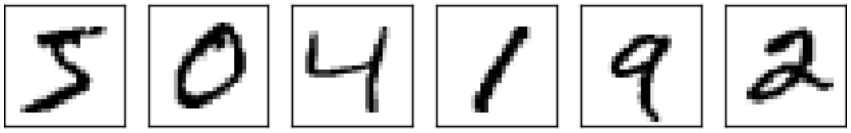
梯度下降

现在，我们有了一个神经网络的设计，那么这个神经网络是如何学习识别图像中的数字呢？第一件事情，我们需要一个数据集来进行我们的实验，使用这个数据集来训练我们的神经网络，让他能够学习去识别手写数字，这个数据集被称为训练集。我们将使用MNIST数据集，它包含有上万个手写数字的扫面图像，MNIST这个名字的由来是因为他是两个NIST提供的数据集的修改的子集，NIST是 united states national institute of standards and technology.下面是一些MNIST的样本：



你可能会发现了，这些数字实际上就是我们在上文中展示的数据。不过请放心，在最后我们用来测试神经网络的时候绝对不会使用和训练数据有重叠的数据的。

MNIST数据由两部分组成，第一部分包含有60000个图片是训练集。这些图片是扫描了250个人的手写数字，其中，一半的人在US Census Bureau工作，另一半是高中生。这些图片都是28*28个像素点的灰度图片。第二部分是10000个测试图片，同样，也是28*28像素的灰度图片。我们会使用测试集来验证我们的神经网络的对于手写数字是别的好坏。为了对神经网络的表现做出一个好的测试，测试数据的来源是250个和写训练数据不一样的人（不过还是两组人一组是census bureau的员工还有高中生）。这可以让我们相信我们的系统可以识别出手写数字，而不是仅仅学会了识别这几个人写的数字。

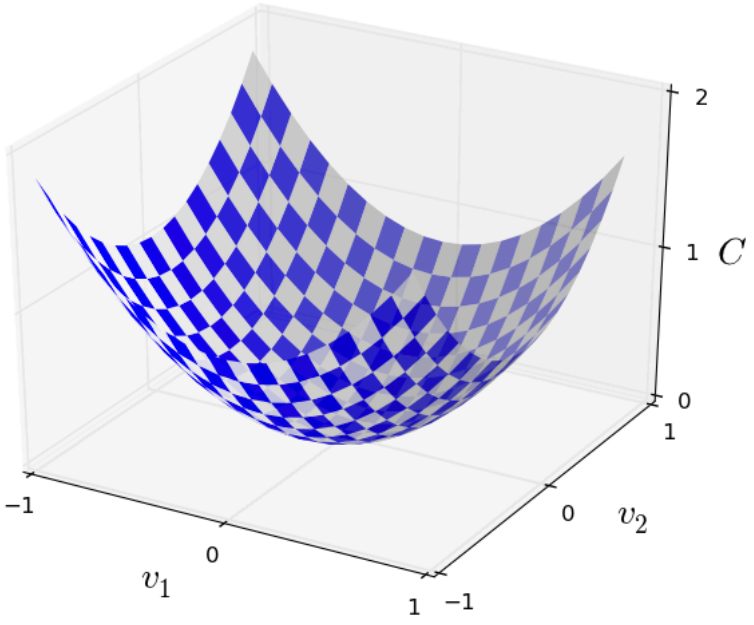
我们将使用x来标记训练的输入数据，把每一个训练数据的输入x看做一个28*28=784维的向量是很方便的。向量中的每一项都代表图像中的每一个像素点的灰度。我们将期望的正确的输出标记为$y = y(x)$, y是一个10维的向量。举个栗子对于一个特定的训练图像，x，画了一个6，那么输出$y(x) = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)^T$就是我们期望从这个网络得到的输出。其中的T表示转置，就是把一个行向量变成一个列向量。

我们所期望的是有一个算法可以让我们找到合适的权重和偏移量，使得网络对于每一个训练输入x都可以有近似于y(x)的输出。为了衡量我们的网络的效果（这里就是识别手写数字的准确程度），我们定义了一个代价函数（有些时候会被称为缺失函数或者公正函数，我们在本书中称其为代价函数，但是你需要知道别人不一定这么叫，因为这个东西经常在一些paper中出现，但是名字可能不尽相同）：

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

其中，w表示网络中所有权重的集合，b是所有的偏移量，n是训练集的个数，a是当网络中的输入为x时的输出向量，求和是在整个训练集的数据域上。当然了，我们的输出a，依赖于x，w和b，但是为了让我们的描述更加的简洁，在这里我们就不明确的写出其中的依赖关系了。符号$\|v\|$表示一个向量v的模。我们把C称为二次代价函数，或者我们会称之为均方误差（mean squared error）或者缩写成MSE。我们再来看一下这个代价函数的函数形式就会发现，c(w,b)是一个非负的值，因为，其中的每一项——求和的每一项||y(x)-a||的平方一定是一个非负的值。而且，当对于每个输入x的输出值a，越来越接近于期望值y(x)的时候我们的代价函数会逼近于0。所以如果我们的学习算法可以通过训练来学习权值和偏移量，使得C(w,b)近乎于零，那我们的训练算法就是一个好的算法。从另一个方面来讲，如果我们的C(w,b)的值很大，那么我们的训练算法就不是很好，也就意味着a和我们对于输入x的期望输出y(x)相差很大。这时我们训练模型的算法就是通过调整权重和偏移量来减小代价函数C(w,b)的值。换句话说，我们想要找到一个权重和偏移量的集合，使得代价函数的值尽可能的小，我们将通过梯度下降算法来完成我们的目标。

