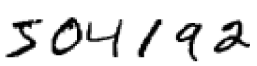
人类视觉系统是大自然的一大奇迹。 考虑下面的手写数字序列：



大部分人能够毫不费力的识别出这些数字是 504192。这种简单性只是一个幻觉。在我们大脑各半球，有一个主要的视觉皮层，即V1，它包含1.4亿个神经元以及数以百亿的神经元连接。而且人类不只是有V1，还有一系列的视觉皮层——V2,V3,V4和V5，它们能够执行更加复杂的图像处理。我们可以将大脑想象成一台超级计算机，在几亿年的进化中不断改进，最终非常适合理解这个视觉世界。要识别手写数字不是一件非常容易的事。然而，我们人类却能非常惊人的通过我们的眼睛理解所看到的一切，但是几乎所有的工作都是在不知不觉中完成的，以至于我们不会赞叹我们的视觉系统解决的问题有多么艰难。

视觉模式识别困难在于如何让计算机程序识别上面数字变得显而易见。看上去非常简单的事操作起来却变得非常困难。我们识别这个形状的简单直觉是——“一个9头上有一个圆圈，右下角有一笔竖线”——但是对于识别算法却不是那么简单。当试图让这些规则更加精确，你将迅速迷失在大量的例外、警告和特殊案例，而且似乎看不到解决的希望。

神经网络用不同的方法来处理这个问题。它的思想就是利用大量的手写数字（训练样本），

然后开发出一套从训练样本中进行学习的系统。换句话说，神经网络使用样本来自动推理出识别手写数字的规则。此外，通过增加训练样本规模，神经网络能学到手写体的更多规则从而提升它的识别精度。因此在我们像上面一样只有100张训练数字同时，有可能我们能通过成千上万更多的训练样本来构建更好的手写识别算法。

本章我们将编写一段计算机程序来实现一个能识别手写数字的神经网络。这个程序大概有74行，而且没有使用其他特别的神经网络库，但是这段程序能够具有96%的数字识别精度，而且没有人工干预。此外，在后面的章节中我们将改进算法达到99%以上的精度。实际上，最好的商业神经网络已经很好的应用在银行支票处理以及邮局识别地址等。

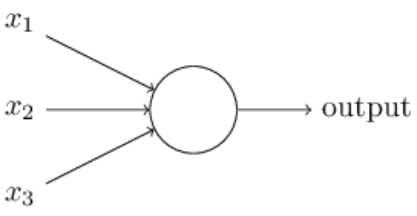
我们专注在手写体识别是因为它是一个很好的学习神经网络的原型问题。做为一个原型，它刚好合适：识别手写数字是一个挑战，不是那么容易，而它也不需要一个极其复杂的方案或者巨大的计算能力。此外，它也是开发更多高级技术的好方法，比如深度学习。因此整本书我们将不断重复回到手写识别这个问题。这本书的后面，我们将讨论这些算法思想如何应用到计算机视觉的其他问题，还包括语言识别、自然语言处理和其他领域。

当然，如果本章只是写一个识别手写数字的计算机程序，那么将非常短小！在编写过程中，我们将介绍很多神经网络的关键思想，包括人工神经网络的两大类别（感知器和sigmoid神经元）以及神经网络标准学习算法，即随机梯度下降。整个过程我们将关注在阐述*为什么*这样处理是有效的，从而让你构建起神经网络直觉。这比只陈述基础的机制更加冗长，但这对更深入理解所学到的内容是值得的。在这些收获上，本章结束你将明白深度学习是什么，且它为什么这么重要。

感知器(Perceptrons)

什么是神经网络？在开始之前，我将介绍一种人工的神经元，即*感知器(perceptron)*。感知器是由Frank Rosenblatt 在上世纪50到60年代发明的，灵感来源于 Warren McCulloch 和 Walter Pitts 的早期工作。今天人工神经网络使用更加通用的其他模型，本书以及许多现代的神经网络工作中，主要的神经网络模型是*sigmoid*神经元。 我们将很快讨论sigmoid神经元，但是为了更好了解它的原理，我们首先要理解感知器。

那么感知器如何工作呢？一个感知器获取几个二进制输入$x_1, x_2, \ldots$，并且产生一个二进制数出：



在这个例子中，感知器具有三个输入x_1, x_2, x_3。通常它会具有更多或更少的输入。Rosenblatt 提出了一个简单规则来计算最后数出。他引入了*权重(weights)* $w_1, w_2, \ldots$，这些实数表示各个输入对输出的重要性。这个神经元输出(output) 0 或者 1 是由这些输入的加权求和 $\sum_j w_j x_j$ 是否大于或者小于某一个阈值(threshold)。不像这些权重，阈值是这个神经元的实数参数。将它们放入更加精确的代数术语中：

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

这就是一个感知器如何工作的全部内容！

这是一个基础的数学模型。你可以这么理解感知器，它是一个通过加权凭据来进行决策的设备。让我们来看这个例子，可能它不是一个真实的例子，但是非常好理解，后面我们将很快进入真实的例子。假设周末到了，你听说在你所在的城市将有一个奶酪节，你很喜欢吃奶酪，并且正决定是否要去参加这个节日，你可能会通过以下三个方面来权衡你的决定:

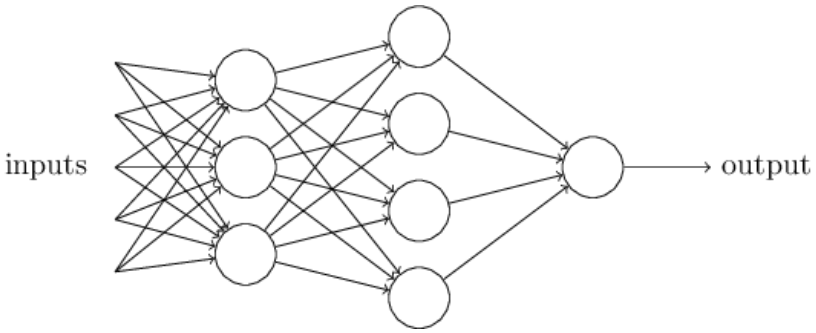
1. 天气好吗？
2. 你的男(女)朋友是否愿意陪你去？
3. 是否这个活动距离公共交通很近？（你自己没车）

我们将这三个因素用对应的二进制变量x_1, x_2和x_3表示。比如，当天气还不错时，我们有$x_1 = 1$ ，天气不好时$x_1 = 0$；相似的，如果男或女朋友愿意去，$x_2 = 1$，否则$x_2 = 0$；对于公共交通x_3同理赋值。

现在假设奶酪是你的最爱，以致于即便你的男或女朋友不感兴趣而且去那里也不太方便，你仍然非常想去参加这个节日活动。但是也许你真的讨厌坏天气，而且如果天气很糟，你也没办法去。你能使用感知器来模拟这类决策。一种决策方式是，让天气权重 $w_1 = 6$，其他条件权重分别为$w_2 = 2$，$w_3 = 2$。权重w_1值越大表示天气影响最大，比起男或女朋友加入或者交通距离的影响都大。最后，假设你选择5做为感知器阈值，按照这种选择，这个感知器就能实现这个决策模型：当天气好时候输出1，天气不好时候输出0，无论你男或女朋友是否愿意去，或者交通是否比较近。

通过更改权重和阈值，我们能得到不同的决策模型。例如，我们将阈值设为3，那么感知器会在以下条件满足时决定去参加活动：如果天气很好、*或者*男(女)朋友愿意去*并且*交通很近。换句话说，它将是决策的不同模型，阈值越低，表明你越想去参加这个节日活动。

显然，这个感知器不是人类决策的完整模型！但是这个例子说明了一个感知器如何能将各种凭据进行加权和来制定决策，而且一个复杂的感知器网络能做出非常微妙的决策：



在这个网络中，第一*列*感知器（我们称其为第一层感知器）通过加权输入凭据来做出三个非常简单的决策。那第二列感知器是什么呢？其中每一个感知器都是通过将第一列的决策结果进行加权和来做出自己的决策。通过这种方式，第二层感知器能够比第一层感知器做出更加复杂和抽象层的决策。第三层感知器能做出更加复杂的决策，以此类推，更多层感知器能够进行更加复杂的决策。

顺便说一句，当我们定义感知器时，它们都只有一个输出。但上面的网络中，这些感知器看上去有多个输出。实际上，它们也仍然只有一个输出，只不过为了更好的表明这些感知器输出被其他感知器所使用，因此采用了多个输出的箭头线表示，这比起绘制一条输出线然后分裂开更好一些。

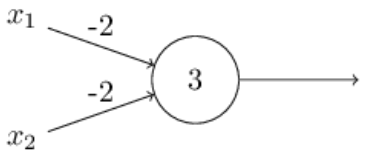
让我们简化描述感知器的方式。加权求和大于阈值的条件 $\sum_j w_j x_j > \text{threshold}$比较麻烦，我们能用两个符号变化来简化它。第一个改变是将 $\sum_j w_j x_j$ 改写成点乘方式，$w \cdot x \equiv \sum_j w_j x_j$，其中$w$ 和 x 分别是权重和输入的向量。第二个改变是将阈值放在不等式另一边，并用*偏移* $b \equiv -\text{threshold}$ 表示。通过使用偏移代替阈值，感知器规则能够重写为：

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

你可以将偏移想象成使感知器如何更容易输出 1，或者用更加生物学术语，偏移是指衡量感知器触发的难易程度。对于一个大的偏移，感知器更容易输出 1。如果偏移负值很大，那么感知器将很难输出 1。显然，引入偏移对于描述感知器改动不大，但是我们后面将看到这将简化符号描述。正因如此，本书剩下部分都将使用偏移，而不是用阈值。

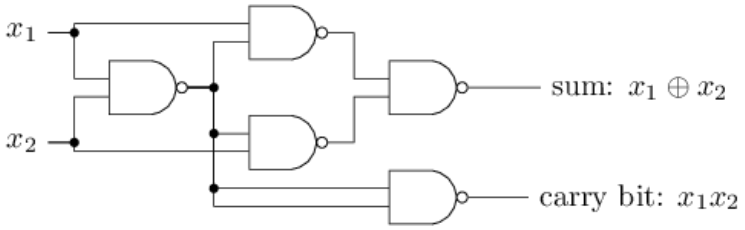
我们已经介绍了用感知器，基于凭据(evidence)的加权求和来进行决策。感知器也能够被用来计算基本逻辑函数，像 AND, OR, and

NAND 这些通常被看做是实现计算的基础。比如说， 假设我们有一个两个输入的感知器，每个输入具有权重-2，整个偏移为3，如下所示感知器：

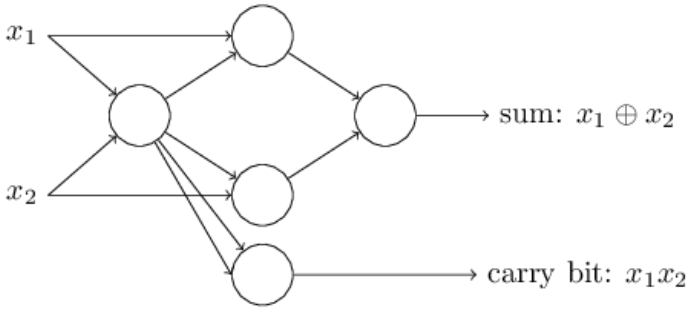


那么我们将看到输入00会产生输出1，因为$(-2)*0+(-2)*0+3=3$是正数。这里，我们引入$*$符号来进行显示乘法。与此类似，输入01和10都将产生输出1。但是输入11将产生输出0，因为$(-2)*1+(-2)*1+3=-1$是负数。因此这个感知器实现了一个NAND门!

