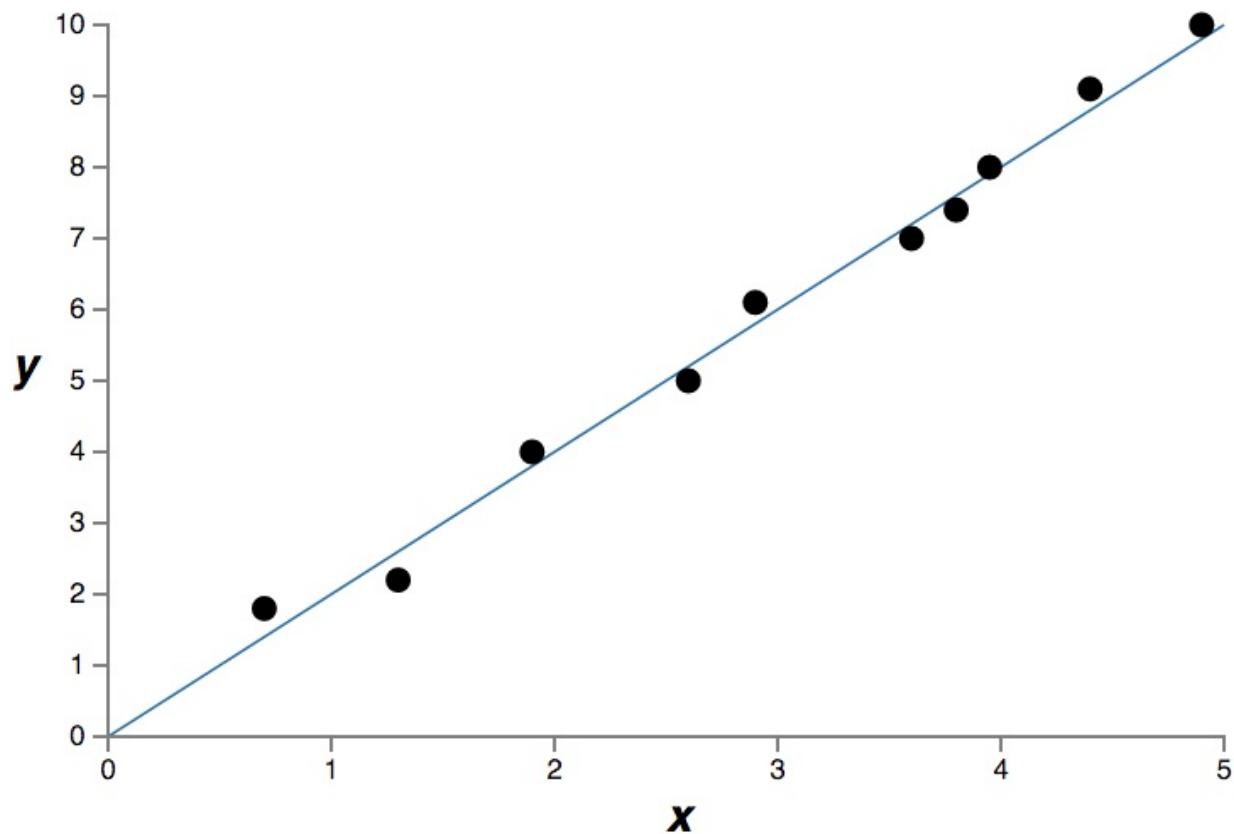
null

null

the program which produces the graph.



这给出了一个完美的拟合。但是我们同样也能够使用线性模型 $y = 2x$ 得到一个好的拟合效果：



null

null

哪个是更好的模型？哪个更可能是真的？还有哪个模型更可能泛化到其他的拥有同样现象的样本上？

这些都是很难回答的问题。实际上，我们如果没有关于现象背后的信息的话，并不能确定给出上面任何一个问题的答案。但是让我们考虑两种可能的情况：(1) 9 阶多项式实际上是完全描述了真实情况的模型，最终它能够很好地泛化；(2) 正确的模型是 $y = 2x$ ，但是存在着由于测量误差导致的额外的噪声，使得模型不能够准确拟合。

先验假设无法说出哪个是正确的（或者，如果还有其他的情况出现）。逻辑上讲，这些都可能出现。并且这不是易见的差异。在给出的数据上，两个模型的表现其实是差不多的。但是假设我们想要预测对应于某个超过了图中所有的 x 的 y 的值，在两个模型给出的结果之间肯定有一个极大的差距，因为 9 阶多项式模型肯定会被 x^9 主导，而线性模型只是线性的增长。

在科学中，一种观点是我们除非不得已应该追随更简单的解释。当我们找到一个简单模型似乎能够解释很多数据样本的时候，我们都会激动地认为发现了规律！总之，这看起来简单的解决仅仅会是偶然出现的不大可能。我们怀疑模型必须表达出某些关于现象的内在的真理。如上面的例子，线性模型加噪声肯定比多项式更加可能。所以如果简单性是偶然出现的话就很令人诧异。因此我们会认为线性模型加噪声表达除了一些潜在的真理。从这个角度看，多项式模型仅仅是学习到了局部噪声的影响效果。所以尽管多是对于这些特定的数据点表现得很好。模型最终会在未知数据上的泛化上出现问题，所以噪声线性模型具有更强大的预测能力。

让我们从这个观点来看神经网络。假设神经网络大多数有很小的权重，这最可能出现在规范化的网络中。更小的权重意味着网络的行为不会因为我们随便改变了一个输入而改变太大。这会让规范化网络学习局部噪声的影响更加困难。将它看做是一种让单个的证据不会影响网络输出太多的方式。相对的，规范化网络学习去对整个训练集中经常出现的证据进行反应。对比看，大权重的网络可能会因为输入的微小改变而产生比较大的行为改变。所以一个无规范化的网络可以使用大的权重来学习包含训练数据中的噪声的大量信息的复杂模型。简言之，规范化网络受限于根据训练数据中常见的模式来构造相对简单的模型，而能够抵抗训练数据中的噪声的特性影响。我们的想法就是这可以让我们的网络对看到的现象进行真实的学习，并能够根据已经学到的知识更好地进行泛化。

所以，倾向于更简单的解释的想法其实会让我们觉得紧张。人们有时候将这个想法称为“奥卡姆剃刀原则”，然后就会热情地将其当成某种科学原理来应用这个法则。但是，这就不是一个一般的科学原理。也没有任何先验的逻辑原因来说明简单的解释就比更为负责的解释要好。实际上，有时候更加复杂的解释其实是正确的。

让我介绍两个说明复杂正确的例子。在 1940 年代，物理学家 Marcel Schein 发布了一个发现新粒子的声明。而他工作的公司，GE，非常欢喜，就广泛地推广这个发现。但是物理学及 Hans Bethe 就有怀疑。Bethe 访问了 Schein，看着那些展示 Schein 的新粒子的轨迹的盘子。但是在每个 plate 上，Bethe 都发现了某个说明数据需要被去除的问题。最后 Schein 展示给 Bethe 一个看起来很好的 plate。Bethe 说这可能就是一个统计上的侥幸。Schein 说，“使得，但是这个可能就是统计学，甚至是根据你自己的公式，也就是 $1/5$ 的概率。” Bethe 说：“但我们可以看过了这 5 个 plate 了”。最终，Schein 说：“但是在我的 plate 中，每

null

null

个好的plate，每个好的场景，你使用了不同的理论（说它们是新的粒子）进行解释，而我只有一种假设来解释所有的 plate。”Bethe 回答说，“在你和我的解释之间的唯一差别就是你是错的，而我所有的观点是正确的。你单一的解释是错误的，我的多重解释所有都是正确的。”后续的工作证实了，Bethe 的想法是正确的而 Schein 粒子不再正确。

注意：这一段翻译得很不好，请参考原文

第二个例子，在 1859 年，天文学家 Urbain Le Verrier 观察到水星并没有按照牛顿万有引力给出的轨迹进行运转。与牛顿力学只有很小的偏差，那时候一些解释就是牛顿力学需要一些微小的改动了。在 1916 年爱因斯坦证明偏差用他的广义相对论可以解释得更好，这是一种和牛顿重力体系相差很大的理论，基于更复杂的数学。尽管引入了更多的复杂性，现如今爱因斯坦的解释其实是正确的，而牛顿力学即使加入一些调整，仍旧是错误的。这部分因为我们知道爱因斯坦的理论不仅仅解释了这个问题，还有很多其他牛顿力学无法解释的问题也能够完美解释。另外，令人印象深刻的是，爱因斯坦的理论准确地给出了一些牛顿力学没能够预测到的显现。但是这些令人印象深刻的现像其实在先前的时代是观测不到的。如果一个人仅仅通过简单性作为判断合理模型的基础，那么一些牛顿力学的改进理论可能会看起来更加合理一些。

从这些故事中可以读出三点。第一，确定两种解释中哪个“更加简单”其实是一件相当微妙的工作。第二，即使我们可以做出这样一个判断，简单性也是一个使用时需要相当小心的指导！第三，对模型真正的测试不是简单性，而是它在新场景中对新的活动中的预测能力。

所以，我们应当时时记住这一点，规范化的神经网络常常能够比非规范化的泛化能力更强，这只是一个实验事实 (empirical fact)。所以，本书剩下的内容，我们也会频繁地使用规范化技术。我已经在上面讲过了为何现在还没有一个人能够发展出一整套具有说服力的关于规范化可以帮助网络泛化的理论解释。实际上，研究者们不断地在写自己尝试不同的规范化方法，然后看看哪种表现更好，尝试理解为何不同的观点表现的更好。所以你可以将规范化看做某种任意整合的技术。尽管其效果不错，但我们并没有一套完整的关于所发生情况的理解，仅仅是一些不完备的启发式规则或者经验。

这里也有更深的问题，这个问题也是有关科学的关键问题——我们如何泛化。规范化能够给我们一种计算上的魔力帮助神经网络更好地泛化，但是并不会带来原理上理解的指导，甚至不会告诉我们什么样的观点才是最好的。

这个问题要追溯到 [归纳问题](#)，最先由苏格兰哲学家大卫 休谟在 "An Enquiry Concerning Human Understanding" (1748) 中提出。在现代机器学习领域中归纳问题被 David Wolpert 和 William Macready 描述成 [无免费午餐定理](#)。

这实在是令人困扰，因为在日常生活中，我们人类在泛化上表现很好。给一个儿童几幅大象的图片，他就能快速地学会认识其他的大象。当然，他们偶尔也会搞错，很可能将一只犀牛误认为大象，但是一般说来，这个过程会相当准确。所以我们有个系统——人的大脑——拥有超大量的自由变量。在受到仅仅少量的训练图像后，系统学会了在其他图像的推广。某种程度上，我们的大脑的规范化做得特别好！怎么做的？现在还不得而知。我期望若干年后，我们能够发展出更加强大的技术来规范化神经网络，最终这些技术会让神经网络甚至在小的训练集上也能够学到强大的泛化能力。

null

null

实际上，我们的网络已经比我们预先期望的要好一些了。拥有 100 个隐藏元的网络会有接近 80,000 个参数。我们的训练集仅有 50,000 幅图像。这好像是用一个 80,000 阶的多项式来拟合 50,000 个数据点。我们的网络肯定会过匹配得很严重。但是，这样的网络实际上却泛化得很好。为什么？这一点并没有很好滴理解。这里有个猜想：梯度下降学习的动态有一种自规范化的效应。这真是一个意料之外的巧合，但也带来了对于这种现象本质无知的不安。不过，我们还是会在后面依照这种实践的观点来应用规范化技术的。神经网络也是由于这点表现才更好一些。

现在我们回到前面留下来的一个细节：L2 规范化没有限制偏差，以此作为本节的结论。当然了，对规范化的过程稍作调整就可以对偏差进行规范了。实践看来，做出这样的调整并不会对结果改变太多，所以，在某种程度上，对不对偏差进行规范化其实就是一种习惯了。然而，需要注意的是，有一个大的偏差并不会像大的权重那样会让神经元对输入太过敏感。所以我们不需要对大的偏差所带来的学习训练数据的噪声太过担心。同时，允许大的偏差能够让网络更加灵活——因为，大的偏差让神经元更加容易饱和，这有时候是我们所要达到的效果。所以，我们通常不会对偏差进行规范化。

规范化的其他技术

除了 L2 外还有很多规范化技术。实际上，正是由于数量众多，我这里也不回将所有的都列举出来。在本节，我简要地给出三种减轻过匹配的其他的方法：L1 规范化、dropout 和人工增加训练样本。我们不会像上面介绍得那么深入。其实，目的只是想让读者熟悉这些主要的思想，然后来体会一下规范化技术的多样性。

L1 规范化：这个方法其实是在代价函数上加上一个权重绝对值的和：

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|. \quad (95)$$

直觉上看，这和 L2 规范化相似，惩罚大的权重，倾向于让网络的权重变小。当然，L1 规范化和 L2 规范化并不相同，所以我们不应该期望 L1 规范化是进行同样的行为。让我们来看看试着理解使用 L1 规范化和 L2 规范化所不同的地方。

首先，我们会研究一下代价函数的偏导数。对(95)求导我们有：

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} \operatorname{sgn}(w), \quad (96)$$

其中 $\operatorname{sgn}(w)$ 就是 w 的正负号。使用这个表达式，我们可以轻易地对反向传播进行修改从而使用基于 L1 规范化的随机梯度下降进行学习。对 L1 规范化的网络进行更新的规则就是

null

null

$$w \rightarrow w' = w - \frac{\eta\lambda}{n} \operatorname{sgn}(w) - \eta \frac{\partial C_0}{\partial w}, \quad (97)$$

其中和往常一样，我们可以用 minibatch 的均值来估计 $\partial C_0 / \partial w$ 。对比公式(93)的 L2 规范化，

$$w \rightarrow w' = w \left(1 - \frac{\eta\lambda}{n} \right) - \eta \frac{\partial C_0}{\partial w}. \quad (98)$$

在两种情形下，规范化的效果就是缩小权重。这和我们想要让权重不会太大的直觉目标相符。在 L1 规范化中，权重按照一个接近 0 的常量进行缩小。在 L2 规范化中，权重同按照一个和 w 成比例的量进行缩小的。所以，当一个特定的权重绝对值 $|w|$ 很大时，L1 规范化缩小得远比 L2 规范化要小得多。而一个特定的权重绝对值 $|w|$ 很小时，L1 规范化权值要比 L2 规范化缩小得更大。最终的结果就是：L1 规范化倾向于聚集网络的权重在相对少量的高重要度连接上，而其他权重就会被驱使向 0 接近。

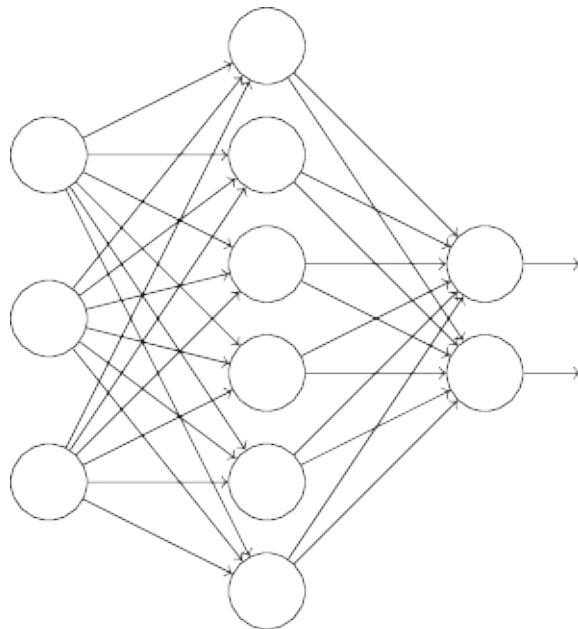
我在上面的讨论中其实忽略了一个问题——在 $w = 0$ 的时候，偏导数 $\partial C / \partial w$ 未定义。原因在于函数 $|w|$ 在 $w = 0$ 时有个直角，事实上，导数是不存在的。不过也没有关系。我们下面要做的就是应用无规范化的通常的梯度下降的规则在 $w = 0$ 处。这应该不会有什问题，直觉上看，规范化效果就是缩小权重，显然，不能对一个已经是 0 的权重进行缩小了。更准确地说，我们将会使用方程(96)(97)并约定 $\operatorname{sgn}(0) = 0$ 。这样就给出了一种精致的规则来进行采用 L1 规范化的随机梯度下降学习。

Dropout : Dropout 是一种相当激进的技术。和 L1、L2 规范化不同，dropout 并不依赖对代价函数的变更。而是，在 dropout 中，我们改变了网络本身。让我在给出为何工作的原理之前描述一下 dropout 基本的工作机制和所得到的结果。

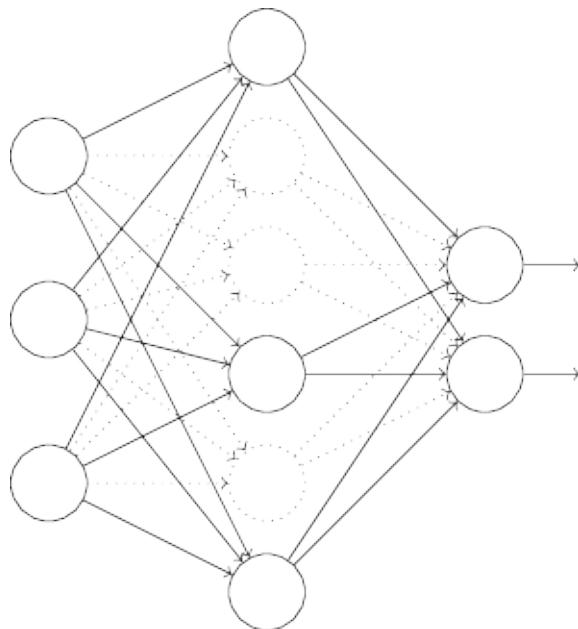
假设我们尝试训练一个网络：

null

null



特别地，假设我们有一个训练数据 x 和对应的目标输出 y 。通常我们会通过在网络中前向传播 x ，然后进行反向传播来确定对梯度的共现。使用 dropout，这个过程就改了。我们会从随机（临时）地删除网络中的一半的神经元开始，让输入层和输出层的神经元保持不变。在此之后，我们会得到最终的网络。注意那些被 dropout 的神经元，即那些临时性删除的神经元，用虚圈表示在途中：



我们前向传播输入，通过修改后的网络，然后反向传播结果，同样通过这个修改后的网络。在 minibatch 的若干样本上进行这些步骤后，我们对那些权重和偏差进行更新。然后重复这个过程，首先重置 dropout 的神经元，然后选择新的随机隐藏元的子集进行删除，估计对一个不同的minibatch的梯度，然后更新权重和偏差。

通过不断地重复，我们的网络会学到一个权重和偏差的集合。当然，这些权重和偏差也是在一般的隐藏元被丢弃的情形下学到的。当我们实际运行整个网络时，是指两倍的隐藏元将会被激活。为了补偿这个，我们将从隐藏元出去的权重减半了。

这个 dropout 过程可能看起来奇怪和ad hoc。为什么我们期待这样的方法能够进行规范化

null

null

呢？为了解释所发生的事，我希望你停下来想一下没有 dropout 的训练方式。特别地，想象一下我们训练几个不同的神经网络，使用的同一个训练数据。当然，网络可能不是从同一初始状态开始的，最终的结果也会有一些差异。出现这种情况时，我们可以使用一些平均或者投票的方式来确定接受哪个输出。例如，如果我们训练了五个网络，其中三个被分类当做是 3，那很可能它就是 3。另外两个可能就犯了错误。这种平均的方式通常是一种强大（尽管代价昂贵）的方式来减轻过匹配。原因在于不同的网络可能会以不同的方式过匹配，平均法可能会帮助我们消除那样的过匹配。

那么这和 dropout 有什么关系呢？启发式地看，当我们丢掉不同的神经元集合时，有点像我们在训练不同的神经网络。所以，dropout 过程就如同大量不同网络的效果的平均那样。不同的网络以不同的方式过匹配了，所以，dropout 的网络会减轻过匹配。

一个相关的启发式解释在早期使用这项技术的论文中曾经给出：“因为神经元不能依赖其他神经元特定的存在，这个技术其实减少了复杂的互适应的神经元。所以，强制要学习那些在神经元的不同随机子集中更加健壮的特征。”换言之，如果我们就爱那个神经网络看做一个进行预测的模型的话，我们就可以将 dropout 看做是一种确保模型对于证据丢失健壮的方式。这样看来，dropout 和 L1、L2 规范化也是有相似之处的，这也倾向于更小的权重，最后让网络对丢失个体连接的场景更加健壮。