现在，假设我们在尝试最小化代价函数，$C(v)$。我们假设，我们的函数可以是任意的实数域上的多元函数$v = v1, v2, v3, \ldots\ldots$要注意，这里，我们把w,b用v来表示，再次强调，这个函数可以是任意形式的，在这里，我们忽略掉神经网络这个上下文。为了得到最小的C(v)，把函数C假设成一个只有两个变量v1,v2的函数，然后画出来他的图 像，来帮助我们理



解。

我们想要找到，一个点C在这个点可以得到全局最小值。当然了，对于上面那个图像而言，我们用眼就能找到最小值。仅凭直觉，我们就可以找到一个简单函数的最小值。但是，对于一个函数C,可能具有复杂的函数形式，有很多的变量，直接找到最小值，就是不可能的了。

解决这个问题的一个方法就是使用微积分来分析找到最小值。我们可以通过计算偏导，然后试着使用偏导来求出C在什么地方可以达到极值。当C的变量个数不多的时候，多数情况下，我们还是可以做到这一点的，但是，当我们有多个变量的时候这就变成了一场，噩梦。并且，对于神经网络，我们总是希可以有更多的变量——那些大的神经网络的代价函数，依赖于十亿多组权重和偏移量。微积分绝对是不适用的。

（目前为止，我们只讨论了拥有两个变量的代价函数，但是，让我们回过头来，到上面两段的时候，说“hey，如果我们的代价函数有多于两个变量呢？”对于这种情况，我们只能说，很抱歉。 不过，请相信，当我说想象C，这个代价函数只有两个参数，这对于我们的理解是很有帮助的。 It just happens that sometimes that picture breaksdown, and the last two paragraphs were dealing with such breakdowns。（当函数图像发生断点，并且两个子图在断点附近的时候，使用微积分变得不是很好分析了。） 使用函数图像对于我们在数学上的思考是很有帮助的，我们要学会找到最合适使用图像来理解的地方。）

好了，现在我们会发现，使用微积分好像不太好用。幸运的是，我们还有别的数学工具，而且，在这个问题上，我们的新方法是很有效的。首先，我们来想象一下，将我们的函数当做一个山谷。如果你觉得有点怪，看一眼上面的函数图像。然后，我们假设有个球，从山谷的斜坡上滚下来了。我们的生活经验会告诉我们，那个球会一路滚到山谷的底部。那么我们是不是可以模仿这种行为来找到函数的最小值呢。我们首先在图像上随机找一个起点（在滚球的时候我们假设是随机找个位置放球），然后模拟球滚动到谷底的行为。我们可移动过计算代价函数的偏导来简单地模拟这种行为（有些情况下我们需要二阶偏导），这些偏导可以提供我们找到局部“尖点”的条件。然后我们的球就可以开始向下滚动了。

根据我们刚刚说的那些，你可能会认为接下来就应该通过牛顿方程来求解球的行为——计算重力啊，摩擦力之类的。但是，实际上，我们不会那么的，，严肃，我们的目的仅仅是去找到代价函数的最小值，而不是构建一个严格的遵循物理定律的世界。我们需要站在球的角度上来思考整个问题，而不是我们人的角度。所以，我们不需要去考虑那些复杂的，甚至有点混乱的物理世界，让我们以一种更加简单的角度来看整个事情：我们构建一个单纯的世界，自己来定义物理规则，规定这个球要怎么去滚到底部。那么，我们会选择怎样的行为，规则，来保证，这个球始终会到达底部。

为了让这个问题变得更加简单，清晰，我们假设，这个球，在v_1方向上滚动Δv_1这么一小段，然后在v_2方向上运动了Δv_2的距离，那么在代价函数上用微积分的方式表达出来就是

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

我们想要找到的是，使ΔC为负的Δv_1和Δv_2，我们会选择他们作为前进方向,以使小球向谷底滚动。为了搞明白怎么选择，我们需要做一些概念上的定义，定义方向v上的变化为$\Delta v \equiv (\Delta v_1, \Delta v_2)^T$，， 这里T就不用多做解释了吧，同样表示的是转置的意思。同时我们要定义一个叫做梯度的概念，代价函数C的梯度$\left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$是一个由偏导构成的向量我们把这个向量记做$\nabla C$：

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

稍后，我们会用Δv和梯度∇C改写ΔC，不过，在这之前我们还是要对于”梯度”这个概念做一些解释，说明。首先当我们遇到∇C这个符号的时候，大家应该都会对于∇感到好奇。到底这个符号有什么意义?实际上，这仅仅是一个数学记号，表示我们在上面定义的那个向量,所以，就这种观点而言，∇仅仅是符号的一部分，总之，我们要知道的就是“∇C指的是梯度向量”。其实,更进一步去看，∇C在某些情况下有其独立的数学意义，不过在这里，我们并不需要知道。