这个NAND 门例子表明我们能够使用感知器来计算简单的逻辑函数。实际上，我们能够使用感知器网络来计算任意的逻辑函数。原因是NAND 门在计算中是通用的，也就是，我们能够用一些NAND 门来构造任意计算。比如，我们能够用NAND门来构建两个二进制变量x_1和x_2的位加法电路。这需要对它们按位求和x_1+x_2，同时当x_1和x_2都是1时候进位1。进位就是x_1和x_2的按位乘：

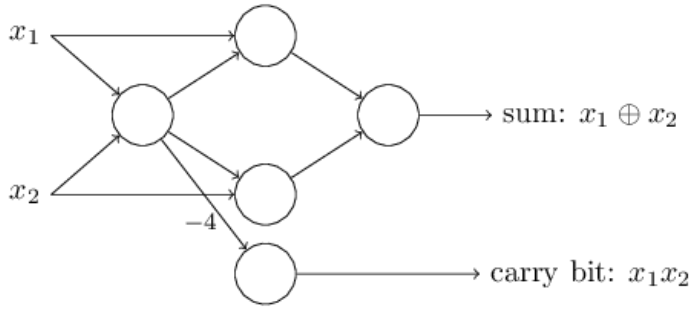


为了得到相同的感知器网络，我们将NAND门用具有两个输入的感知器代替，每一个输入具有权重-2，整体具有偏移3。以下是感知器网络结果，注意我们把右下角的NAND门对应的感知器向左移动了一点，以便绘制图表中的箭头：

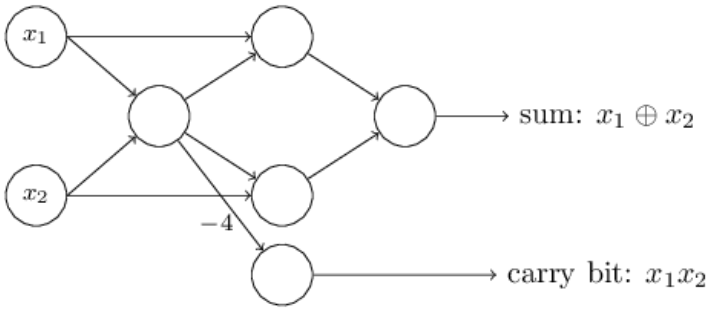


值得注意的是，这个感知器网络中最左边的感知器输出被右下角的感知器使用了两次。当我们定义感知器模型时，我们说这种双倍输出到同一个感知器是不允许的。实际上，这没什么关系，如果我们不允许这种情况，只需要把这两条线合并成一条线，并且将权重设置为-4即可。（如果不这么认为，那么你需要停下来并证明一下它们等同性）

在这点小改变后，感知器网络如下所示，没有标签的权重为-2，所有偏移为3，唯一权重为-4的边进行了标记：



直到现在，我们将变量输入x_1和x_2绘制到感知器网络的左边。实际上，更变的应该绘制一层额外的感知器——*输入层(input layer)*来编码输入：



这个只有一个输出，没有输入的输入感知器，

是一个简写。这么看，假设我们有一个没有输入的感知器，那么权重和 $\sum_j w_j x_j$ 将始终为0，那么如果偏差$b > 0$，感知器将输出1，否则输出0。也就是，这个感知器将简单的输出一个固定的值，而不是希望的值（如上面的例子中x_1）。更好的理解输入感知器可以将其不当作感知器，而是一个简单定义期望值$x_1, x_2, \ldots$输出的特殊单元。

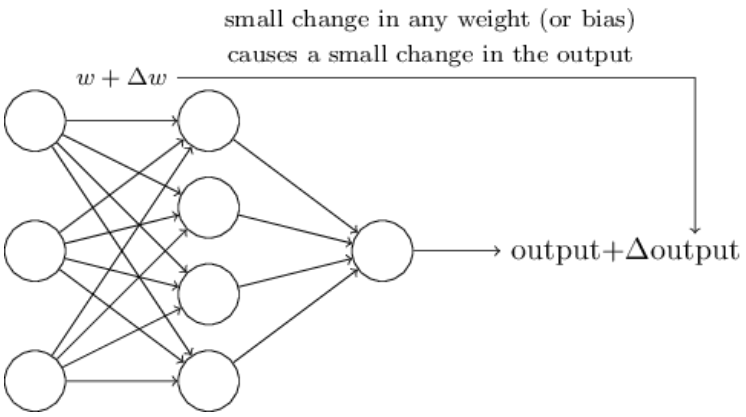
这个加法器例子演示了感知器网络能够模拟具有许多与非门的电路。并且由于与非门对于计算是通用的，因此感知器也是对计算通用的。

感知器的计算通用性同时让人满意，也让人失望。它让人满意是因为感知器网络能够像其他计算装置那么强大，但是也是让人失望是因为它看起来只是一种新的与非门，不是什么大新闻！

不过，情况比这个观点更好一些。它能够让我们设计*学习算法*，这种算法能够自动调整人工神经网络的权重和偏移。这会在响应外部刺激时候发生，而且没有程序员的直接干预。这些学习算法能让我们用一种新的方式使用人工神经网络，它将与传统的逻辑门方法完全不同。替代显示的摆放一个具有与非门和其他门的电路，我们的神经网络能够学会简单地解决这些问题，有些问题对于直接设计 传统电路来说非常困难。

Sigmoid神经元

学习算法听起来好极了。但是我们如何对一个神经网络设计算法。假设我们有一个想通过学习来解决问题的感知器网络。比如，网络的输入可能是一些扫描来的手写数字图像原始像素数据。且我们希望这个网络能够学会调权和偏移以便正确的对它们进行数字分类。为了看到学习如何进行，假设我们在网络中改变一点权重（或者偏移）。我们希望很小的权重改变能够引起网络输出的细微改变。后面我们将看到，这个特性将使学习成为可能。以下示意图就是我们想要的（显然这个网络对于手写识别太简单！）：

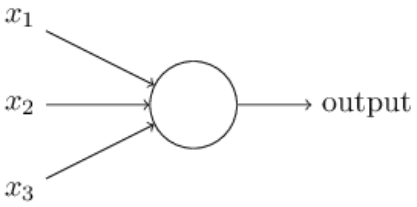


如果权重或者偏移的细小改变能够轻微影响到网络输出，那么我们会逐步更改权重和偏移来让网络按照我们想要的方式发展。比如，假设网络错误的将数字“9”的图像识别为“8”，我们将指出如何在权重和偏移上进行细小的改动来使其更加接近于“9”。（tensorfly.cn社区原创）并且这个过程将不断重复，不断地改变权重和偏移来产生更好的输出。于是这个网络将具有学习特性。

问题是在我们的感知器网络中并没有这样发生。实际上，任何一个感知器的权重或者偏移细小的改变有时都能使得感知器输出彻底翻转，从0到1。这种翻转可能导致网络剩余部分的行为以某种复杂的方式完全改变。因此当“9”可能被正确分类后，对于其他图片，神经网络行为结果将以一种很难控制的方式被完全改变。这使得很难看出如何逐渐的调整权重和偏移以至于神经网络能够逐渐按照预期行为接近。也许对于这个问题有一些聪明的方式解决，但是目前如何让感知器网络学习还不够清楚。

我们能通过引入一种新的人工神经元（*sigmoid*神经元）来克服这个问题。它和感知器类似，但是细微调整它的权重和偏移只会很细小地影响到输出结果。这就是让sigmoid神经元网络学习的关键原因。

好的，让我们来描述一下这个sigmoid神经元。我们将用描绘感知器一样来描绘它：



就像感知器，sigmoid神经元有输入$x_1, x_2, \ldots$。但是输入值不仅是0或者1，还可以是0*到*1的任意值。因此，比如$0.638\ldots$对于sigmoid神经元是一个合理的输入。也像感知器一样，sigmoid神经元对每一个输入都有对应的权重$w_1, w_2, \ldots$，还有一个全局的偏移b。但是输出不是0或者1，而是$\sigma(w \cdot x + b)$，σ是*sigmoid函数**，定义如下：

*顺便说一下，σ有时叫做*逻辑函数*，并且这种新的神经元叫做*逻辑神经元*。记住这个术语非常重要，因为许多研究中的神经网络都在使用它。

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

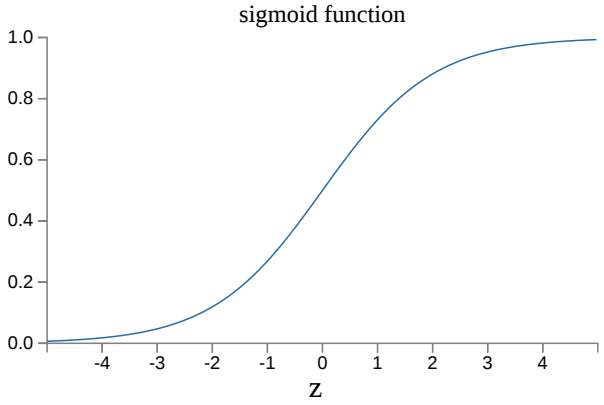
把这些更直接的放在一起，具有输入$x_1, x_2, \ldots$，权重$w_1, w_2, \ldots$和偏移b的sigmoid神经元输出为：

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

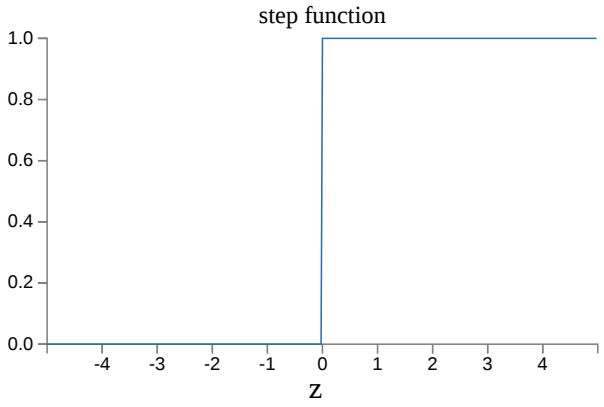
眨眼一看，sigmoid神经元跟感知器非常不同。对于不熟悉的人来说，sigmoid函数的代数形式看起来很晦涩和令人生畏。实际上，感知器和sigmoid神经元有很多相似之处，并且sigmoid函数的代数形式展现了更多的技术细节。

为了理解和感知器模型的相似点，假设$z \equiv w \cdot x + b$是一个大的正数，那么$e^{-z} \approx 0$且$\sigma(z) \approx 1$。换句话说，当$z = w \cdot x + b$是一个很大的正数，sigmoid神经元的输出接近于1，这和感知器结果类似。另外假设$z = w \cdot x + b$是一个很小的负数，那么$e^{-z} \to \infty$，并且$\sigma(z) \approx 0$。因此当$z = w \cdot x + b$是一个很小负数时候，sigmoid神经元和感知器的输出也是非常接近的。只有当$w \cdot x + b$在一个适度的值，sigmoid神经元和感知器偏差才较大。