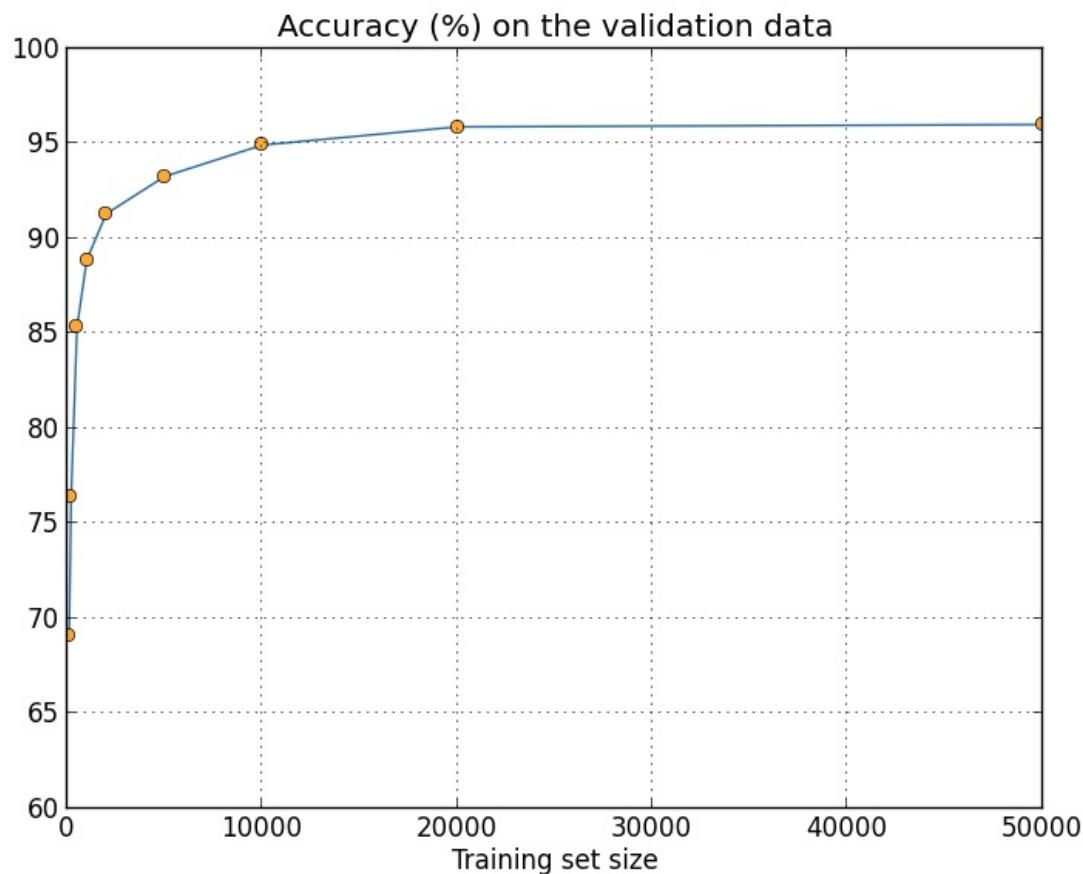
当然，真正衡量 dropout 的方式在提升神经网络性能上应用得相当成功。原始论文介绍了用来解决很多不同问题的技术。对我们来说，特别感兴趣的是他们应用 dropout 在 MNIST 数字分类上，使用了一个和我们之前介绍的那种初级的前向神经网络。这篇文章关注到最好的结果是在测试集上去得到 98.4% 的准确度。他们使用 dropout 和 L2 规范化的组合将其提高到了 98.7%。类似重要的结果在其他不同的任务上也取得了一定的功效。dropout 已经在过匹配问题尤其突出的训练大规模深度网络中。

人工扩展训练数据：我们前面看到了 MNIST 分类准确度在我们使用 1,000 幅训练图像时候下降到了 80 年代的准确度。这种情况并不奇怪，因为更少的训练数据意味着我们的网络所接触到较少的人类手写的数字中的变化。让我们训练 30 个隐藏元的网络，使用不同的训练数据集，来看看性能的变化情况。我们使用 minibatch 大小为 10，学习率是 $\eta = 0.5$ ，规范化参数是 $\lambda = 5.0$ ，交叉熵代价函数。我们在全部训练数据集合上训练 30 个回合，然后会随着训练数据量的下降而成比例变化回合数。为了确保权重下降因子在训练数据集上相同，我们会在全部训练集上使用规范化参数为 $\lambda = 5.0$ ，然后在使用更小的训练集的时候成比例地下降 λ 值。

This and the next two graph are produced with the program [more_data.py](#).

null

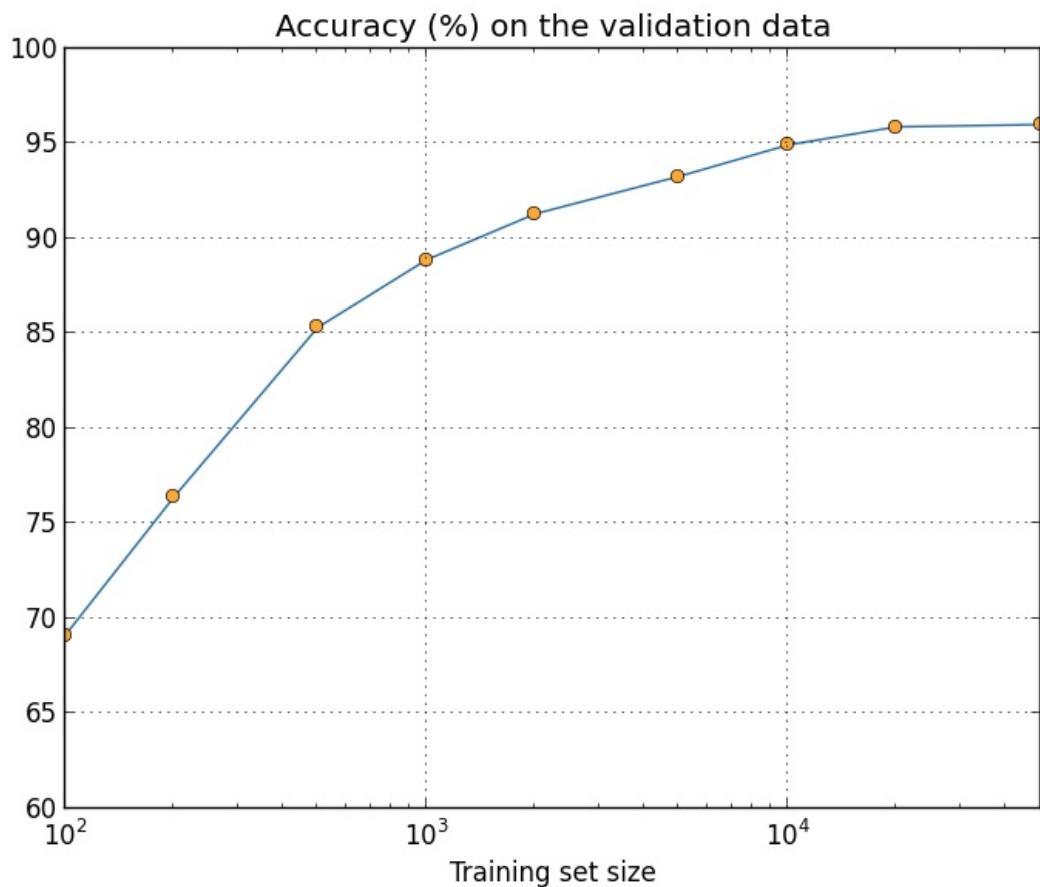
null



如你所见，分类准确度在使用更多的训练数据时提升了很大。根据这个趋势的话，提升会随着更多的数据而不断增加。当然，在训练的后期我们看到学习过程已经进入了饱和状态。然而，如果我们使用对数作为横坐标的话，可以重画此图如下：

null

null

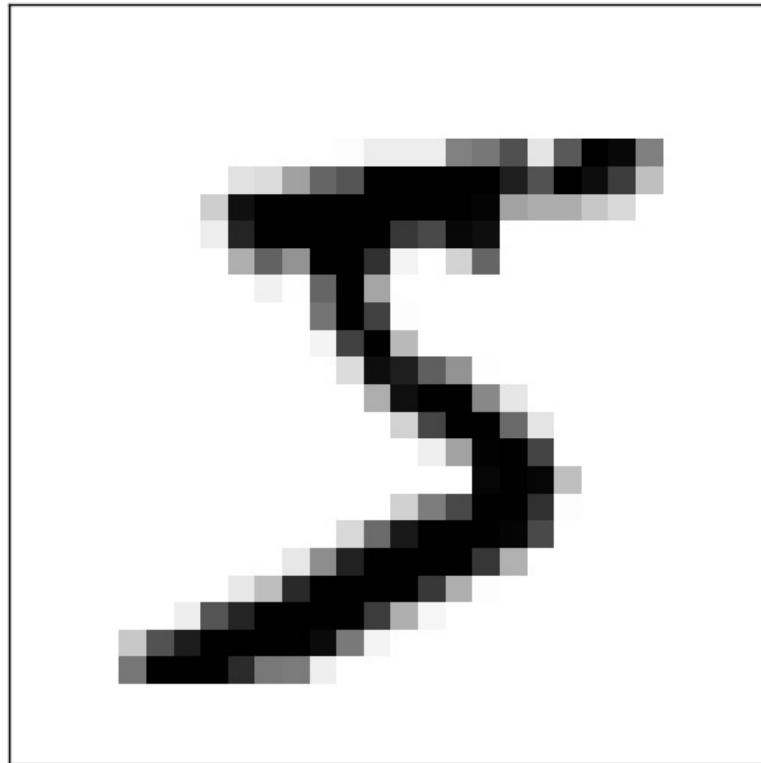


这看起来到了后面结束的地方，增加仍旧明显。这表明如果我们使用大量更多的训练数据——不妨设百万或者十亿级的手写样本——那么，我们可能会得到更好的性能，即使是用这样的简单网络。

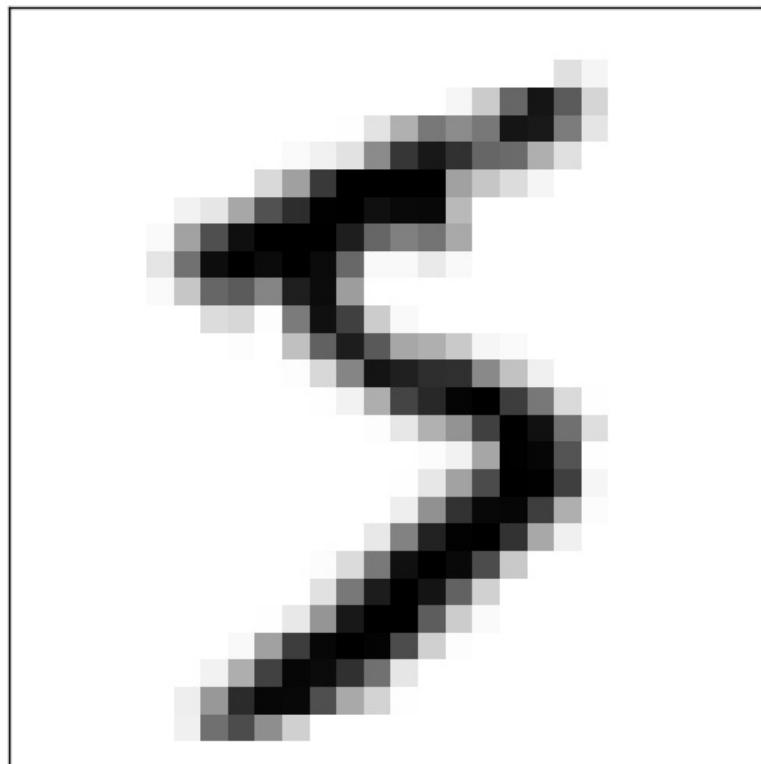
获取更多的训练样本其实是很重要的想法。不幸的是，这个方法代价很大，在实践中常常是很难达到的。不过，还有一种方法能够获得类似的效果，那就是进行人工的样本扩展。假设我们使用一个 5 的训练样本，

null

null



将其进行旋转，比如说 15° ：



null

null

这还是会被识别为同样的数字的。但是在像素层级这和任何一幅在 MNIST 训练数据中的图像都不相同。所以将这样的样本加入到训练数据中是很可能帮助我们学习有关手写数字更多知识的方法。而且，显然，我们不会就只对这幅图进行人工的改造。我们可以在所有的 MNIST 训练样本上通过和多小的旋转扩展训练数据，然后使用扩展后的训练数据来提升我们网络的性能。

这个想法非常强大并且已经被广泛应用了。让我们看看一些在 MNIST 上使用了类似的方法进行研究成果。其中一种他们考虑的网络结构其实和我们已经使用过的类似——一个拥有 800 个隐藏元的前驱神经网络，使用了交叉熵代价函数。在标准的 MNIST 训练数据上运行这个网络，得到了 98.4% 的分类准确度，其中使用了不只是旋转还包括转换和扭曲。通过在这个扩展后的数据集上的训练，他们提升到了 98.9% 的准确度。然后还在“弹性扭曲 (elastic distortion) ”的数据上进行了实验，这是一种特殊的为了模仿手部肌肉的随机抖动的图像扭曲方法。通过使用弹性扭曲扩展的数据，他们最终达到了 99.3% 的分类准确度。他们通过展示训练数据的所有类型的变体来扩展了网络的经验。

[Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis](#),
by Patrice Simard, Dave Steinkraus, and John Platt (2003).

这个想法的变体也可以用在提升手写数字识别之外不同学习任务上的性能。一般就是通过应用反映真实世界变化的操作来扩展训练数据。找到这些方法其实并不困难。例如，你要构建一个神经网络来进行语音识别。我们人类甚至可以在有背景噪声的情况下识别语音。所以你可以通过增加背景噪声来扩展你的训练数据。我们同样能够对其进行加速和减速来获得相应的扩展数据。所以这是另外的一些扩展训练数据的方法。这些技术并不总是有用——例如，其实与其在数据中加入噪声，倒不如先对数据进行噪声的清理，这样可能更加有效。当然，记住可以进行数据的扩展，寻求应用的机会还是相当有价值的一件事。

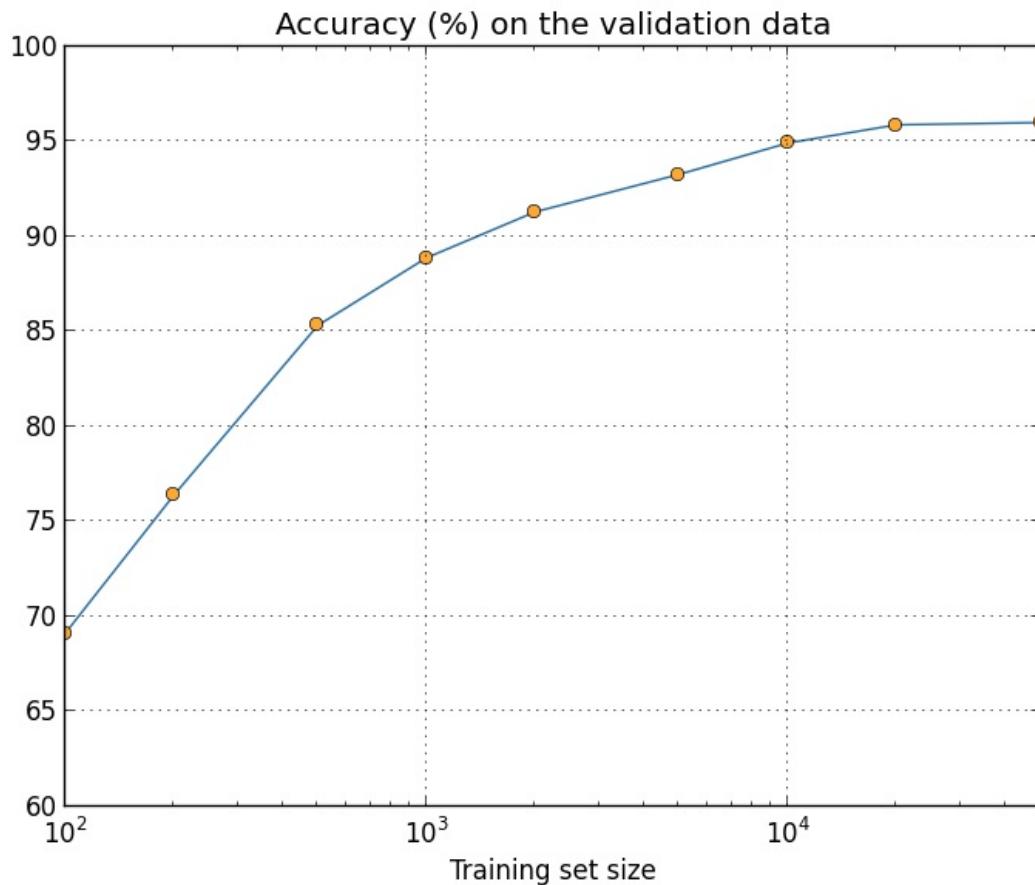
练习

- 正如上面讨论的那样，一种扩展 MNIST 训练数据的方式是用一些微小的旋转。如果我们允许过大的旋转，则会出现什么状况呢？

大数据的旁白和对分类准确度的影响：让我们看看神经网络准确度随着训练集大小变化的情况：

null

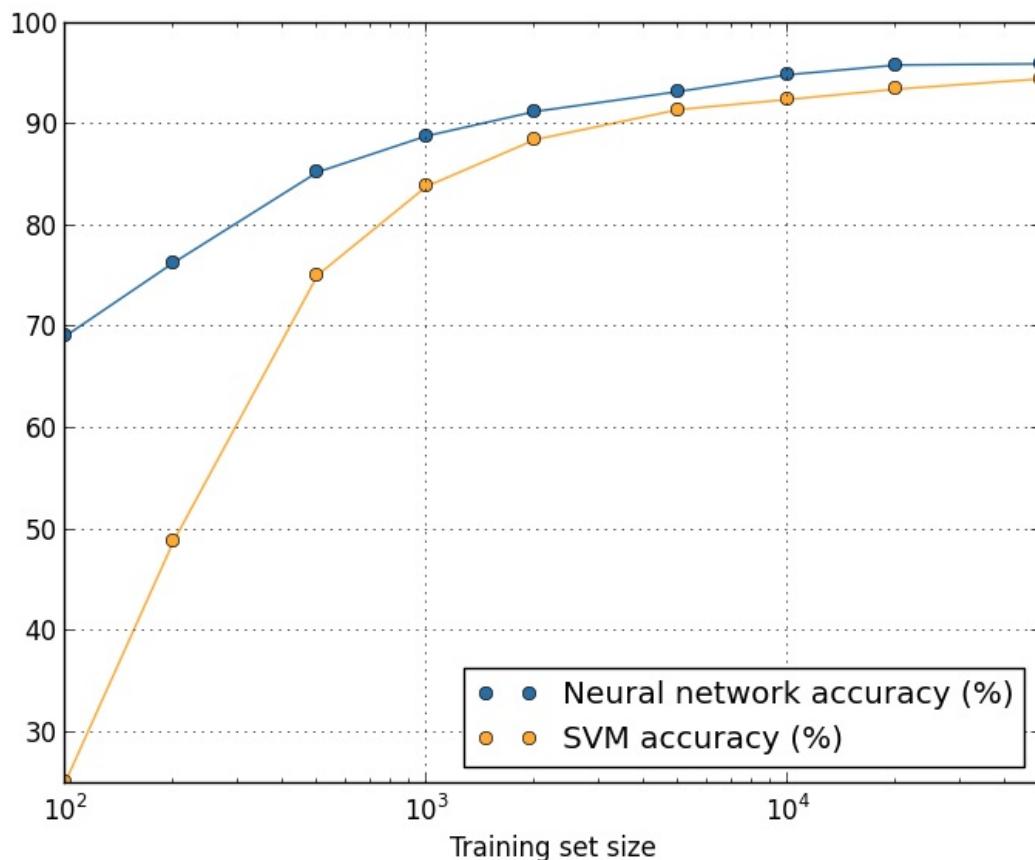
null



假设，我们使用别的什么方法来进行分类。例如，我们使用 SVM。正如第一章介绍的那样，不要担心你熟不熟悉 SVM，我们不进行深入的讨论。下面是 SVM 模型的准确度随着训练数据集的大小变化的情况：

null

null



可能第一件让你吃惊的是神经网络在每个训练规模下性能都超过了 SVM。这很好，尽管你对细节和原理可能不太了解，因为我们只是直接从 scikit-learn 中直接调用了这个方法，而对神经网络已经深入讲解了很多。更加微妙和有趣的现象其实是如果我们训练 SVM 使用 50,000 幅图像，实际上 SVM 已经能够超过我们使用 5,000 幅图像的准确度。换言之，更多的训练数据可以补偿不同的机器学习算法的差距。

还有更加有趣的现象也出现了。假设我们试着用两种机器学习算法去解决问题，算法 A 和算法 B。有时候出现，算法 A 在一个训练集合上超过 算法 B，却在另一个训练集上弱于算法 B。上面我们并没有看到这个情况——因为这要求两幅图有交叉的点——这里并没有。对“算法 A 是不是要比算法 B 好？”正确的反应该是“你在使用什么训练集合？”

在进行开发时或者在读研究论文时，这都是需要记住的事情。很多论文聚焦在寻找新的技术来给出标准数据集上更好的性能。“我们的超赞的技术在标准测试集 Y 上给出了百分之 X 的性能提升。”这是通常的研究声明。这样的声明通常比较有趣，不过也必须被理解为仅仅在特定的训练数据机上的应用效果。那些给出基准数据集的人们会拥有更大的研究经费支持，这样能够获得更好的训练数据。所以，很可能他们由于超赞的技术的性能提升其实在更大的数据集合上就丧失了。换言之，人们标榜的提升可能就是历史的偶然。所以需要记住的特别是在实际应用中，我们想要的是更好的算法和更好的训练数据。寻找更好的算法很重，不过需要确保你在此过程中，没有放弃对更多更好的数据的追求。

