人工神经网络与深度学习

从入门到放弃

作者：Michael Nielsen

时间：October 22, 2021

版本：0.1

翻译：Euler Meng

关于人工智能的最大的问题在于，人们认为他们真正理解这一技术。

目录

第一章 使用神经网络识别手写数字

内容提要

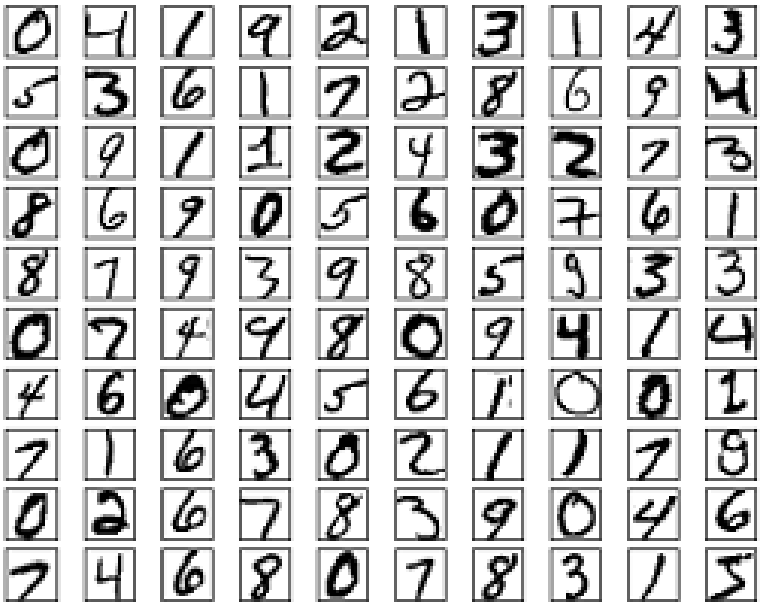
- 感知机
- Sigmoid 神经元
- 神经网络架构
- 手写数字分类网络
- 梯度下降
- 神经网络实现

人类的视觉系统是世界上的奇迹之一。考虑一下以下的手写数字序列：

大多数人毫不费力地认出这些数字是 504192，但这种轻松是有欺骗性的。在人类大脑的每个半球，都有一个初级视觉皮层，也被称为 $V1$，包含 1.4 亿个神经元，它们之间有几百亿个连接。然而，人类的视觉不仅涉及 $V1$，还涉及整个视觉皮层——$V2$、$V3$、$V4$ 和 $V5$——逐步进行更加复杂的图像处理。人类的大脑是一台超级计算机，经过数亿年的进化调整，极好地适应了对视觉世界的理解。识别手写的数字并不容易，恰恰相反，人类在理解眼睛所看到的东西方面有着令人震惊的能力，但因为几乎所有工作都是在无意识中完成的，所以通常我们并不了解我们的视觉系统解决了多么艰难的问题。

如果你试图编写一个计算机程序来识别上述数字，那么视觉模式识别的难度就会变得很明显。人类识别形状似乎很容易，但想转化为程序突然变得极其困难，关于我们如何识别形状的简单直觉——“9 在顶部有一个环，右下方有一个垂直的笔画”——用算法来表达这些规则并不那么简单。并且当你试图使这种规则精确化时，你很快就会迷失在异常和特殊情况的泥潭中，看上去毫无希望。

神经网络以一种不同的方式处理这个问题，这个想法是获取大量的手写数字，称为训练样本，



然后开发一个可以从这些训练样本中学习的系统。换句话说，神经网络使用这些样本来自动推断出识别手写数字的规则。此外，通过增加训练样本的数量，网络可以学习更多关于笔迹的知识，从而提高其准确性。因此，虽然我在上面只展示了 100 个训练数字样本，但也许我们可以通过使用数千甚至数百万或数十亿的训练样本来创建一个更好的手写识别器。

在本章中，我们将编写程序实现一个学习识别手写数字的神经网络。这个程序只有 74 行，而且没有使用特殊的神经网络库。但这个短小精悍的程序可以识别数字，准确率超过 96%，而且无需人工干预。此外，在后面的章节中，我们将发展一些想法，将准确率提升到 99% 以上。事实上，现在最好的商业神经网络已经非常出色，银行用它来处理支票，邮局用它来识别地址。

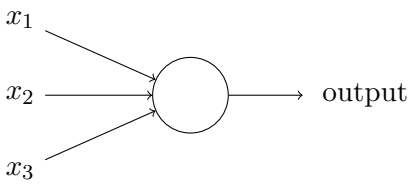
我们专注于手写识别，大体而言，它是学习神经网络的优秀原型问题。作为一个原型，它击中了一个甜点：它具有挑战性——识别手写数字是一个不小的成就——但它又并不过于困难，以至于需要一个极其复杂的解决方案，或巨大的计算能力。此外，它也是发展更多高级技术的好起点，比如深度学习。因此，在本书中，我们将反复回到手写识别的问题上。本书后面部分，我们将讨论如何将这些想法应用于其他计算机视觉问题，以及语音、自然语言处理和其他领域。

当然，如果只是编写一个识别手写数字的计算机程序，那么这一章可以短得多！但在这一过程中，我们会发展出许多关于神经网络的关键思想，包括两种重要的人工神经元（感知机和Sigmoid 神经元），以及标准神经网络学习算法，即随机梯度下降算法。在整个过程中，我着重于解释事情为什么要这样做，以便建立起你对于神经网络的直觉。这比我只介绍基本的机械原理需要更多的讨论，但这是值得的，因为你会由此获得更深的理解。作为回报，在本章结束时，我们将能够理解什么是深度学习，以及它为什么重要。

1.1 感知机

什么是神经网络？首先我将解释一种叫做**感知机**的人工神经元模型，感知机是在 20 世纪五、六十年代由科学家Frank Rosenblatt 发明的，灵感来自Warren McCulloch和 Walter Pitts的早期 工作。今天，使用其他类型的人工神经元模型更为普遍——在本书中，以及在许多现代的神经网络工作中，主要使用一种叫做**Sigmoid 神经元**的模型。我们很快就会了解到Sigmoid 神经元，但是为了理解为什么要以这种方式定义Sigmoid 神经元，值得花时间先了解一下感知机。

那么，感知机是如何工作的？一个感知机接受几个二进制输入，$x_1, x_2, \ldots$，并产生一个单一的二进制输出：



在上面的例子中，感知机有三个输入，x_1, x_2, x_3，一般来说，它可以有更多或更少的输入。Rosenblatt 提出了一个简单的规则来计算输出。他引入了**权重**，$w_1, w_2, \ldots$，用于表征每个输入对输出重要性的实数。神经元的输出，0 或 1，是由加权和 $\sum_j w_j x_j$ 是否小于或大于某个**阈值**决定的。就像权重一样，阈值也是一个实数，是神经元的一个参数。用更精确的代数形式表示如下：

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

这就是感知机工作原理的全部内容！

这就是基本的数学模型，理解感知机的一种方式是，它是一种依据权重做出决定的设备。让我举一个例子，这不是一个很现实的例子，但它很容易理解，我们很快就会有更现实的例子。假设周末快到了，你听说在你的城市会有一个奶酪节，你喜欢奶酪，并试图决定是否要去参加。你可能会通过给三个因素设置权重来做决定。

1. 天气好吗？
2. 你的男朋友或女朋友是否愿意陪你去？
3. 节日举办地点是否靠近公共交通？(你没有自己的车)

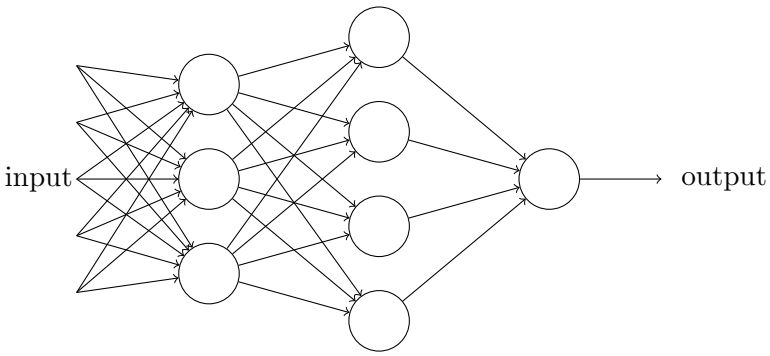
我们可以用相应的二进制变量 $x1$，$x2$，和 $x3$ 来表示这三个因素。例如，如果天气好，我们会有 $x1 = 1$，如果天气不好，$x1 = 0$。同样地，如果你的男朋友或女朋友想去，$x2 = 1$，如果不想去，$x2 = 0$。同样，$x3$ 和公共交通也是如此。

现在，假设你酷爱奶酪，以至于即使你的男朋友或女朋友对奶酪不感兴趣，或交通不便，你都愿意去参加。但也许你非常厌恶坏天气，如果天气不好，你就不会出门。你可以用感知机来模拟这种决策。一种方法是为天气选择一个权重$w1 = 6$，其他条件选择 $w2 = 2$ 和 $w3 = 2$。$w1$ 的值越大，表明天气对你来说越重要，比你的男朋友或女朋友是否和你一起去，或者公交站点的远近都重要。最后，假设你为感知机选择一个 5 的阈值。有了这些，感知机就实现了所需的决策模型，只要天气好就输出 1，天气不好就输出 0。你的男朋友或女朋友是否想去，或者附近是否有公交站点，对决策结果没有影响。

通过改变权重和阈值，我们可以得到不同的决策模型。例如，假设我们选择了一个 3 的阈值，那

么感知机就会决定，只要天气好，或者节日举办地附近有公交站点，而且你的男朋友或女朋友愿意和你一起去，你就应该去参加节日。换句话说，这将是一个不同的决策模型，降低阈值意味着你更愿意去参加节日。

很明显，感知机并不是人类做决策时使用的完整模型！但这个例子说明了感知机可以如何权衡不同种类的依据以做出决策。一个复杂的感知机网络应该可以做出相当微妙的决定，这似乎是合理的推测。



在这个网络中，第一列感知机——我们称之为第一层感知机——正在通过权衡作为依据的输入值，做三个非常简单的决定。那么第二层的感知机呢？每一个感知机都在通过权衡第一层的决策结果来做决定。这样，第二层的感知机可以在比第一层的感知机更复杂、更抽象的水平上做出决定。而第三层的感知机甚至可以做出更复杂的决策。这样一来，多层感知机网络就可以从事复杂的决策。

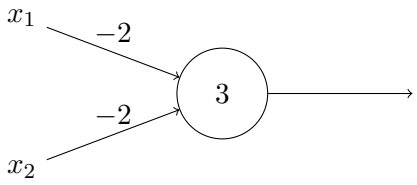
顺便说一下，当我定义感知机时，我说一个感知机只有一个输出值。但在上面的网络中，感知机看起来像是有多个输出值。事实上，它们仍然只有一个单一的输出值。多个输出的箭头只是一种有用的简化方式，表明一个感知机的输出被用作其他几个感知机的输入。这比画一个单一的输出线，然后再分割开来要方便一些。

让我们简化一下描述感知机的方式，$\sum_j w_j x_j > \text{threshold}$ 的条件有些冗长，我们可以做两个符号上的改变来简化它。第一个变化是把 $\sum_j w_j x_j$ 写成点积，$w \cdot x \equiv \sum_j w_j x_j$，其中 w 和 x 是向量，其组成部分分别是权重和输入的向量。第二个变化是把阈值移到不等式的另一边，用感知机的**偏置** $b \equiv -threshold$ 来代替它。用偏置代替阈值后，感知机规则可以被重写为：

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

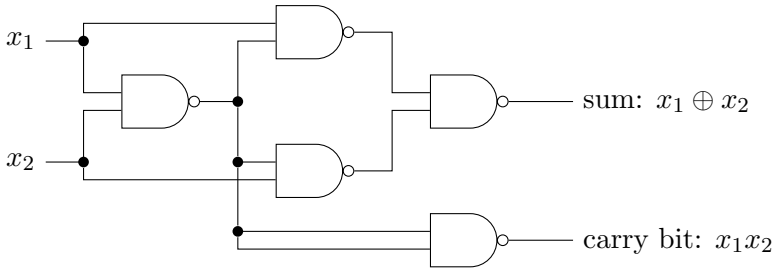
你可以把偏置看作是衡量让感知机输出 1 的难易程度。或者用更加偏向生物学的术语来说，偏置是衡量让感知机激活的难易程度。对于一个有很大偏置的感知机来说，感知机输出 1 是非常容易的。但是如果偏置是绝对值非常大的负数，那么感知机就很难输出 1。显然，引入偏置只是我们描述感知机的一个小变化，但是我们以后会看到它可以导致符号的进一步简化。正因为如此，在本书的其余部分，我们将不使用阈值，我们总是使用偏置。