σ函数的代数形式是什么？我们如何理解它呢？实际上，σ函数的准确形式不太重要——真正有用的是绘制的函数形状，看看下面这个形状：



这是一个阶跃函数的平滑版本：



如果σ函数实际上是一个阶跃函数，那么sigmoid神经元就*是*一个感知器，因为输出是1或者0，取决于$w \cdot x + b$是正还是负。*使用上面我们说明的实际σ函数，将得到一个平滑的感知器。实际上，σ函数的平滑程度是至关重要的，而不是它的具体形式。σ函数越平滑意味着权重微调Δw_j和偏移微调Δb将对神经元产生一个细小的输出改变Δoutput。实际上，微积分告诉我们细小的输出Δoutput近似等于：

*实际上，当$w \cdot x + b = 0$时，感知器趋于0，而阶跃函数输出1。所以，严格的说，我们只需要在一个点上修改阶跃函数。

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

其中总和基于所有的权重w_j，且∂ output/∂w_j和∂ output/∂b分别表示output基于w_j和b的偏导数。不用见到偏导就惊慌！虽然上面具有偏导运算的表达式看起来很复杂，但实际上很简单（这是一个好消息）：输出改变Δoutput是权重和偏移改变Δw_j和Δb的*线性函数*。这种线性使得权重和偏移的细微改变就能很容易使得输出按期望方式微小改变。因此sigmoid神经元具有和感知器同样的定性行为，同时还能方便的找出权重与偏移如何对输出的产生影响。

如果σ函数的形状的确起作用，而不是其精确的形式，那么在表达式(3) 中为什么使用这种形式呢？实际上在本书后面，我们将偶尔考虑神经元的输出函数为$f(w \cdot x + b)$来做为*激活函数*$f(\cdot)$。主要的改变是

当我们使用不同的激活函数之后，等式(5) 中的偏导值将跟随改变。结果就是当我们后面计算偏导时，使用σ函数将简化代数计算，因为指数形式区别于其他具有更好的偏导特性。在任何情况下，σ函数在神经网络中被广泛使用，在本书中也是我们经常使用的激励函数。

我们如何解读sigmoid神经元输出？显然，感知器和sigmoid神经元最大的区别是sigmoid神经元不是只输出0或者1。它能够输出0到1之间任意实数，如0.173 …和0.689 …都是合理的输出。这将非常有用，比如，如果你想使用输出值来表示神经网络中一张输入图像像素的平均强度。但是有时候这又非常讨厌。假设我们想从神经网络中指明要么“输入图像是9”或者“输入图像不是9”。显然像感知器一样只输出0或1更容易处理。但实际上，我们将建立一个约定来处理这个问题，比如，将至少是0.5的输出当做为“9”，其他比0.5小的输出当做为“非9”。我们使用这样约定总是明确的，因此不会产生如何混淆。

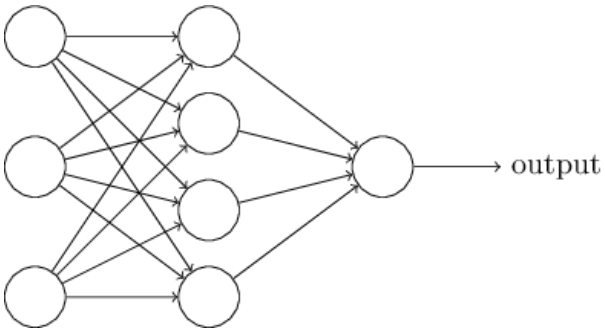
练习

- **Sigmoid神经元模拟感知器，第一部分**
  假设我们将感知器网络中的权重和偏移乘以一个正常数$c > 0$。证明该网络的输出行为不会改变。

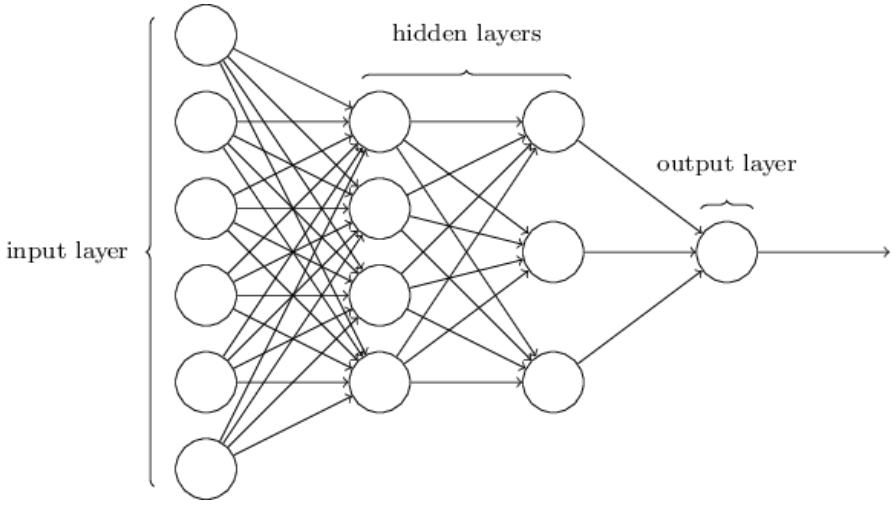
- **Sigmoid神经元模拟感知器，第二部分**
  假设和感知器网络最后一个问题一样，整个输入都设定好，不需要真实的输入值，只需要固定输入值。假设对于网络中某些特殊感知器输入x，使得$w \cdot x + b \neq 0$。将这些感知器用神经元替换，那么再将权重和偏移乘以一个正常数$c > 0$，可以证明当$c \to \infty$时，神经元网络和感知器网络的行为将是一样的。那么对于感知器中$w \cdot x + b = 0$的那些呢，是否不成立？

神经网络体系结构

在下一区段，我将介绍用神经网络能够很好的对手写数字进行区分。先做点准备，让我们命名一下网络中不同部分的术语。假如我们有如下网络：



就像先前说的，网络的最左边一层被称为输入层，其中的神经元被称为*输入神经元*。最右边及输出层包含*输出神经元*，在这个例子中，只有一个单一的输出神经元。中间层被称为*隐含层*，因为里面的神经元既不是输入也不是输出。“隐含”这个术语可能听起来很神秘——当我第一次听到时候觉得一定有深层的哲学或者数学意义——但实际上它只表示“不是输入和输出”而已。上面的网络只包含了唯一个隐含层，但是一些网络可能有多层。比如，下面的4层网络具有2个隐含层：



令人困扰的是，由于一些历史原因，这样的多层网络有时被叫做*多层感知器*或者*MLPs*尽管它是由sigmoid神经元构成的，而不是感知器。在这本书中，我将不会使用MLP，因为我觉得它太混淆了，但是提醒你它可能在其它地方出现。

网络中的输入和输出层设计通常很简单，比如，假设我们试着确定手写图片是否代表“9”。设计网络更自然的方式是将图像像素强度编码进输入神经元。如果图像是一幅64 by 64的灰度图，那么我们将有$4,096 = 64 \times 64$个输入神经元，每一个是介于0和1的强度值。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”。

虽然神经元网络的输入输出层很简单，设计好隐含层却是一门艺术。特别是很难将隐含层的设计过程总结出简单的经验规则。相反，神经元研究者已经为隐含层开发出许多启发式设计，它们能帮助大家获取所期望行为的网络。例如，一些启发式算法能帮助确定如何平衡隐含层数量与网络训练花费时间。我们将在本书后面见到一些这样的启发式设计。

直到现在，我们已经讨论了某一层的输出当作下一层的输入的神经网络。这样的网络被称为*前向反馈*神经网络。这意味着在网络中没有循环——信息总是向前反馈，决不向后。如果真的有循环，我们将终止这种输入依赖于输出的σ函数。这会很难理解，因此我们不允许这样的循环。

但是，人工神经网络也有一些具有循环反馈。这样的模型被称为递归神经网络。这样的模型思想是让神经元在不活跃之前激励一段有限的时间。这种激励能刺激其它神经元，使其也能之后激励一小会。这就导致了更多神经元产生激励，等过了一段时间，我们将得到神经元的级联反应。在这样的模型中循环也不会有太大问题，因为一个神经元的输出过一会才影响它的输入，而不是瞬间马上影响到。

递归神经网络没有前馈神经网络具有影响力，因为递归网络的学习算法威力不大（至少到目前）。但是递归网络仍然很有趣，它们比起前馈网络更加接近于我们人脑的工作方式。并且递归神经网络有可能解决那些前馈网络很难解决的重要问题。但是为了限制本书的范围，我们将集中在已经被大量使用的前馈神经网络上。

一个分类手写数字的简单神经网络

定义了神经网络后，让我们回到手写识别。我们能将识别手写数字分成两个子问题。首先，我们想办法将一个包含很多数字的图像分成一系列独立的图像，每张包含唯一的数字。比如我们将把下面图像

分成6个分离的小图像，

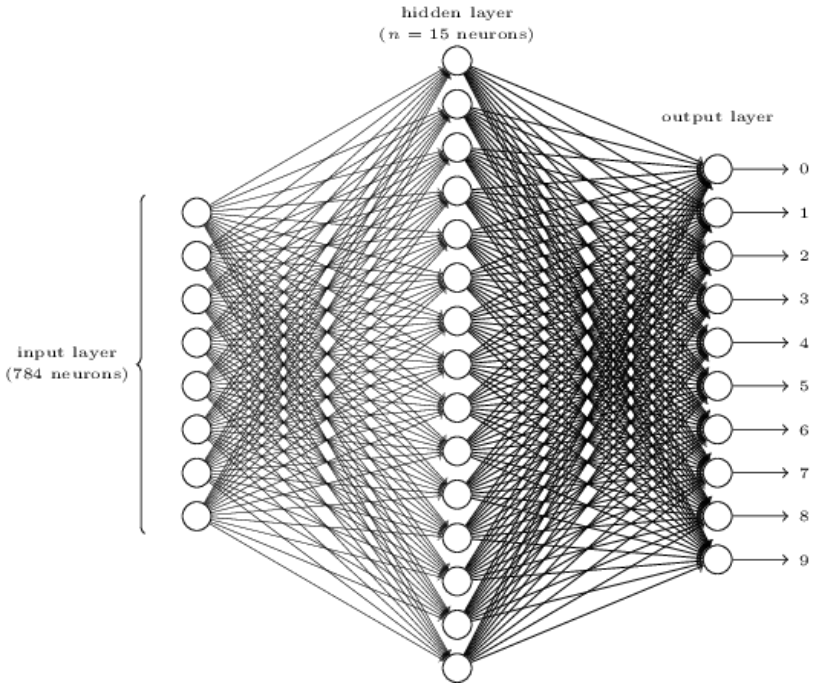
我们人类能够很容易解决这个*分段问题*，但是对于计算机程序如何正确的分离图像却是一个挑战。然后，一旦这幅图像被分离，程序需要将各个数字进行分类。因此，比如，我们希望程序能够将上面的第一个数字

识别为5。

我们主要关注在第二个问题，也就是，分类这些独立的数字图像。我们这么做是因为一旦你能够找到一个好方式来分类独立的图像，分离问题就不是那么难解决了。有许多途径能够解决分离问题，一种途径是试验很多不同的分离图像方法，并采用独立图像分类器给每个分离试验打分。如果独立数字分类器坚信某种分离方式，那么打分较高。

如果分类器在某些片断上问题很多，那么得分就低。这个思想就是如果分类器分类效果很有问题，那么很可能是分离方式不对造成。这种想法和一些变型能够很好解决分离问题。因此无须担心分离，我们将集中精力开发一个神经网络，它能解决更有趣和复杂的问题，即识别独立的手写数字。