问题

null

null

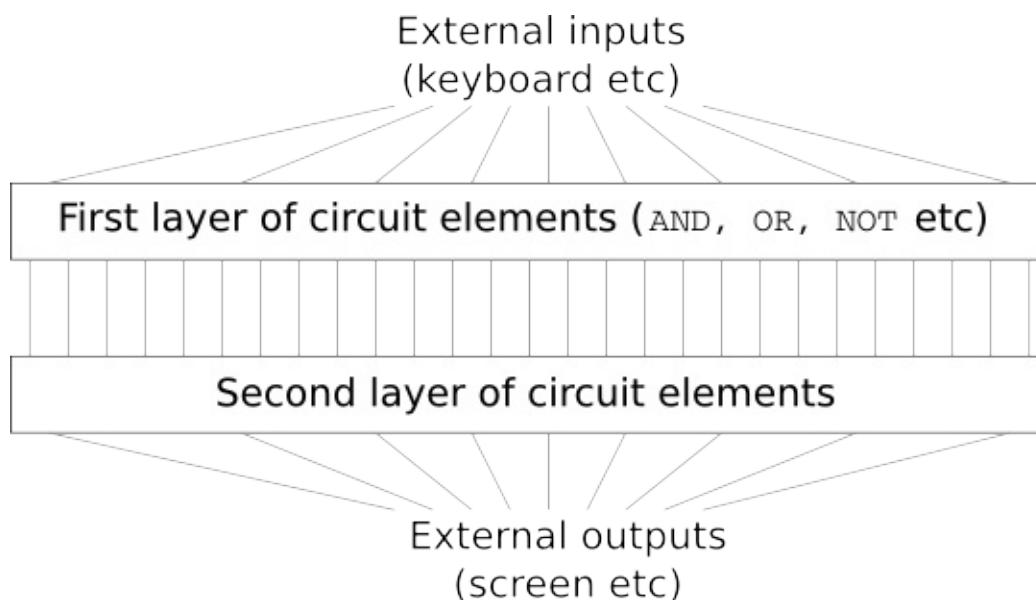
- 研究问题：我们的机器学习算法在非常大的数据集上如何进行？对任何给定的算法，其实去定义一个随着训练数据规模变化的渐近的性能是一种很自然的尝试。一种简单粗暴的方法就是简单地进行上面图中的趋势分析，然后将图像推进到无穷大。而对此想法的反驳是曲线本身会给出不同的渐近性能。你能够找到拟合某些特定类别曲线的理论上的验证方法吗？如果可以，比较不同的机器学习算法的渐近性能。

总结：我们现在已经介绍完了过匹配和规范化。当然，我们重回这个问题。正如我们前面讲过的那样，尤其是计算机越来越强大，我们有训练更大的网络的能力时。过匹配是神经网络中一个主要的问题。我们有迫切的愿望来开发出强大的规范化技术来减轻过匹配，所以，这也是当前研究的极其热门的方向之一。

null

null

假设你是一名工程师，接到一项从头开始设计计算机的任务。某天，你在工作室工作，设计逻辑电路，构建 *AND* 门，*OR* 门等等时，老板带着坏消息进来：客户刚刚添加了一个奇特的设计需求：整个计算机的线路的深度必须只有两层：



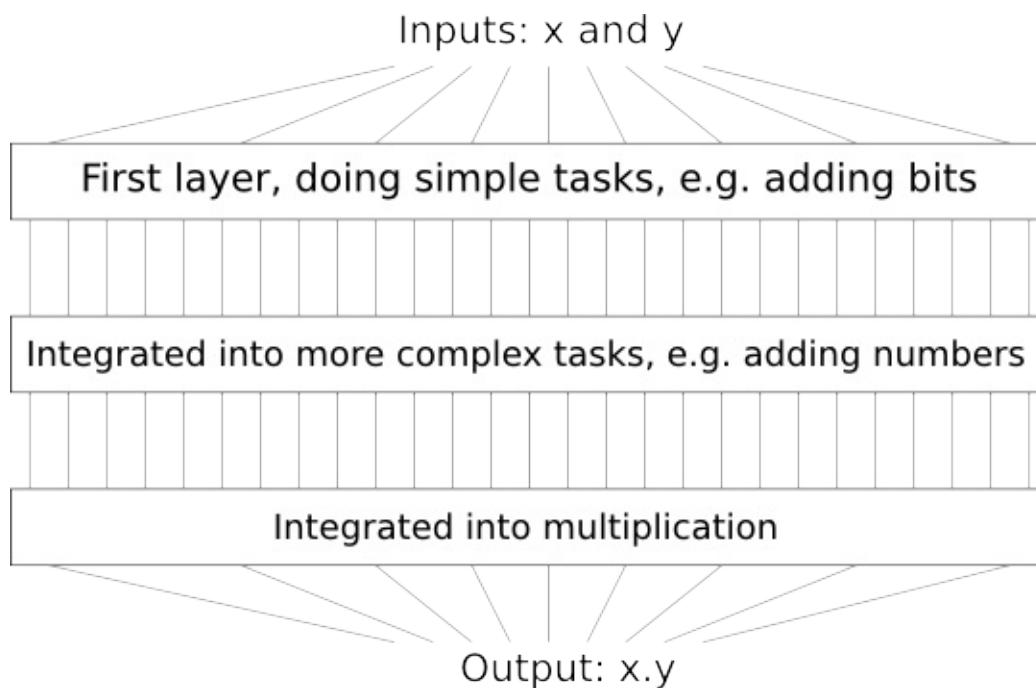
你惊呆了，跟老板说道：“这货疯掉了吧！”

老板说：“他们确实疯了，但是客户的需求比天大，我们要满足它。”

实际上，在某种程度上看，他们的客户并没有太疯狂。假设你可以使用贵重特殊的逻辑门可以 *AND* 起来你想要的那么多的输入。同样也能使用多值输入的 *NAND* 门——可以 *AND* 多个输入然后求否定的门。有了这类特殊的门，构建出来的两层的深度的网络便可以计算任何函数。但是仅仅因为某件事是理论上可能的，就代表这是一个好的想法。在实践中，在解决线路设计问题（或者大多数的其他算法问题）时，我们通常考虑如何解决子问题，然后逐步地集成这些子问题的解。换句话说，我们通过多层的抽象来获得最终的解答。例如，我们来设计一个逻辑线路来做两个数的乘法。我们希望在已经有了计算两个数加法的子线路基础上创建这个逻辑线路。计算两个数和的子线路也是构建在用语两个比特相加的子子线路上的。最终的线路就长成这个样子：

null

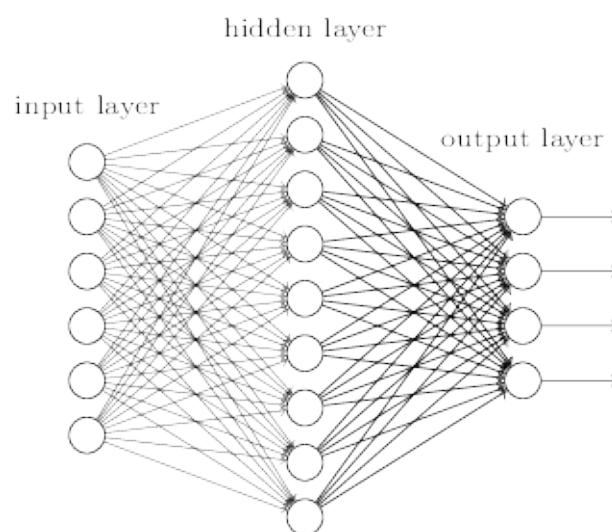
null



最终的线路包含至少三层线路的单元。实际上，这个线路很可能会超过三层，因为我们可以将子任务分解成比上述更小的单元。但是基本思想就是这样。

因此深度线路让这样的设计过程变得更加简单。但是这对于设计本身帮助并不大。其实，数学证明对于某些函数设计的非常浅的线路可能需要指数级的线路单元来计算。例如，在1980年代早期的一系列著名的论文已经给出了计算比特的集合的奇偶性通过浅的线路来计算需要指数级的门。另一当面，如果你使用更深的线路，那么可以使用规模很小的线路来计算奇偶性：仅仅需要计算比特的对的奇偶性，然后使用这些结果来计算比特对的对的奇偶性，以此类推，构建出总共的奇偶性。深度线路这样就能从本质上获得超过浅线路的更强的能力。

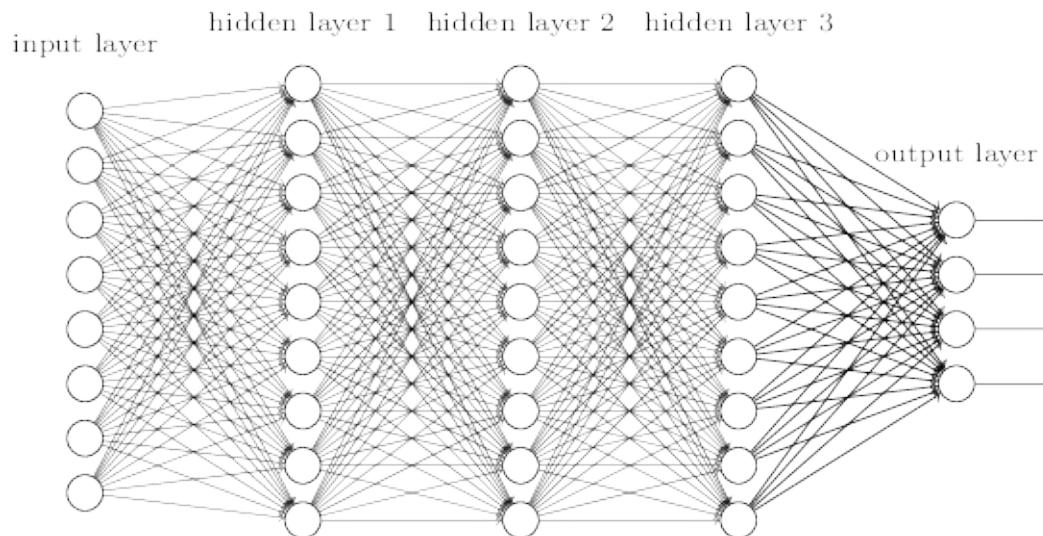
到现在为止，本书讲神经网络看作是疯狂的客户。几乎我们遇到的所有的网络就只包括一层隐含神经元（另外还有输入输出层）：



这些简单的网络已经非常有用了：在前面的章节中，我们使用这样的网络可以进行准确率高达 98% 的手写数字的识别！而且，直觉上看，我们期望拥有更多隐含层的神经网络能够变的更加强大：

null

null



这样的网络可以使用中间层构建出多层的抽象，正如我们在布尔线路中做的那样。例如，如果我们在进行视觉模式识别，那么在第一层的神经元可能学会识别边，在第二层的神经元可以在边的基础上学会识别出更加复杂的形状，例如三角形或者矩形。第三层将能够识别更加复杂的形状。依此类推。这些多层的抽象看起来能够赋予深度网络一种学习解决复杂模式识别问题的能力。然后，正如线路的示例中看到的那样，存在着理论上的研究结果告诉我们深度网络在本质上比浅层网络更加强大。

对某些问题和网络结构，Razvan Pascanu, Guido Montúfar, and Yoshua Bengio 在2014年的这篇文章[On the number of response regions of deep feed forward networks with piece-wise linear activations](#)给出了证明。更加详细的讨论在Yoshua Bengio 2009年的著作[Learning deep architectures for AI](#) 的第二部分。

那我们如何训练这样的深度神经网络呢？在本章中，我们尝试使用基于BP的随机梯度下降的方法来训练。但是这会产生问题，因为我们的深度神经网络并不能比浅层网络性能好太多。

这个失败的结果好像与上面的讨论相悖。这就能让我们退缩么，不，我们要深入进去试着理解使得深度网络训练困难的原因。仔细研究一下，就会发现，在深度网络中，不同的层学习的速度差异很大。尤其是，在网络中后面的层学习的情况很好的时候，先前的层次常常会在训练时停滞不变，基本上学不到东西。这种停滞并不是因为运气不好。而是，有着更加根本的原因是的学习的速度下降了，这些原因和基于梯度的学习技术相关。

当我们更加深入地理解这个问题时，发现相反的情形同样会出现：先前的层可能学习的比较好，但是后面的层却停滞不变。实际上，我们发现在深度神经网络中使用基于梯度下降的学习方法本身存在着内在不稳定性。这种不稳定性使得先前或者后面的层的学习过程阻滞。

这个的确是坏消息。但是真正理解了这些难点后，我们就能够获得高效训练深度网络的更深洞察力。而且这些发现也是下一章的准备知识，我们到时会介绍如何使用深度学习解决图像识别问题。

(消失的恋人，哦不) 消失的梯度问题

null

```
null
```

那么，在我们训练深度网络时究竟哪里出了问题？

为了回答这个问题，让我们重新看看使用单一隐藏层的神经网络示例。这里我们也是用 MNIST 数字分类问题作为研究和实验的对象。

MNIST 问题和数据在 ([这里](#)) 和 ([这里](#)) .

这里你也可以在自己的电脑上训练神经网络。或者就直接读下去。如果希望实际跟随这些步骤，那就需要在电脑上安装 python 2.7, numpy 和代码，可以通过下面的命令复制所需要的代码

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

如果你不使用 *git*，那么就直接从这里 ([here](#)) 下载数据和代码。然后需要转入 *src* 子目录。

接着从 python 的 shell 就可以载入 MNIST 数据：

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

然后设置我们的网络：

```
>>> import network2  
>>> net = network2.Network([784, 30, 10])
```

这个网络拥有 784 个输入层神经元，对应于输入图片的 $28 * 28 = 784$ 个像素点。我们设置隐藏层神经元为 30 个，输出层为 10 个神经元，对应于 MNIST 数字 $(0, 1, \dots, 9)$ 。让我们训练 30 轮，使用 mini batch 大小为 10，学习率 $\eta = 0.1$ ，正规化参数 $\lambda = 5.0$ 。在训练时，我们也会在验证集上监控分类的准确度：

```
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,  
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

最终我们得到了分类的准确率为 96.48（也可能不同，每次运行实际上会有一点点的偏差）这和我们前面的结果相似。现在，我们增加另外一层隐藏层，同样地是 30 个神经元，试着使用相同的超参数进行训练：

```
>>> net = network2.Network([784, 30, 30, 10])  
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,  
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

最终的结果分类准确度提升了一点，96.90。这点令人兴奋：一点点的深度带来了效果。那么就再增加一层同样的隐藏层：

```
null
```

null

```
>>> net = network2.Network([784, 30, 30, 30, 10])
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

哦，这里并没有什么提升，反而下降到了 96.57%，这与最初的浅层网络相差无几。再增加一层：

```
>>> net = network2.Network([784, 30, 30, 30, 30, 10])
>>> net.SGD(training_data, 30, 10, 0.1, lmbda=5.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

分类准确度又下降了，96.53。这可能不是一个统计显著地下降，但是会让人们觉得沮丧。

这里表现出来的现象看起非常奇怪。直觉地，额外的隐藏层应当让网络能够学到更加复杂的分类函数，然后可以在分类时表现得更好吧。可以肯定的是，事情并没有变差，至少新的层次增加上，在最坏的情形下也就是没有影响。事情并不是这样子的。

那么，应该是怎样的呢？假设额外的隐藏层的确能够在原理上起到作用，问题是我们的学习算法没有发现正确地权值和偏差。那么现在就要好好看看学习算法本身有哪里出了问题，并搞清楚如何改进了。

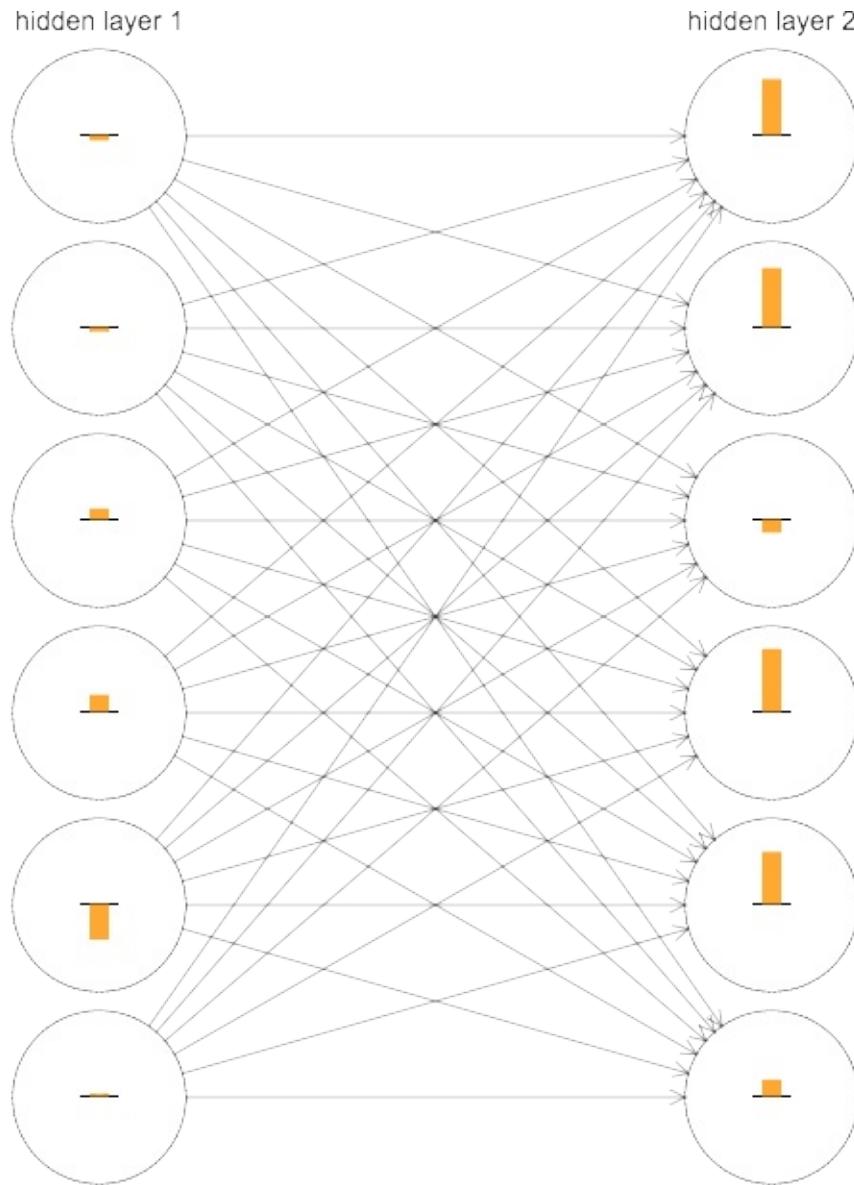
为了获得一些关于这个问题直觉上的洞察，我们可以将网络学到的东西进行可视化。下面，我画出了一部分 [784, 30, 30, 10] 的网络，也就是包含两层各有 30 个隐藏神经元的隐藏层。图中的每个神经元有一个条形统计图，表示这个神经元在网络进行学习时改变的速度。更大的条意味着更快的速度，而小的条则表示变化缓慢。更加准确地说，这些条表示了每个神经元上的 $\frac{\partial C}{\partial b}$ ，也就是代价函数关于神经元的偏差更变的速率。回顾第二章（Chapter 2），我们看到了这个梯度的数值不仅仅是在学习过程中偏差改变的速度，而且也控制了输入到神经元权重的变量速度。如果没有回想起这些细节也不要担心：目前要记住的就是这些条表示了每个神经元权重和偏差在神经网络学习时的变化速率。

为了让图里简单，我只展示出来最上方隐藏层上的 6 个神经元。这里忽略了输入层神经元，因为他们并不包含需要学习的权重或者偏差。同样输出层神经元也忽略了，因为这里我们做的是层层之间的比较，所以比较相同数量的两层更加合理啦。在网络初始化后立即得到训练前期的结果如下：