上面把感知机描述为一种权衡依据以做出决定的设备，另一种理解感知机的方式是计算基本的逻辑功能，即我们通常认为的运算基础，如“**与**”，“**或**”和“**与非**”等函数。例如，假设我们有一个有两个输入的感知机，每个输入的权重为 -2，总体偏置为 3，这是我们的感知机：

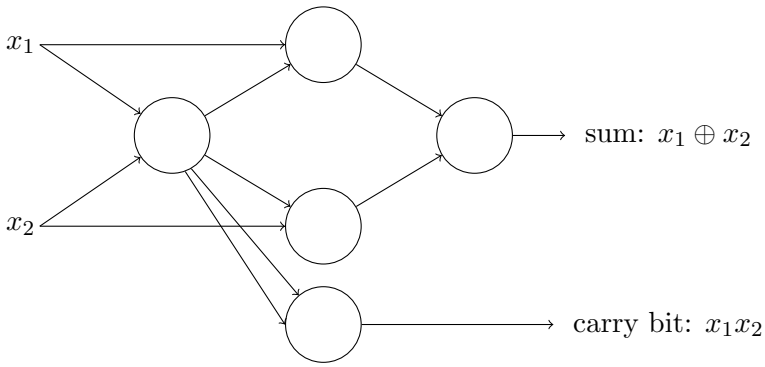


然后我们看到，输入 00 产生输出 1，因为 $(-2)\cdot 0+(-2)\cdot 0+3=3$ 是正数。在这里，我引入了 $\cdot$ 符号以使乘法明确。类似的计算表明，输入 01 和 10 产生输出 1。但是输入 11 产生输出 0，因为 $(-2)\cdot 1+(-2)\cdot 1+3=-1$ 是负数。所以我们的感知机实现了一个与非门!

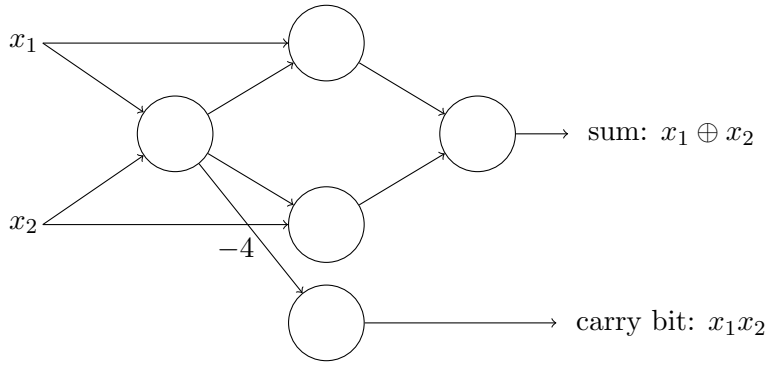
与非门的例子表明，我们可以用感知机来计算简单的逻辑函数。事实上，我们可以使用感知机网络来计算任何逻辑函数，其原因是与非门是通用的计算方法，也就是说，我们可以用与非门来构建任何计算。例如，我们可以使用与非门来构建一个电路，将两个二进制位 $x1$ 和 $x2$ 相加。这需要计算位相加，$x_1 \oplus x_2$，以及一个进位，当 $x1$ 和 $x2$ 都是 1 时，进位被设置为 1，也就是说，进位只是位相乘 $x_1 \cdot x_2$：



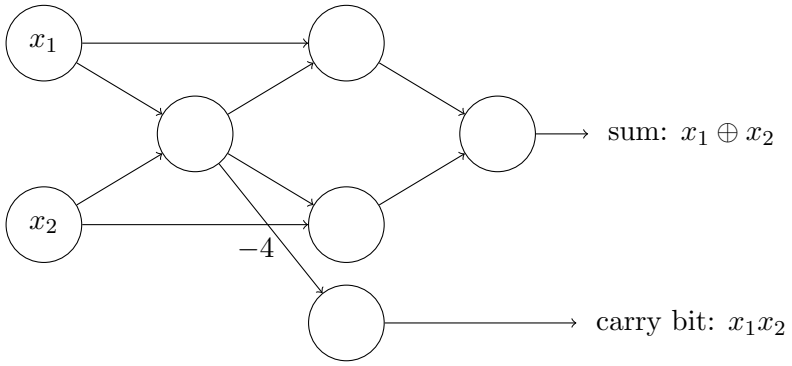
为了得到一个等效的感知机网络，我们用具有两个输入的感知机替换所有的与非门，每个感知机的权重为 -2，总的偏置为 3，这是得到的网络。注意，我把对应于右下角与非门的感知机移动了一点，只是为了让图上的箭头更容易画出来。



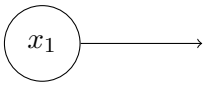
这个感知机网络的一个值得注意的方面是，最左边的感知机的输出被两次用作最下面的感知机的输入。当我定义感知机模型的时候，我没有说这种双重输出到同一位置的做法是否被允许，实际上，这并不太重要。如果我们不想允许这种事情，那么可以简单地将两条线合并，变成一个权重为 -4 的单一连接，而不是两个权重为 -2 的连接。（如果你不觉得这很明显，你应该停下来，向自己证明这是等价的）有了这个变化，网络看起来如下，所有未标记的权重都等于 -2，所有的偏置都等于 3，而单一的权重为 -4，如标记的那样。



到目前为止，我把像 x_1 和 x_2 这样的输入画成感知机网络左边浮动的变量，实际上，可以画一层额外的感知机——输入层——来方便对输入编码：



这个感知机的符号，有一个输出，但没有输入，



是一个速记法，它实际上并不意味着一个没有输入的感知机。要看到这一点，假设我们确实有一个没有输入的感知机。那么加权和 $\sum_j w_j x_j$ 将总是零，所以如果 $b > 0$，感知机将输出 1，如果 $b \leq 0$，则输出 0。也就是说，感知机只是输出一个固定值，而不是期望值（上面的例子中是 $x1$）。最好是把这个输入层看作根本不是真正的感知机，而是简单定义为输出期望值 $x_1, x_2, \ldots$ 的特殊单元。

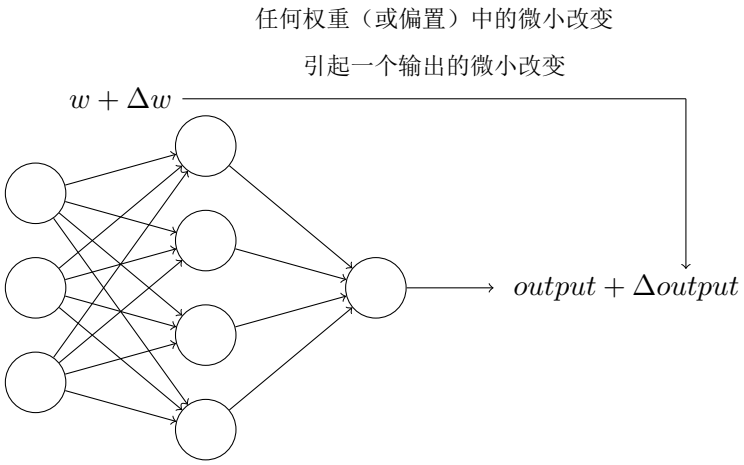
加法器的例子展示了一个感知机网络如何被用来模拟一个包含许多与非门的电路，因为与非门是通用的计算方法，所以感知机也是通用的计算方法。

感知机的通用计算性既令人欣慰又令人失望。令人欣慰的是，它告诉我们，感知机网络可以像任何其他计算设备一样强大。但它也令人失望，因为它使我们觉得感知机似乎只是一种新型的与非门，这并不是什么大新闻！

然而，实际情况比这种观点所认为的要好。事实证明，我们可以设计**学习算法**，自动调整人工神经元网络的权重和偏置。这种调整是对外部刺激的反应，无需程序员的直接干预。这些学习算法使我们能够以一种与传统逻辑门完全不同的方式使用人工神经元，我们的神经网络可以简单地通过学习来解决问题，而不是明确地布置一个由与非门和其他门组成的电路，这些问题有时候直接用传统的电路设计是很难解决的。

1.2 Sigmoid 神经元

学习算法听起来非常棒，但我们如何为神经网络设计这样的算法呢？假设我们有一个感知机网络，我们想用它来学习解决一些问题。例如，网络的输入可能是一个数字的手写扫描图像的原始像素数据，我们希望网络能够学习权重和偏置，从而使网络的输出能够正确地对数字进行分类。为了了解学习是如何进行的，假设我们对网络中的一些权重（或偏置）做一个小的改变。我们希望这个权重的微小改变只会导致网络输出产生相应的微小改变。稍后我们将看到，这一特性将使学习成为可能。从原理上讲，这就是我们想要的东西（很明显这个网络太简单了，无法进行手写识别！）。

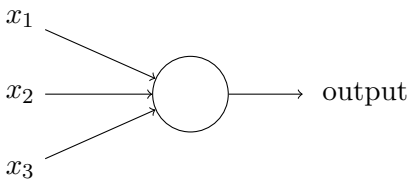


如果权重（或偏置）的微小改变真的只导致输出的微小改变，那么我们就可以利用这一事实来修改权重和偏置，使我们的网络表现得更符合我们想要的方式。例如，假设该网络错误地将一张图片分类为“8”，而它应该是“9”，我们可以想办法对权重和偏置做一个小小的改变，使网络更接近于将图像分类为“9”。然后我们会重复这个过程，不断地改变权重和偏置，以产生越来越好的输出，这时网络就在学习。

问题是，当我们的网络包含感知机时，情况并非如此。事实上，网络中任何一个感知机的权重或偏置的微小变化有时会导致该感知机的输出完全翻转，例如从 0 到 1。那样的翻转可能接下来引起网络其余部分的行为以极其复杂的方式被完全改变。因此，虽然你的“9”现在可能被正确分类，但网络在所有其他图像上的行为可能以某种难以控制的方式完全改变这个分类结果，这使得我们很难看到应该如何逐步修改权重或偏置，以使网络更接近理想的行为。也许有一些聪明的方法来解决这个问题，但是目前为止，我们还没发现有什么办法能让感知机网络进行学习。

我们可以通过引入一种新型的人工神经元来克服这个问题，这种神经元被称为Sigmoid 神经元。Sigmoid 神经元类似于感知机，但经过修改后，其权重和偏置的微小改变只导致其输出的微小改变，这是一个关键的事实，它将使Sigmoid 神经元网络具备学习的能力。

好吧，让我来描述一下Sigmoid 神经元，我们将以描述感知机的同样方式来描述Sigmoid 神经元：



就像感知机一样，Sigmoid 神经元有输入 $x_1, x_2, \ldots$，但这些输入不是只有 0 或 1，而是可以取 0 和 1 之间的任何数值。因此，例如，0.638... 是一个Sigmoid 神经元的有效输入。另外，就像感知机一样，Sigmoid 神经元对每个输入都有权重，$w_1, w_2, \ldots$，还有一个整体偏置，b。Sigmoid 神经元输出不是 0 或 1，相反，它是 $\sigma(w \cdot x + b)$)，其中 σ 被称为Sigmoid 函数[1]，其定义如下:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

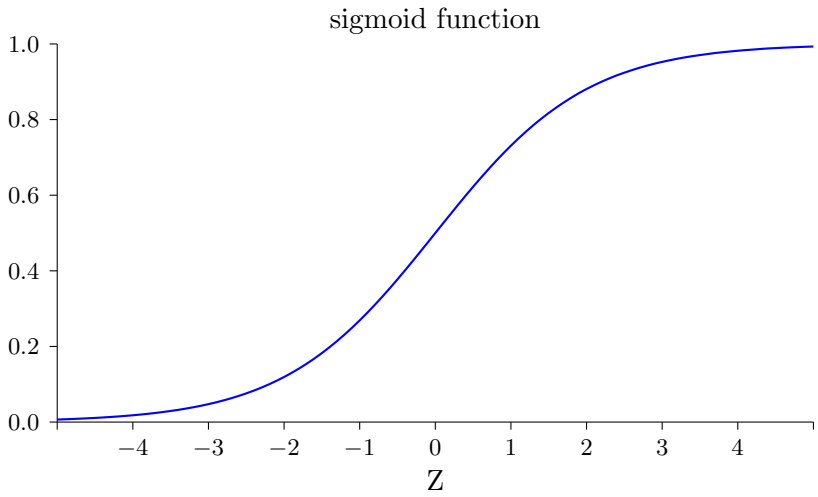
更明确地说，输入为 $x_1, x_2, \ldots$，权重为 $w_1, w_2, \ldots$，偏置为 b 的Sigmoid 神经元的输出为:

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

乍一看，Sigmoid 神经元似乎与感知机非常不同，如果你还不熟悉，sigmoid 函数的代数形式可能看起来晦涩且令人望而生畏。事实上，感知机和Sigmoid 神经元之间有许多相似之处，Sigmoid 函数的代数形式更像是一个技术细节，而不是理解的真正障碍。

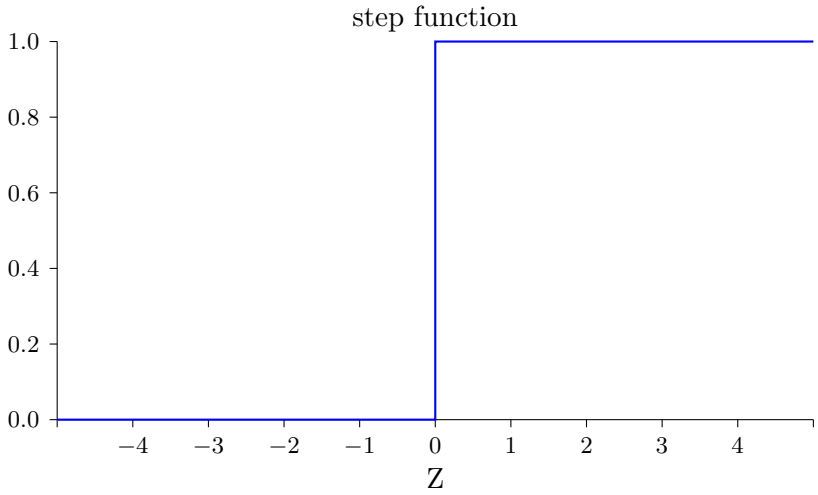
为了理解与感知机模型的相似性，假设 $z = w \cdot x + b$ 是一个很大的正数。那么 $e^{-z} \approx 0$ ，所以 $\sigma(z) \approx 1$。换句话说，当 $z = w \cdot x + b$ 为大正数时，Sigmoid 神经元的输出约为 1，就像感知机一样。另一方面，假设 $z = w \cdot x + b$ 是绝对值非常大的负数时，那么 $e^{-z} \to \infty$，$\sigma(z) \approx 0$。因此，当 $z = w \cdot x + b$ 为绝对值非常大的负数时，Sigmoid 神经元的行为也近似于感知机。只有当 $w \cdot x + b$ 的大小适中时，才会与感知机模型有很大偏差。

σ 的代数形式是什么样的呢？我们怎么去理解它呢？事实上，σ 的精确形式并不那么重要——真正重要的是绘图时的函数形状，这里是 σ 的形状:



这个形状是阶跃函数平滑后的版本：

[1]顺便说一下，σ 有时被称为**逻辑函数**，而这一类新的神经元被称为**逻辑神经元**。记住这些术语是很有用的，因为许多从事神经网的人都使用这些术语，然而，我们将坚持使用Sigmoid 神经元这个术语



如果 σ 实际上是一个阶跃函数，那么Sigmoid 神经元将是一个感知机，因为输出是 1 或 0，取决于 $w \cdot x + b$ 是正还是负。[2] 通过使用实际的 σ 函数，我们得到了，正如上面已经暗示的，一个平滑的感知机。事实上，σ 函数的平滑性才是关键的事实，而不是它的详细形式。σ 的平滑性意味着权重的微小变化 Δw_j 和偏置的微小变化 Δb 将使神经元的输出产生微小变化 Δoutput。事实上，微积分告诉我们，Δoutput 可以很好地被近似为:

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

其中，求和是在所有权重 w_j 上进行的，而 $\partial \, \text{output}/\partial w_j$ 和 $\partial \, \text{output}/\partial b$ 分别表示输出相对于 w_j 和 b 的偏导。如果你对偏导不适应，也不要惊慌! 虽然上面的表达式看起来很复杂，包含所有权重的偏导，但它实际上说的是非常简单的事情（这可是个好消息）：Δoutput 是权重和偏置变化 Δw_j 和 Δb 的线性函数。这种线性使得我们很容易选择权重和偏置的微小变化来实现输出中任何想要的微小变化。因此，虽然Sigmoid 神经元具有与感知机相同的定性行为，但它们更容易弄清改变权重和偏置将如何改变输出。

如果真正重要的是 σ 的形状，而不是它的确切形式，那么为什么要使用公式(3)中 σ 的特定形式? 事实上，在本书的后面，我们将偶尔考虑使用其他激活函数 $f(\cdot)$，输出为 $f(w \cdot x + b)$ 的神经元。当我们使用不同的激活函数时，主要的变化是公式(5)中偏导数的特定值发生了变化。实际上，当我们以后计算这些偏导数时，使用 σ 将会简化代数计算，这只是因为指数在求导时具有可爱的特性。无论如何，σ 是神经网络工作中常用的，也是我们在本书中最常使用的激活函数。

我们应该如何解释一个Sigmoid 神经元的输出? 很明显，感知机和Sigmoid 神经元之间的一个很大的区别是 Sigmoid 神经元不只是输出 0 或 1，它们可以将 0 和 1 之间的任何实数作为输出，所以诸如 0.173... 和 0.689... 这样的值都是合法的输出。这可能很有用，例如，如果我们想用输出值来代表输入到神经网络的图像中的像素的平均强度。但有时它也会成为一种困扰，例如，我们想让网络的输出表示“输入图像是 9”或“输入图像不是 9”，显然，就像在感知机中一样，如果输出是 0 或 1，这是最简单的。但在实践中，我们可以建立一个约定来处理这个问题，例如，约定将任何至少为 0.5 的输出解释为表示“9”，而任何小于 0.5 的输出则表示“不是 9”。当我们使用这样的约定时，我总是会明确说明，

[2]实际上，当 $w \cdot x + b = 0$ 时，感知机输出 0，而阶跃函数输出 1。因此，严格来说，我们需要在这一点上修改阶跃函数。但你会明白这个道理

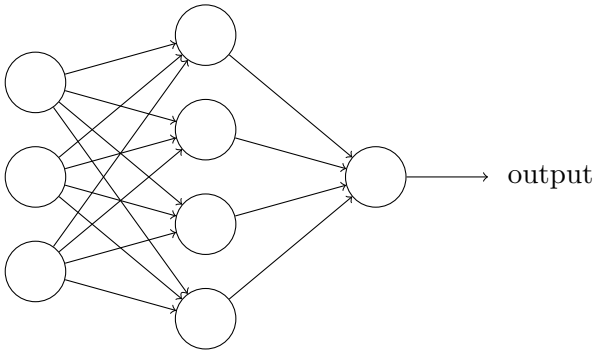
所以它应该不会引起任何混淆。

✍ **练习 1.1** 假设我们把一个感知机网络中的所有权重和偏置，乘以一个正的常数 $c > 0$，证明网络的行为没有改变。

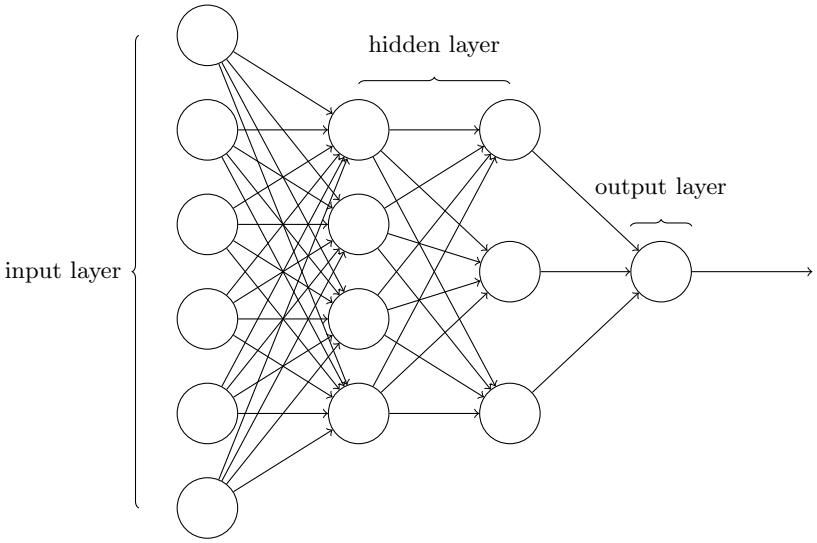
✍ **练习 1.2** 假设我们有和上一个问题一样的设置——一个感知机网络，假设对感知机网络的总体输入已经选定，我们不需要实际的输入值，我们只需要假设输入值已经固定。假设权重和偏置是这样的：对于网络中任何特定感知机的输入 x，$w \cdot x + b \neq 0$。现在用Sigmoid 神经元替换网络中的所有感知机，并将权重和偏置乘以一个正常数 $c > 0$。证明在 $c \to \infty$ 的极限情况下，Sigmoid 神经元网络的行为和感知机网络的完全一致。当一个感知机的 $w \cdot x + b = 0$ 时两者又为什么会存在差异？

1.3 神经网络的架构

在下一节中，我将介绍一个神经网络，它可以很好地对手写数字进行分类，作为准备，解释一些可以让我们命名网络中不同部分的术语是很有帮助的。假设我们有了这个网络:



如前所述，该网络中最左边的一层被称为输入层，该层内的神经元被称为**输入神经元**。最右边的，即输出层包含有**输出神经元**，在本例中，只包含一个输出神经元。中间层被称为**隐藏层**，因为该层的神经元既不是输入也不是输出。“隐藏”这个词听起来也许有点神秘——我第一次听到这个词时，以为它一定有什么深刻的哲学或数学意义——但它的真正含义不过是“不是输入或输出”。上面的网络只有一个隐藏层，但有些网络有多个隐藏层。例如，下面这个四层网络有两个隐藏层：



有点令人困惑的是，由于历史原因，这种多层网络有时被称为多层感知机或MLP，尽管它是由Sigmoid神经元组成的，而不是感知机。我不打算在本书中使用 MLP 的术语，因为我认为这会引起混淆，但这里想提醒你它的存在。

网络中输入层和输出层的设计通常是直截了当的，例如，假设我们试图确定一个手写图像是否描绘了一个“9”，设计网络的一个自然方法是将图像像素的强度编码到输入神经元中，如果图像是 64×64 的灰度图像，那么我们就会有 $4,096 = 64 \times 64$ 的输入神经元，其强度在 0 和 1 之间取合适的值。输出层将只包含一个神经元，输出值小于 0.5 表示“输入图像不是 9”，大于 0.5 表示“输入图像是 9”。

虽然神经网络的输入层和输出层的设计通常是简单明了的，但隐藏层的设计则堪称一门艺术。特别是，不可能用几个简单的经验法则来总结隐藏层的设计过程。相反，神经网络研究人员已经为隐藏层设计开发了许多启发式方法，帮助人们从他们的网络中获得他们想要的行为。例如，这种启发式方

法可以用来帮助在隐藏层的数量和训练网络所需的时间二者中找到一个均衡。在本书后面，我们会遇到几个这样的设计启发式方法。

到目前为止，我们讨论的神经网络，都是其中一个层的输出被用作下一个层的输入。这种网络被称为**前馈**神经网络。这意味着网络中没有回路——信息总是向前传输，而从不向后回馈。如果存在回路，我们最终会遇到 σ 函数的输入取决于输出的情况。这就很难理解了，所以我们不允许有这种回路。

然而，还有其他的人工神经网络模型，其中反馈回路是可能的，这些模型被称为rnn。这种模型的想法是让神经元在某个有限的时间段内激活，然后变成休眠状态。处于激活态的神经元可以激活其他的神经元，使这些神经元随后被激活且同样保持一段有限的时间，这将导致更多的神经元被激活，因此随着时间的推移，我们得到了一个神经元激活的级联系统。在这个的模型中，因为神经元的输出只在一段时间后影响其输入，而不是即刻影响，所以回路不会造成问题。