为了识别这些数字，我们将采用三层神经网络：



网络的输入层含有输入像素编码的神经元。跟下节讨论的一样，我们用于网络训练的数据将包含许多28 by 28像素手写数字扫描图像，因此输入层包含$784 = 28 \times 28$个神经元。为了简化，我们在上图中忽略了许多输入神经元。输入像素是灰度值，白色值是0.0，黑色值是1.0，它们之间的值表明灰度逐渐变暗的程度。

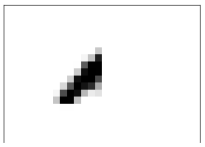
网络的第二层是隐含层。我们将其神经元的数量表示为n，后面还对其采用不同的n值进行实验。这个例子中使用了一个小的隐含层，只包含$n = 15$个神经元。

网络的输出层包含10个神经元。如果第一个神经元被触发，例如它有一个≈ 1的输出，那么这就表明这个网络认为识别的数字是0。更精确一点，我们将神经元输出标记为0到9，然后找出具有最大激励值的神经元。如果这个神经元是6，那么这个网络将认为输入数字是6。对于其他神经元也有类似结果。

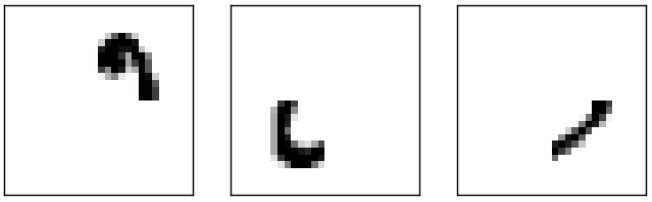
你可能想知道为什么用10个输出神经元。别忘了，网络的目标是识别出输入图像对应的数字（$0, 1, 2, \ldots, 9$）。一种看起来更自然的方法是只用4个输出神经元，每个神经元都当做一个二进制值，取值方式取决于神经元输出接近0，还是1。4个神经元已经足够用来编码输

出，因为$2^4 = 16$大于输入数字的10种可能值。（tensorfly社区原创，qq群： 472113439）为什么我们的网络使用10个神经元呢？是否这样效率太低？最终的理由是以观察和实验为依据的：我们能尝试两种网络设计，最后结果表明对于这个特殊问题，具有10个输出神经元的网络能比只有4个输出神经元的网络更好的学习识别手写数字。这使得我们想知道*为什么*具有10个输出神经元的网络效果更好。这里是否有一些启发，事先告诉我们应该用10个输出神经元，而不是4个输出神经元？

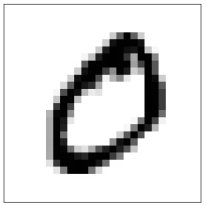
为了弄懂我们为什么这么做，从原理上想清楚神经网络的工作方式很重要。首先考虑我们使用10个输出神经元的情况。让我们集中在第一个输出神经元上，它能试图确定这个手写数字是否为0。它通过对隐含层的凭据进行加权和求出。隐含层的神经元做了什么呢？假设隐含层中第一个神经元目标是检测到是否存在像下面一样的图像：



它能够对输入图像与上面图像中像素重叠部分进行重加权，其他像素轻加权来实现。相似的方式，对隐含层的第二、三和四个神经元同样能检测到如下所示的图像：



你可能已经猜到，这四幅图像一起构成了我们之前看到的0的图像：



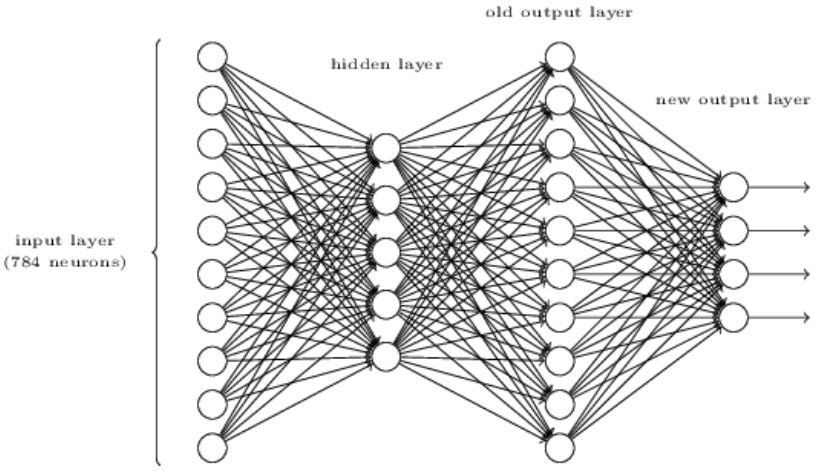
因此如果这四个神经元一起被触发，那么我们能够推断出手写数字是0。当然，这些不*只*是我们能推断出0的凭据类别——我们也可以合理的用其他方式得到0（通过对上述图像的平移或轻微扭曲）。但是至少这种方式推断出输入是0看起来是安全的。

假设神经网络按这种方式工作，我们能对为什么10个输出神经元更好，而不是4个，给出合理的解释。如果我们有4个输出神经元，那么第一个输出神经元将试着确定什么是数字图像中最重要的位。这里没有很容易的方式来得到如上图所示简单的形状的重要位。很难想象，这里存在任何一个很好的根据来说明数字的形状组件与输出重要位的相关性。

现在，就像所说的那样，这就是一个启发式。没有什么能说明三层神经网络会按照我描述的那样运作，这些隐含层能够确定简单的形状组件。可能一个更聪明的学习算法将找到权重赋值以便我们使用4个输出神经元。但是我描述的这个启发式方法已经运作很好，能够节约很多时间来设计更好的神经元网络架构。

练习

- 有一个方法能够对上面的三层网络增加一个额外层来确定数值位表示。这个额外层将先前输出层转换为二进制表示，就像下图说明一样。找到这层新的输出神经元权重和偏移。假定第3层神经元（例如，原来的输出层）正确的输出有至少0.99的激励值，不正确的输出有小于0.01的激励值。



梯度下降学习算法

现在我们已经有一个设计好的神经网络，它怎样能够学习识别数字呢？首要的事情是我们需要一个用于学习的数据集——也叫做训练数据集。我们将使用MNIST数据集，它包含成千上万的手写数字图像，同时还有它们对应的数字分类。MNIST的名字来源于NIST（美国国家标准与技术研究所）收集的两个修改后的数据集。这里是一些来自于MNIST的图像：

就像你看到的，事实上这些数字和本章开始展示的用于识别的图像一样。当然，测试我们的网络时候，我们会让它识别不在训练集中的图像！

MNIST数据来自两部分。第一部分包含60,000幅用于训练的图像。这些图像是扫描250位人员的手写样本得到的，他们一半是美国人口普查局雇员，一半是高中学生。这些图像都是28乘28像素的灰度图。MNIST数据的第二部分包含10,000幅用于测试的图像。它们也是28乘28的灰度图像。我们将使用这些测试数据来评估我们用于学习识别数字的神经网络。为了得到很好的测试性能，测试数据来自和训练数据*不同*的250位人员（仍然是来自人口普查局雇员和高中生）。 这帮助我们相信这个神经网络能够识别在训练期间没有看到的其他人写的数字。

我们将使用符号x来表示训练输入。可以方便的把训练输入x当作一个$28 \times 28 = 784$维的向量。每一个向量单元代表图像中一个像素的灰度值。我们将表示对应的输出为$y = y(x)$，其中y是一个10维向量。比如，如果一个特殊的训练图像x，表明是6，那么$y(x) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$就是网络期望的输出。注意$T$是转置运算，将一个行向量转换为列向量。

我们想要的是一个能让我们找到合适的权重和偏移的算法，以便网络输出$y(x)$能够几乎满足所有训练输入x。为了量化这个匹配度目标，我们定义了一个*代价函数**:

*有时被称为*损失*或*目标*函数。我们在本书中都使用代价函数，但是你需要注意其他术语，因为它通常在一些研究文章和神经网络讨论中被使用到。

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

这里，w表示网络中的所有权重，b是所有偏移，n训练输入的总数，a是网络输入为x时的输出向量，总和是对所有输入x进行的累加。当然输出a取决于x，w和b，但是为了符号简化，我没有指明这种依赖关系。符号$\|v\|$只是表示向量v的长度。我们称C为*二次型*代价函数；它有时候也叫做*均方误差*或*MSE*。检验二次型代价函数，我们能看到$C(w, b)$是一个非负值，因为求和中的每一项都是非负的。此外，对于所有训练数据x，$y(x)$和输出a近似相等时候，$C(w, b)$会变得很小，例如，$C(w, b) \approx 0$。因此如果我们的训练算法能找到合理的权重和偏移使得$C(w, b) \approx 0$，这将是一个很好的算法。相反，如果$C(w, b)$很大——这意味着$y(x)$和大量输出a相差较大。所以训练算法的目标就是找到合适的权重和偏移来最小化$C(w, b)$。换句话说，我们想找到一组

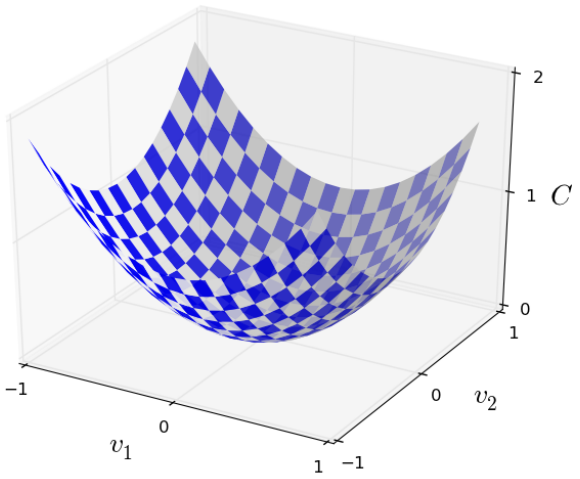
权重和偏移集合，使得代价函数值尽可能小。我们将使用*梯度下降*算法来实现。

为什么引入二次型代价函数？毕竟，我们不是主要关注在网络对多少图像进行了正确分类？为什么不直接最大化这个数值，而不是最小化一个像二次型代价函数一样的间接测量值？问题就在于正确识别的图像数量不是权重和偏移的平滑的函数。对于大多数，权重和偏移的很小改变不会让正确识别的图像数量值有任何改变。这就使得很难指出如何改变权重和偏移来提高性能。如果我们使用一个像二次型的平滑代价函数，它将很容易指出如何细微的改变权重和偏移来改进代价函数。这就是为什么我们首先关注在最小化二次型代价函数，而且只有那样我们才能检测分类的准确性。

即使我们想使用一个平滑的代价函数，你可能仍然想知道为什么在等式(6) 中选择了二次型代价函数。它难道不是一个*临时的*选择？也许如果我们选择一个不同的代价函数，我们将获取到完全不同的最小化权重和偏移？这是一个很好的考虑，后面我们将回顾这个代价函数，并且进行一些改变。但是，这个等式(6) 中的二次型代价函数对理解神经网络学习基础非常好，因此目前我们将坚持使用它。

简要回顾一下，我们训练神经网络的目标是找出能最小化二次型代价函数$C(w,b)$的权重和偏移。这是一个适定问题，但是它会产生许多当前提出的干扰结构——权重w和偏移b的解释、背后隐藏的σ函数、网络架构的选择、MNIST等等。结果说明我们能通过忽略许多这类结构来理解大量内容，且只需要集中在最小化方面。因此现在我们将忘掉所有关于代价函数的特殊形式，神经网络的连接关系等等。相反，我们将想象成只简单的给出一个具有许多变量的函数，且我们想要最小化它的值。我们将开发一个新技术叫*梯度下降*，它能被用来解决这类最小化问题。然后我们将回到神经网络中这个想最小化的特殊函数。