这个程序给出了计算梯度的方法[generate_gradient.py](#)，也包含了其他一些在本章后面提到的计算方法。

null

null



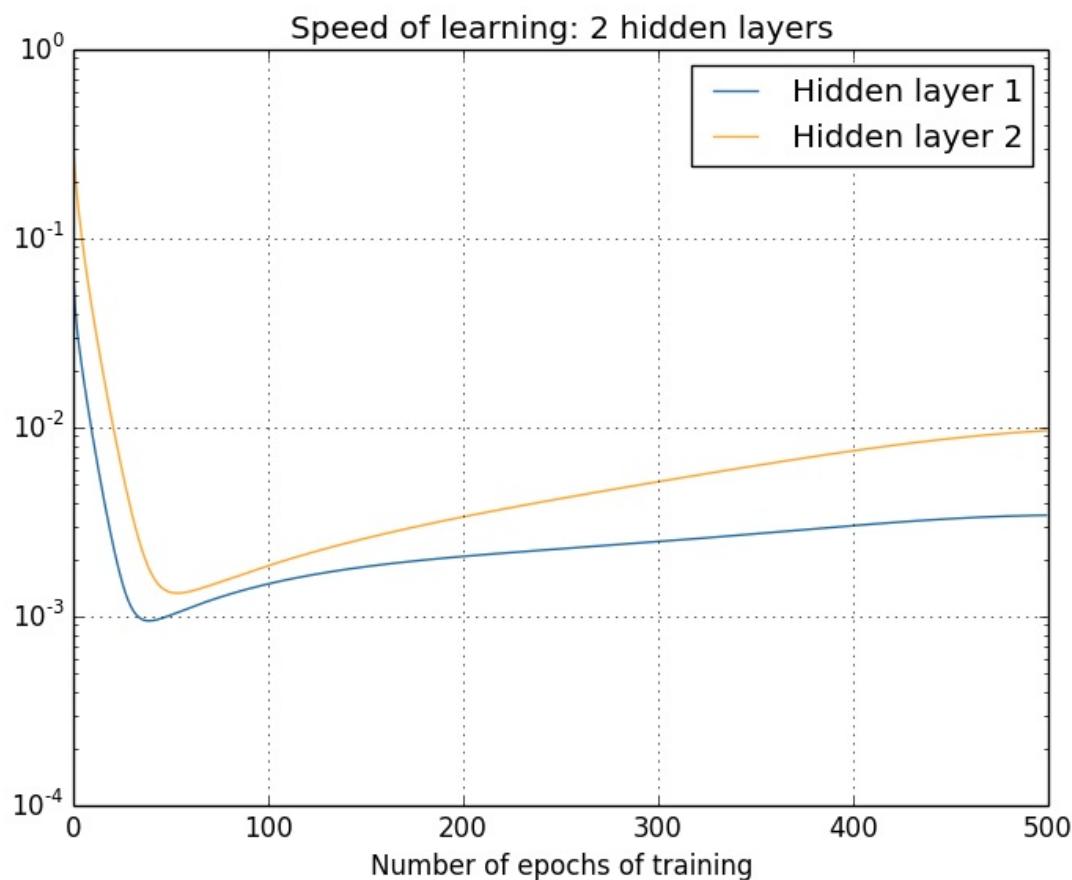
该网络是随机初始化的，因此看到了神经元学习的速度差异其实很大。而且，我们可以发现，第二个隐藏层上的条基本上都要比第一个隐藏层上的条要大。所以，在第二个隐藏层的神经元将学习得更加快速。这仅仅是一个巧合么，或者第二个隐藏层的神经元一般情况下都要比第一个隐藏层的神经元学习得更快？为了确定我们的猜测，拥有一种全局的方式来比较学习速度会比较有效。我们这里将梯度表示为 $\delta_j^l = \partial C / \partial b_j^l$ 在第 l 层的第 j 个神经元的梯度。我们可以将 δ^1 看做是一个向量其中元素表示第一层隐藏层的学习速度， δ^2 则是第二层隐藏层的学习速度。接着使用这些向量的长度作为全局衡量这些隐藏层的学习速度的度量。因此， $\|\delta^1\|$ 就代表第一层隐藏层学习速度，而 $\|\delta^2\|$ 就代表第二层隐藏层学习速度。借助这些定义，在和上图同样的配置下， $\|\delta^1\| = 0.07$ 而 $\|\delta^2\| = 0.31$ ，所以这就确认了之前的疑惑：在第二层隐藏层的神经元学习速度确实比第一层要快。

如果我们添加更多的隐藏层呢？如果我们有三个隐藏层，比如说在一个 $[784, 30, 30, 10]$ 的网络中，那么对应的学习速度就是 $0.012, 0.060, 0.283$ 。这里前面的隐藏层学习速度还是要低于最后的隐藏层。假设我们增加另一个包含 30 个隐藏神经元的隐藏层。那么，对应的学习速度就是： $0.003, 0.017, 0.070, 0.285$ 。还是一样的模式：前面的层学习速度低于后面的层。

null

null

现在我们已经看到了训练开始时的学习速度，这是刚刚初始化之后的情况。那么这个速度会随着训练的推移发生什么样的变化呢？让我们看看只有两个隐藏层。学习速度变化如下：

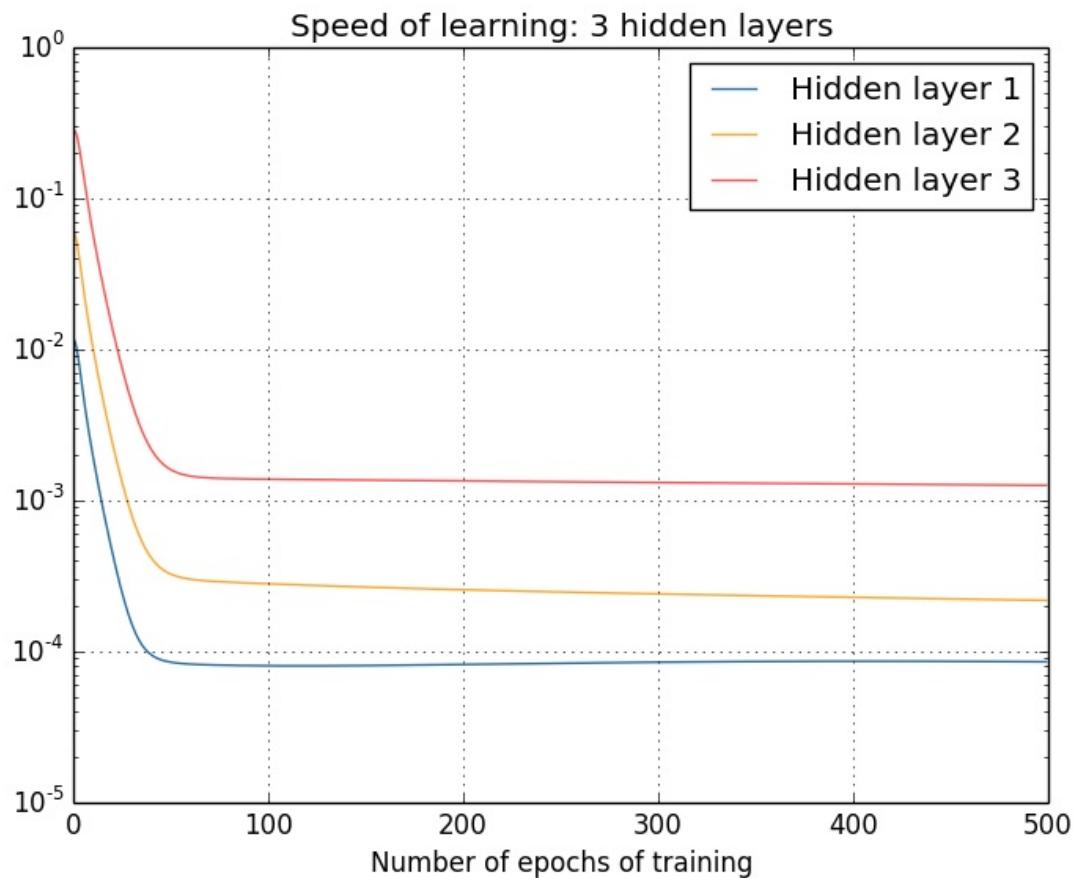


为了产生这些结果，我在 1000 个训练图像上进行了 500 轮 batch 梯度下降。这和我们通常训练方式还是不同的——我没有使用 minibatch，仅仅使用了 1000 个训练图像，而不是全部的 50,000 幅图。我并不是想做点新鲜的尝试，或者蒙蔽你们的双眼，但因为使用 minibatch 随机梯度下降会在结果中带来更多的噪声（尽管在平均噪声的时候结果很相似）。使用我已经确定的参数可以对结果进行平滑，这样我们可以看清楚真正的情况是怎样的。如图所示，两层在开始时就有着不同的速度。然后两层的学习速度在触底前迅速下落。在最后，我们发现第一层的学习速度变得比第二层更慢了。

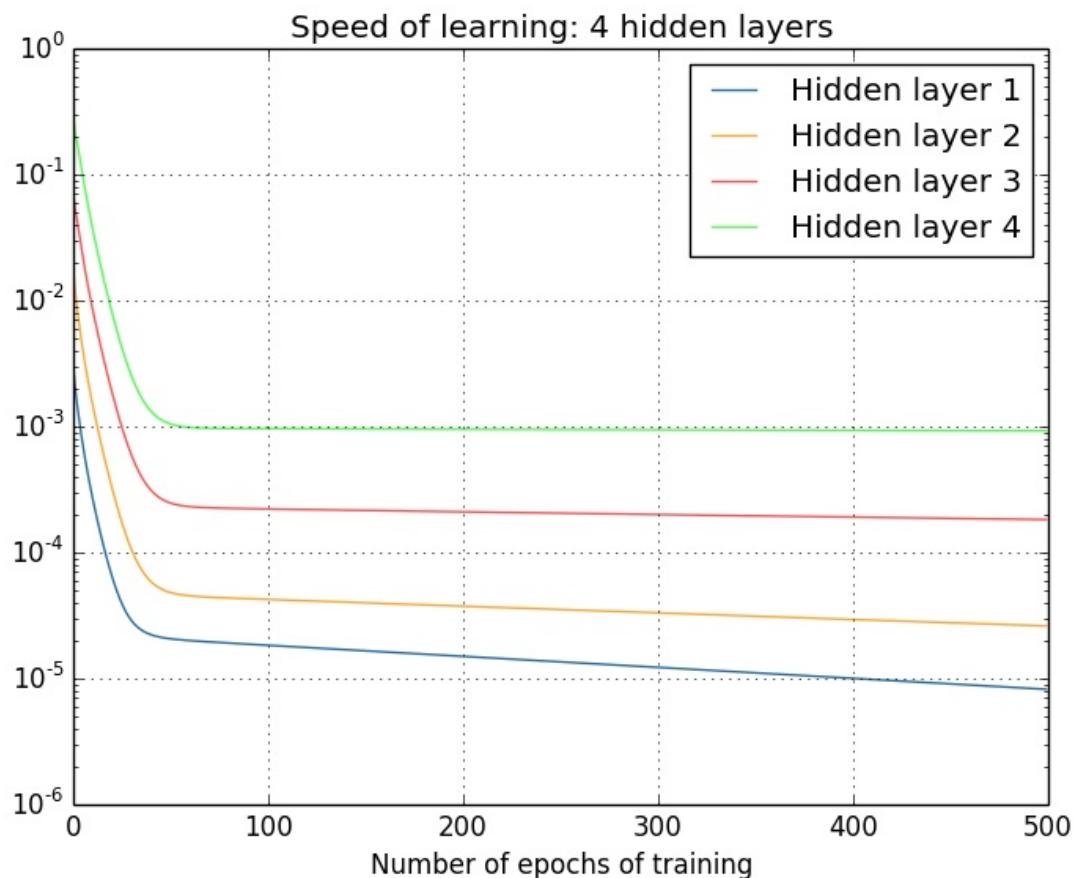
那么更加复杂的网络是什么情况呢？这里是一个类似的实验，但是这次有三个隐藏层 ([784, 30, 30, 30, 10])：

null

null



同样，前面的隐藏层要比后面的隐藏层学习的更慢。最后一个实验，就是增加第四个隐藏层([784, 30, 30, 30, 30, 10])，看看这里会发生什么：



null

null

同样的情况出现了，前面的隐藏层的学习速度要低于后面的隐藏层。这里，第一层的学习速度和最后一层要差了两个数量级，也就是比第四层慢了100倍。难怪我们之前在训练这些网络的时候遇到了大麻烦！

现在我们已经有了一项重要的观察结果：至少在某些深度神经网络中，在我们在隐藏层 BP 的时候梯度倾向于变小。这意味着在前面的隐藏层中的神经元学习速度要慢于后面的隐藏层。这儿我们只在一个网络中发现了这个现象，其实在多数的神经网络中存在着更加根本的导致这个现象出现的原因。这个现象也被称作是 消失的梯度问题 (**vanishing gradient problem**)。

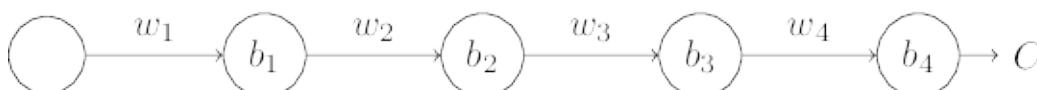
为何消失的梯度问题会出现呢？我们可以通过什么方式避免它？还有在训练深度神经网络时如何处理好这个问题？实际上，这个问题是可以避免的，尽管替代方法并不是那么有效，同样会产生问题——在前面的层中的梯度会变得非常大！这也叫做 爆炸的梯度问题 (**exploding gradient problem**)，这也没比消失的梯度问题更好处理。更加一般地说，在深度神经网络中的梯度是不稳定的，在前面的层中或会消失，或会爆炸。这种不稳定性才是深度神经网络中基于梯度学习的根本问题。这就是我们需要理解的东西，如果可能的话，采取合理的步骤措施解决问题。

一种有关消失的（不稳定的）梯度的看法是确定这是否确实是一个问题。此刻我们暂时转换到另一个话题，假设我们正要数值优化一个一元的函数 $f(x)$ 。如果其导数 $f'(x)$ 很小，这难道不是一个好消息么？是不是意味着我们已经接近极值点了？同样的方式，在深度神经网络中前面隐藏层的小的梯度是不是表示我们不需要对权重和偏差做太多调整了？

当然，实际情况并不是这样的。想想我们随机初始网络中的权重和偏差。在面对任意的一种任务，单单使用随机初始的值就能够获得一个较好的结果是太天真了。具体讲，看看 MNIST 问题的网络中第一层的权重。随机初始化意味着第一层丢失了输入图像的几乎所有信息。即使后面的层能够获得充分的训练，这些层也会因为没有充分的信息而很难识别出输入的图像。因此，在第一层不进行学习的尝试是不可能的。如果我们接着去训练深度神经网络，我们需要弄清楚如何解决消失的梯度问题。

什么导致了消失的梯度问题？也就是在深度神经网络中的所谓的梯度不稳定性

为了弄清楚为何会出现消失的梯度，来看看一个极简单的深度神经网络：每一层都只有一个单一的神经元。下图就是有三层隐藏层的神经网络：



这里， w_1, w_2, \dots 是权重，而 b_1, b_2, \dots 是偏差， C 则是某个代价函数。回顾一下，从第 j 个神经元的输出 $a_j = \sigma(z_j)$ ，其中 σ 是通常的 sigmoid 函数，而 $z_j = w_j * a_{j-1} + b_j$ 是神经元的带权输入。我已经在最后表示出了代价函数 C 来强调代价是网络输出 a_4 的函数：如果实际输出越接近目标输出，那么代价会变低；相反则会变高。

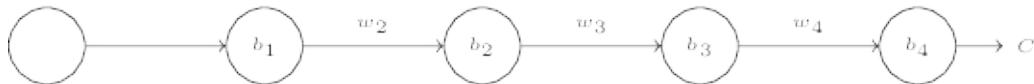
null

null

现在我们要来研究一下关于第一个隐藏神经元梯度 $\partial C / \partial b_1$ 。我们将会计算出 $\partial C / \partial b_1$ 的表达式，通过研究表达式来理解消失的梯度发生的原因。

开始就简单地给出 $\partial C / \partial b_1$ 的表达式。初看起来有点复杂，但是其结构是相当简单的，我一会儿会解释。下图给出了具体的表达式：

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



表达式结构如下：对每个神经元有一个 $\sigma'(z_j)$ 项；对每个权重有一个 w_j 项；还有一个 $\partial C / \partial a_4$ 项，表示最后的代价函数。注意，我已经将表达式中的每个项置于了对应的位置。所以网络本身就是表达式的解读。

你可以直接认可这个表达式，直接跳到[该表达式如何关联于小时的梯度问题的](#)。这对理解没有影响，因为实际上上面的表达式只是前面对于[BP 的讨论](#)的特例。但是也包含了一个表达式正确的解释，所以去看看那个解释也是很有趣的（也可能更有启发性吧）。

假设我们对偏差 b_1 进行了微小的调整 Δb_1 。这会导致网络中剩下的元素一系列的变化。首先会对第一个隐藏元输出产生一个 Δa_1 的变化。这样就会导致第二个神经元的带权输入产生 Δz_2 的变化。从第二个神经元输出随之发生 Δa_2 的变化。以此类推，最终会对代价函数产生 ΔC 的变化。这里我们有：

$$\frac{\partial C}{\partial b_1} \approx \frac{\Delta C}{\Delta b_1}$$

这表示我们可以通过仔细追踪每一步的影响来搞清楚 $\partial C / \partial b_1$ 的表达式。现在我们看看 Δb_1 如何影响第一个神经元的输出 a_1 的。我们有 $a_1 = \sigma(z_1) = \sigma(w_1 * a_0 + b_1)$ ，所以有 $\Delta a_1 \approx \frac{\partial \sigma(w_1 a_0 + b_1)}{\partial b_1} \Delta b_1 = \sigma'(z_1) \Delta b_1$

$\sigma'(z_1)$ 这项看起很熟悉：其实是我们上面关于 $\partial C / \partial b_1$ 的表达式的第一项。直觉上看，这项将偏差的改变 Δb_1 转化成了输出的变化 Δa_1 。 Δa_1 随之又影响了带权输入

$$z_2 = w_2 * a_1 + b_2 : \Delta z_2 \approx \frac{\partial z_2}{\partial a_1} \Delta a_1 = w_2 \Delta a_1$$

将 Δz_2 和 Δa_1 的表达式组合起来，我们可以看到偏差 b_1 中的改变如何通过网络传输影响到 z_2 的： $\Delta \approx \sigma'(z_1) w_2 \Delta b_1$

现在，又能看到类似的结果了：我们得到了在表达式 $\partial C / \partial b_1$ 的前面两项。以此类推下去，跟踪传播改变的路径就可以完成。在每个神经元，我们都会选择一个 $\sigma'(z_j)$ 的项，然后在每个权重我们选择出一个 w_j 项。最终的结果就是代价函数中变化 ΔC 的相关于偏差 Δb_1 的表达式：

$$\Delta C \approx \sigma'(z_1) w_2 \sigma'(z_2) \dots \sigma'(z_4) \frac{\partial C}{\partial a_4} \Delta b_1$$

除以 Δb_1 ，我们的确得到了梯度的表达式：

null

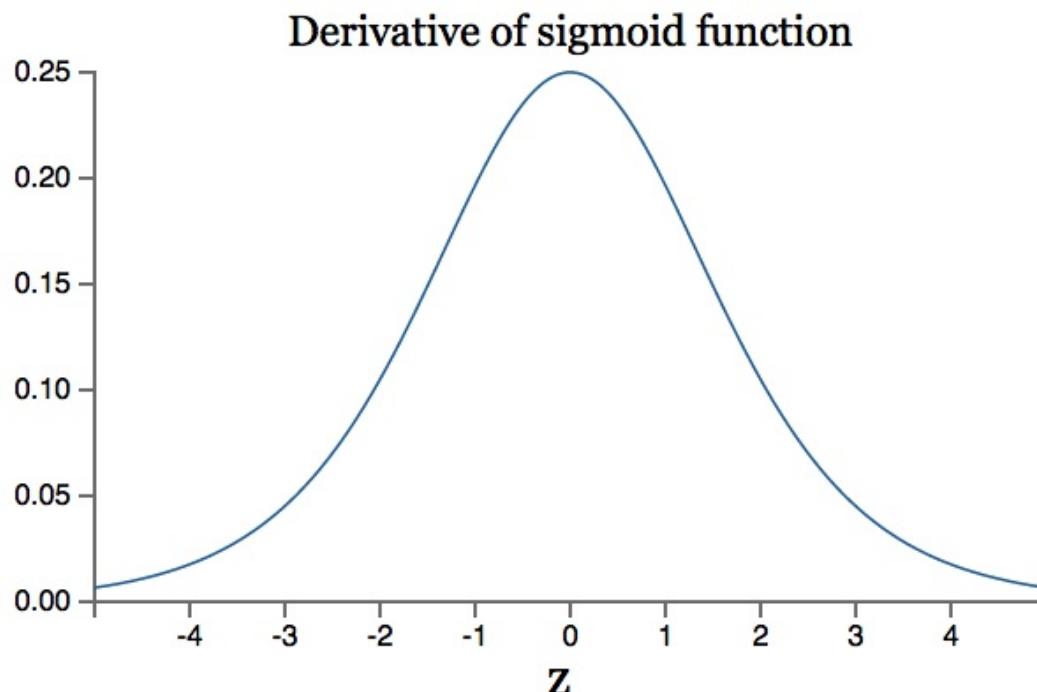
null

$$\partial C / \partial b_1 = \sigma'(z_1) w_2 \sigma'(z_2) \dots \sigma'(z_4) \frac{\partial C}{\partial a_4}$$

为何出现梯度消失：现在把梯度的整个表达式写下来：

$$\partial C / \partial b_1 = \sigma'(z_1) w_2 \sigma'(z_2) w_3 \sigma'(z_3) \sigma'(z_4) \frac{\partial C}{\partial a_4}$$