循环神经网络的影响力不如前馈网络，部分原因是循环神经网络的学习算法（至少到目前为止）不够强大，但是，循环神经网络仍然非常有吸引力，它们在原理上比前馈网络更接近我们的大脑工作方式，而且，循环神经网络有可能解决一些重要的问题，这些问题如果仅仅用前馈网络来解决，则更加困难。然而，为了限制篇幅，在这本书中，我们将集中讨论更广泛使用的前馈网络。

1.4 一个简单的分类手写数字的网络

在定义了神经网络之后，我们再来看看手写识别，我们可以把识别手写数字的问题分成两个子问题。首先，我们希望有一种方法可以将包含许多数字的图像分解为一连串独立的图像，每张图像都包含一个数字。例如，我们想把图像

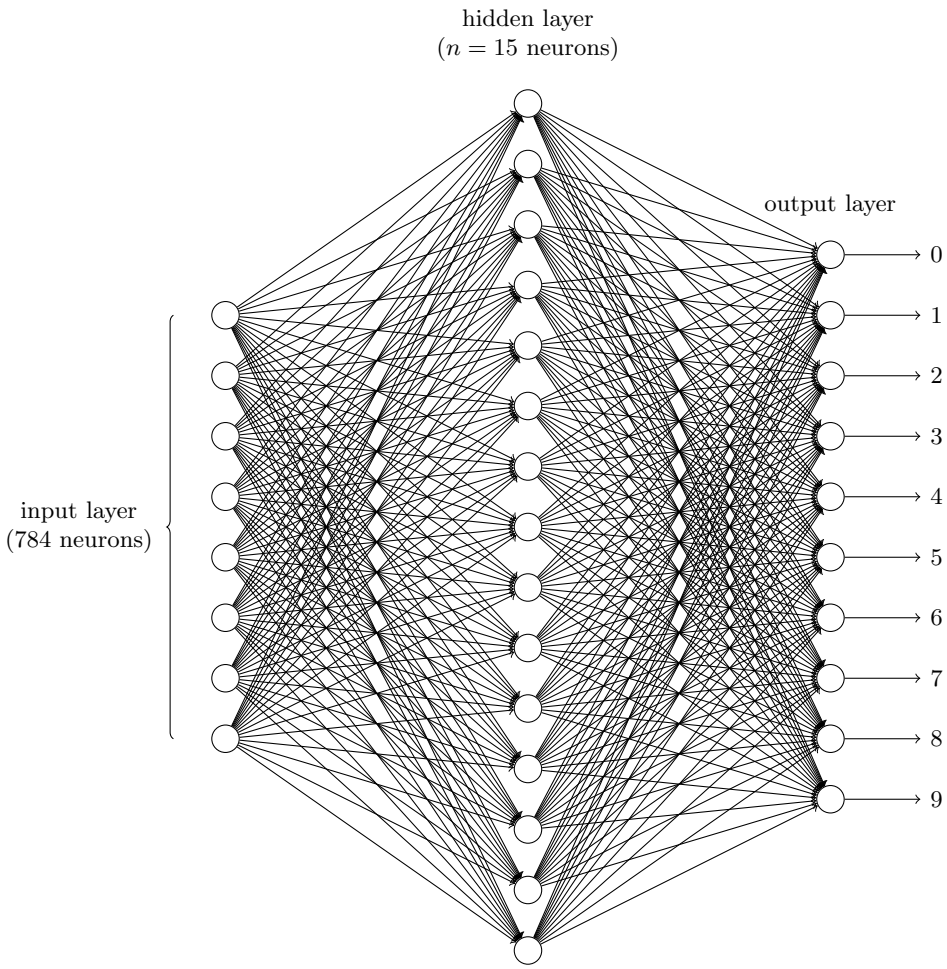
分成六个独立的图像，

我们人类很容易解决这个分割问题，但对于计算机程序来说，要正确分割图像是很有挑战性的。一旦图像被分割，程序就需要对每个单独的数字进行分类，例如，我们希望我们的程序能够识别出上面的第一个数字，

是 5。

我们将专注于编写程序来解决第二个问题，也就是对单个数字进行分类。我们这样做是因为事实证明，一旦你有了对单个数字进行分类的好方法，分割问题就不难解决。解决分割问题的方法有很多。一种方法是试验多种不同的图像分割方式，使用数字分类器对每个分割片段进行评分，如果数字分类器对其在所有片段的分类置信度比较高，则这种分割方式得到高分；如果分类器在一个或多个片段中出现问题，则得到低分。这种方法的思想是，如果分类器在某个地方遇到困难，那么它可能是因为图像分割出错导致的，这个想法和它的变化形式可以用来很好地解决分割问题。因此，与其担心分割问题，我们不如专注于开发一个神经网络，它可以解决更有趣和更困难的问题，即单个手写数字的识别。

为了识别单个数字，我们将使用一个三层的神经网络：



网络的输入层包含编码输入像素值的神经元，正如下一节会讨论的，我们的网络训练数据将包括许多 28×28 像素的手写数字扫描图像，因此输入层包含 $784 = 28 \times 28$ 个神经元。为了简单起见，我省略了大部分的输入的神经元。输入的像素是灰度的，值为 0.0 表示白色，值为 1.0 表示黑色，中间数值表示逐渐暗淡的灰色。

网络的第二层是一个隐藏层。我们用 n 表示这个隐藏层中的神经元数量，我们将对 n 的不同值进行实验. 图中的例子用一个小的隐藏层来说明，只包含 $n = 15$ 个神经元。

网络的输出层包含 10 个神经元。如果第一个神经元激活，即有一个输出 ≈ 1，那么，这将表明网络认为该数字是 0，如果第二个神经元激活，那么这将表明网络认为该数字是一个 1，以此类推。更准确地说，我们把输出神经元的编号从 0 到 9，然后找出哪个神经元的激活值最高。比如，如果 6 号神经元激活，那么我们的网络将猜测输入的数字是一个 6。以此类推，其他输出神经元也是如此。

你可能想知道为什么我们要用 10 个输出神经元。毕竟，哪个数字（ $0, 1, 2, \ldots, 9$ ）对应于输入图像才是网络该告诉我们的。一个看似自然的方法是只用 4 个输出神经元，将每个神经元视为二进制值，结果取决于该神经元的输出是否接近于 0 还是接近 1。四个神经元足以对答案进行编码，因为 $2^4 = 16$ 比输入数字的 10 个可能值要多。为什么我们的网络要使用 10 个神经元呢？这不是效率低下吗？最终的理由是经验性的：我们可以尝试两种网络设计，结果发现，对于这个特定的问题，具有 10 个输出神经元的网络在识别数字方面比 4 个神经元的网络要好。但这让我们想知道为什么使用 10 个输出神经元效果更好，是否有一些启发式的方法可以提前告诉我们，我们应该使用 10 个输出的编码，而不是 4 个输出编码？

为了理解我们为什么要这样做，从根本原理出发，思考一下神经网络在做什么是有帮助的。首先考虑的是我们使用 10 个输出神经元的情况。让我们专注于第一个输出神经元，即试图决定数字是否为 0 的那个神经元，它通过权衡来自隐藏层神经元的依据来完成这一工作。这些隐藏的神经元在做什么？假设隐藏层的第一个神经元只是用于检测如下的图像是否存在：

它可以通过重度加权与图像重叠的输入像素，而只轻度加权其他输入来做到这一点。同理，我们假设隐藏层的第二、第三和第四个神经元以类似的方式检测是否存在以下图像：

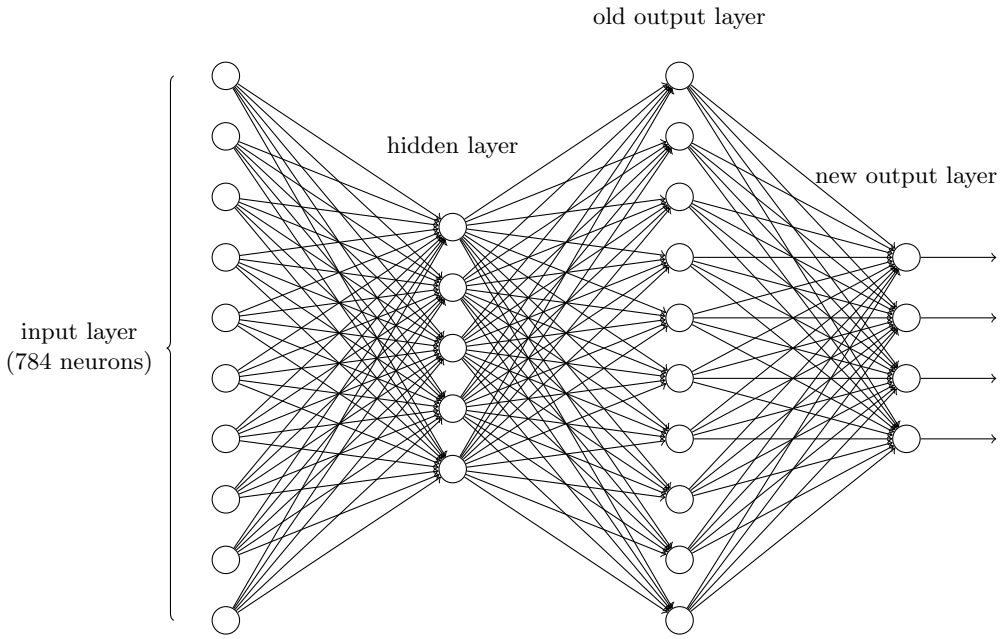
正如你可能已经猜到的，这四张图片共同构成了 0 图像，我们在前面显示的一行数字中看到了这一点。

因此，如果所有隐藏神经元中的这四个被激活，那么我们可以推断出数字是 0。当然，这并不是我们可以用来断定该图像是 0 的唯一证据，我们能通过很多其他合理的方式得到 0（例如，通过上述图像的转换，或轻微的扭曲）。但似乎可以说，至少在这种情况下，我们会得出结论，输入的是一个 0。

假设神经网络以这种方式运作，我们可以给出一个合理的解释，为什么最好有 10 个输出，而不是 4 个输出。如果我们有 4 个输出，那么第一个输出神经元将试图决定数字中最高有效位是什么。而且，没有简单的方法可以将最高有效位与上面显示的那些简单的形状联系起来。很难想象有什么好的历史原因能让数字的组成形状与（比如说）输出中最高有效位密切相关。

上面我们说的只是一个启发式的方法，没有人说三层神经网络必须以我描述的方式运作，即隐藏的神经元用于检测简单的数字组成形状。也许一个聪明的学习算法会找到一些合适的权重，让我们只使用 4 个输出神经元也能完成数字识别的工作。但是，我所描述的启发式的思维方式非常有效，可以为你设计好的神经网络架构节省大量的时间。

✍ **练习 1.3** 通过在上面的三层网络中增加一个额外的层，就可以确定一个数字的按位表示。额外的一层将前一层的输出转换为二进制表示，如下图所示。为新的输出层找到一组权重和偏置。假设前 3 层的神经元是这样的：第三层（即旧的输出层）的正确输出的激活值至少为 0.99，而错误输出的激活值小于 0.01.


old output layer
hidden layer
new output layer
input layer
(784 neurons)

1.5 使用梯度下降算法进行学习

现在我们有了一个神经网络的设计，那么它如何才能学会识别数字呢？首先，我们需要一个数据集来学习——所谓的训练数据集。我们将使用MNIST 数据集，其中包含数万张手写数字的扫描图像，以及它们的正确分类。MNIST 的名字来自于这样一个事实：它是由NIST 美国国家标准与技术研究所（NIST）收集的两个数据集的一个修改后的子集。下面是 MNIST 的一些图片：

正如你所看到的，这些数字实际上与本章开始时显示的那些数字是一样的。当然，在测试我们的网络时，我们会要求它识别不在训练集中的图像!