好的，假设我们将最小化一些函数$C(v)$。它可能是具有许多变量$v = v_1, v_2, \ldots$的任意真实值函数。请注意，我们将用v替换符号w和b来强调它可以适合任意一个函数——我们并不是一定在神经网络的背景下考虑。为了最小化$C(v)$，可以把C想象成只具有两个变量，即v_1和v_2：



我们想要找到怎样才能使C达到全局最小化。现在，当然，对上面绘制的函数，我们能看出该图中最小化位置。从这种意义上讲，我可能展示了一个*太*简单的函数！一个通常的函数C可能由许多变量构成，并且很难看出该函数最小化位置在哪里。

一种解决办法就是使用微积分来解析地找到这个最小值。我们能计算导数，然后使用它们来找到C函数最小极值的位置。当C只有一个或少量变量时，这个办法可能行得通。但是当我们有许多变量时，这将变成一场噩梦。且对于神经网络，我们通常需要*更多*的变量——最大的神经网络在极端情况下，具有依赖数十亿权重和偏移的代价函数。使用微积分来求最小值显然行不通！

(在断言我们通过把C当作只具有两个变量的函数来获得领悟后，我将用两段第二次转回来，并且说：“嘿，一个具有超过两个变量的函数是什么呢？”，对此很抱歉。请相信我，把C想象成只具有两个变量的函数对我们理解很有帮助。有时候想象会中断，且最后两段将处理这种中断。好的数学思维通常会涉及应付多个直观想象，学会什么时候合适使用每一个想象，什么时候不合适。）

Ok，所以用微分解析的方法行不通。幸运的是，我们可以通过一种非常直观的类比来找到一种“行得通”的算法。首先将我们的函数看作是一个凹形的山谷。（瞄一眼上面的插图，这种想法应该是很直观的。）好，现在让我们想象有一个小球沿着山谷的坡面向低处滚。生活经验告诉我们这个小球最终会到达谷底。或许我们可以采用类似的思路找到函数的最小值？好，首先我们将随机选取一个点作为这个（想象中的）球的起点，然后再模拟这个小球沿着坡面向谷底运动的轨迹。我们可以简单地通过计算 C 的偏导数（可能包括某些二阶偏

导数）来进行轨迹的模拟——这些偏导数蕴涵了山谷坡面局部“形状”的所有信息，因此也决定了小球应该怎样在坡面上滚动。

根据我刚才所写的内容，你可能会以为我们将从小球在坡面上滚动的牛顿运动方程出发，然后考虑重力和坡面阻力的影响，如此等等……实际上，我们不会把“小球在坡面滚动”的这一类比太过当真——毕竟，我们我们的初衷是要设计一种最小化 C 的算法，而不是真地想精确模拟现实世界的物理定律！“小球在坡面滚动”的直观图像是为了促进我们的理解和想象，而不应束缚我们具体的思考。所以，与其陷入繁琐的物理细节，我们不妨这样问自己：如果让我们来当一天上帝，那么我们会怎样设计我们自己的物理定律来引导小球的运动？我们该怎样选择运动定律来确保小球一定会最终滚到谷底？

为了将这个问题描述得更确切，现在让我们来想想，当我们将小球沿着 v_1 方向移动一个小量Δv_1，并沿着 v_2 方向移动一个小量 Δv_2 之后会发生什么。微分法则告诉我们，C 将作如下改变:

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

我们将设法选择 Δv_1 和 Δv_2 以确保 ΔC 是负数，换句话说，我们将通过选择 Δv_1 和 Δv_2 以确保小球是向着谷底的方向滚动的。为了弄清楚该怎样选择 Δv_1 和 Δv_2，我们定义一个描述 v 改变的矢量 $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$，这里的 T 同样是 转置（transpose）算符，用来将一个行矢量转化为列矢量。我们还将定义 C 的“梯度”，它是由 C 的偏导数构成的矢量，$\left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$。我们将“梯度”矢量记为 ∇C，即：

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

很快我们将会用 Δv 和梯度 ∇C 来重写 ΔC 的表达式。在这之前，为了避免读者可能对“梯度”产生的困惑，我想再多做一点解释。当人们首次碰到∇C这个记号的时候，可能会想究竟该怎样看待∇这个符号。∇的含义到底是什么？其实，我们完全可以把∇C整体看作是单一的数学对象——按照上述公式定义的矢量，仅仅是偶然写成了用两个符号标记的形式。根据这个观点，∇像是一种数学形式的旗语，来提醒我们“嘿，∇C是一个梯度矢量”，仅此而已。当然也有更抽象的观点，在这种观点下，∇ 被看作是一个独立的数学实体（例如，被看作一个微分算符），不过我们这里没有必要采用这种观点。

使用上述定义，ΔC 的表达式 (7) 可以被重写为：

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

这个方程有助于我们理解为什么 ∇C 被称为“梯度”矢量：∇C 将 v 的改变和 C 的改变联系在了一起，而这正是我们对于“梯度”这个词所期望的含义。然而，真正令我们兴奋的是这个方程让我们找到一种 Δv 的选择方法可以确保 ΔC 是负的。具体来说，如果我们选择

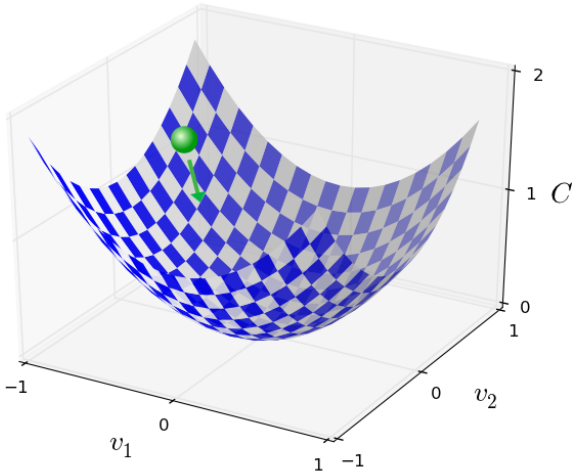
$$\Delta v = -\eta \nabla C, \tag{10}$$

这里 η 是一个正的小参数，被称为“*学习率*”（learning rate)。这样方程span id="margin_859162866010_reveal" class="equation_link">(9) 化为 $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$。如果我们根据公式 (10) 指定 v 的改变，由于 $\|\nabla C\|^2 \geq 0$ ，它确保了 $\Delta C \leq 0$，即 C 的值将一直减小，不会反弹。（当然要在近似关系 (9)) 成立的范围内）。这正是我们期望的性质！因此，我们将在梯度下降算法（gradient descent algorithm）中用公式 (10) 来定义小球的“运动定律”。也就是说，我们将用公式(10) 来计算 Δv 的值，然后将小球的位置 v 移动一小步：

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

然后，我们使用这一规则再将小球移动一小步。如果我们不断这样做，C 将不断减小，符合期望的话，C 将一直减小到全局最小值。

总之，梯度下降算法（gradient descent algorithm）是通过不断计算梯度∇C，然后向着梯度*相反*的方向小步移动的方式让小球不断顺着坡面滑向谷底。这个过程可以形象地用下图表示：



注意：采用这一规则，梯度下降算法并没有模拟真实的物理运动。在现实世界，小球具有动量，而动量可能让小球偏离最陡的下降走向，甚至可以让其暂时向着山顶的方向逆向运动。只有考虑了摩擦力的作用，才能确保小球是最终落在谷底的。但这里，我们选择Δv的规则

却是说“立马给我往下滚”，而这个规则仍然是寻找最小值的一个好方法！

为了让“梯度下降法”能够正确发挥作用，我们必须选择一个足够小的“学习率” η 以确保方程 (9) 是一个好的近似。如果我们不这样做，最终可能会导致 $\Delta C > 0$，这当然不是我们想要的。（机器学习社区tensorfly.cn原创）与此同时，我们也不希望 η 太小，因为η太小的话每一步的 Δv 就会非常小, 从而导致梯度下降算法的效率非常低。在实际运用过程中，η 经常是变化的从而确保一方面方程 (9) 是好的近似，另一方面算法也不会太慢。后面我们将会明白具体该怎么做。

前面我们已经解释了当函数 C 仅仅依赖两个变量时的梯度下降算法。但其实，当 C 依赖更多变量的时候，前面所有的论述依然成立。具体来说，假设函数 C 依赖m个变量， $v_1, \ldots, v_m$，那么由微小改变 $\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$ 引起的 C 的变化ΔC是:

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

这里梯度矢量 ∇C 定义为

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

类似于两个自变量的情形，我们可以选择

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

那么就可以确保由公式 (12) 得到的 ΔC 是负的。这给了我们一种沿着梯度找到最小值的方法，即使 C 依赖于很多变量。即通过不断重复地使用以下更新规则，

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

你可以把这种更新规则看作是“梯度下降算法”的定义。它给了我们一种不断改变 v 的位置从而找到函数 C 最小值的方法。这一规则并不总是有效——有些情况可能会导致出错，阻止梯度下降算法找到 C 的全局最小值。在后面的章节我们会回过头来讨论这种可能性。不过，在实际应用中，梯度下降算法常常非常有效，并且在神经网络中，我们发现它也是一种最小化代价函数（cost function）的强有力方法，因此也有助于神经网络的学习。

的确，从某种意义上说梯度下降其实是寻找最小值的最佳策略。假设我们正试图通过移动一小步 Δv 来让函数 C 尽可能地减小。这等价于要要最小化 $\Delta C \approx \nabla C \cdot \Delta v$。我们将限定移动步幅的大小，即令 $\|\Delta v\| = \epsilon$ 其中 $\epsilon > 0$ 是一个固定的小量. 换句话说，我们想移动步长

限定的一小步，并试图找到一个移动方向能够让C尽可能减小。可以证明这里最小化$\nabla C \cdot \Delta v$的Δv选择是 $\Delta v = -\eta \nabla C$, 这里 $\eta = \epsilon/\|\nabla C\|$ 是由步长约束 $\|\Delta v\| = \epsilon$ 所要求的。因此梯度下降可以被认为是一种沿着 C 局域下降最快的方向一步一步向前走，从而找到最小值的方法。

练习

- 证明上面最后一段的断言。*提示：*如果你还不熟悉柯西-施瓦茨不等式，你最好学习一下，这会对你很有帮助。

- 我们阐述当C有两个变量时的梯度下降，也阐述了超过两个变量的梯度下降。当C只有一个变量时候，会发生什么呢？你能提供在一个维度上梯度下降的几何解释吗？

人们调研过很多从梯度下降变种而来的算法，包括精确模仿小球在真实物理世界运动的方法。这些模仿小球真实物理运动的方法有一些优点，但是也有一个显著的缺点：需要计算 C 的二阶偏导数，而那样的话，计算成本会很高。为了看清楚为什么计算成本会很高，假设我们要计算所有的二阶偏导数 $\partial^2 C/\partial v_j \partial v_k$。 如果这里需要计算的 v_j 变量有一百万个，我们就要计算大概万亿（百万的平方）量级的二阶偏导数。*！这个计算成本非常高。 正因为上面所说的，人们搞出不少小技巧来避免计算成本过高的问题，同时，寻找梯度下降的替代算法也是非常活跃的研究领域。不过，本书将始终把梯度下降（及其变种）算法作为我们训练神经网络的主要方法。