除了最后一项，该表达式是一系列形如 $w_j \sigma'(z_j)$ 的乘积。为了理解每个项的行为，先看看下面的sigmoid 函数导数的图像：



该导数在 $\sigma'(0) = 1/4$ 时达到最高。现在，如果我们使用标准方法来初始化网络中的权重，那么会使用一个均值为 0 标准差为 1 的高斯分布。因此所有的权重通常会满足 $|w_j| < 1$ 。有了这些信息，我们发现会有 $w_j \sigma'(z_j) < 1/4$ 。并且在我们进行了所有这些项的乘积时，最终结果肯定会指数级下降：项越多，乘积的下降的越快。**这里我们敏锐地嗅到了消失的梯度问题的合理解释。

更明白一点，我们比较一下 $\partial C / \partial b_1$ 和一个更后面一些的偏差的梯度，不妨设为 $\partial C / \partial b_3$ 。当然，我们还没有显式地给出这个表达式，但是计算的方式是一样的。

null

null

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \underbrace{w_2 \sigma'(z_2)}_{< \frac{1}{4}} \underbrace{w_3 \sigma'(z_3)}_{< \frac{1}{4}} \underbrace{w_4 \sigma'(z_4)}_{\text{common terms}} \frac{\partial C}{\partial a_4}$$

$$\frac{\partial C}{\partial b_3} = \sigma'(z_3) w_4 \sigma'(z_4) \underbrace{\frac{\partial C}{\partial a_4}}$$

两个表示式有很多相同的项。但是 $\partial C / \partial b_1$ 还多包含了两个项。由于这些项都是 $< 1/4$ 的。所以 $\partial C / \partial b_1$ 会是 $\partial C / \partial b_3$ 的 $1/16$ 或者更小。这其实就是消失的梯度出现的本质原因了。当然，这里并非严格的关于消失的梯度微调的证明而是一个不太正式的论断。还有一些可能的产生原因了。特别地，我们想要知道权重 w_j 在训练中是否会增长。如果会，项 $w_j \sigma'(z_j)$ 会不会不在满足之前 $w_j \sigma'(z_j) < 1/4$ 的约束。事实上，如果项变得很大——超过 1，那么我们将不再遇到消失的梯度问题。实际上，这时候梯度会在我们 BP 的时候发生指数级地增长。也就是说，我们遇到了梯度爆炸的问题。梯度爆炸问题：现在看看梯度爆炸如何出现的把。这里的例子可能不是那么自然：固定网络中的参数，来确保产生爆炸的梯度。但是即使是不自然，也是包含了确定会产生爆炸梯度（而非假设的可能）的特质的。

共两个步骤：首先，我们将网络的权重设置得很大，比如 $w_1 = w_2 = w_3 = w_4 = 100$ 。然后，我们选择偏差使得 $\sigma'(z_j)$ 项不会太小。这是很容易实现的：方法就是选择偏差来保证每个神经元的带权输入是 $z_j = 0$ （这样 $\sigma'(z_j) = 1/4$ ）。比如说，我们希望 $z_1 = w_1 * a_0 + b_1$ 。我们只要把 $b_1 = -100 * a_0$ 即可。我们使用同样的方法来获得其他的偏差。这样我们可以发现所有的项 $w_j * \sigma'(z_j)$ 都等于 $100 * 1/4 = 25$ 。最终，我们就获得了爆炸的梯度。

不稳定的梯度问题：根本的问题其实并非是消失的梯度问题或者爆炸的梯度问题，而是在前面的层上的梯度是来自后面的层上项的乘积。当存在过多的层次时，就出现了内在本质上的不稳定场景。唯一让所有层都接近相同的学习速度的方式是所有这些项的乘积都能得到一种平衡。如果没有某种机制或者更加本质的保证来达成平衡，那网络就很容易不稳定了。简而言之，真实的问题就是神经网络受限于不稳定梯度的问题。所以，如果我们使用标准的基于梯度的学习算法，在网络中的不同层会出现按照不同学习速度学习的情况。

练习

- 在我们对于消失的梯度问题讨论中，使用了 $|\sigma'(z) < 1/4|$ 这个结论。假设我们使用一个

null

个不同的激活函数，其导数值是非常大的。这会帮助我们避免不稳定梯度的问题么？

消失的梯度问题普遍存在：我们已经看到了在神经网络的前面的层中梯度可能会消失也可能爆炸。实际上，在使用 sigmoid 神经元时，梯度通常会消失。为什么？再看看表达式 $|w\sigma'(z)|$ 。为了避免消失的梯度问题，我们需要 $|w\sigma'(z)| >= 1$ 。你可能会认为如果 w 很大的时候很容易达成。但是这比看起来还是困难很多。原因在于， $\sigma'(z)$ 项同样依赖于 w ： $\sigma'(z) = \sigma'(w * a + b)$ ，其中 a 是输入的激活函数。所以我们在让 w 变大时，需要同时不让 $\sigma'(wa + b)$ 变小。这将是很大的限制了。原因在于我们让 w 变大，也会使得 $wa + b$ 变得非常大。看看 σ' 的图，这会让我们走到 σ' 的两翼，这里会去到很小的值。唯一避免发生这个情况的方式是，如果输入激活函数掉入相当狭窄的范围内（这个量化的解释在下面第一个问题中进行）。有时候，有可能会出现。但是一般不会发生。所以一般情况下，会遇到消失的梯度。

问题

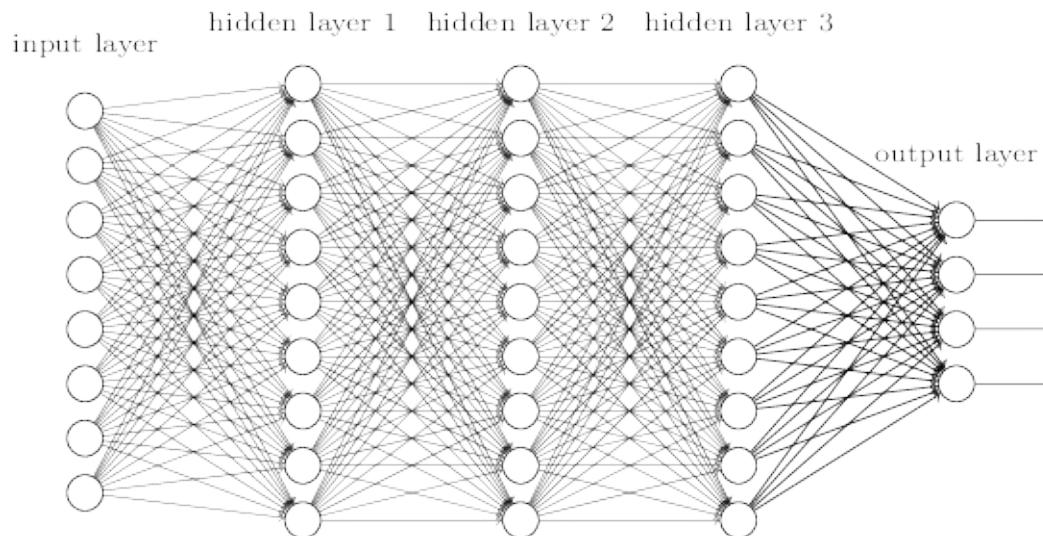
- 考虑乘积 $|w\sigma'(wa + b)|$ 。假设有 $|w\sigma'(wa + b)| >= 1$ 。
 - 这种情况只有在 $|w| >= 4$ 的时候才会出现。
 - 假设 $|w| >= 4$ ，考虑那些满足 $|w\sigma'(wa + b)| >= 1$ 的输入激活 a 集合。证明：
满足上述条件的该集合能够充满一个不超过 $\frac{2}{|w|} \ln(\frac{|w|(1+\sqrt{1-4/|w|})}{2} - 1)$ 宽度的区间。
◦ 数值上说明上述表达式在 $|w| = 6.9$ 时候去的最高值约 0.45。所以即使每个条件都满足，我们仍然有一个狭窄的输入激活区间，这样来避免消失的梯度问题。
- 弧神经元：考虑一个单一输入的神经元， x ，对应的权重 w_1 ，偏差 b ，输出上的权重 w_2 。证明，通过合理选择权重和偏差，我们可以确保 $w_2\sigma(w_1 * x + b) = x$ 其中 $x \in [0, 1]$ 。这样的神经元可用来作为弧元试用，输出和输入相同（成比例）。提示：可以重写 $x = 1/2 + \Delta$ ，可以假设 w_1 很小，和在 $w_1 * \Delta$ 使用 Taylor 级数展开。

在更加复杂网络中的不稳定梯度

现在已经研究了简单的网络，每一层只包含一个神经元。那么那些每层包含很多神经元的更加复杂的深度网络呢？

null

null



实际上，在这样的神经网络中，同样的情况也会发生。在前面关于 BP 的章节中，我们看到了在一个共 L 层的第 l 层的梯度：

$$\delta^l = \Sigma'(z^l) * (w^{l+1})^T \Sigma'(z^{l+1})(w^{l+2})^T \dots \Sigma'(z^L) \nabla_a C$$

这里 $\Sigma'(z^l)$ 是一个对角矩阵，每个元素是对第 l 层的带权输入 $\sigma'(z)$ 。而 w^l 是对不同层的权值矩阵。 $\nabla_a C$ 是对每个输出激活的偏导数向量。

这是更加复杂的表达式。不过，你仔细看，本质上的形式还是很相似的。主要是包含了更多的形如 $(w^j)^T \Sigma'(z^j)$ 的对 (pair)。而且，矩阵 $\Sigma'(z^j)$ 在对角线上的值挺小，不会超过 $1/4$ 。由于权值矩阵 w^j 不是太大，每个额外的项 $(w^j)^T \sigma'(z^l)$ 会让梯度向量更小，导致梯度消失。更加一般地看，在乘积中大量的项会导致不稳定的梯度，和前面的例子一样。实践中，一般会发现在 sigmoid 网络中前面的层的梯度指数级地消失。所以在这些层上的学习速度就会变得很慢了。这种减速不是偶然现象：也是我们采用的训练的方法决定的。

深度学习其他的障碍

本章我们已经聚焦在消失的梯度上，并且更加一般地，不稳定梯度——深度学习的一大障碍。实际上，不稳定梯度仅仅是深度学习的众多障碍之一，尽管这一点是相当根本的。当前的研究集中在更好地理解在训练深度神经网络时遇到的挑战。这里我不会给出一个详尽的总结，仅仅想要给出一些论文，告诉你人们正在寻觅探究的问题。

首先，在 2010 年 Glorot 和 Bengio 发现证据表明 sigmoid 函数的选择会导致训练网络的问题。特别地，他们发现 sigmoid 函数会导致最终层上的激活函数在训练中会聚集在 0，这也导致了学习的缓慢。他们的工作中提出了一些取代 sigmoid 函数的激活函数选择，使得不会被这种聚集性影响性能。

第二个例子，在 2013 年 Sutskever, Martens, Dahl 和 Hinton 研究了深度学习使用随机权重初始化和基于 *momentum* 的 SGD 方法。两种情形下，好的选择可以获得较大的差异的训练效果。这些例子告诉我们，“什么让训练深度网络非常困难”这个问题相当复杂。本章，我们已经集中于深度神经网络中基于梯度的学习方法的不稳定性。结果表明了激活函数的选

null

null

择，权重的初始化，甚至是学习算法的实现方式也扮演了重要的角色。当然，网络结构和其他超参数本身也是很重要的。因此，太多因子影响了训练神经网络的难度，理解所有这些因子仍然是当前研究的重点。尽管这看起来有点悲观，但是在下一章中我们会介绍一些好的消息，给出一些方法来一定程度上解决和迂回所有这些困难。

null

null

在上一章，我们学习了深度神经网络通常比浅层神经网络更加难以训练。我们有理由相信，若是可以训练深度网络，则能够获得比浅层网络更加强大的能力，但是现实很残酷。从上一章我们可以看到很多不利的消息，但是这些困难不能阻止我们使用深度神经网络。本章，我们将给出可以用来训练深度神经网络的技术，并在实战中应用它们。同样我们也会从更加广阔的视角来看神经网络，简要地回顾近期有关深度神经网络在图像识别、语音识别和其他应用中的研究进展。然后，还会给出一些关于未来神经网络又或人工智能的简短的推测性的看法。

这一章比较长。为了更好地让你们学习，我们先粗看一下整体安排。本章的小结之间关联并不太紧密，所以如果读者熟悉基本的神经网络的知识，那么可以任意跳到自己最感兴趣的的部分。

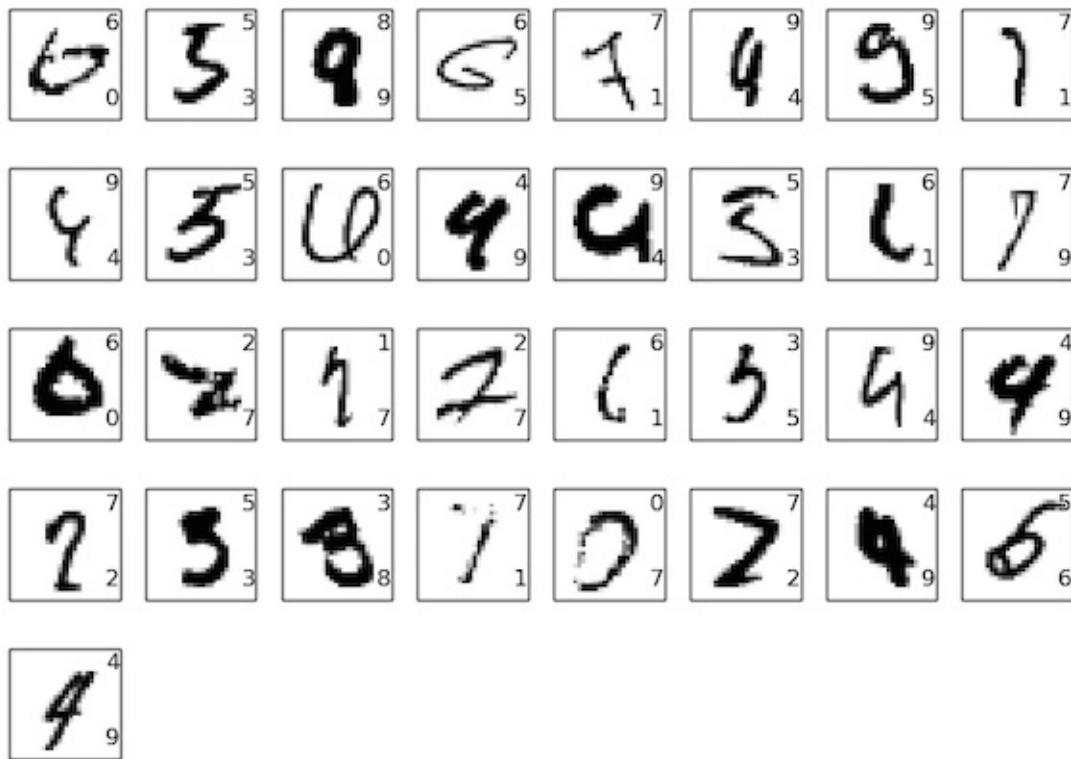
本章主要的部分是对最为流行神经网络之一的深度卷积网络的介绍。我们将细致地分析一个使用卷积网络来解决 MNIST 数据集的手写数字识别的例子（包含了代码和讲解）：



我们将从浅层的神经网络开始来解决上面的问题。通过多次的迭代，我们会构建越来越强大的网络。在这个过程中，也将要探究若干强大技术：卷积、pooling、使用GPU来更好地训练、训练数据的算法性扩展（避免过匹配）、dropout 技术的使用（同样为了防止过匹配现象）、网络的 ensemble 使用和其他技术。最终的结果能够接近人类的表现。在 10,000 幅 MNIST 测试图像上——模型从未在训练中接触的图像——该系统最终能够将其中 9,967 幅正确分类。这儿我们看看错分的 33 幅图像。注意正确分类是右上的标记；系统产生的分类在右下：

null

null



可以发现，这里面的图像对于正常人类来说都是非常困难区分的。例如，在第一行的第三幅图。我看的话，看起来更像是“9”而非“8”，而“8”却是给出的真实的结果。我们的网络同样能够确定这个是“9”。这种类型的“错误”最起码是容易理解的，可能甚至值得我们赞许。最后用对最近使用深度（卷积）神经网络在图像识别上的研究进展作为关于图像识别的讨论的总结。

本章剩下的部分，我们将会从一个更加宽泛和宏观的角度来讨论深度学习。概述一些神经网络的其他模型，例如 RNN 和 LSTM 网络，以及这些网络如何在语音识别、自然语言处理和其他领域中应用的。最后会试着推测一下，神经网络和深度学习未来发展的方向，会从 intention-driven user interfaces 谈谈深度学习在人工智能的角色。这章内容建立在本书前面章节的基础之上，使用了前面介绍的诸如 BP、规范化、softmax 函数，等等。然而，要想阅读这一章，倒是不需要太过细致地掌握前面章节中内容的所有的细节。当然读完第一章关于神经网络的基础是非常有帮助的。本章提到第二章到第五章的概念时，也会在文中给出链接供读者去查看这些必需的概念。

需要注意的一点是，本章所没有包含的那一部分。这一章并不是关于最新和最强大的神经网络库。我们也不是想训练数十层的神经网络来处理最前沿的问题。而是希望能够让读者理解深度神经网络背后核心的原理，并将这些原理用在一个 MNIST 问题的解决中，方便我们的理解。换句话说，本章目标不是将最前沿的神经网络展示给你看。包括前面的章节，我们都是聚焦在基础上，这样读者就能够做好充分的准备来掌握众多的不断涌现的深度学习领域最新工作。本章仍然在Beta版。期望读者指出笔误，bug，小错和主要的误解。如果你发现了可疑的地方，请直接联系 mn@michaelnielsen.org。

卷积网络简介

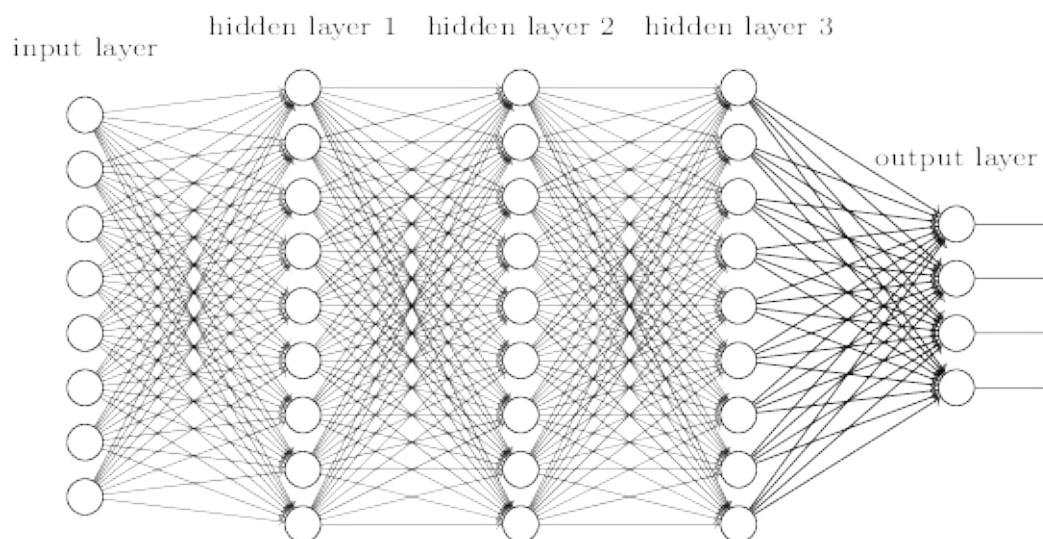
null

null

在前面的章节中，我们教会了神经网络能够较好地识别手写数字：



我们在深度神经网络中使用全连接的邻接关系。网络中的神经元与相邻的层上的所有神经元均连接：



特别地，对输入图像中的每个像素点，我们将其光强度作为对应输入层神经元的输入。对于 28×28 像素的图像，这意味着我们输入神经元需要有 $784 (= 28 \times 28)$ 个。

实践中的卷积神经网络

我们现已看到卷积神经网络中核心思想。现在我们就来看看如何在实践中使用卷积神经网络，通过实现某些卷积网络，应用在 MNIST 数字分类问题上。我们使用的程序是 `network3.py`，这是 `network.py` 和 `network2.py` 的改进版本。代码可以在[GitHub](#) 下载。注意我们会在下一节详细研究一下代码。本节，我们直接使用 `network3.py` 来构建卷积网络。

`network.py` 和 `network2.py` 是使用 python 和矩阵库 numpy 实现的。这些程序从最初的理论开始，并实现了 BP、随机梯度下降等技术。我们既然已经知道原理，对 `network3.py`，我们现在就使用 Theano 来构建神经网络。使用 Theano 可以更方便地实现卷积网络的 BP，因为它会自动计算所有包含的映射。Theano 也会比我们之前的代码（容易看懂，运行蛮）运行得快得多，这会更适合训练更加复杂的神经网络。特别的一点，Theano 支持 CPU 和 GPU，我们写出来的 Theano 代码可以运行在 GPU 上。这会大幅度提升学习的速度，这样就算是很复杂的网络也是可以用在实际的场景中的。