MNIST 的数据分为两部分。第一部分包含 60,000 张图像，作为训练数据使用。这些图像是 250 人的扫描笔迹样本，其中一半是美国人口普查局的雇员，另一半是高中生。这些图像是灰度的，大小为 28×28 像素。MNIST 数据集的第二部分是 10,000 张作为测试数据的图像。同样，这些是 28×28 的灰度图像。我们将使用测试数据来评估我们的神经网络在识别数字方面的学习程度。为了使学习程度得到很好的测试，测试数据取自与原始训练数据不同的 250 人的群体（尽管仍然是人口普查局雇员和高中学生之间的群体）。这有助于让我们相信，我们的系统可以从训练时没有看到的人的书写中识别出数字。

我们将使用符号 x 来表示训练输入，把每个训练输入 x 看作是一个 $28 \times 28 = 784$ 维的向量是很方便的。向量中的每个条目代表图像中单个像素的灰度值。我们将用 $y = y(x)$ 表示相应的期望输出，其中 y 是一个 10 维向量。例如，如果一个特定的训练图像 x 描绘了一个 6，那么 $y(x) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$ 就是网络的期望输出。请注意，这里的 T 是转置操作，将行向量变成普通（列）向量。

我们想要的是一种算法，它可以让我们找到权重和偏置，从而使网络的输出在所有训练输入 x 时都接近 $y(x)$。为了量化我们实现这一目标的程度，我们定义了一个**代价函数**[3]：

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

这里，w 表示网络中所有权重的集合，b 表示所有的偏置，n 是训练输入的总数，a 是输入 x 时网络输出的向量，求和则是在总的训练输入 x 上进行的。当然，输出 a 取决于 x、w 和 b，但为了保持符号的简单，我没有明确指出这种依赖性。符号 $\|v\|$ 只是表示向量 v 的模。我们将 C 称为**二次**代价函数；它有时也被称为**均方误差**或者 **MSE**。观察二次代价函数的形式，我们可以看到 $C(w, b)$ 是非负的，因为求和公式中的每项都是非负的。此外，代价 $C(w, b)$ 变得很小，即 $C(w, b) \approx 0$ 时，正是当 $y(x)$ 近似等于所有训练输入 x 的输出 a 时。因此，如果我们的训练算法能够找到权重和偏置，使 $C(w, b) \approx 0$，它就能工作的很好。因此，我们训练算法的目的将是使作为权重和偏置的代价函数 $C(w, b)$ 最小化，换句话说，我们要找到一组权重和偏置，使代价尽可能地小，我们将使用一种被称为**梯度下降**的算法来实现这一目标。

为什么要引入二次代价？毕竟，我们主要关心的不是被网络正确分类的图像的数量吗？为什么不尝试直接使这个数量最大化，而不是使二次代价这样的间接措施最小化？这么做是因为在神经网络中，正确分类的图像数量不是权重和偏置的平滑函数，在大多数情况下，对权重和偏置进行小的改变不会

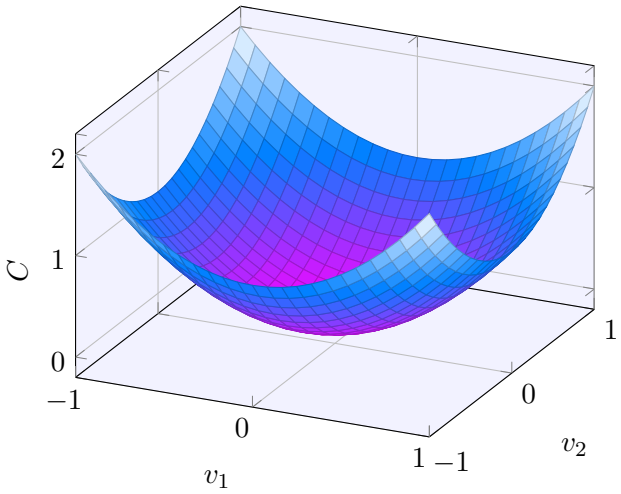
[3]有时被称为**损失**或**目标**函数。我们在本书中一直使用代价函数这个术语，但你应该注意到其他术语，因为它经常被用于研究论文和其他关于神经网络的讨论中。

导致正确分类的图像数量有任何变化。这使得我们很难弄清楚如何改变权重和偏置以获得更好的性能。如果我们使用一个平滑的代价函数，如二次代价，就很容易找出如何对权重和偏置进行小的改变，以获得性能的改善。这就是为什么我们首先专注于最小化二次代价，然后才会考察分类的准确性。

即使考虑到我们想使用一个平滑的代价函数，你可能仍然会想，为什么我们选择方程(6)中使用的二次函数。这不是一个相当**临时**的选择吗？也许如果我们选择一个不同的代价函数，我们会得到一组完全不同的最小化权重和偏置？这种顾虑是合理的，以后我们将重新审视代价函数，并做一些修改。尽管如此，方程(6)的二次代价函数对于理解神经网络学习的基本原理非常有效，所以我们现在还是坚持使用它。

回顾一下，我们训练神经网络的目标是找到最小化二次代价函数 $C(w,b)$ 的权重和偏置。这是一个很好解决的问题，但目前提出的问题有很多分散注意力的结构——将 w 和 b 解释为权重和偏置，潜伏在背景中的 σ 函数，网络结构的选择，MNIST，等等。事实证明，我们可以通过忽略这些结构中的大部分，而只是集中在最小化方面来理解它。所以现在我们要忘记所有关于代价函数的具体形式、与神经网络的联系等等，相反，我们要想象，我们只是面对一个多变量函数，并且我们要使该函数最小化。我们将开发一种叫做**梯度下降**的技术，可以用来解决这种最小化问题。问题解决后我们再回到我们想要最小化的神经网络的具体函数。

好吧，让我们假设我们试图最小化一些函数，$C(v)$。这可以是任意多变量的实值函数，$v = v_1, v_2, \ldots$。请注意，我用 v 代替了 w 和 b 的符号，以强调这可能是任意函数——我们不再特别考虑神经网络的背景。为了最小化 $C(v)$，我们可以把 C 想象成两个变量 $v1$ 和 $v2$ 的函数：



我们要做的是找到 C 达到全局最小值的地方。当然，对于上面绘制的函数，我们可以用眼睛看图并找到最小值。从这个意义上说，我所展示的函数也许**过于**简单了！通常 C 可能是一个由许多变量组成的复杂的函数，仅仅用眼睛看图形来找到最小值是不可能的。

解决这个问题的一个方法是使用微积分来尝试解析性地找到最小值。我们可以计算导数，然后尝试用它们来寻找 C 是极值的地方。运气好的话，当 C 是一个或几个变量的函数时，这可能会奏效，但当我们有更多的变量时，它就会变成一场噩梦。而对于神经网络来说，我们往往需要**大量**的变量——最大的神经网络的代价函数以极其复杂的方式依赖于数十亿的权重和偏置，使用微积分来计算最小值是行不通的。

(在确定我们将通过把 C 想象成一个只有两个变量的函数来理解神经网络之后，我已经两次提到说，“嘿，但如果它是一个不止两个变量的函数呢?”对此我很抱歉。请相信我，我说把 C 想象成两个

变量的函数是有助于我们理解的。有时这种想象的画面会出现障碍，而上面两段就是在帮助你克服这种障碍。对数学的良好思考往往涉及处理多种直觉上的想象画面，我们需要学习何时适合使用想象的画面，何时不适合使用。）

好的，所以微积分是行不通的。幸运的是，有一个漂亮的比喻暗示有一个相当好的算法。首先把我们的函数看成是一种山谷，如果你稍微眯起眼睛看一下上面的图，这应该不是太难。我们想象一个球在山谷的斜坡上滚动，我们的日常经验告诉我们，球最终会滚到山谷的底部。也许我们可以用这个想法来寻找函数的最小值？我们为一个（假想的）球随机选择一个起点，然后模拟球滚落到谷底的运动。我们可以通过计算 C 的导数（也许还有一些二阶导数）来进行这种模拟——这些导数将告诉我们关于山谷的局部“形状”所需要知道的一切，由此我们知道球应该如何滚动。

根据我刚才所写的，你可能会认为我们会试图写出球的牛顿运动方程，考虑摩擦和重力的影响，等等。事实上，我们不会把滚球的比喻看得那么认真——我们是在设计一个最小化 C 的算法，而不是在开发一个精确的物理定律仿真！。球的视角是为了激发我们的想象力，而不是限制我们的思维，因此，与其纠结于物理学的所有混乱细节，不如简单地问问我们自己：如果我们被宣布扮演一天上帝，并且可以制定我们自己的物理定律，告诉球应该如何滚动，那么我们可以选择什么样的运动定律，使球总是滚动到谷底？

为了更加精确描述这个问题，让我们想一想，当我们把球在 $v1$ 方向上移动一小段 $\Delta v1$，在 $v2$ 方向上移动一小段 $\Delta v2$ 时会发生什么。微积分告诉我们，C 的变化如下：

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \tag{7}$$

我们要寻找一种选择 Δv_1 和 Δv_2 的方法使得 ΔC 为负；也就是说，我们将选择它们，使球滚落到山谷中。为了弄清楚如何做出这样的选择，我们可以将 Δv 定义为 v 变化的向量，$\Delta v \equiv (\Delta v_1, \Delta v_2)^T$，其中 T 是转置操作，将行向量变成列向量。我们还定义 C 的梯度为偏导数的向量，$\left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$。我们用 ∇C 来表示梯度向量，即：

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

稍后我们将用 Δv 和梯度 ∇C 来重写变化 ΔC。在这之前，我想澄清一些有时会让人们对梯度感到困惑的事情。当第一次见到 ∇C 符号时，人们有时会想他们应该如何理解 ∇ 符号。∇ 究竟是什么意思？事实上，我们完全可以把 ∇C 看成是一个单一的数学记号——上面定义的向量——碰巧用两个符号来写。这样来看，∇ 只是一个符号，犹如风中挥舞的旗帜，告诉你" 嘿，∇C 是一个梯度向量"。有一些更高级的观点，∇ 可以被看作是一个独立的数学对象（例如，作为一个微分算子），但是我们不需要这样的观点。

有了这些定义，ΔC 的表达式(7) 可以被重写为：

$$\Delta C \approx \nabla C \cdot \Delta v \tag{9}$$

这个方程有助于解释为什么 ∇C 被称为梯度向量：∇C 将 v 的变化与 C 的变化联系起来，正如我们期望的用梯度来表示。但这个方程真正令人兴奋的是它让我们看到如何选择 Δv 来使 ΔC 为负值。特别是，假设我们选择：

$$\Delta v = -\eta \nabla C \tag{10}$$

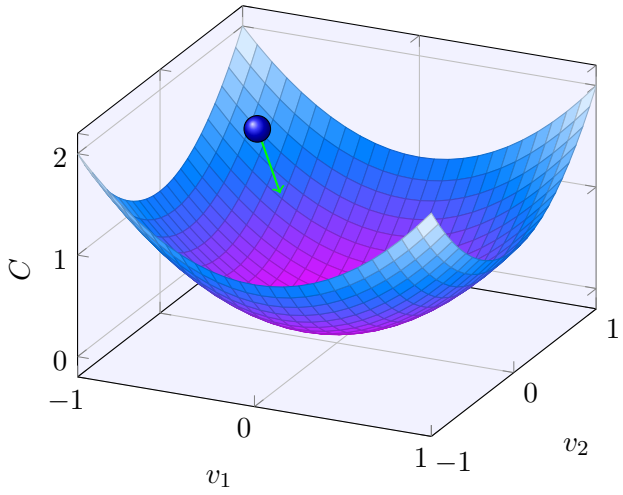
其中 η 是一个小的、正的参数（称为**学习速率**）。那么公式(9)告诉我们，$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$。因为 $\|\nabla C\|^2 \geq 0$，这就保证了 $\Delta C \leq 0$，也就是说，如果我们根据(10)中的规则改变 v，C

将总是减少，而不会增加。(当然，在方程(9)中的近似值的限制范围内)。这正是我们想要的特性。因此，我们将用方程(10)来定义我们梯度下降算法中球的" 运动规律"。也就是说，我们将使用方程(10)来计算 Δv 的值，然后将球的位置 v 按这个量移动：

$$v \to v' = v - \eta \nabla C \tag{11}$$

然后我们将再次使用这个更新规则，进行另一次移动。如果我们一直这样做，一次又一次，我们将不断减少 C，直到——我们希望的——我们达到一个全局最小值。

总结一下，梯度下降算法的工作方式是反复计算梯度 ∇C，然后向**相反**的方向移动，沿着山谷 “滚落”。我们可以这样形象地描述它：



请注意，有了这个规则，梯度下降并不能重现真实的物理运动。在现实生活中，球是有动量的，这个动量可能会让它滚过斜坡，甚至（瞬间）滚上坡。只有在摩擦力的作用下，球才会被保证滚到山谷中。相比之下，我们选择 Δv 的规则只是说" 往下走，就现在"。但这仍然是一个很好的寻找最小值的规则!