*实际上，稍微估准一点的话，需要计算的偏导数大概有万亿的一半个，因为∂2C/∂vj∂vk=∂2C/∂vk∂vj。不管怎样，你依然很容易看得出计算成本会非常高。

我们怎样把梯度下降应用于神经网络的学习过程呢？具体思路是用梯度下降来寻找权重（weights）w_k 和 偏移（bias） b_l ，从而最小化代价函数(6) 。为了看得更清楚，我们将梯度下降更新规则中的变量 v_j用权重和偏移替换，进行重新表述。换句话说，我们现在的“位置”有 w_k and b_l 这些分量, 而梯度矢量 ∇C 也具有相应的分量 $\partial C/\partial w_k$ 和 $\partial C/\partial b_l$。将梯度下降规则用这些分量的形式写出来，我们有

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

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

通过不断重复应用这一更新规则，我们可以“从山上滚下来”，符合预期的话，我们将找到代价函数的最小值。换句话说，这个规则可以用于神经网络的学习过程。

在运用梯度下降规则的时候会碰到许多挑战难题。在后面的章节我们会仔细研究这些困难。但这里我只想提及其中的一个难题。为了更好

地理解是什么难题，让我们回过头来看一下公式 (6) 定义的二次型代价函数。注意到这个代价函数具有形式$C = \frac{1}{n}\sum_x C_x$, 即，每一个训练样例误差 $C_x \equiv \frac{\|y(x)-a\|^2}{2}$ 的平均。实际上，为了计算梯度 ∇C ，我们需要对每个训练输入（trainning input）分别计算其梯度 ∇C_x ，然后再对它们求平均。不幸的是，如果训练输入的样本数量非常大，那么这样会消耗很多时间，因此导致神经网络的学习过程非常慢。

有一个可用于加速学习过程的办法被称作 *“随机梯度下降”*（stochastic gradient descent）。 这个办法是通过随机选择训练输入的少量样本，并只计算这些样本的∇C_x的平均值来估算梯度 nablaC。通过计算这些少量样本的平均值，我们可以很快估算出梯度∇C的真实值，从而加速梯度下降过程，即加速神经网络的学习过程。

具体来说， 首先“随机梯度下降” 要随机挑出 m（一个较小的数目）个训练输入。我们将这些随机的训练输入标记为 $X_1, X_2 \ldots X_m$, 并将它们称为 一个 “小组”（mini-batch）。假设样本数m足够大，我们预计这个随机小组 ∇C_{X_j} 的平均值应该和所有 ∇C_x 的平均值大致相等，即

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

这里第二个求和符号表示对整个训练数据集的求和。交换等号两边的顺序，我们有

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

这个公式明确了我们可以通过仅仅计算随机选择的“小组”梯度来估算整体的梯度值。

为了将其和神经网络的学习过程直接联系起来，假设我们用 w_k 和 b_l 标记神经网络中的权重（weights）和 偏移（biases），那么随机梯度下降的工作方式为：随机挑出一“小组”训练输入，并用如下方式来训练

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

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

这里的求和是对当前小组里的所有训练样本 X_j 进行的。然后我们再随机挑出另一小组进行训练。如此反复下去，直到我们用尽了所有的训练输入。这个过程被称为完成了一“*代*”（epoch）训练。到那个时候，我们就可以重新开始另一代训练。

有的时候，我们需要注意：人们在标定代价函数(cost function)和对权重、偏移进行“组”(mini-batch)更新时所采用的惯例可能有所差异。公式 (6) 中，我们用了一个 $\frac{1}{n}$ 的因子对整体代价函数进行了标定。有时候，人们会忽略这个 $\frac{1}{n}$ 因子，即对所有训练样例“求和”而不是“求平均”。这一点在训练样例总数不能提前确定的时候非常有用，比如当更多的训练数据在不断实时产生的时候就会发生这种情况。同样类似，公式(20) 和(21) 中的“组”更新规则，有时候也会忽略求和前面的 $\frac{1}{m}$因子。从概念上说，两者几乎没什么不同，因为这等价于重新标定了学习率 η 而已。不过当我们在仔细对比不同的研究工作时，需要对此留心一点。

我们可以将随机梯度下降看做是某种类似于民意调查的过程：将梯度下降应用于一“小组”样例要比应用于全部样例要容易得多，就如同采样民意调查要比全民公投要容易得多。例如，在MNIST样例中，整个训练集的样本数 $n = 60000$ 个，而比如说我们选定的“小组”样本数为$m = 10$个。这意味着我们在估算梯度的时候会快6000倍！当然，估算结果不一定精确——因为有统计涨落的误差——不过，我们不需要它精确：因为我们关心的仅仅是这个过程是不是整体向着C减小的方向在走，而不是要严格计算出梯度值。实际上，随机梯度下降在训练神经网络的过程中是非常常见和有效的手段，也是本书将要演绎的许多训练技巧的基础。

练习

- 梯度下降的一种极端方式是使用大小为1的“小组”。也就是，给定一个训练输入x，我们按照规则$w_k \rightarrow w_k' = w_k - \eta \partial C_x / \partial w_k$和$b_l \rightarrow b_l' = b_l - \eta \partial C_x / \partial b_l$来更新权重和偏差。然后选择另外一个训练输入，再来更新权重和偏差。反复这样，这个过程被称为*在线*，*在线*或*增量*学习。在线学习中，神经网络每次从一个训练输入中进行学习（就像人来开始那样）。请指出在线学习的优点和缺点，相对于具有“小组”大小为20的随机梯度下降算法。

好，这一节的最后，我们来讨论一个对于刚刚接触梯度下降算法的人常有的困扰。在神经网络中，代价函数 C 当然依赖于很多变量，即，所有的权重和偏移。因此，从某种意义上说，C 定义了一个非常非常高维空间中的曲面。有些人可能会觉得；“嗯，我需要想象这些额外的维度”，然后开始担心：“我甚至连四维空间都没法想象，更别说五维或者五百万维了”。是不是这些人缺少了某种“真正的”超级数学家所具有的超能力？当然不是。即便是最牛哄哄的数学家也很难想象四维空间，如果可以的话。技巧在于这些数学家发展了一套描述问题的替代方法，而这正是我们刚才所做的：我们用一套代数形式的

（而非直观的）算法来表征 ΔC，从而进一步思考怎样让C不断减小。那些擅长思考高维空间的人，其实头脑中内建了一个“库”，这个“库”包含了许多类似于刚才描述的这些抽象方法。这些方法也许不像我们所熟知的三维空间那样简单直观，不过一旦你在头脑中也建立了这个“库”，你就可以驾轻就熟地“想象”高维空间了。这里我就不再深入讨论下去了，如果你对于数学家们怎样想象高维空间感兴趣的话，你可能会喜欢读 这篇讨论。虽然其中讨论到的一些方法非常复杂，但主要的方法还是比较直观，也可以被大家掌握的。

使用神经网络识别手写数字

好了，让我们使用随机梯度下降算法和MNIST训练数据来写一个能够学习识别手写数字的程序。首先我们需要获得MNIST数据。如果你是git的使用者，你可以通过克隆这本书的源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，你可以通过这里下载所需的数据和源代码。

顺便说一下，早些时候我在描述MNIST数据时，我说过它被分成了两份，其中60000幅图片用于训练，另外10000幅用于测试。那是MNIST官方的陈述。事实上，我们将要使用的分法会有一点区别。用于测试的数据我们不会改动，但是60000幅用于训练的图片将被分成两部分：其中一份包括50000幅图片，将用于训练我们的神经网络，剩下的10000幅图片将作为*验证集（validation set）*。虽然我们在这一章不会用这个验证集，但是在这本书之后的部分我们会发现验证集对确定怎么设置神经网络的一些*超参数(hyper-parameters)*很有用，比如学习速率等，这些参数没有在我们的学习算法中直接选取。尽管验证集不是MNIST数据的原始规范，但是很多人都是用这种方式使用MNIST，验证集的使用在神经网络中非常普遍。从现在开始，当我说到“MNIST训练数据集”时，我指的是我们分出来的50000幅图片，而不是原来的60000幅图片集。*

*像之前提到的，MNIST数据集是基于NIST（美国国家标准与技术研究所）收集的两种数据。为了构建MNIST数据集，原始数据被Yann LeCun，Corinna Cortes和Christopher J. C. Burges转换为更加方便使用的格式。请参考这个连接查看细节。我的git仓库中存放的数据集是用一种更容易在Python中使用的格式。我也是从蒙特利尔大学（链接）LISA机器学习实验室获得的这个特殊格式。

除去MNIST数据，我们还需要一个叫Numpy的Python函数库用于做快速线性代数运算。如果你还没有安装Numpy，你可以从这里获得它。

在给出完整的列表之前，下面让我先解释一下神经网络程序的核心特征。最重要的部分就是一个叫Network的类（class），它被用来表示一个神经网络。下面是用于初始化Network对象的代码:

```
class Network(object):

    def __init__(self, sizes):
```

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

在这段代码中，列表sizes包含在对应层的神经元的个数。因此，比如说，如果我们想要构造一个Network对象，其中第一层有2个神经元，第二层有3个神经元，最后一层只有1个神经元，我们可以通过下面的代码做到些：

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

Network对象中所有的偏移（biases）和权重（weights）都通过Numpy中的np.random.randn函数产生标准正态分布的随机数进行初始化。这个随机的初始化给了随机梯度下降算法一个起始点。在随后的章节中我们会找到更好的初始化的方法，但是现在会用这个方法。注意到Network初始化的代码假设了第一层神经元为输入层，并且没有对那些神经元设置偏移量，因为只有在计算之后层的输出时才用到偏移量。

同时也要注意到，偏移量和权重系数被储存为Numpy矩阵的列表。因此，比如说net.weights[1] 就是一个Numpy矩阵，它储存着连接第2层和第3层神经元的权重系数。（注意不是第1层和第2层，因为python列表的索引指标是从0开始的。）因为net.weights[1]比较冗长，让我们用w表示这个矩阵。矩阵元w_{jk}就是连接第2层中第k个神经元和第3层中第j个神经元的权重系数。索引指标j和k这样的排序是不是看上去有点奇怪，交换j和k的顺序是不是看上去更合理？这样排序的最大的好处就是第三层的神经元的激活向量（vector of activations）可以直接写成：

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

这个公式描述的过程有点多，因此让我们一点一点地来解开。a是第2层神经元的激活向量（vector of activations）。为了得到a'，我们先将a乘以权重矩阵w，然后加上偏移向量b，最后我们对向量$wa + b$中的每一个元使用函数σ。（这叫函数σ的*向量化（vectorizing）*。）很容验证，在计算sigmoid神经元的输出结果时，公式(22) 给出的结果会和我们之前用公式(4) 所说的方式给出的结果是一样的。

练习

- 将公式(22) 写成向量元素的形式，验证在算sigmoid神经元的输出时，它给出的结果和公式(4) 的结果是一样的。

有了这些在心中，很容易写一段程序用于计算一个Network 对象的输出结果。我们从定义sigmoid函数开始：

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

注意到，当输入量z是一个向量或者Numpy数组，Numpy自动地对向量的每一个元运用sigmoid函数。

然后我们对Network 类加一个feedforward方法。当一个输入量a 给了network，这个方法就给出对应的输出*. 这个方法所做的就是将公式 (22) 应用到每一层:

*假定输入a是一个(n, 1)Numpy的n维数组，不是一个(n,)向量。这里n是网络的输入数量。如果你试着用一个(n,)的向量作为输入，你将得到一个奇怪的结果。虽然用一个(n,)的向量看起来是一种更自然的选择，但是使用一个(n, 1)的n维数组使得更容易修改代码来一次性前向反馈多个输入，这时常更加方便。