如果你要继续跟下去，就需要安装 Theano。跟随[这些参考](#) 就可以安装 Theano 了。后面的例

null

null

子在 Theano 0.6 上运行。有些是在 Mac OS X Yosemite 上，没有 GPU。有些是在 Ubuntu 14.4 上，有 NVIDIA GPU。还有一些在两种情况都有运行。为了让 `network3.py` 运行，你需要在 `network3.py` 的源码中将 GPU 置为 True 或者 False。除此之外，让 Theano 在 GPU 上运行，你可能要参考 [the instructions here](#)。网络上还有很多的教程，用 Google 很容易找到。如果没有 GPU，也可以使用 [Amazon Web Services](#) EC2 G2 spot instances。注意即使是 GPU，训练也可能花费很多时间。很多实验花了数分钟或者数小时才完成。在 CPU 上，则可能需要好多天才能运行完最复杂的实验。正如在前面章节中提到的那样，我建议你搭建环境，然后阅读，偶尔回头再检查代码的输出。如果你使用 CPU，可能要降低训练的次数，甚至跳过这些实验。

为了获得一个基准，我们将启用一个浅层的架构，仅仅使用单一的隐藏层，包含 100 个隐藏元。训练 60 次，使用学习率为 $\eta = 0.1$ ，mini-batch 大小为 10，无规范化。Let's go：

```
>>> import network3
>>> from network3 import Network
>>> from network3 import ConvPoolLayer, FullyConnectedLayer, SoftmaxLayer
>>> training_data, validation_data, test_data = network3.load_data_shared()
>>> mini_batch_size = 10
>>> net = Network([ FullyConnectedLayer(n_in=784, n_out=100), SoftmaxLayer(n_in=100, n_out=10)])
>>> net.SGD(training_data, 60, mini_batch_size, 0.1, validation_data, test_data)
```

卷积网络的代码

好了，现在来看看我们的卷积网络代码，`network3.py`。整体看来，程序结构类似于 `network2.py`，尽管细节有差异，因为我们使用了 Theano。首先我们来看 `FullyConnectedLayer` 类，这类似于我们之前讨论的那些神经网络层。下面是代码

```
class FullyConnectedLayer(object):

    def __init__(self, n_in, n_out, activation_fn=sigmoid, p_dropout=0.0):
        self.n_in = n_in
        self.n_out = n_out
        self.activation_fn = activation_fn
        self.p_dropout = p_dropout
        # Initialize weights and biases
        self.w = theano.shared(
            np.asarray(
                np.random.normal(
                    loc=0.0, scale=np.sqrt(1.0/n_out), size=(n_in, n_out)),
                dtype=theano.config.floatX),
            name='w', borrow=True)
        self.b = theano.shared(
            np.asarray(np.random.normal(loc=0.0, scale=1.0, size=(n_out,))),
            dtype=theano.config.floatX),
            name='b', borrow=True)
        self.params = [self.w, self.b]

    def set_inpt(self, inpt, inpt_dropout, mini_batch_size):
```

null

```
null

    self.inpt = inpt.reshape((mini_batch_size, self.n_in))
    self.output = self.activation_fn(
        (1-self.p_dropout)*T.dot(self.inpt, self.w) + self.b)
    self.y_out = T.argmax(self.output, axis=1)
    self.inpt_dropout = dropout_layer(
        inpt_dropout.reshape((mini_batch_size, self.n_in)), self.p_dropout)
    self.output_dropout = self.activation_fn(
        T.dot(self.inpt_dropout, self.w) + self.b)

    def accuracy(self, y):
        "Return the accuracy for the mini-batch."
        return T.mean(T.eq(y, self.y_out))
```

`__init__` 方法中的大部分都是可以自解释的，这里再给出一些解释。我们根据正态分布随机初始化了权重和偏差。代码中对应这个操作的一行看起来可能很吓人，但其实只在进行载入权重和偏差到 Theano 中所谓的共享变量中。这样可以确保这些变量可在 GPU 中进行处理。对此不做过深的解释。如果感兴趣，可以查看[Theano documentation](#)。而这种初始化的方式也是专门为 sigmoid 激活函数设计的（参见[这里](#)）。理想的情况是，我们初始化权重和偏差时会根据不同的激活函数（如 tanh 和 Rectified Linear Function）进行调整。这个在下面的问题中会进行讨论。初始方法 `__init__` 以 `self.params = [self.w, self.b]` 结束。这样将该层所有需要学习的参数都归在一起。后面，`Network.SGD` 方法会使用 `params` 属性来确定网络实例中什么变量可以学习。

`set_inpt` 方法用来设置该层的输入，并计算相应的输出。我使用 `inpt` 而非 `input` 因为在 python 中 `input` 是一个内置函数。如果将两者混淆，必然会导致不可预测的行为，对出现的问题也难以定位。注意我们实际上用两种方式设置输入的：`self.inpt` 和 `self.inpt_dropout`。因为训练时我们可能要使用 dropout。如果使用 dropout，就需要设置对应丢弃的概率 `self.p_dropout`。这就是在 `set_inpt` 方法的倒数第二行 `dropout_layer` 做的事。所以 `self.inpt_dropout` 和 `self.output_dropout` 在训练过程中使用，而 `self.inpt` 和 `self.output` 用作其他任务，比如衡量验证集和测试集模型的准确度。

`ConvPoolLayer` 和 `SoftmaxLayer` 类定义和 `FullyConnectedLayer` 定义差不多。所以我这儿不会给出代码。如果你感兴趣，可以参考本节后面的 `network3.py` 的代码。

尽管这样，我们还是指出一些重要的微弱的细节差别。明显一点的是，在 `ConvPoolLayer` 和 `SoftmaxLayer` 中，我们采用了相应的合适的计算输出激活值方式。幸运的是，Theano 提供了内置的操作让我们计算卷积、max-pooling 和 softmax 函数。

不大明显的，在我们引入softmax layer 时，我们没有讨论如何初始化权重和偏差。其他地方我们已经讨论过对 sigmoid 层，我们应当使用合适参数的正态分布来初始化权重。但是这个启发式的论断是针对 sigmoid 神经元的（做一些调整可以用于 tanh 神经元上）。但是，并没有特殊的原因说这个论断可以用在 softmax 层上。所以没有一个先验的理由应用这样的初始化。与其使用之前的方法初始化，我这里会将所有权值和偏差设置为 0。这是一个 ad hoc 的过程，但在实践使用过程中效果倒是很不错。

好了，我们已经看过了所有关于层的类。那么 Network 类是怎样的呢？让我们看看 `__init__` 方法：

```
null
```

null

```
class Network(object):

    def __init__(self, layers, mini_batch_size):
        """Takes a list of `layers`, describing the network architecture, and
        a value for the `mini_batch_size` to be used during training
        by stochastic gradient descent.

        ...
        self.layers = layers
        self.mini_batch_size = mini_batch_size
        self.params = [param for layer in self.layers for param in layer.params]
        self.x = T.matrix("x")
        self.y = T.ivector("y")
        init_layer = self.layers[0]
        init_layer.set_inpt(self.x, self.x, self.mini_batch_size)
        for j in xrange(1, len(self.layers)):
            prev_layer, layer = self.layers[j-1], self.layers[j]
            layer.set_inpt(
                prev_layer.output, prev_layer.output_dropout, self.mini_batch_size)
        self.output = self.layers[-1].output
        self.output_dropout = self.layers[-1].output_dropout
```

这段代码大部分是可以自解释的。`self.params = [param for layer in ...]` 此行代码对每层的参数捆绑到一个列表中。`Network.SGD` 方法会使用 `self.params` 来确定 `Network` 中哪些变量需要学习。而 `self.x = T.matrix("x")` 和 `self.y = T.ivector("y")` 则定义了 Theano 符号变量 `x` 和 `y`。这些会用来表示输入和网络得到的输出。

这儿不是 Theano 的教程，所以不会深度讨论这些变量指代什么东西。但是粗略的想法就是这些代表了数学变量，而非显式的值。我们可以对这些变量做通常需要的操作：加减乘除，作用函数等等。实际上，Theano 提供了很多对符号变量进行操作方法，如卷积、max-pooling 等等。但是最重要的是能够进行快速符号微分运算，使用 BP 算法一种通用的形式。这对于应用随机梯度下降在若干种网络结构的变体上特别有效。特别低，接下来几行代码定义了网络的符号输出。我们通过下面这行

```
init_layer.set_inpt(self.x, self.x, self.mini_batch_size)
```

设置初始层的输入。

请注意输入是以每次一个 mini-batch 的方式进行的，这就是 mini-batch size 为何要指定的原因。还需要注意的是，我们将输入 `self.x` 传了两次：这是因为我们可能会以两种方式（有dropout和无dropout）使用网络。`for` 循环将符号变量 `self.x` 通过 `Network` 的层进行前向传播。这样我们可以定义最终的输出 `output` 和 `output_dropout` 属性，这些都是 `Network` 符号式输出。

现在我们理解了 `Network` 是如何初始化了，让我们看看它如何使用 `SGD` 方法进行训练的。代码看起来很长，但是它的结构实际上相当简单。代码后面也有一些注解。

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        validation_data, test_data, lmbda=0.0):
    """Train the network using mini-batch stochastic gradient descent."""
    training_x, training_y = training_data
```

null

null

```
validation_x, validation_y = validation_data
test_x, test_y = test_data

# compute number of minibatches for training, validation and testing
num_training_batches = size(training_data)/mini_batch_size
num_validation_batches = size(validation_data)/mini_batch_size
num_test_batches = size(test_data)/mini_batch_size

# define the (regularized) cost function, symbolic gradients, and updates
l2_norm_squared = sum([(layer.w**2).sum() for layer in self.layers])
cost = self.layers[-1].cost(self)+\
        0.5*lmbda*l2_norm_squared/num_training_batches
grads = T.grad(cost, self.params)
updates = [(param, param-eta*grad)
            for param, grad in zip(self.params, grads)]

# define functions to train a mini-batch, and to compute the
# accuracy in validation and test mini-batches.
i = T.lscalar() # mini-batch index
train_mb = theano.function(
    [i], cost, updates=updates,
    givens={
        self.x:
        training_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
        self.y:
        training_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
    })
validate_mb_accuracy = theano.function(
    [i], self.layers[-1].accuracy(self.y),
    givens={
        self.x:
        validation_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
        self.y:
        validation_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
    })
test_mb_accuracy = theano.function(
    [i], self.layers[-1].accuracy(self.y),
    givens={
        self.x:
        test_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
        self.y:
        test_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
    })
self.test_mb_predictions = theano.function(
    [i], self.layers[-1].y_out,
    givens={
        self.x:
        test_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
    })

# Do the actual training
best_validation_accuracy = 0.0
for epoch in xrange(epochs):
    for minibatch_index in xrange(num_training_batches):
        iteration = num_training_batches*epoch+minibatch_index
        if iteration % 1000 == 0:
            print("Training mini-batch number %d" % iteration)
        cost_ij = train_mb(minibatch_index)
        if (iteration+1) % num_training_batches == 0:
            validation_accuracy = np.mean(
                [validate_mb_accuracy(j) for j in xrange(num_validation_batches)])
            print("Epoch %d: validation accuracy %.2f%%" % (epoch, validation_accuracy))
            if validation_accuracy >= best_validation_accuracy:
```

null

null

```
        print("This is the best validation accuracy to date.")
        best_validation_accuracy = validation_accuracy
        best_iteration = iteration
        if test_data:
            test_accuracy = np.mean(
                [test_mb_accuracy(j) for j in xrange(num_test_batches)])
            print('The corresponding test accuracy is {0:.2%}'.format(
                test_accuracy))
        print("Finished training network.")
        print("Best validation accuracy of {0:.2%} obtained at iteration {1}".format(
            best_validation_accuracy, best_iteration))
        print("Corresponding test accuracy of {0:.2%}".format(test_accuracy))
```

前面几行很直接，将数据集分解成 x 和 y 两部分，并计算在每个数据集中 mini-batch 的数量。接下来的几行更加有意思，这也体现了 Theano 有趣的特性。那么我们就摘录详解一下：

```
# define the (regularized) cost function, symbolic gradients, and updates
l2_norm_squared = sum([(layer.w**2).sum() for layer in self.layers])
cost = self.layers[-1].cost(self)+ 0.5*lambda*l2_norm_squared/num_training_batches
grads = T.grad(cost, self.params)
updates = [(param, param-eta*grad) for param, grad in zip(self.params, grads)]
```

这几行，我们符号化地给出了规范化的 log-likelihood 代价函数，在梯度函数中计算了对应的导数，以及对应参数的更新方式。Theano 让我们通过这短短几行就能够获得这些效果。唯一隐藏的是计算 `cost` 包含一个对输出层 `cost` 方法的调用；该代码在 `network3.py` 中其他地方。但是，总之代码很短而且简单。有了所有这些定义好的东西，下面就是定义 `train_mini_batch` 函数，该 Theano 符号函数在给定 minibatch 索引的情况下使用 `updates` 来更新 `Network` 的参数。类似地，`validate_mb_accuracy` 和 `test_mb_accuracy` 计算在任意给定的 minibatch 的验证集和测试集上 `Network` 的准确度。通过对这些函数进行平均，我们可以计算整个验证集和测试数据集上的准确度。

`SGD` 方法剩下的就是可以自解释的了——我们对次数进行迭代，重复使用训练数据的 minibatch 来训练网络，计算验证集和测试集上的准确度。

好了，我们已经理解了 `network3.py` 代码中大多数的重要部分。让我们看看整个程序，你不必过分仔细地读下这些代码，但是应该享受粗看的过程，并随时深入研究那些激发出你好奇地代码段。理解代码的最好的方法就是通过修改代码，增加额外的特征或者重新组织那些你认为能够更加简洁地完成的代码。代码后面，我们给出了一些对初学者的建议。这儿是代码：

在 GPU 上使用 Theano 可能会有点难度。特别地，很容易在从 GPU 中拉取数据时出现错误，这可能会让运行变得相当慢。我已经试着避免出现这样的情况，但是也不能肯定在代码扩充后出现一些问题。对于你们遇到的问题或者给出的意见我洗耳恭听
(mn@michaelnielsen.org)。

```
"""network3.py
```

null

null

```
A Theano-based program for training and running simple neural
networks.

Supports several layer types (fully connected, convolutional, max
pooling, softmax), and activation functions (sigmoid, tanh, and
rectified linear units, with more easily added).

When run on a CPU, this program is much faster than network.py and
network2.py. However, unlike network.py and network2.py it can also
be run on a GPU, which makes it faster still.

Because the code is based on Theano, the code is different in many
ways from network.py and network2.py. However, where possible I have
tried to maintain consistency with the earlier programs. In
particular, the API is similar to network2.py. Note that I have
focused on making the code simple, easily readable, and easily
modifiable. It is not optimized, and omits many desirable features.

This program incorporates ideas from the Theano documentation on
convolutional neural nets (notably,
http://deeplearning.net/tutorial/lenet.html ), from Misha Denil's
implementation of dropout (https://github.com/mdenil/dropout ), and
from Chris Olah (http://colah.github.io ).

"""

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

# Third-party libraries
import numpy as np
import theano
import theano.tensor as T
from theano.tensor.nnet import conv
from theano.tensor import softmax
from theano.tensor import shared_randomstreams
from theano.tensor.signal import downsample

# Activation functions for neurons
def linear(z): return z
def ReLU(z): return T.maximum(0.0, z)
from theano.tensor.nnet import sigmoid
from theano.tensor import tanh

#### Constants
GPU = True
if GPU:
    print "Trying to run under a GPU. If this is not desired, then modify "+\
          "network3.py\n\tto set the GPU flag to False."
    try: theano.config.device = 'gpu'
    except: pass # it's already set
    theano.config.floatX = 'float32'
else:
    print "Running with a CPU. If this is not desired, then the modify "+\
          "network3.py to set\n\tthe GPU flag to True."

#### Load the MNIST data
def load_data_shared(filename='../data/mnist.pkl.gz'):
    f = gzip.open(filename, 'rb')
    training_data, validation_data, test_data = cPickle.load(f)
    f.close()
    def shared(data):
        """Place the data into shared variables. This allows Theano to copy
        the data to the GPU, if one is available.
        """
        return theano.shared(np.asarray(data, dtype='float32'))
    training_data = shared(training_data)
    validation_data = shared(validation_data)
    test_data = shared(test_data)
    return training_data, validation_data, test_data
```

null

null

```
shared_x = theano.shared(
    np.asarray(data[0], dtype=theano.config.floatX), borrow=True)
shared_y = theano.shared(
    np.asarray(data[1], dtype=theano.config.floatX), borrow=True)
return shared_x, T.cast(shared_y, "int32")
return [shared(training_data), shared(validation_data), shared(test_data)]
```

Main class used to construct and train networks

```
class Network(object):
```

```
def __init__(self, layers, mini_batch_size):
    """Takes a list of `layers`, describing the network architecture, and
    a value for the `mini_batch_size` to be used during training
    by stochastic gradient descent.
    """
    self.layers = layers
    self.mini_batch_size = mini_batch_size
    self.params = [param for layer in self.layers for param in layer.params]
    self.x = T.matrix("x")
    self.y = T.ivector("y")
    init_layer = self.layers[0]
    init_layer.set_inpt(self.x, self.x, self.mini_batch_size)
    for j in xrange(1, len(self.layers)):
        prev_layer, layer = self.layers[j-1], self.layers[j]
        layer.set_inpt(
            prev_layer.output, prev_layer.output_dropout, self.mini_batch_size)
    self.output = self.layers[-1].output
    self.output_dropout = self.layers[-1].output_dropout
```

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        validation_data, test_data, lmbda=0.0):
    """Train the network using mini-batch stochastic gradient descent."""
    training_x, training_y = training_data
    validation_x, validation_y = validation_data
    test_x, test_y = test_data

    # compute number of minibatches for training, validation and testing
    num_training_batches = size(training_data)/mini_batch_size
    num_validation_batches = size(validation_data)/mini_batch_size
    num_test_batches = size(test_data)/mini_batch_size

    # define the (regularized) cost function, symbolic gradients, and updates
    l2_norm_squared = sum([(layer.w**2).sum() for layer in self.layers])
    cost = self.layers[-1].cost(self)+\
        0.5*lmbda*l2_norm_squared/num_training_batches
    grads = T.grad(cost, self.params)
    updates = [(param, param-eta*grad)
               for param, grad in zip(self.params, grads)]

    # define functions to train a mini-batch, and to compute the
    # accuracy in validation and test mini-batches.
    i = T.lscalar() # mini-batch index
    train_mb = theano.function(
        [i], cost, updates=updates,
        givens={
            self.x:
            training_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
            self.y:
            training_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
        })
    validate_mb_accuracy = theano.function(
        [i], self.layers[-1].accuracy(self.y),
        givens={}
```

null

null

```
        self.x:
        validation_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
        self.y:
        validation_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
    })
test_mb_accuracy = theano.function(
    [i], self.layers[-1].accuracy(self.y),
    givens={
        self.x:
        test_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size],
        self.y:
        test_y[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
    })
self.test_mb_predictions = theano.function(
    [i], self.layers[-1].y_out,
    givens={
        self.x:
        test_x[i*self.mini_batch_size: (i+1)*self.mini_batch_size]
    })
# Do the actual training
best_validation_accuracy = 0.0
for epoch in xrange(epochs):
    for minibatch_index in xrange(num_training_batches):
        iteration = num_training_batches*epoch+minibatch_index
        if iteration % 1000 == 0:
            print("Training mini-batch number {}".format(iteration))
        cost_ij = train_mb(minibatch_index)
        if (iteration+1) % num_training_batches == 0:
            validation_accuracy = np.mean(
                [validate_mb_accuracy(j) for j in xrange(num_validation_batches)])
            print("Epoch {}: validation accuracy {:.2%}".format(
                epoch, validation_accuracy))
            if validation_accuracy >= best_validation_accuracy:
                print("This is the best validation accuracy to date.")
                best_validation_accuracy = validation_accuracy
                best_iteration = iteration
            if test_data:
                test_accuracy = np.mean(
                    [test_mb_accuracy(j) for j in xrange(num_test_batches)])
                print('The corresponding test accuracy is {:.2%}'.format(
                    test_accuracy))
    print("Finished training network.")
    print("Best validation accuracy of {:.2%} obtained at iteration {}".format(
        best_validation_accuracy, best_iteration))
    print("Corresponding test accuracy of {:.2%}".format(test_accuracy))
```