为了使梯度下降正常工作，我们需要选择足够小的学习率学习速率 η，使方程(9)能得到很好的近似。如果我们不这样做，我们可能会得到 $\Delta C > 0$ 的结果，这显然是不可取的。同时，我们也不希望 η 太小，因为这将使 Δv 的变化变得很小，从而使梯度下降算法工作得很慢。在实际实现中，η 经常被改变，以便使方程(9)能保持很好的近似度，但算法又不会太慢。我们稍后会看到这一点是如何实现的。

我已经解释了当 C 是一个只有两个变量的函数时的梯度下降。但是，事实上，即使 C 是多变量的函数，一切也同样有效。假设 C 是 m 个变量 $v_1, \ldots, v_m$ 的多元函数，那么，由一个小的变化 $\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$ 所产生的 C 的变化 ΔC 是：

$$\Delta C \approx \nabla C \cdot \Delta v \tag{12}$$

这里的梯度 ∇C 是向量

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m} \right)^T \tag{13}$$

类似两个变量的情况，我们可以选取

$$\Delta v = -\eta \nabla C \tag{14}$$

而我们保证 ΔC 的（近似）表达式(12)将是负的。这为我们提供了一种方法，即使 C 是多变量函

数，也可以通过重复应用更新规则，按照梯度指示达到最小值。

$$v \to v' = v - \eta \nabla C \tag{15}$$

你可以把这个更新规则看成是对梯度下降算法的**定义**。这给我们提供了一种方式去通过重复改变 v 来找到函数 C 的最小值。这条规则并不总是有效的——有几件事情可能出错，使梯度下降无法找到 C 的全局最小值，这一点我们将在以后的章节中再来探讨。但在实践中，梯度下降通常工作得非常好，在神经网络中，我们会发现它是一种使代价函数最小化的强大方式，因此有助于网络的学习。

事实上，甚至在某种意义上，梯度下降是寻找最小值的最佳策略。假设我们试图在位置上做一个动作 Δv，以便尽可能地减少 C。这就相当于最小化 $\Delta C \approx \nabla C \cdot \Delta v$。我们将限制移动的步长大小，使 $\|\Delta v\| = \epsilon$ 为某个小的固定的 $\epsilon > 0$。换句话说，我们想要的移动是一个固定大小的小步，并且我们试图找到尽可能减少 C 的移动方向。可以证明，使 $\nabla C \cdot \Delta v$ 最小化的 Δv 的选择是 $\Delta v = -\eta \nabla C$，其中 $\eta = \epsilon / \|\nabla C\|$ 由大小约束条件 $\|\Delta v\| = \epsilon$ 决定。所以梯度下降可以被看作是在最快减少 C 的方向上迈出的一步。

✍ **练习 1.4** 证明上一段落的推断。提示：可以利用柯西-施瓦茨不等式。

✍ **练习 1.5** 我给出了当 C 是二元及多元函数时梯度下降的解释。当 C 只是单变量函数时，会发生什么？你能对梯度下降在一维情况下的作用提供一个几何学的解释吗？

人们已经研究了梯度下降的许多变化，包括一些更接近真实模拟球体物理运动的变化形式。这些模仿球体的变化有一些优点，但也有一个主要的缺点：事实证明有必要计算 C 的二次偏导，而这可能是相当昂贵的。为了了解为什么代价高，假设我们想计算所有的二次偏导 $\partial^2 C / \partial v_j \partial v_k$。如果有一百万个这样的 v_j 变量，那么我们就需要计算大约一万亿个（即一百万个平方）二次偏导数[4]！这将耗费大量的计算代价，存在有一些技巧可以避免这种问题，寻找梯度下降的替代品是一个活跃的研究领域。但在本书中，我们将使用梯度下降法（包括变化形式）作为我们在神经网络中学习的主要方法。

我们如何在神经网络中应用梯度下降法进行学习？我们的想法是使用梯度下降来找到权重 w_k 和偏置 b_l，使方程(6)中的代价最小化。为了了解这一点，让我们重述一下梯度下降的更新规则，用权重和偏置来代替变量 v_j。换句话说，我们的"位置"现在有分量 w_k 和 b_l，而梯度向量 ∇C 有相应的分量 $\partial C / \partial w_k$ 和 $\partial C / \partial b_l$。用分量来写出梯度下降的更新规则，我们有：

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{16}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l} \tag{17}$$

通过反复应用这个更新规则，我们可以让球"滚下山"，并有希望找到代价函数的最小值。换句话说，这是一个可以用来在神经网络中学习的规则。

在应用梯度下降法则时，有一些挑战，我们将在后面的章节中深入研究这些问题。但现在我只想提一个问题。为了理解这个问题，让我们回顾一下方程(6)中的二次代价。注意这个代价函数的形式是 $C = \frac{1}{n} \sum_x C_x$，也就是说，它是每个训练样本的代价 $C_x \equiv \frac{\|y(x)-a\|^2}{2}$ 的平均值。在实践中，为了计算梯度 ∇C，我们需要对每个训练输入 x 分别计算梯度 ∇C_x，然后对其进行平均，$\nabla C = \frac{1}{n} \sum_x \nabla C_x$。不幸的是，当训练输入的数量非常大时，这可能需要很长的时间，因此学习变得很慢。

一个叫做**随机梯度下降**的想法可以用来加快学习速度。这个想法是通过计算随机选择的训练输入的一个小样本的 ∇C_x 来估计梯度 ∇C。通过对这个小样本的平均化，我们可以很快得到对真实梯度 ∇C 的良好估计，这有助于加快梯度下降的速度，从而加快学习速度。

[4]实际上，更接近万亿次的一半，因为 $\partial^2 C / \partial v_j \partial v_k = \partial^2 C / \partial v_k \partial v_j$。同样，你知道怎么做。

更加精确的说，随机梯度下降的工作方式是随机挑选出少量的 m 个随机训练输入。我们将这些随机训练输入标记为 $X_1, X_2, \ldots, X_m$，并将它们称为一个**小批量数据**。只要样本量 m 足够大，我们期望 ∇C_{X_j} 的平均值将大致等于所有 ∇C_x 的平均值，也就是说：

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

这里的第二个求和符号是在整个训练数据上进行的，对换一下我们得到：

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j} \tag{19}$$

证实了我们可以通过仅仅计算随机选取的小批量数据的梯度来估算整体的梯度。

为了将这一点明确地与神经网络的学习联系起来，假设 w_k 和 b_l 表示我们神经网络中的权重和偏置。那么，随机梯度下降法的工作原理是随机选择一批小批量数据的训练输入，然后用这些输入进行训练。

$$w_k \to w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{20}$$

$$b_l \to b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l} \tag{21}$$

其中，两个求和符号为当前小批量数据中所有训练实例 X_j 的总和。然后我们再随机选取一个小批量数据进行训练。以此类推，直到我们用尽了训练输入，这被称为完成了一个训练**迭代期**。然后我们用一个新的迭代期重新开始。

顺便说一下，值得注意的是，关于改变代价函数的比例参数，和用于计算权重和偏置的小批量数据更新的约定是不同的。在方程(6)中，我们将整个代价函数的比例定为 $\frac{1}{n}$。人们有时会省略 $\frac{1}{n}$，直接对单个训练实例的代价进行求和，而不是求平均数。当训练样本的总数事先不知道时，这一点特别有用。例如，这种情况可能发生在训练数据是实时产生的情况下。而且，以类似的方式，小批量数据更新规则(20)和(21)有时也会省略和前面的 $\frac{1}{m}$ 项。从概念上讲，这没有什么区别，因为这相当于重新调整了学习速率 η 的大小。但在对不同的工作进行详细对比时，这是值得注意的。

我们可以把随机梯度下降看成是民意调查：对一个小的小批量数据进行采样比对整个批次应用梯度下降要容易得多，就像进行一次民意调查比进行一次完整的选举要容易。例如，如果我们有一个 n=60,000 的训练集，如 MNIST，并选择一个（比如）$m = 10$ 的小批量数据，这意味着我们在估计梯度时将得到 6000 倍的加速。当然，这个估计不会是完美的——会有统计学上的波动——但它不需要完美：我们真正关心的是向一个有助于减少 C 的大方向移动，这意味着我们不需要精确计算梯度。在实践中，随机梯度下降是一种常用的、强大的神经网络学习技术，它是我们将在本书中开发的大多数学习技术的基础。

✍ **练习 1.6** 梯度下降算法一个极端的版本是把小批量数据的大小设为 1。也就是说，给定一个训练输入样本 x，我们根据规则 $w_k \to w_k' = w_k - \eta \partial C_x / \partial w_k$ 和 $b_l \to b_l' = b_l - \eta \partial C_x / \partial b_l$ 更新我们的权重和偏置。然后我们选择另一个样本输入，并再次更新权重和偏置。以此类推，反复进行。这个过程被称为**在线**、**online**、**on-line**、或者**增量**学习。在 online 学习中，神经网络一次只从一个训练输入样本中学习（就像人类一样）。与小批量数据为 20 的随机梯度下降法相比，说出在线学习的一个优点和一个缺点。

在这一节的最后，让我来讨论一个有时会让刚接触梯度下降的人感到困惑的问题。在神经网络中，代价 C 当然是多变量的函数--所有的权重和偏置——因此在某种意义上定义了一个非常高维空间的平面。有些人可能担心的想："嘿，我必须能够将所有这些额外的维度可视化"。他们可能开始担心："我

不能在四维空间中思考，更不用说五维（或五百万维）了"。他们是否缺少一些特殊的能力，一些" 真正的" 超级数学家所拥有的能力？当然，答案是否定的。即使是大多数专业数学家也不能很好地想象四维空间。他们使用的诀窍是扩展出其它的方法来描绘发生了什么事。这正是我们在上面所做的：我们用代数（而不是图像）来表示 ΔC，以找出如何移动以减少 C。善于在高维度上思考的人有包含有许多不同的技术的知识库；我们的代数技巧只是一个例子。这些技巧可能没有我们在三维可视化时的简单性，但是一旦你建立了这样的技巧库，你就可以很好地进行高维思考。如果你有兴趣，那么你可能会喜欢这篇关于专业数学家用来进行高维思考的一些技术的讨论。虽然所讨论的一些技术相当复杂，但许多最好的内容都是直观和容易掌握的，任何人都可以掌握。

1.6 实现我们的网络来分类数字

好吧，让我们写一个程序，使用随机梯度下降法和 MNIST 训练数据来学习如何识别手写数字。我们将用一个简短的 Python（2.7）程序来完成这个任务，只有 74 行的代码，我们首先需要的是获得 MNIST 数据。如果你是git用户，那么你可以通过克隆本书的代码库来获得这些数据。

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

如果你不使用 git，也可以从 这里下载数据和代码。

顺便说一下，当我之前描述 MNIST 数据的时候，我说它被分成了 60,000 张训练图像和 10,000 张测试图像。那是 MNIST 的官方描述。实际上，我们将以一种不同的方式来分割数据。我们将保留测试图像，但将 60,000 张 MNIST 训练集分成两部分：一组是 50,000 张图像，我们将用它来训练我们的神经网络，另一组是 10,000 张图像的验证集。我们不会在本章中使用验证数据，但在本书的后面，我们会发现它对于弄清如何设置神经网络的某些**超参数**很有帮助——比如学习率等，这些参数并不是由我们的学习算法直接选择的。虽然验证数据并不是原始 MNIST 规范的一部分，但很多人都以这种方式使用 MNIST，而且验证数据的使用在神经网络中很常见。从现在开始，当我提到“MNIST 训练数据”时，我指的是我们的 50,000 张图片数据集，而不是原来的 60,000 张图片数据集。[5]）

除了 MNIST 数据之外，我们还需要一个叫做Numpy的 Python 库，用于进行快速线性代数。如果你还没有安装 Numpy，你可以在这里下载得到它。

在列出一个完整的代码清单之前，让我解释一下神经网络代码的核心特性。

核心片段是一个Network 类，我们用来表示一个神经网络。这是我们用来初始化一个 Network对象的代码：

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

在这段代码中，列表的 sizes 包含了各层中神经元的数量。例如，如果我们想创建一个第一层有 2 个神经元、第二层有 3 个神经元、最后一层有 1 个神经元的 Network 对象，我们可以用代码来完成：

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

Network 对象中的偏置和权重都是随机初始化的，使用 Numpy 的 np.random.randn 函数来生成平均值为 0、标准差为 1 的高斯分布。这种随机初始化给我们的随机梯度下降算法提供了一个起点。在后面的章节中，我们会找到更好的初始化权重和偏置的方法，但现在这样就可以了。请注意，Network 初