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

当然，我们想让Network 对象做的最主要的事情是学习。为了达到这个目的，我们要给它们一个SGD方法，用这个方法实现随机梯度下降算法。下面是代码。有几处地方有一点难以理解，但是我们会在给出代码后加以讲解。

```
    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The "training_data" is a list of tuples
        "(x, y)" representing the training inputs and the desired
        outputs.  http://tensorfly.cn/home 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是一个由(x, y)类型元组组成的列表，其中x表示训练数据的输入，y为对应的输出。变量epochs和mini_batch_size正如你所预期的一样,分别是训练的epochs数和取样时mini_batch_size的大小。eta是学习的速率，η。如果可选参量test_data被提供了，程序在每一个训练“代”（epoch）结束之后都会评估神经网络的表现，然后输出部分进展。这对跟踪进展有用，但是会大大减慢速度。

这段代码按照下面的方式运行。在每一代（epoch），程序首先随机地打乱训练数据的顺序，然后将数据分成合适大小的mini_batch。这个很容，就是随机从训练数据中抽样。接下来，对每一个epochs，我们使用一次梯度下降算法。这个通过self.update_mini_batch(mini_batch, eta)所对应的程序代码实现，这段代码会利用mini_batch中的训练数据，通过一个梯度下降的循环来更新神经网络的权重系数和偏移量。下面是update_mini_batch方法的源代码：

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

大部分工作都是下面这一行做的：

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

这个牵涉到一个叫*逆传播（backpropagation）*的算法，这个算法可以快速计算代价函数（cost function）的梯度。因此update_mini_batch 所作的仅仅就是对 mini_batch中每一个训练数据样本计算这些梯度，然后更新self.weights和self.biases。

现在我不会展示出self.backprop的源代码。我们会在下一章学习逆传播是怎么工作的，以及self.backprop的源代码。现在，就假设它如上面所说的方式工作，返回每一个训练数据样本x所对应成本的正确梯度。让我们看一下整个程序，包括之前忽略了的说明文档。除了self.backprop，整个程序是无需解释的 - - 所有重要的事情是在self.SGD和self.update_mini_batch中做的，我们之前已经讨论过了。self.backprop方法在计算梯度的时候还要用到一些额外的函数，包括sigmoid_prime，用来计算σ函数的导数，和self.cost_derivative。后面这个我暂时不会在这里描述。只要看一看这些方法的代码或者说明文档，你就可以了解它们的主要意思（甚至细节）。我们将在下一章详细研究它们。注意到这个程序看上去很长，但是大部分都是说明文档，让这个程序更容易理解。实际上，整个程序只有74行非空白、非注释代码。所有的代码可以在GitHub上找到。

```
"""
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))
```

这个程序识别手写数字的效果怎么样呢? 好，让我们从加载MNIST数据开始。我将用下面这个小脚本mnist_loader.py来做这件事, 我们将在python shell中运行下面的指令，

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

当然，这也可以通过一个独立的python程序来完成，不过如果你一直在跟着这个教程走，在python shell中完成应该是最简单的。

加载完MNIST数据后，我们将要建立一个有30个隐藏神经元的Network。这件事是在我们导入上面的Python程序之后去做的，对应的变量我们称其为network,

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

最终，我们将用随机梯度下降法，由MNIST training_data完成学习过程。该过程将历经30代（epoch），其每“组”（mini-batch）的样本数为10，学习率（learning rate）为 $\eta = 3.0$,

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

注意：如果你在一边在读教程一边在跑程序，你会发现这些程序的执行需要花点时间 —— 对于一般的主机（2015年左右），大概需要几分钟。我建议你一边运行，一边继续阅读，并且周期性地查看代码的输出结果。如果你时间比较赶，你可以通过减少学习的epoch数目，减少隐藏神经元的数目，或者只用部分训练数据等方法来加速程序的运行过程。注意用于生产环境的代码（production code）会远远快于现在的速度：这里的python脚本是用来帮助你理解神经网络是怎样工作的，而并非以高效运行为目标的代码！当然，一旦我们将神经网络训练好，我们就可以在几乎任何计算平台上运行得飞快。例如，一旦我们通过神经网络的学习过程获得了一组很好的权重和偏移，我们就可以将其非常容易地移植为网页浏览器的javascript版本，或者是移动设备的原生应用版本。不管怎样，下面是训练神经网络过程的一组结果。这组结果显示了经历每一代训练后，可以正确辨认出来的测试图片的个数。正如你所看到的，经历过第一代训练后，其准确率已经达到10,000张图片中识别出9,129张图片，并且这个数字在不断增加，

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

就是说，训练后的网络给出了一个分类准确率峰值大约在 95% 到 95.42% 之间的结果 ("第28代")! 这个首次试验的结果非常鼓舞人心。不过我需要提醒你的是，如果你来运行代码的话，可能结果和我的不完全一样，因为我们初始化权重和偏移参数的时候是随机选择的。为了获得这里的结果，我实际上挑了三次运行中最好的那次结果。

让我们将隐藏神经元的数目改为100个，重新跑一遍上面的程序试验。跟前面的情况一样，如果你在一边读本文一边运行代码，需要提醒你的是，这个运行时间可能会有点长（在我的主机上，这个试验每一代大约需要几十秒的时间），因此在代码执行的时候，你最好并行地接着往下读。

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

我们能非常确切地看到，这个试验将结果的准确率提高到 96.59%。至少，对于当前情形，增加隐藏神经元的个数让结果变得更好。*

*读者反馈指出这个试验的许多变化，有一些训练结果更加糟糕。使用在第三章将引入的技术可以大幅度减少不同网络训练的性能变化。

当然，为了或者这些准确率，我们还要选择一个具体的训练“代”数（number of epochs），mini-batch 的大小（即每个batch中的样本个数），以及学习率 η。正如上面提到的，这些都是我们神经网络的超参数（hyper-parameters）。所谓“超参数”是要将它们和学习算法的训练“参数”（权重和偏移）相区分。如果我们选定的“超参数”比较差，我们的结果可能也比较糟糕。例如，如果我们选择学习率为 $\eta = 0.001$ 的话,

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

这些结果就显得不那么鼓舞人心了,

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

不管怎样，你还是看到神经网络的表现还是在随着学习时间的推移不断慢慢变好。这个结果暗示我们需要增加学习率，比如说 $\eta = 0.01$。如果我们这么做之后，结果更好了，这就意味着我们应该进一步增加学习率。（如果某种改变让结果更好，那就试着再多改变一点！）如果我们这么重复若干次后，我们的学习率可能就比较大了，比如说 $\eta = 1.0$（有可能甚至会调到$\eta = 3.0$），这个值就和我们之前的试验值相近了。所以，就算我们一开始“超参数”选得不好，我们至少有足够的信息量可以知道该怎样改进我们“超参数”的选择。

一般来说，分析神经网络的错误是很有挑战性的。尤其是，当初始选择的“超参数”，其得到的结果和随机噪声没什么差别的话，这句话就更对了。假设我们选择神经网络的隐藏神经元数目还是30个，不过将学习率变为 $\eta = 100.0$:

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

这种选择就有点过了，即，学习率的值有太高了:

```
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
```

现在设想一下我们第一次遇到这个问题。当然，我们从之前的试验*知道*正确的减少学习率的方式。但是如果是我们第一次遇到这个问题，那么在结果中没有什么能指导我们怎么去做。我们可能不只担心学习率，而且还担心神经网络的其他每个方面。我们可能想知道是否采用了一种让网络很难学习的初始化权重和偏移？或者我们没有足够的训练数据来正确学习？可能我们没有进行足够的“代”来学习？或者用这种架构的神经网络不可能学会去识别手写数字？可能学习率太*低*？或者可能学习率太高？当你第一次遇到这个问题，你不可能总是很确定它们。

我们应该从中吸取的经验是，调试分析神经网络的工作不是那么简单的事。就像一般的编程调试一样，这其实是个手艺活儿。为了能够从神经网络中获得漂亮的结果，你需要学习调试这门手艺。更普遍地来说，我们需要发展一套启发性的思路来选择好的“超参数”和网络层次结构。本书我们将对此展开详细论述，包括上面的程序我是怎样选择“超参数”的。

练习

- 试着构造一个只有两层的神经网络——一个输入层和一个输出层，没有隐含层——即分别784和10个神经元。使用随机梯度下降法训练这个网络，你能得到什么样的分类精确度？

之前，我跳过了关于怎样加载MNIST数据的细节。这个其实非常直截了当。出于完整性，这里是代码。用于存储 MNIST数据的数据结构在代码说明中作了解释—— 都是很直接的东东, tuples 和 Numpy ndarray 对象 的lists (如果你对ndarray不熟，可以将其看作是矢量):

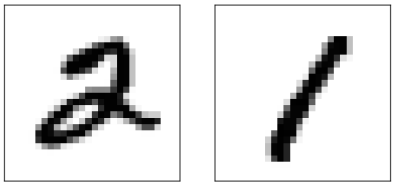
```
"""
mnist_loader
```

```
~~~~~~~~~~~~

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
```

上面我说我们程序的结果相当好。这具体是啥意思呢？究竟是相对于啥东东好? 如果我们有一些可以对比的测试基准，所谓的“好”才更有信息量一些。最简单的基准，当然是纯随机地去猜数字。这个准确率大概是 10%。我们的结果当然远比这要好！

对于一个不那么普通的比较基准呢？让我们先试试非常简单的办法：去看图片的暗度值。比如说，一张2的图片，一般来说比1的图片要暗一些，仅仅是因为2相对于1有更多像素是黑的，比如下面的例子：



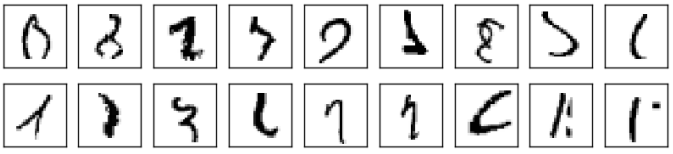
这暗示我们可以用训练数据来计算每个数字，$0, 1, 2, \ldots, 9$，的平均暗度值。每当提供我们一张新图，我们就计算这张图有多暗，然后根据那个数字的平均暗度值和这张图的暗度值最接近，来猜测这张图是什么数字。这个方法很简单，也很容易编程实现，所以这里我就不把代码写出来了，如果你对此感兴趣，可以看一看这个Github库。即便如此，这相对于随机猜测已经有了很大进度，对于$10,000$张测试图片，已经可以猜对大概 $2,225$ 张，即准确率大约是22.25%。

要求准确率在20% 和 50% 之间，找到其他办法来实现并不困难。如果你再努力一点，超过50% 的准确率也不难。但是，如果想要准确率远远高于这个区间，建立机器学习的算法就很有帮助了。让我们来使用知名度最高的算法之一，*向量机*（support vector machine，简记 *SVM*）。如果你对向量机不太熟悉，不用担心，我们不需要关心向量机的具体工作细节。我们将使用一个叫 scikit-learn的Python库。它提供了SVM著名的C语言库LIBSVM的一个简单的Python界面。

如果我们使用默认的设置来跑scikit-learn的SVN分类器，对于10,000张测试图片，大概能猜对9,435张。（代码可以在 这里找到。）这相对于根据图片暗度值来猜，显然已经有很大的进步。确实，这说明SVM的表现相对于神经网络大致相当，仅仅稍差一点点。在后面的章节，我们将引入新的手段让神经网络的表现可以远远超过SVM。