Define layer types

```
class ConvPoolLayer(object):
    """Used to create a combination of a convolutional and a max-pooling
    layer. A more sophisticated implementation would separate the
    two, but for our purposes we'll always use them together, and it
    simplifies the code, so it makes sense to combine them.
    """

    def __init__(self, filter_shape, image_shape, poolsize=(2, 2),
                 activation_fn=sigmoid):
        """`filter_shape` is a tuple of length 4, whose entries are the number
        of filters, the number of input feature maps, the filter height, and the
        filter width.
        `image_shape` is a tuple of length 4, whose entries are the
        mini-batch size, the number of input feature maps, the image
        height, and the image width.
```

null

null

```
`poolsize` is a tuple of length 2, whose entries are the y and
x pooling sizes.
"""
self.filter_shape = filter_shape
self.image_shape = image_shape
self.poolsize = poolsize
self.activation_fn=activation_fn
# initialize weights and biases
n_out = (filter_shape[0]*np.prod(filter_shape[2:])/np.prod(poolsize))
self.w = theano.shared(
    np.asarray(
        np.random.normal(loc=0, scale=np.sqrt(1.0/n_out), size=filter_shape),
        dtype=theano.config.floatX),
    borrow=True)
self.b = theano.shared(
    np.asarray(
        np.random.normal(loc=0, scale=1.0, size=(filter_shape[0],)),
        dtype=theano.config.floatX),
    borrow=True)
self.params = [self.w, self.b]

def set_inpt(self, inpt, inpt_dropout, mini_batch_size):
    self.inpt = inpt.reshape(self.image_shape)
    conv_out = conv.conv2d(
        input=self.inpt, filters=self.w, filter_shape=self.filter_shape,
        image_shape=self.image_shape)
    pooled_out = downsample.max_pool_2d(
        input=conv_out, ds=self.poolsize, ignore_border=True)
    self.output = self.activation_fn(
        pooled_out + self.b.dimshuffle('x', 0, 'x', 'x'))
    self.output_dropout = self.output # no dropout in the convolutional layers

class FullyConnectedLayer(object):

    def __init__(self, n_in, n_out, activation_fn=sigmoid, p_dropout=0.0):
        self.n_in = n_in
        self.n_out = n_out
        self.activation_fn = activation_fn
        self.p_dropout = p_dropout
        # Initialize weights and biases
        self.w = theano.shared(
            np.asarray(
                np.random.normal(
                    loc=0.0, scale=np.sqrt(1.0/n_out), size=(n_in, n_out)),
                dtype=theano.config.floatX),
            name='w', borrow=True)
        self.b = theano.shared(
            np.asarray(np.random.normal(loc=0.0, scale=1.0, size=(n_out,)),
                      dtype=theano.config.floatX),
            name='b', borrow=True)
        self.params = [self.w, self.b]

    def set_inpt(self, inpt, inpt_dropout, mini_batch_size):
        self.inpt = inpt.reshape((mini_batch_size, self.n_in))
        self.output = self.activation_fn(
            (1-self.p_dropout)*T.dot(self.inpt, self.w) + self.b)
        self.y_out = T.argmax(self.output, axis=1)
        self.inpt_dropout = dropout_layer(
            inpt_dropout.reshape((mini_batch_size, self.n_in)), self.p_dropout)
        self.output_dropout = self.activation_fn(
            T.dot(self.inpt_dropout, self.w) + self.b)

    def accuracy(self, y):
```

null

null

```
    "Return the accuracy for the mini-batch."
    return T.mean(T.eq(y, self.y_out))

class SoftmaxLayer(object):

    def __init__(self, n_in, n_out, p_dropout=0.0):
        self.n_in = n_in
        self.n_out = n_out
        self.p_dropout = p_dropout
        # Initialize weights and biases
        self.w = theano.shared(
            np.zeros((n_in, n_out), dtype=theano.config.floatX),
            name='w', borrow=True)
        self.b = theano.shared(
            np.zeros((n_out,), dtype=theano.config.floatX),
            name='b', borrow=True)
        self.params = [self.w, self.b]

    def set_inpt(self, inpt, inpt_dropout, mini_batch_size):
        self.inpt = inpt.reshape((mini_batch_size, self.n_in))
        self.output = softmax((1-self.p_dropout)*T.dot(self.inpt, self.w) + self.b)
        self.y_out = T.argmax(self.output, axis=1)
        self.inpt_dropout = dropout_layer(
            inpt_dropout.reshape((mini_batch_size, self.n_in)), self.p_dropout)
        self.output_dropout = softmax(T.dot(self.inpt_dropout, self.w) + self.b)

    def cost(self, net):
        "Return the log-likelihood cost."
        return -T.mean(T.log(self.output_dropout)[T.arange(net.y.shape[0]), net.y])

    def accuracy(self, y):
        "Return the accuracy for the mini-batch."
        return T.mean(T.eq(y, self.y_out))

#### Miscellanea
def size(data):
    "Return the size of the dataset `data`."
    return data[0].get_value(borrow=True).shape[0]

def dropout_layer(layer, p_dropout):
    srng = shared_randomstreams.RandomStreams(
        np.random.RandomState(0).randint(999999))
    mask = srng.binomial(n=1, p=1-p_dropout, size=layer.shape)
    return layer*T.cast(mask, theano.config.floatX)
```

问题

- 目前，`SGD` 方法需要用户手动确定训练的次数（epoch）。早先在本书中，我们讨论了一种自动选择训练次数的方法，也就是`early stopping`。修改 `network3.py` 以实现 Early stopping。
- 增加一个 `Network` 方法来返回在任意数据集上的准确度。
- 修改 `SGD` 方法来允许学习率 η 可以是训练次数的函数。提示：在思考这个问题一段时间后，你可能会在[this link](#) 找到有用的信息。

null

null

- 在本章前面我曾经描述过一种通过应用微小的旋转、扭曲和变化来扩展训练数据的方法。改变 `network3.py` 来加入这些技术。注意：除非你有充分多的内存，否则显式地产生整个扩展数据集是不大现实的。所以要考虑一些变通的方法。
- 在 `network3.py` 中增加 `load` 和 `save` 方法。
- 当前的代码缺点就是只有很少的用来诊断的工具。你能想出一些诊断方法告诉我们网络过匹配到什么程度么？加上这些方法。
- 我们已经对rectified linear unit 及 sigmoid 和 tanh 函数神经元使用了同样的初始方法。正如[这里所说](#)，这种初始化方法只是适用于 sigmoid 函数。假设我们使用一个全部使用 RLU 的网络。试说明以常数 c 倍调整网络的权重最终只会对输出有常数 c 倍的影响。如果最后一层是 softmax，则会发生什么样的变化？对 RLU 使用 sigmoid 函数的初始化方法会怎么样？有没有更好的初始化方法？注意：这是一个开放的问题，并不是说有一个简单的自包含答案。还有，思考这个问题本身能够帮助你更好地理解包含 RLU 的神经网络。
- 我们对于不稳定梯度问题的分析实际上是对 sigmoid 神经元的。如果是 RLU，那分析又会有什么差异？你能够想出一种使得网络不太会受到不稳定梯度问题影响的好方法么？注意：好实际上就是一个研究性问题。实际上有很多容易想到的修改方法。但我现在还没有研究足够深入，能告诉你们什么是真正的好技术。

图像识别领域中的近期进展

在 1998 年，MNIST 数据集被提出来，那时候需要花费数周能够获得一个最优的模型，和我们现在使用 GPU 在少于 1 小时内训练的模型性能差很多。所以，MNIST 已经不是一个能够推动技术边界前进的问题了；不过，现在的训练速度让 MNIST 能够成为教学和学习的样例。同时，研究重心也已经发生了转变，现代的研究工作包含更具挑战性的图像识别问题。在本节，我们简短介绍一些近期使用神经网络进行图像识别上的研究进展。

本节内容和本书其他大部分都不一样。整本书，我都专注在那些可能会成为持久性的方法上——诸如 BP、规范化、和卷积网络。我已经尽量避免提及那些在我写书时很热门但长期价值未知的研究内容了。在科学领域，这样太过热门容易消逝的研究太多了，最终对科学发展的价值却是很微小的。所以，可能会有人怀疑：“好吧，在图像识别中近期的发展就是这种情况么？两到三年后，事情将发生变化。所以，肯定这些结果仅仅是一些想在研究前沿阵地领先的专家的专属兴趣而已？为何又费力来讨论这个呢？”

这种怀疑是正确的，近期研究论文中一些改良的细节最终会失去其自身的重要性。过去几年里，我们已经看到了使用深度学习解决特别困难的图像识别任务上巨大进步。假想一个科学史学者在 2100 年写起计算机视觉。他们肯定会将 2011 到 2015（可能再加上几年）这几年作为使用深度卷积网络获得重大突破的时段。但这并不意味着深度卷积网络，还有dropout、RLU 等等，在 2100 年仍在使用。但这确实告诉我们在思想的历史上，现在，正发生着重要的转变。这有点像原子的发现，抗生素的发明：在历史的尺度上的发明和发现。所以，尽管我

null

null

们不会深入这些细节，但仍值得从目前正在发生的研究成果中获得一些令人兴奋的研究发现。

The 2012 LRMD paper：让我们从一篇来自 Stanford 和 Google 的研究者的论文开始。后面将这篇论文简记为 LRMD，前四位作者的姓的首字母命名。LRMD 使用神经网络对 [ImageNet](#) 的图片进行分类，这是一个具有非常挑战性的图像识别问题。2011 年 ImageNet 数据包含了 16,000,000 的全色图像，有 20,000 个类别。图像从开放的网络上爬去，由 Amazon Mechanical Turk 服务的工人分类。下面是几幅 ImageNet 的图像：



上面这些分别属于 圆线刨，棕色烂根须，加热的牛奶，及 通常的蚯蚓。如果你想挑战一下，你可以访问[hand tools](#)，里面包含了一系列的区分的任务，比如区分 圆线刨、短刨、倒角刨以及其他十几种类型的刨子和其他的类别。我不知道读者你是怎么样一个人，但是我不能将所有这些工具类型都确定地区分开。这显然是比 MNIST 任务更具挑战性的任务。LRMD 网络获得了不错的 15.8% 的准确度。这看起很不给力，但是在先前最优的 9.3% 准确度上却是一个大的突破。这个飞跃告诉人们，神经网络可能会成为一个对非常困难的图像识别任务的强大武器。

The 2012 KSH paper：在 2012 年，出现了一篇 LRMD 后续研究 Krizhevsky, Sutskever and Hinton (KSH)。KSH 使用一个受限 ImageNet 的子集数据训练和测试了一个深度卷积神经网络。这个数据集是机器学习竞赛常用的一个数据集——ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)。使用一个竞赛数据集可以方便比较神经网络和其他方法之间的差异。ILSVRC-2012 训练集包含 120,000 幅 ImageNet 的图像，共有 1,000 类。验证集和测试集分别包含 50,000 和 150,000 幅，也都是同样的 1,000 类。

ILSVRC 竞赛中一个难点是许多图像中包含多个对象。假设一个图像展示出一只拉布拉多犬追逐一只足球。所谓“正确的”分类可能是拉布拉多犬。但是算法将图像归类为足球就应该被惩罚么？由于这样的模糊性，我们做出下面设定：如果实际的ImageNet分类是出于算法给出的最可能的 5 类，那么算法最终被认为是正确的。KSH 深度卷积网络达到了 84.7% 的准确度，比第二名的 73.8% 高出很多。使用更加严格度量，KSH 网络业达到了 63.3% 的准确度。

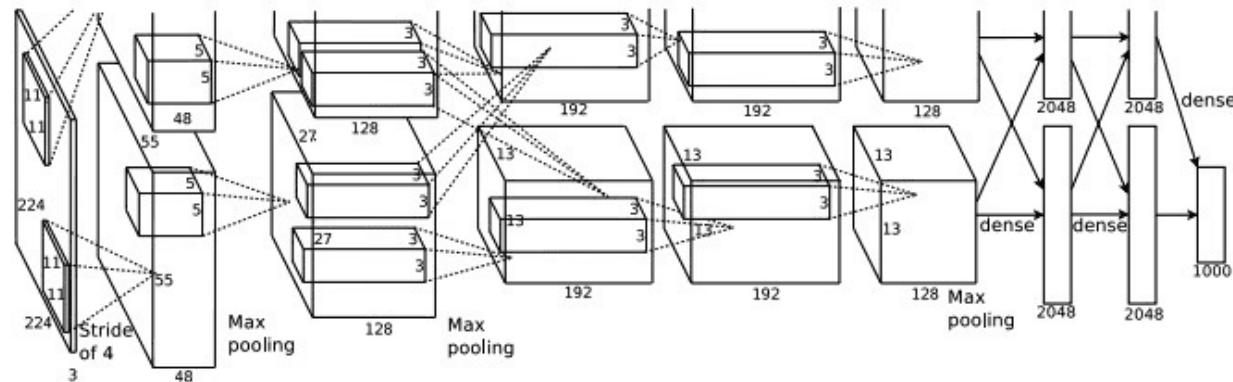
我们这里会简要说明一下 KSH 网络，因为这是后续很多工作的源头。而且它也和我们之前给出的卷积网络相关，只是更加复杂精细。KSH 使用深度卷积网络，在两个 GPU 上训练。使用两个 GPU 因为 GPU 的型号使然（NVIDIA GeForce GTX 580 没有足够的大的内存来存放整个网络）所以用这样的方式进行内存的分解。

KSH 网络有 7 个隐藏层。前 5 个隐藏层是卷积层（可能会包含 max-pooling），而后两个隐藏层则是全连接层。输出层则是 1,000 的 softmax，对应于 1,000 种分类。下面给出了网络的架构图，来自 KSH 的论文。我们会给出详细的解释。注意很多层被分解为 2 个部分，对应

null

null

于 2 个 GPU。



输出层包含 $3 \times 224 \times 224$ 神经元，表示一幅 224×224 的图像的 RGB 值。回想一下，ImageNet 包含不同分辨率的图像。这里也会有问题，因为神经网络输入层通常是固定的小。KSH 通过将每幅图进行的重设定，使得短的边长度为 256。然后在重设后的图像上裁剪出 256×256 的区域。最终 KSH 从 256×256 的图像中抽取出随机的 224×224 的子图（和水平反射）。他们使用随机的方式，是为了扩展训练数据，这样能够缓解过匹配的情况。在大型网络中这样的方法是很有效的。这些 224×224 的图像就成为了网络的输入。在大多数情形下，裁剪的图像仍会包含原图中主要的对象。

现在看看 KSH 的隐藏层，第一隐藏层是一个卷积层，还有 max-pooling。使用了大小为 11×11 的局部感应区，和大小为 4 的步长。总共有 96 个特征映射。特征映射被分成两份，分别存放在两块 GPU 上。max-pooling 在这层和下层都是 3×3 区域进行，由于允许使用重叠的 pooling 区域，pooling 层其实会产生两个像素值。

Pooling layers in CNNs summarize the outputs of neighboring groups of neurons in the same kernel map. Traditionally, the neighborhoods summarized by adjacent pooling units do not overlap (e.g., [17, 11, 4]). To be more precise, a pooling layer can be thought of as consisting of a grid of pooling units spaced s pixels apart, each summarizing a neighborhood of size $z \times z$ centered at the location of the pooling unit. If we set $s = z$, we obtain traditional local pooling as commonly employed in CNNs. If we set $s < z$, we obtain overlapping pooling. This is what we use throughout our network, with $s = 2$ and $z = 3$. This scheme reduces the top-1 and top-5 error rates by 0.4% and 0.3%, respectively, as compared with the non-overlapping scheme $s = 2$, $z = 2$, which produces output of equivalent dimensions. We generally observe during training that models with overlapping pooling find it slightly more difficult to overfit.

第二隐藏层同样是卷积层，并附加一个 max-pooling 步骤。使用了 5×5 的局部感知区，总共有 256 个特征映射，在每个 GPU 上各分了 128 个。注意到，特征映射只使用 48 个输入信道，而不是前一层整个 96 个输出。这是因为任何单一的特征映射仅仅使用来自同一个 GPU 的输入。从这个角度看，这个网络和此前我们使用的卷积网络结构还是不同的，尽管本质上仍一致。

第三、四和五隐藏层也是卷积层，但和前两层不同的是：他们不包含 max-pooling 步。各层参数分别是：(3) 384 个特征映射， 3×3 的局部感知区，256 个输入信道；(4) 384 个 null

null

特征映射，其中 3×3 的局部感知区和 192 个输入信道；(5) 256 个特征映射， 3×3 的局部感知区，和 192 个输入信道。注意第三层包含一些 GPU 间的通信（如图所示）这样可使得特征映射能够用上所有 256 个输入信道。

第六、七隐藏层是全连接层，其中每层有 4,096 个神经元。

输出层是一个 1,000 个单元的 softmax 层。

KSH 网络使用了很多的技术。放弃了 sigmoid 或者 tanh 激活函数的使用，KSH 全部采用 RLU，显著地加速了训练过程。KSH 网络用有将近 60,000,000 的参数，所以，就算有大规模的数据训练，也可能出现过匹配情况。为了克服这个缺点，作者使用了随机剪裁策略扩展了训练数据集。另外还通过使用 [l2 regularization](#) 的变体和 dropout 来克服过匹配。网络本身使用 [基于momentum](#) 的 mini-batch 随机梯度下降进行训练。

这就是 KSH 论文中诸多核心想法的概述。细节我们不去深究，你可以通过仔细阅读论文获得。或者你也可以参考 Alexandrite Krizhevsky 的[cuda-convnet](#) 及后续版本，这里包含了这些想法的实现。还有基于 Theano 的实现也可以在[这儿](#)找到。尽管使用多 GPU 会让情况变得复杂，但代码本身还是类似于之前我们写出来的那些。Caffe 神经网络框架也有一个 KSH 网络的实现，看[Model Zoo](#)。

The 2014 ILSVRC 竞赛：2012年后，研究一直在快速推进。看看 2014年的 ILSVRC 竞赛。和 2012 一样，这次也包括了一个 120,000 张图像，1,000 种类别，而最终评价也就是看网络输出前五是不是包含正确的分类。胜利的团队，基于 Google 之前给出的结果，使用了包含 22 层的深度卷积网络。他们称此为 GoogLeNet，向 LeNet-5 致敬。GoogLeNet 达到了 93.33% 的准确率远超2013年的 88.3% [Clarifai](#) 和 2012 年的KSH 84.7%。

那么 GoogLeNet 93.33% 的准确率又是多好呢？在2014年，一个研究团队写了一篇关于 ILSVRC 竞赛的综述文章。其中有个问题是人类在这个竞赛中能表现得如何。为了做这件事，他们构建了一个系统让人类对 ILSVRC 图像进行分类。其作者之一 Andrej Karpathy 在一篇博文中解释道，让人类达到 GoogLeNet 的性能确实很困难：

...the task of labeling images with 5 out of 1000 categories quickly turned out to be extremely challenging, even for some friends in the lab who have been working on ILSVRC and its classes for a while. First we thought we would put it up on [Amazon Mechanical Turk]. Then we thought we could recruit paid undergrads. Then I organized a labeling party of intense labeling effort only among the (expert labelers) in our lab. Then I developed a modified interface that used GoogLeNet predictions to prune the number of categories from 1000 to only about 100. It was still too hard - people kept missing categories and getting up to ranges of 13-15% error rates. In the end I realized that to get anywhere competitively close to GoogLeNet, it was most efficient if I sat down and went through the painfully long training process and the subsequent careful annotation process myself... The labeling happened at a rate of about 1 per minute, but this decreased over time... Some images are easily recognized, while some images (such as those of fine-grained breeds of dogs, birds, or monkeys) can require multiple minutes of concentrated effort. I became very good at identifying breeds of dogs...

null

null

Based on the sample of images I worked on, the GoogLeNet classification error turned out to be 6.8%... My own error in the end turned out to be 5.1%, approximately 1.7% better.