[5]如前所述，MNIST 数据集是基于美国国家标准与技术研究所（NIST）收集的两个数据集。为了构建 MNIST，Yann LeCun、Corinna Cortes 和 Christopher J. C. Burges 对 NIST 的数据集进行了删减，并将其转换成更方便的格式。更多细节见此链接。我的代码仓库中的数据集的形式使得在 Python 中加载和操作 MNIST 数据变得更加容易。我从蒙特利尔大学的 LISA 机器学习实验室获得了这种特殊形式的数据（链接）

始化代码假定第一层神经元是输入层，并且省略了为这些神经元设置任何偏置，因为偏置只用于计算后面各层的输出。

还要注意的是，偏置和权重是以 Numpy 矩阵列表的形式存储的。因此，例如 net.weights[1] 是一个 Numpy 矩阵，存储连接第二和第三层神经元的权重。(它不是第一层和第二层，因为 Python 的列表索引从 0 开始)。由于 net.weights[1] 比较冗长，让我们直接表示这个矩阵 w。它是一个矩阵，使得 w_{jk} 是第二层的第 k^{th} 个神经元和第三层的第 j^{th} 个神经元之间的连接权重。这种 j 和 k 索引的顺序可能看起来很奇怪——当然，把 j 和 k 指数互换一下会更有意义？使用这种顺序的最大好处是，它意味着第三层神经元的激活向量是:

$$a' = \sigma(wa + b) \tag{22}$$

这个方程有很多内容，所以我们来逐一解读。a 是第二层神经元的激活向量，为了得到 a，我们用 a 乘以权重矩阵 w，并加上偏置的向量 b。然后，我们将函数 以元素方式应用于向量 $wa + b$ 中的每个条目。(称为将函数 **向量化**。) 很容易验证方程(22)给出的结果与我们先前采用的规则，即方程(4)，计算一个Sigmoid 神经元的输出是一样的。

✍ **练习 1.7** 将方程(22)以分量形式写出来，并验证它所得到的结果与计算Sigmoid 神经元输出的规则(4)相同。

有了这些，写代码计算一个 Network 实例的输出是很容易的。我们首先定义 sigmoid 函数：

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

请注意，当输入的 z 是一个向量或 Numpy 数组时，Numpy 会自动将函数 sigmoid 以元素方式应用，也就是以向量的形式。

然后，我们在 Network 类中添加一个 feedforward 方法，给定网络的输入 a，返回相应的输出。[6] 该方法所做的就是对每一层应用公式(22)：

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

当然，我们希望我们的 Network 对象能够做的主要事情是学习。为此，我们将给它们一个实现随机梯度下降的 SGD 方法，下面是代码。它在一些地方有点神秘，但我将在下面的代码列表后面逐个进行分析。

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
```

[6]假设输入 a 是一个 (n,1) 的 Numpy ndarray，而不是一个 (n,) 向量。这里，n 是网络的输入数。如果你试图使用一个 (n,) 向量作为输入，你会得到奇怪的结果。尽管使用 (n,) 向量似乎是更自然的选择，但使用 (n,1)ndarray 使得修改代码以一次前馈多个输入变得特别容易，而这有时是很方便的。

```
        gradient descent.  The "training_data" is a list of tuples
        "(x, y)" representing the training inputs and the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  If "test_data" is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out.  This is useful for
        tracking progress, but slows things down substantially."""
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)
```

training_data 是一个代表训练输入和相应期望输出的 (x, y) 元组的列表。epochs 和 mini_batch_size 这两个变量正如你所预料的——训练的迭代期数量，以及采样时使用的小批量数据的大小。eta 是学习速率，η。如果提供了可选的参数 test_data，那么程序将在每次训练后评估网络，并打印出部分进展。这对跟踪进度很有用，但会大大降低速度。

该代码的工作原理如下：在每个迭代期中，它首先随机打乱训练数据，然后将其划分为适当大小的小批量数据。这是一种从训练数据中随机抽样的简单方法。然后，对于每一个mini_batch，我们应用一次梯度下降。这是通过代码 self.update_mini_batch(mini_batch, eta) 完成的，它只使用 mini_batch 中的训练数据，根据梯度下降的单次迭代来更新网络权重和偏置。下面是 update_mini_batch 方法的代码：

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

```
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]
```

大部分的工作是由这行完成：

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

这调用了称为**反向传播**的算法，这是一种快速计算代价函数梯度的方法。所以 update_mini_batch 的工作方式很简单，就是为 mini_batch 中的每个训练样本计算这些梯度，然后适当地更新 self.weights 和 self.biases。

我现在不打算展示 self.backprop 的代码，我们将在下一章研究**反向传播**的工作原理及代码。现在，只需假设它的行为与声称的一样，返回与训练样本 x 相关代价的适当梯度值。

让我们看看完整的程序，包括我上面省略的文档注释。除了 self.backprop 之外，这个程序是不言自明的——所有繁重的工作都在 self.SGD 和 self.update_mini_batch 中完成，这些我们已经讨论过了。self.backprop 方法使用了一些额外的函数来帮助计算梯度，即 sigmoid_prime（计算 σ 函数的导数）和 self.cost_derivative（我在此不作描述）。你可以通过查看代码和文档注释来了解这些的要点（和细节），我们将在下一章中详细介绍它们。请注意，虽然该程序看起来很冗长，但大部分是文档注释，目的是使代码易于理解。事实上，该程序只包含了 74 行非空格、非注释的代码。所有的代码都可以在 GitHub 上这里找到。

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

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

#### Libraries
# Standard library
import random

# Third-party libraries
import numpy as np
```

```
class Network(object):

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

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

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

```
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)

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

    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
```

```
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

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

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

#### Miscellaneous functions
```

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

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

该程序对手写数字的识别效果如何？好吧，让我们从加载 MNIST 数据开始。我将使用一个小的辅助程序 mnist_loader.py 来完成这个工作，我们在 Python shell 中执行以下命令。

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

当然，这也可以在一个单独的 Python 程序中完成，但如果你正在照着本书做，在 Python shell 中完成可能是最容易的。

在加载 MNIST 数据后，我们将创建一个有 30 个隐藏神经元的 Network。我们在导入上面列出的名为 network 的 Python 程序后可以这样做：

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

最后，我们将使用随机梯度下降法从 MNIST training_data 中学习 30 次迭代期，小批量数据大小为 10，学习速率 为 $\eta = 3.0$，

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

请注意，如果你在阅读过程中运行代码，它将需要一些时间来执行——对于一台典型的机器（截至 2015 年），它可能需要几分钟来运行。我建议你运行代码后，继续你的阅读，并定时检查一下代码的输出情况。如果你很着急，你可以通过减少迭代期的数量，减少隐藏神经元的数量，或者只使用部分训练数据来提升运行速度。

请注意，实际生产环境使用的代码会快得多：这里的 Python 脚本的目的是帮助你理解神经网络的工作原理，而不是成为高性能的代码！当然，一旦我们训练了一个网络，它确实可以在几乎任何计算平台上快速运行。例如，一旦我们为一个网络学习了一套好的权重和偏置，它就可以很容易地被移植到网络浏览器的 Javascript 中运行，或者作为移动设备上的本地应用程序。这里有一个神经网络训练运行的部分打印输出。该记录显示了神经网络在每个训练周期后正确识别的测试图像的数量。正如你所看到的，在仅仅一个迭代期后，这个数字已经达到了 10000 张中的 9129 张，而且这个数字还在继续增长。

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

更确切的说，经过训练的网络给我们提供了大约 95% 的识别率——在峰值时（“第 28迭代期”）为 95.42%！作为第一次尝试，这相当令人鼓舞。然而，我应该提醒你，如果你运行这段代码，那么你的结果不一定会和我的完全一样，因为我们使用了（不同的）随机权重和偏置来初始化我们的网络。上面我选取了三次运行中的最佳一次作为结果。

让我们重新运行上述实验，将隐藏神经元的数量改为 100。和前面的情况一样，如果你边读边运行代码，我应该警告你它需要相当长的时间来执行（在我的机器上，这个实验每个训练周期需要几十秒），所以在代码执行时继续阅读是明智的。

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

当然，这将结果提升至 96.59%。至少在这种情况下，使用更多的隐藏神经元有助于我们获得更好的结果[7]。

当然，为了获得这些准确性，我必须对训练的迭代期数、小批量数据大小和学习速率 η 做出具体的选择。正如我在上面提到的，这些被称为我们神经网络的超参数，以区别于我们的学习算法所学习到的参数（权重和偏置）。如果我们的超参数选择得不好，我们就会得到不好的结果。例如，假设我们选择的学习速率为 $\eta = 0.001$，

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

结果就不那么令人鼓舞了，

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

[7]读者的反馈表明，这个实验的结果有相当大的变化，有些训练运行的结果要差一些。使用第三章中介绍的技术将大大减少我们的网络在不同训练运行中的性能变化。

然而，你可以看到，随着时间的推移，网络的性能正在慢慢变好。这表明应该增加学习速率，比如说增加到 $\eta = 0.01$。如果我们这样做，我们会得到更好的结果，这表明我们应该再次增加学习速率。(如果做一个改变能改善情况，就尝试做更多！) 如果我们反复做几次，我们最终会得到一个类似 $\eta = 1.0$ 的学习率 (也许可以微调到 3.0)，这与我们早期的实验很接近。因此，即使我们最初对超参数的选择不理想，但我们至少得到了足够的信息来帮助我们改进对超参数的选择。

一般来说，调试神经网络可能是一个挑战。当最初选择的超参数产生的结果不比随机噪声好时，这一点尤其真实。假设我们尝试先前成功的 30 个隐藏神经元的网络结构，但将学习率改为 $\eta = 100.0$：

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

在这点上，我们实际走的太远，学习速率太高了：

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

现在想象一下，我们是第一次遇到这个问题。当然，我们从先前的实验中知道，正确的做法是降低学习速率。但是，如果我们是第一次遇到这个问题，那么输出中就不会有太多的信息来指导我们该怎么做。我们可能不仅担心学习速率，还担心我们的神经网络的其他方面。我们可能会想，是否我们初始化权重和偏置的方式使网络难以学习？或者是我们没有足够的训练数据来获得有意义的学习？也许我们还没有运行足够的迭代期？或者也许这种结构的神经网络不可能学会识别手写数字？也许学习速率太低了？或者，也许是学习速率太高了？当你第一次遇到一个问题时，你总是很难确定。

从中得到的教训是，调试神经网络并非易事，就像常规编程一样，这也是一门艺术。你需要学习这门调试的艺术，以便从神经网络中获得好的结果。更广泛地说，我们需要启发式方法来选择好的超参数和好的架构。我们将在整本书中详细讨论这些问题，包括我如何选择上述的超参数。

✍ **练习 1.8** 尝试创建一个只有两层的网络——一个输入层和一个输出层，没有隐藏层——分别有 784 个和 10 个神经元。使用随机梯度下降法训练该网络。你能达到什么样的分类精度？

早些时候，我略过了如何加载 MNIST 数据的细节。这是很简单的，这里是完整的代码。用于存储 MNIST 数据的数据结构在文档注释中有描述——都是简单的类型，Numpy ndarray 对象的元组和列表 (如果你对 ndarray 不熟悉，可以把它们想象成向量)：

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

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

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

# Third-party libraries
import numpy as np

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

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

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

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

    This is a nice data format, but for use in neural networks it's
    helpful to modify the format of the ``training_data`` a little.
    That's done in the wrapper function ``load_data_wrapper()``, see
    below.
    """
    f = gzip.open('../data/mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = cPickle.load(f)
```

```
    f.close()
    return (training_data, validation_data, test_data)

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

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

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

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

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

```
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

我在上面说，我们的项目得到了相当好的结果。那是什么意思？与什么相比算好？有一些简单的（非神经网络的）基线测试来进行比较是很有意义的，可以了解表现良好意味着什么。当然，最简单的基线是随机猜测数字。这将有百分之十的次数是正确的。我们做得比这要好得多!