然而这并不是故事的全部。对于scikit-learn向量机，10,000张图片中猜对9,435的识别率是对于默认设置而言的。SVM中有不少可以调节的参数，我们可以寻找那些可以让SVM表现更好的参数值。这里我就不具体去找了，有兴趣的可以参考 Andreas Mueller 的 这篇博客。Mueller 说明了通过优化SVM的参数值，有可能让其准确率超过98.5%。话句话说，参数合适的SVM在70个数字中只会认错1个。这个表现已经非常出色了！我们的神经网络可以做得更好吗？

事实上，确实可以更好。现在，对于解决MNIST问题，一个良好设计的神经网络，其表现超过所有其他方法，包括向量机。(机器学习社区tensorfly.cn原创, qq群号472113439) 目前（2013）的记录是10,000张图片中认对9,979张。这个工作是由 Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, 和 Rob Fergus 实现的。后面我们会见到他们用过的大部分技术手段。对于那种程度的准确率，其表现已经和人眼相当了，某种意义上甚至比人眼更高了，因为 MNIST 中的不少图片对于人眼来说也很难认对，例如:



我相信你会同意这些图确实很难认！注意到MNIST数据集中还有这些难认的图片存在，神经网络居然可以精准识别10,000张除去21张以外的其他所有图片，这确实是非常出色的。编程的时候，我们通常会认为要解决类似于识别MNIST数字这样的复杂问题，需要一个复杂的算法。但即便是刚提到的Wan *et al* 文献，他们所使用的神经网络算法也是相当简单的。所有的复杂性都从训练数据中自动习得。某种意义上说，从我们的结果和其他更复杂文献中的结果中，对于许多问题，我们有这样的结论：

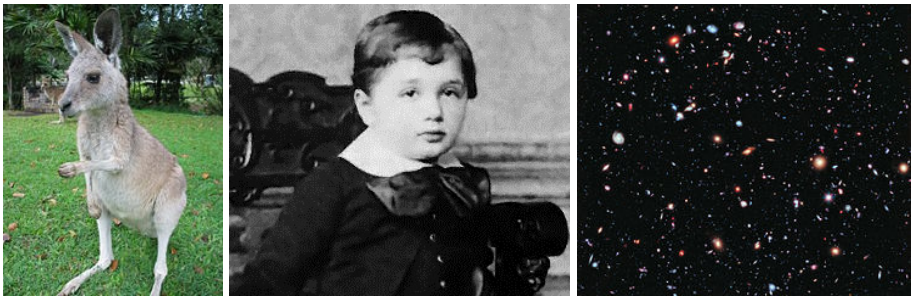
复杂的算法 ≤ 简单的学习算法 + 好的训练数据

通往深度学习（deep learning）

尽管神经网络的优异表现让人印象深刻，但这种优异变现却似乎是神秘难解的。神经网络中的权重和偏移是自动发现的。而这意味着我们没法对网络怎样工作给出一个直截了当的解释。对于分类手写文字的网络，我们可以找到某种方法去理解其工作原理吗？如果知道了这些原理，我们可以让神经网络表现得更好吗？

为了让问题表现得更尖锐，假设再过几十年之后，神经网络最终实现了真正意义的人工智能（AI），我们就最终理解那样的智能神经网络了吗？也许这个神经网络依然对于我们来说是个“黑匣子”，其中的权重和偏移我们都没法理解，因为这些参数都是通过学习自动获得的。早年研究AI的人员希望通过努力创建某种人工智能来帮助人们理解人类“智能”的原理或者人脑的工作方式。然而，最终结果可能是，我们最终对人脑和人工智能的工作方式都理解不了！

为了强调这些问题，让我们回到本章一开始对人工神经元所做的解释，即，它是一种衡量凭据重要性（weighing evidence）的手段。假设我们利用它来决定一张图是否是人脸：



Credits: 1. Ester Inbar. 2. Unknown. 3. NASA, ESA, G. Illingworth, D. Magee, and P. Oesch (University of California, Santa Cruz), R. Bouwens (Leiden University), and the HUDF09 Team. Click on the images for more details.

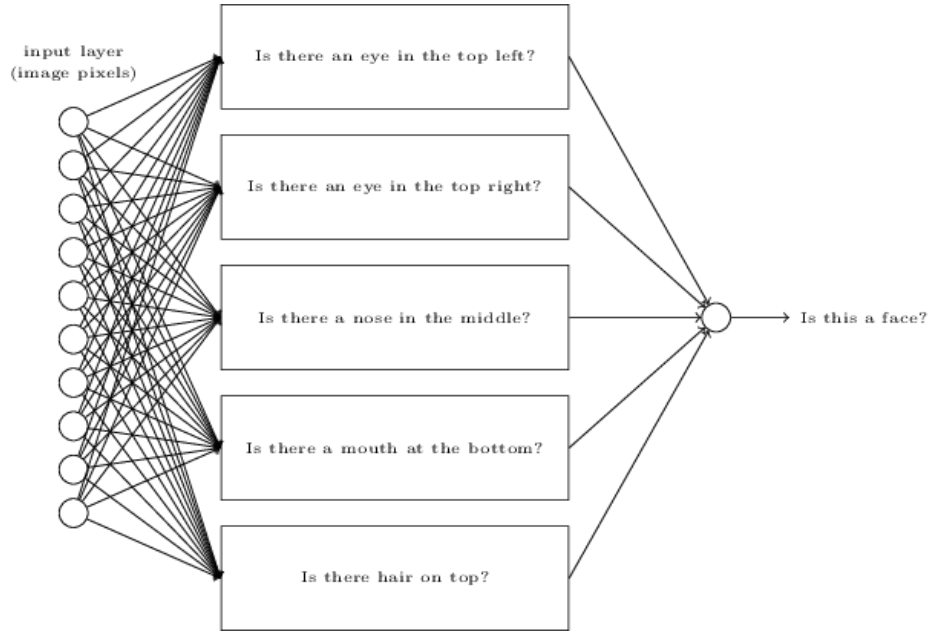
我们可以采用识别手写数字一样的方法来处理这个问题，即，将图片的像素值作为神经网络的输入，而将某个神经元的最终输出作为结果来表示“是的，这是一张脸” 或者 “不是，这不是人脸”。

假设让我们来做这件事，但是不采用学习算法。取而代之，我们将试图选择恰当的权重和偏移参数，手工设计一个网络。我们具体将怎么办呢？让我们暂时将整个神经网络都忘掉，一种启发性的办法是我们可以将这个问题细化为以下“子问题”：是不是有个眼睛在图片的左上方？是不是有另一个眼睛在图片的右上方？是不是有个鼻子在中间？是不是有张嘴在中下方？是不是有头发在顶上？如此等等。

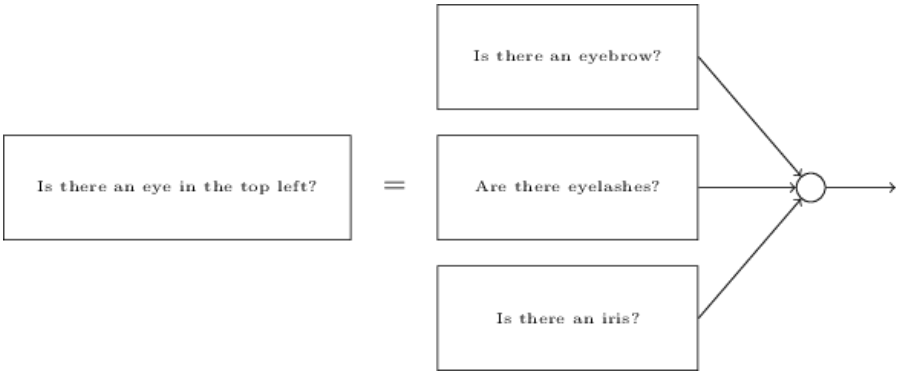
如果这些问题的许多答案都是 “是的”，或者仅仅是 “很可能是”，那么我们就可以得出结论：这张图片很可能是人脸。与此相反，如果这些问题的绝大多数答案都是“不是”，那么这张图片就很可能不是人脸。

当然，这仅仅是种大体上的启发性方法，并且也有很多缺陷。比如说，这个人也许是个秃子，因此没有头发。也许我们只能看到脸的一

部分，或者是从某个特殊视角看到的脸，因此脸部的某些特征可能是模糊的。不过，这种启发性的思路依然暗示着：如果我们可以用神经网络来解决这些“子问题”，那么我们也许可以通过将解决这些“子问题”的网络合并起来，来建立一个可以实现人脸识别的神经网络。这里有一种可能的网络结构，其中方框标记的是解决“子问题”的网络。注意：我们并不是真打算用这个方法来解决人脸识别问题，而是将其作为一种对理解神经网络工作方式建立某种直观印象和感觉的手段。下面是网络结构：



这些子网络似乎也应该可以进一步细化。假设我们来考虑这个问题：“是不是有个眼睛在图片的左上方？”这个问题可以被进一步细化为如下“子问题”：“是不是有眉毛？”；“是不是有睫毛？”；“是不是有瞳孔？”；如此等等。当然，这些问题其实应该也包含相关的位置信息，例如：“眉毛是在左上方，并且在瞳孔上方吗？”。不过，为了简化表述，我们认为对于“是不是有个眼睛在图片的左上方？”这个问题，我们可以进一步细化为：



这些问题当然可以通过多层结构（multiple layers）被一步一步更加细化。直到最终，我们要通过子网络来解决的问题如此简单，以至于可以在像素级别上很容易地回答它们。比如说，这些问题可能是：某种

很简单的形状是否出现于图片某个特定的位置上。这些问题可以由直接与图片像素相连接的神经元来回答。

最终的结果是，神经网络通过一系列的“多层结构”将一个复杂的问题：“图片上有没有人脸？”逐步简化为在像素级别就可以轻易回答的简单问题。其中靠前的结构层用来回答关于输入图片的非常简单而具体的问题，靠后的结构层用来回答较复杂而抽象的问题。这种多层结构，即有两个或更多“隐藏层”（hidden lyaers）的网络，被称为*深度神经网络*。

当然，我前面并没有说该怎样将一个问题逐渐一步一步细化为“子网络”。并且，通过手工设计来决定网络中的权重和偏移显然行不通的。因此，我们将通过训练数据，采用学习算法来自动决定网络中权重和偏移参数，当然也决定了问题的概念层次结构。在上世纪八九十年代，研究者们试图用“随机梯度下降法” 和 “逆传播”（backpropagation）算法来训练深度神经网络。不幸的是，除了某些特殊的结构，他们没有成功。这些网络可以学习，但是非常之慢，以至于没有实用性。

2006年以来，一系列的新技术被发展出来，让深度神经网络的学习成为可能。这些深度学习的技术同样是基于“随机梯度下降法” 和 “逆传播”，但是引入了一些新的办法。这些新技术让训练更深（也更大）的神经网络成为现实——人们现在一般可以训练有5到10个“隐藏层”的神经网络。并且对于许多问题，这些深度网络的表现要远远好于浅的网络，即，只有1个“隐藏层”的网络。其中的原因，当然是因为深度网络可以建立起一个复杂的概念层次结构。这有点像编程语言通过模块化设计和抽象出类来实现一个复杂的计算机程序。深度网络和潜网络的区别有点像一种可以调用函数的编程语言和不能调用函数的编程语言之间的区别。在神经网络中，抽象方式可能和编程语言有所不同，但其重要性确是一样的。