换言之，一个专家级别的人类，非常艰难地检查图像，付出很大的精力才能够微弱胜过深度神经网络。实际上，Karpathy 指出第二个人类专家，用小点的图像样本训练后，只能达到 12.0% 的 top-5 错误率，明显弱于 GoogLeNet。大概有一半的错误都是专家“难以发现和认定正确的类别究竟是什么”。

这些都是惊奇的结果。根据这项工作，很多团队也报告 top-5 错误率实际上好过 5.1%。有时候，在媒体上被报道成系统有超过人类的视觉。尽管这项发现是很振奋人心的，但是这样的报道只能算是一种误解，认为系统在视觉上超过了人类，事实上并非这样。ILSVRC 竞赛问题在很多方面都是受限的——在公开的网络上获得图像并不具备在实际应用中的代表性！而且 top-5 标准也是非常人工设定的。我们在图像识别，或者更宽泛地说，计算机视觉方面的研究，还有很长的路要走。当然看到近些年的这些进展，还是很鼓舞人心的。

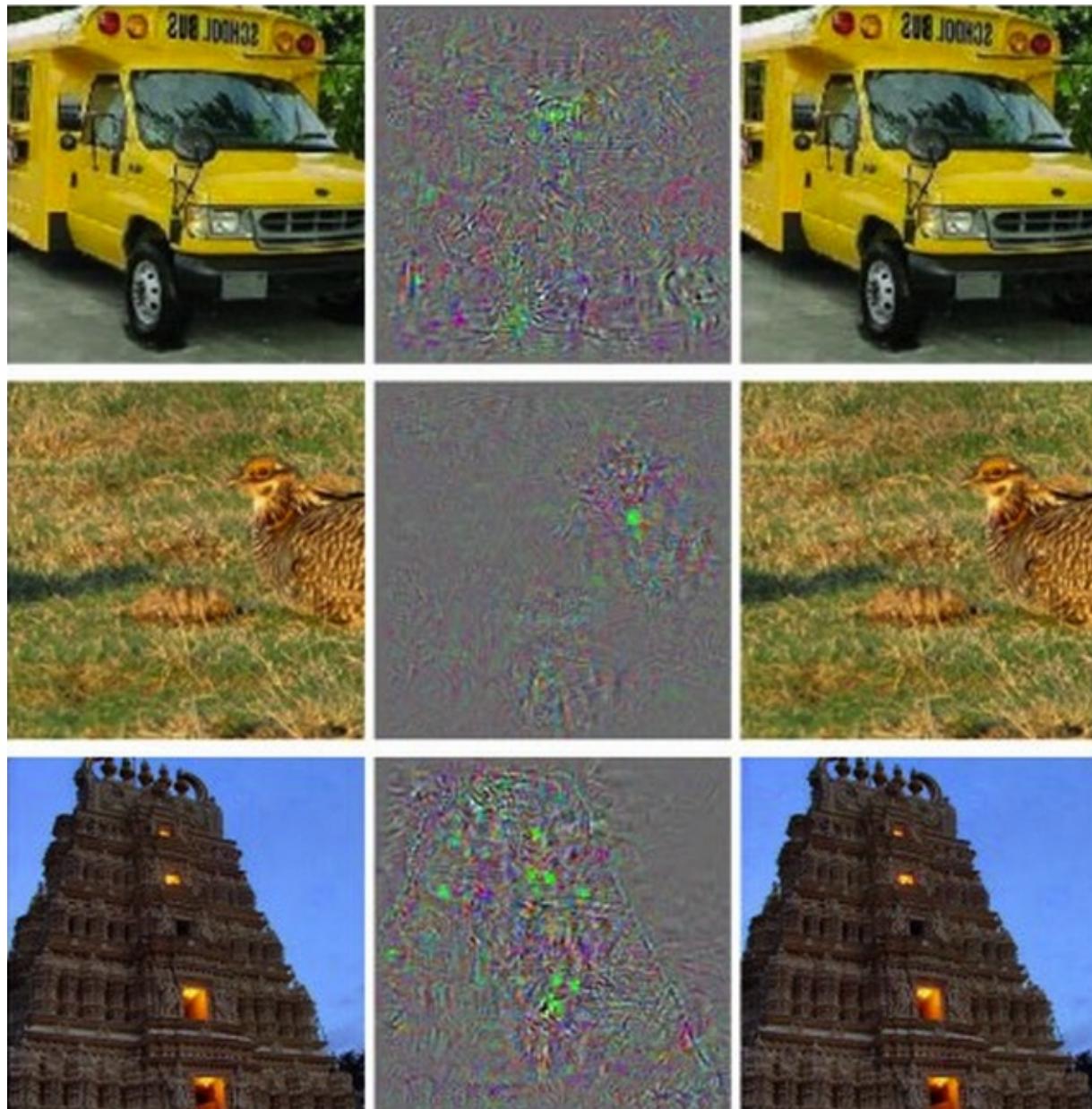
其他研究活动：上面关注于 ImageNet，但是也还有一些其他的使用神经网络进行图像识别的工作。我们也介绍一些进展。

一个鼓舞人心的应用上的结果就是 Google 的一个团队做出来的，他们应用深度卷积网络在识别 Google 的街景图像库中街景数字上。在他们的论文中，对接近 100,000,000 街景数字的自动检测和自动转述已经能打到与人类不相上下的程度。系统速度很快：在一个小时内将法国所有的街景数字都转述了。他们说道：“拥有这种新数据集能够显著提高 Google Maps 在一些国家的地理精准度，尤其是那些缺少地理编码的地区。”他们还做了一个更一般的论断：“我们坚信这个模型，已经解决了很多应用中字符短序列的 OCR 问题。”

我可能已经留下了印象——所有的结果都是令人兴奋的正面结果。当然，目前一些有趣的研究工作也报道了一些我们还没能够真的理解的根本性的问题。例如，2013 年一篇论文指出，深度网络可能会受到有效干扰的影响。看看下面的图示。左侧是被网络正确分类的 ImageNet 图像。右边是一幅稍受干扰的图像（使用中间的噪声进行干扰）结果就没有能够正确分类。作者发现对每幅图片都存在这样的“对手”图像，而非少量的特例。

null

null



这是一个令人不安的结果。论文使用了基于同样的被广泛研究使用的 KSH 代码。尽管这样的神经网络计算的函数在理论上都是连续的，结果表明在实际应用中，可能会碰到很多非常不连续的函数。更糟糕的是，他们将会以背离我们直觉的方式变得不连续。真是烦心啊。另外，现在对这种不连续性出现的原因还没有搞清楚：是跟损失函数有关么？或者激活函数？又或是网络的架构？还是其他？我们一无所知。

现在，这些问题也没有听起来这么吓人。尽管对手图像会出现，但是在实际场景中也不常见。正如论文指出的那样：

对手反例的存在看起来和网络能获得良好的泛化性能相违背。实际上，如果网络可以很好地泛化，会受到这些难以区分出来的对手反例怎么样的影响？解释是，对手反例集以特别低的概率出现，因此在测试集中几乎难以发现，然而对手反例又是密集的（有点像有理数那样），所以会在每个测试样本附近上出现。

我们对神经网络的理解还是太少了，这让人觉得很沮丧，上面的结果仅仅是近期的研究成果。当然了，这样结果带来一个主要好处就是，催生出一系列的研究工作。例如，最近一篇文章说明，给定一个训练好的神经网络，可以产生对人类来说是白噪声的图像，但是网络能

null

null

够将其确信地分类为某一类。这也是我们需要追寻的理解神经网络和图像识别应用上的研究方向。

虽然遇到这么多的困难，前途倒还是光明的。我们看到了在众多相当困难的基准任务上快速的研究进展。同样还有实际问题的研究进展，例如前面提到的街景数字的识别。但是需要注意的是，仅仅看到在那些基准任务，乃至实际应用的进展，是不够的。因为还有很多根本性的现象，我们对其了解甚少，就像对手图像的存在问题。当这样根本性的问题还亟待发现（或者解决）时，盲目地说我们已经接近最终图像识别问题的答案就很不合适了。这样的根本问题当然也会催生出不断的后续研究。

其他的深度学习模型

在整本书中，我们聚焦在解决 MNIST 数字分类问题上。这一“下金蛋的”问题让我们深入理解了一些强大的想法：随机梯度下降，BP，卷积网络，正规化等等。但是该问题却也是相当狭窄的。如果你研读过神经网络的研究论文，那么会遇到很多这本书中未曾讨论的想法：RNN，Boltzmann Machine，生成式模型，迁移学习，强化学习等等……等等！（太多了）神经网络是一个广阔的领域。然而，很多重要的想法都是我们书中探讨过的那些想法的变种，在有了本书的知识基础上，可能需要一些额外的努力，便可以理解这些新的想法了。所以在本节，我们给出这些想法的一些介绍。介绍本身不会非常细节化，可能也不会很深入——倘若要达成这两点，这本书就得扩展相当多内容了。因此，我们接下来的讨论是偏重思想性的启发，尝试去激发这个领域的产生丰富的概念，并将一些丰富的想法关联于前面已经介绍过的概念。我也会提供一些其他学习资源的连接。当然，链接给出的很多想法也会很快被超过，所以推荐你学会搜索最新的研究成果。尽管这样，我还是很期待众多本质的想法能够受到足够久的关注。

Recurrent Neural Networks (RNNs)：在前馈神经网络中，单独的输入完全确定了剩下的层上的神经元的激活值。可以想象，这是一幅静态的图景：网络中的所有事物都被固定了，处于一种“冰冻结晶”的状态。但假如，我们允许网络中的元素能够以动态方式不断地比那话。例如，隐藏神经元的行为不是完全由前一层的隐藏神经元，而是同样受制于更早的层上的神经元的激活值。这样肯定会带来跟前馈神经网络不同的效果。也可能隐藏和输出层的神经元的激活值不会单单由当前的网络输入决定，而且包含了前面的输入的影响。

拥有之类时间相关行为特性的神经网络就是递归神经网络，常写作 RNN。当然有不同的方式来从数学上给出 RNN 的形式定义。你可以参考[维基百科上的RNN介绍](#)来看看 RNN。在我写作本书的时候，维基百科上介绍了超过 13 种不同的模型。但除了数学细节，更加一般的想法是，RNN 是某种体现出了随时间动态变化的特性的神经网络。也毫不奇怪，RNN 在处理时序数据和过程上效果特别不错。这样的数据和过程正是语音识别和自然语言处理中常见的研究对象。

RNN 被用来将传统的算法思想，比如说 Turing 机或者编程语言，和神经网络进行联系上。[这篇 2014 年的论文](#)提出了一种 RNN 可以以 python 程序的字符级表达作为输入，用这个表达来预测输出。简单说，网络通过学习来理解某些 python 的程序。[第二篇论文](#) 同样是 2014 年的，使用 RNN 来设计一种称之为“神经 Turing 机”的模型。这是一种通用机器整个结构可以

null

null

使用梯度下降来训练。作者训练 NTM 来推断对一些简单问题的算法，比如说排序和复制。

不过正如在文中提到的，这些例子都是极其简单的模型。学会执行 `print(398345+42598)` 并不能让网络称为一个正常的python解释器！对于这些想法，我们能推进得多远也是未知的。结果都充满了好奇。历史上，神经网络已经在传统算法上失败的模式识别问题上取得了一些成功。另外，传统的算法也在神经网络并不擅长的领域里占据上风。今天没有人会使用神经网络来实现 Web 服务器或者数据库程序。研究出将神经网络和传统的算法结合的模型一定是非常棒的。RNN 和 RNN 给出的启发可能会给我们不少帮助。RNN 同样也在其他问题的解决中发挥着作用。在语音识别中，RNN 是特别有效的。例如，基于 RNN 的方法，已经在音位识别中取得了准确度的领先。同样在开发人类语言的上改进模型中得到应用。更好的语言模型意味着能够区分出发音相同的那些词。例如，好的语言模型，可以告诉我们“to infinity and beyond”比“two infinity and beyond”更可能出现，尽管两者的发音是相同的。RNN 在某些语言的标准测试集上刷新了记录。

在语音识别中的这项研究其实是包含于更宽泛的不仅仅是 RNN 而是所有类型的深度神经网络的应用的一部分。例如，基于神经网络的方法在大规模词汇的连续语音识别中获得极佳的结果。另外，一个基于深度网络的系统已经用在了 Google 的 Android 操作系统中（详见 [Vincent Vanhoucke's 2012-2015 papers](#)）

我刚刚讲完了 RNN 能做的一小部分，但还未提及他们如何工作。可能你并不诧异在前馈神经网络中的很多想法同样可以用在 RNN 中。尤其是，我们可以使用梯度下降和 BP 的直接的修改来训练 RNN。还有其他一些在前馈神经网络中的想法，如正规化技术，卷积和代价函数等都在 RNN 中非常有效。还有我们在书中讲到的很多技术都可以适配一下 RNN 场景。

Long Short-term Memory units(LSTMs)：影响 RNN 的一个挑战是前期的模型会很难训练，甚至比前馈神经网络更难。原因就是我们在上一章提到的不稳定梯度的问题。回想一下，这个问题的通常表现就是在反向传播的时候梯度越变越小。这就使得前期的层学习非常缓慢。在 RNN 中这个问题更加糟糕，因为梯度不仅仅通过层反向传播，还会根据时间进行反向传播。如果网络运行了一段很长的时间，就会使得梯度特别不稳定，学不到东西。幸运的是，可以引入一个成为 long short-term memory 的单元进入 RNN 中。LSTM 最早是由 [Hochreiter 和 Schmidhuber 在 1997 年提出](#)，就是为了解决这个不稳定梯度的问题。LSTM 让 RNN 训练变得相当简单，很多近期的论文（包括我在上面给出的那些）都是用了 LSTM 或者相关的想法。

深度信念网络，生成式模型和 **Boltzmann** 机：对深度学习的兴趣产生于 2006 年，最早的论文就是解释如何训练称为 深度信念网络（DBN）的网络。

参见 Geoffrey Hinton, Simon Osindero 和 Yee-Whye Teh 在 2006 年的 [A fast learning algorithm for deep belief nets](#)，及 Geoffrey Hinton 和 Ruslan Salakhutdinov 在 2006 年的相关工作 [Reducing the dimensionality of data with neural networks](#)

DBN 在之后一段时间内很有影响力，但近些年前馈网络和 RNN 的流行，盖过了 DBN 的风头。尽管如此，DBN 还是有几个有趣的特性。一个就是 DBN 是一种生成式模型。在前馈网络中，我们指定了输入的激活函数，然后这些激活函数便决定了网络中后面的激活值。而像 DBN 这样的生成式模型可以类似这样使用，但是更加有用的可能就是指定某些特征神经元的

null

null

值，然后进行“反向运行”，产生输入激活的值。具体讲，DBN 在手写数字图像上的训练同样可以用来生成和手写数字很像的图像。换句话说，DBN 可以学习写字的能力。所以，生成式模型更像人类的大脑：不仅可以读数字，还能够写出数字。用 Geoffrey Hinton 本人的话就是：“要识别对象的形状，先学会生成图像。”（to recognize shapes, first learn to generate images）另一个是 DBN 可以进行无监督和半监督的学习。例如，在使用图像数据学习时，DBN 可以学会有用的特征来理解其他的图像，即使，训练图像是无标记的。这种进行非监督学习的能力对于根本性的科学理由和实用价值（如果完成的足够好的话）来说都是极其有趣的。

所以，为何 DBN 在已经获得了这些引人注目的特性后，仍然逐渐消失在深度学习的浪潮中呢？部分原因在于，前馈网络和 RNN 已经获得了很多很好的结果，例如在图像和语音识别的标准测试任务上的突破。所以大家把注意力转到这些模型上并不奇怪，这其实也是很合理的。然而，这里隐藏着一个推论。研究领域里通常是赢家通吃的规则，所以，几乎所有的注意力集中在最流行的领域中。这会给那些进行目前还不很流行方向上的研究人员很大的压力，虽然他们的研究长期的价值非常重要。我个人的观点是 DBN 和其他的生成式模型应该获得更多的注意。并且我对今后如果 DBN 或者相关的模型超过目前流行的模型也毫不诧异。欲了解 DBN，参考这个[DBN 综述](#)。还有[这篇文章](#)也很有用。虽然没有主要地将 DBN，但是已经包含了很多关于 DBN 核心组件的受限 Boltzmann 机的有价值的信息。

其他想法：在神经网络和深度学习中还有其他哪些正在进行的研究？恩，其实还有很多大量的其他美妙的工作。热门的领域包含使用神经网络来做[自然语言处理 natural language processing](#)、[机器翻译 machine translation](#)，和更加惊喜的应用如[音乐信息学 music informatics](#)。当然其他还有不少。在读者完成本书的学习后，应该可以跟上其中若干领域的近期工作，可能你还需要填补一些背景知识的缺漏。在本节的最后，我再提一篇特别有趣的论文。这篇文章将深度卷积网络和一种称为强化学习的技术来学习[玩电子游戏 play video games well](#)（参考[这里 this followup](#)）。其想法是使用卷积网络来简化游戏界面的像素数据，将数据转化成一组特征的简化集合，最终这些信息被用来确定采用什么样的操作：“上”、“下”、“开火”等。特别有趣的是单一的网络学会 7 款中不同的经典游戏，其中 3 款网络的表现已经超过了人类专家。现在，这听起来是噱头，当然他们的标题也挺抓眼球的——“Playing Atari with reinforcement learning”。但是透过表象，想想系统以原始像素数据作为输入，它甚至不知道游戏规则！从数据中学会在几种非常不同且相当敌对的场景中做出高质量的决策，这些场景每个都有自己复杂的规则集合。所以这的解决是非常干净利落的。

神经网络的未来

意图驱动的用户接口：有个很古老的笑话是这么说的：“一位不耐烦的教授对一个困惑的学生说道，‘不要光听我说了什么，要听懂我说的含义。’”。历史上，计算机通常是扮演了笑话中困惑的学生这样的角色，对用户表示的完全不知晓。而现在这个场景发生了变化。我仍然记得自己在 Google 搜索的打错了一个查询，搜索引擎告诉了我“你是否要的是[这个正确的查询]？”，然后给出了对应的搜索结果。Google 的 CEO Larry Page 曾经描述了最优搜索引擎就是准确理解用户查询的含义，并给出对应的结果。

这就是意图驱动的用户接口的愿景。在这个场景中，不是直接对用户的查询词进行结果的反
null

null

馈，搜索引擎使用机器学习技术对大量的用户输入数据进行分析，研究查询本身的含义，并通过这些发现来进行合理的动作以提供最优的搜索结果。

而意图驱动接口这样的概念也不仅仅用在搜索上。在接下来的数十年，数以千计的公司会将产品建立在机器学习来设计满足更高的准确率的用户接口上，准确地把握用户的意图。现在我们也看到了一些早期的例子：如苹果的Siri；Wolfram Alpha；IBM 的 Watson；可以对照片和视频进行注解的系统；还有更多的。

大多数这类产品会失败。启发式用户接口设计非常困难，我期望有更多的公司会使用强大的机器学习技术来构建无聊的用户接口。最优的机器学习并不会在你自己的用户接口设计很糟糕时发挥出作用。但是肯定也会有能够胜出的产品。随着时间推移，人类与计算机的关系也会发生重大的改变。不久以前，比如说，2005年——用户从计算机那里得到的是准确度。因此，很大程度上计算机很古板的；一个小小的分号放错便会完全改变和计算机的交互含义。但是在以后数十年内，我们期待着创造出意图驱动的用户借款购，这也会显著地改变我们在与计算机交互的期望体验。

机器学习，数据科学和创新的循环：当然，机器学习不仅仅会被用来建立意图驱动的接口。另一个有趣的应用是数据科学中，机器学习可以找到藏在数据中的“确知的未知”。这已经是非常流行的领域了，也有很多的文章和书籍介绍了这一点，所以本文不会涉及太多。但我想谈谈比较少讨论的一点，这种流行的后果：长期看来，很可能机器学习中最大的突破并不会任何一种单一的概念突破。更可能的情况是，最大的突破是，机器学习研究会获得丰厚的成果，从应用到数据科学及其他领域。如果公司在机器学习研究中投入1美元，则有1美元加10美分的回报，那么机器学习研究会有很充足的资金保证。换言之，机器学习是驱动几个主要的新市场和技术成长的领域出现的引擎。结果就是出现拥有精通业务的大团队，能够获取足够的资源。这样就能够将机器学习推向更新的高度，创造出更多市场和机会，一种高级创新的循环。

神经网络和深度学习的角色：我已经探讨过机器学习会成为一个技术上的新机遇创建者。那么神经网络和深度学习作为一种技术又会有什么样的独特的贡献呢？

为了更好地回答这个问题，我们来看看历史。早在1980年代，人们对神经网络充满了兴奋和乐观，尤其是在BP被大家广泛知晓后。而在1990年代，这样的兴奋逐渐冷却，机器学习领域的注意力转移到了其他技术上，如SVM。现在，神经网络卷土重来，刷新了几乎所有的记录，在很多问题上也都取得了胜利。但是谁又能说，明天不会有一种新的方法能够击败神经网络？或者可能神经网络研究的进程又会阻滞，等不来没有任何的进展？

所以，可能更好的方式是看看机器学习的未来而不是单单看神经网络。还有个原因是人们对神经网络的理解还是太少了。为何神经网络能够这么好地泛化？为何在给定大规模的学习的参数后，采取了一些方法后可以避免过匹配？为何神经网络中随机梯度下降很有效？在数据集扩展后，神经网络又能达到什么样的性能？如，如果ImageNet扩大10倍，神经网络的性能会比其他的机器学习技术好多少？这些都是简单，根本的问题。当前，我们都对它们理解的很少。所以，要说神经网络在机器学习的未来要扮演什么样的角色，很难回答。

我会给出一个预测：我相信，深度学习会继续发展。学习概念的层次特性、构建多层抽象的能力，看起来能够从根本上解释世界。这也并不是说未来的深度学习研究者的想法发生变

null

null

化。我们看到了，在那些使用的神经单元、网络的架构或者学习算法上，都出现了重大转变。如果我们不再将最终的系统限制在神经网络上时，这些转变将会更加巨大。但人们还是在进行深度学习的研究。

神经网络和深度学习将会主导人工智能？本书集中在使用神经网络来解决具体的任务，如图像分类。现在更进一步，问：通用思维机器会怎么样？神经网络和深度学习能够帮助我们解决（通用）人工智能（AI）的问题么？如果可以，以目前深度学习领域的发展速度，我们能够期待通用 AI 在未来的发展么？认真探讨这个问题可能又需要另写一本书。不过，我们可以给点意见。其想法基于 [Conway's law](#)：

任何设计了一个系统的组织…… 最终会不可避免地产生一个设计，其结构本身是这个组织的交流结构

所以，打个比方，Conway 法则告诉我们波音 747 客机的设计会镜像在设计波音 747 那时的波音及其承包商的组织结构。或者，简单举例，假设一个公司开发一款复杂的软件应用。如果应用的 dashboard 会集成一些机器学习算法，设计 dashboard 的人员最好去找公司的机器学习专家讨论一下。Conway 法则就是这种观察的描述，可能更加宏大。第一次听到 Conway 法则，很多人的反应是：“好吧，这不是很显然么？”或者“这是不是不对啊？”让我来对第二个观点进行分析。作为这个反对的例子，我们可以看看波音的例子：波音的审计部门会在哪里展示 747 的设计？他们的清洁部门会怎么样？内部的食品供应？结果就是组织的这些部门可能不会显式地出现在 747 所在的任何地方。所以我们应该理解 Conway 法则就是仅仅指那些显式地设计和工程的组织部门。

null