那么，一个不那么差的基准线呢？让我们尝试一个极其简单的想法：我们将看一个图像的黑暗程度。例如，2 的图像通常会比 1 的图像暗很多，只是因为更多的像素被涂黑了，正如下面的例子所说明的。

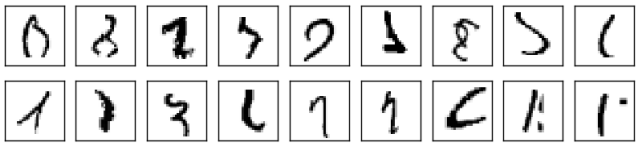
这提示我们可以使用训练数据来计算每个数字的平均暗度，$0, 1, 2, \ldots, 9$。当遇到一个新的图像时，我们计算该图像的暗度，然后猜测它与哪个数字的平均暗度最接近。这是一个简单的程序，很容易编码，所以我不会明确写出代码——如果你有兴趣，它就在GitHub 仓库里。它比随机猜测有很大的改进，在 10000 张测试图片中，有 2225 张是正确的，即 22.25% 的准确性。

找到其他能达到 20% 至 50% 的准确率的办法也不难。如果你再努力一点，你可以达到 50% 以上。但是要想获得更高的准确率，使用成熟的机器学习算法是有帮助的。让我们尝试使用最著名的算法之一，**支持向量机**或 **SVM**。如果你对 SVM 不熟悉，不用担心，我们不需要了解 SVM 的工作细节。相反，我们将使用一个叫做scikit-learn的 Python 库，它提供了一个简单的 Python 接口，包装了一个用于 SVM 的快速的，称为LIBSVM的 C 库。

如果我们使用默认设置运行 scikit-learn 的 SVM 分类器，那么在 10000 张测试图像中，它可以得到 9435 张正确的图像。(代码可在这里获得。) 这比我们根据图像的黑暗程度进行分类的幼稚方法有很大的改进。事实上，这意味着 SVM 的表现与我们的神经网络大致相同，只是稍差一些。在后面的章节中，我们将介绍新的技术来改进我们的神经网络，使它们的表现比 SVM 好得多。

然而，这并不是故事的结局。10,000 个结果中的 9,435 个是针对 scikit-learn 的 SVM 的默认设置。SVM 有许多可调整的参数，而且有可能搜索到能够改善这种默认情况下性能的参数。我不会明确地进行这种搜索，如果你想了解更多，请参考Andreas Mueller的这篇博客。Mueller 表明，通过一些优化 SVM 参数的工作，有可能将性能提高到 98.5% 以上的准确率。换句话说，一个经过良好调整的 SVM 只在 70 个数字中出现一个错误。这已经很不错了! 神经网络能做得更好吗?

事实上，它们可以。目前，设计良好的神经网络在解决 MNIST 问题上的表现优于其他所有技术，包括 SVM。目前（2013 年）的记录是将 10000 张图片中的 9979 张分类正确。这是由Li Wan，Matthew Zeiler，Sixin Zhang，Yann LeCun，和 Rob Fergus 完成的。我们将在本书后面看到他们使用的大部分技术。这个层次的性能接近于人类的水平，而且可以说更好，因为不少 MNIST 图像甚至对人类来说都很难有信心地识别，例如：



我相信你会同意，这些是很难分类的！对于 MNIST 数据集中的这些图像，神经网络能够准确地对 10,000 张测试图像中除 21 张以外的所有图像进行分类，这很了不起。通常，在编程时，我们认为解决像识别 MNIST 数字这样的复杂问题需要一个复杂的算法。但即使是刚才提到的 Wan 等人的论文中的神经网络也只涉及相当简单的算法，是我们在本章中看到的算法的变化形式。所有的复杂性都是从训练数据中自动学习的。从某种意义上说，我们的结果和那些更深奥的论文的寓意是，对于某些问题：

复杂的算法 $\leq$ 简单的学习算法 + 好的训练数据

1.7 迈向深度学习

虽然我们的神经网络有令人印象深刻的表现，但这种表现却有些神秘莫测。网络中的权重和偏置是自动发现的，而这意味着我们无法立即解释该网络是如何做到的。我们能否找到一些方法来理解网络对手写数字进行分类的原则？而且，基于这样的原则，我们能否做得更好？

为了更明确地描述这些问题，假设几十年后神经网络引发了人工智能（*AI*），我们会理解这种智能网络是如何工作的吗？因为它们是自动学习的，有我们不了解的权重和偏置，这些网络对我们来说是不透明的。在人工智能研究的早期阶段，人们希望在构建人工智能的努力过程中，也同时能够帮助我们理解人工智能背后的机制，以及人类大脑的运转方式。但结果可能是，最终我们既不能理解大脑的运转方式，也不能理解人工智能的机制。

为了解决这些问题，回想一下我在本章开始时给出的作为衡量依据的方法——人工神经元的解释。假设我们想确定一张图片是否显示了一张人脸[8]：

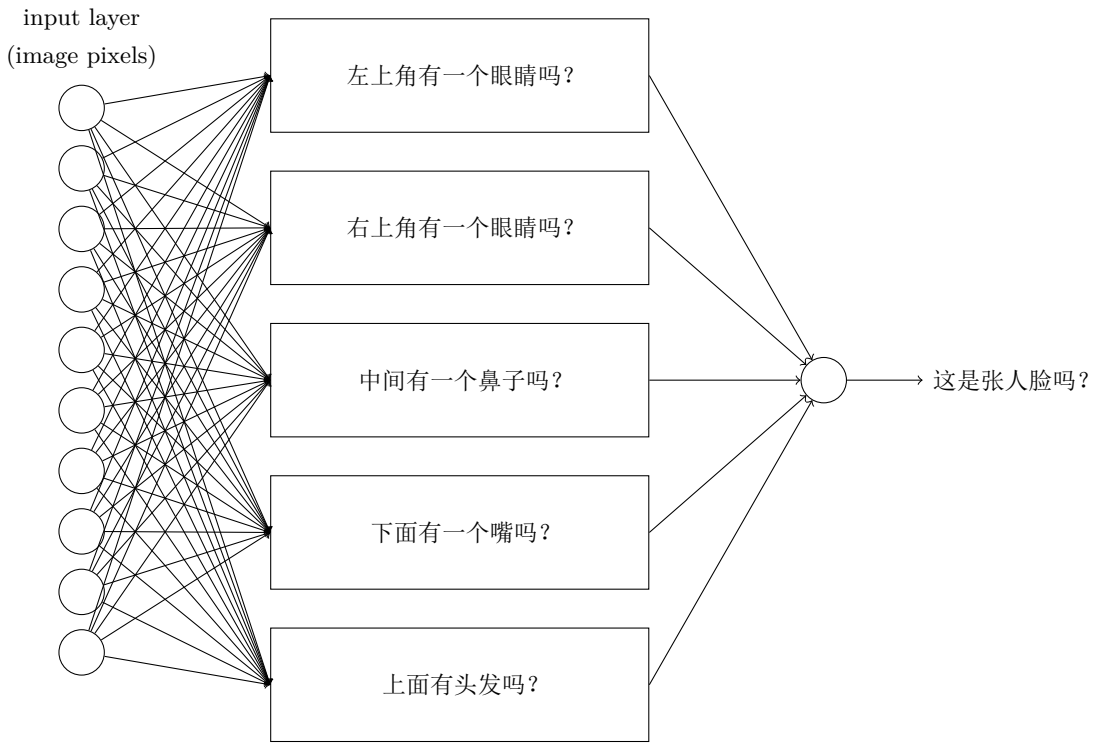
我们可以用解决手写识别的方式来解决这个问题——把图像中的像素作为神经网络的输入，网络的输出是一个单一的神经元，表示“是，这是一张脸”或“不，这不是一张脸”。

假设我们就采取这种方式，但接下来我们不使用学习算法，相反，我们将尝试手动来设计一个网络，并选择适当的权重和偏置。我们可以如何做呢？先暂且忘掉神经网络，我们可以采用的启发式方法是将问题分解为子问题：图像的左上方是否有一只眼睛？右上角有眼睛吗？中间有一个鼻子吗？底部中间有嘴吗？上面有头发吗？等等。

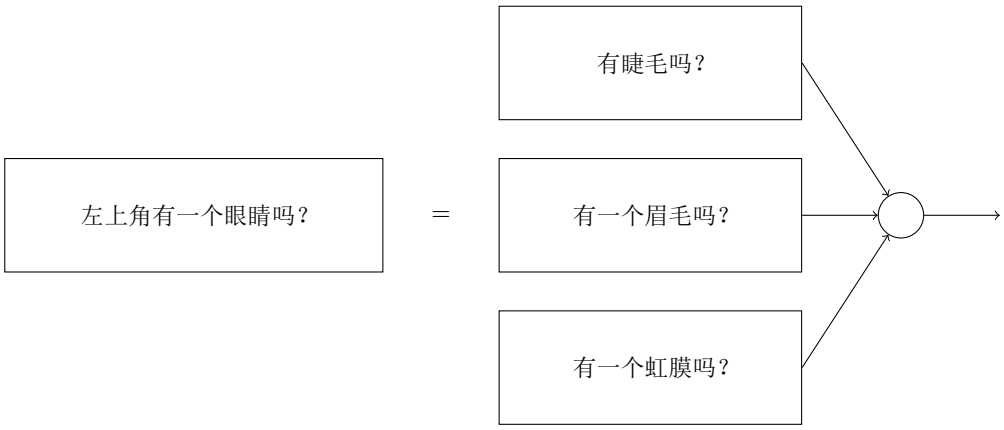
如果其中几个问题的答案是" 是”，甚至只是“可能是”，那么我们就会得出结论，该图像可能是一张脸。相反，如果大多数问题的答案是“否”，那么该图像可能不是一张脸。

当然，这只是一个粗略的启发式方法，它有很多不足之处。也许有个人是秃头，所以他们没有头发；也许我们只能看到脸的一部分，或者脸是在一个角度，所以一些面部特征被遮住了；不过，启发式方法提示我们，如果我们能用神经网络解决这些子问题，那么也许我们可以通过结合这些子问题的网络，构建一个用于面部检测的神经网络。下面是一个可能的架构，矩形表示子网络。请注意，这并不是解决人脸检测问题的现实方法，而是为了帮助我们构建起网络如何运转的直观感受，下图是这个网络的结构：

[8]照片来源：1. Ester Inbar. 2. 未知. 3. NASA, ESA, G. Illingworth, D. Magee, and P. Oesch (University of California, Santa Cruz), R. Bouwens (Leiden University), and the HUDF09 Team. 点击序号查看更多细节。



子网络可以被继续分解，这应该是合理的。假设我们正在考虑这样的问题："左上角有一只眼睛吗?"这可以被分解成如下问题："有眉毛吗?"；"有睫毛吗?"；"有虹膜吗?"；等等。当然，这些问题也应该包括位置信息——"眉毛在左上方，在虹膜上方吗?"，诸如此类的问题——但让我们保持简单，回答"左上角有一只眼睛吗?"这个问题的网络现在可以被分解成：



这些问题也可以通过多个层次进一步分解。最终，我们将使用子网络来回答那些简单到可以在单个像素层面上轻易回答的问题。例如，这些问题可能是关于图像中特定位置是否存在非常简单的形状，这样的问题可以由连接到图像中原始像素的单个神经元来回答。

最终的结果是一个网络，它将一个非常复杂的问题——这幅图像是否显示了一张脸——分解为非常简单的问题，可以在单个像素的层面上进行回答。它通过一系列的多层结构来实现这一目标，前面的层回答关于输入图像的非常简单和具体的问题，后面的层则建立了一个更加复杂和抽象的概念的层次结构，具有这种多层结构的网络——两个或多个隐藏层——被称为**深度神经网络**。

当然，我还没有说如何将网络递归分解成子网络，手工设计网络中的权重和偏置肯定是不现实的。相反，我们希望使用学习算法，使网络能够自动从训练数据中学习权重和偏置——也就是概念的层次

结构。20 世纪 80 年代和 90 年代的研究人员尝试使用随机梯度下降法和反向传播法来训练深度网络。不幸的是，除了一些特殊的架构，他们并没有取得很好的效果。网络虽然会学习，但学习速度非常缓慢，无法在实践中发挥作用。

自 2006 年以来，人们开发了一套技术使深度神经网络也能够学习。这些深度学习技术仍然基于随机梯度下降和反向传播，但也引入了新的理念。这些技术使得更深（和更大）的网络可以被训练——现在训练有 5 到 10 个隐藏层的网络都是很常见的。而且事实证明，这些网络在许多问题上的表现远远好于只有一个隐藏层的浅层神经网络。当然，其原因在于深层网络有能力建立起复杂的概念层次。这有点像传统的编程语言使用模块化设计和抽象的思想来创建复杂的计算机程序。将深层网络与浅层网络相比较，有点像将有能力进行函数调用的编程语言与没有能力进行这种调用的精简语言相比较。抽象在神经网络中采取的形式与在传统编程中不同，但它同样重要。

中英对照术语表