基于以上的定义，我们可以将表达式(7)中的ΔC改写:

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

这个方程也有助于我们去理解∇C为什么被称为梯度向量：∇C和C中表示方向的v密切相关，所以我们形象的将其称之为梯度。但是，这个方程想要表达的是，让我们明白如何选择Δv来让ΔC变成负的。尤其是，假设我们选择：

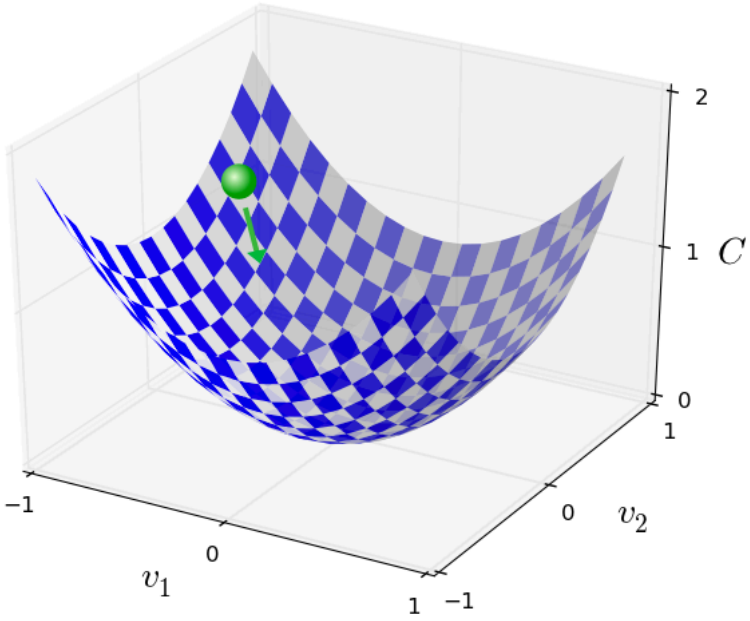
$$\Delta v = -\eta \nabla C, \tag{10}$$

这里η是一个很小的正数（被称为学习速度）。结合方程（9）我们就有 $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$因为$\|\nabla C\|^2$大于等于0，所以这就保证了$\Delta C$是一个非正数,也就是说如果我们根据方程10来对v进行改变，那么C的值必然是递减的，绝不会增加（当然，这个要在方程9的约束下成立）。这正是我们想要的属性啊。所以，我们会把方程10定义为我们梯度下降算法中小球的“行动规则”。也就是说，我们会使用方程10来计算Δv的值，然后我们就开始移动我们的小球，从v移动到v’：

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

接着，我们使用这个规则进行下一次移动。如果我们一直这么一遍一遍的做，我们就会一直减少C的值——按照我们希望的那样——到达全局最小值。

总结一下，梯度下降算法的工作就是不断地计算梯度∇C，然后反向移动，沿着山谷（函数图像）的斜坡“下滑”，我们可以将其可视化如下：



不过，我们要记得这个梯度下降是在我们假设的物理规则下作用的而不是真正的物理运动。实际情况下，我们的小球如果具有动能的话它可能穿过这个斜坡甚至上坡。在仅有重力和摩擦力的情况下，我们的小球才会滚到谷底。相比于实际情况，我们选择的Δv仅仅是表示“滚下去，现在就滚”。不过这依然是一个找到最小值的好方法。

为了保证我们的梯度下降可以正确的工作，我们需要选择一个足够好的学习速度。它要足够的小才能让方程9达到足够的近似度。如果不是，我们可能会让$\Delta C > 0$，这肯定是不好的。同时我们又不能让学习速度η太小了，如果我们将学习速度变得太小了，么ΔC就会变得非常小，因此，整个梯度下降就会变得非常缓慢。通常情况下，η有很多选择的，可以让我们的方程足够的近似，同时也不至于让学习过程变得太慢，稍后我们会看到如何去选择η。（做个通俗点的比喻，这个时候用小球可能还有点不够通俗，我们假设现在是人要下山了，那么学习速度其实就是我们下山的步伐，如果我们的步伐过大那么我们很有可能就迈过最低点了，同样的，如果我们的步伐太小，那么我们的下山速度就会非常非常的慢了。）

我们已经展示过梯度下降算法在我们的代价函数C只有两个参数的时候是有效地。但是，事实上当C有更多的参数时梯度下降依然是很有效的。假设C是一个具有m个参数（$v_1, v_2, v_3, , , v_m$）的函数。那么，C上由$\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$引起的变化$\Delta C$就是：

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

这里的梯度∇C就变成了：

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

和两个参数的时候一样，我们可以选择：

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

来保证我们的12中的ΔC是一个负值。这就让我们可以通过这个梯度来不断地改变位置来找到C的最小值，即使我们的代价函数C拥有多个变量，这个关于位置的更新规则还是一样的：

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

你可以把上面的规则视为梯度下降的算法，他给了我们一种不断去改变小球位置的方法，让我们可以找到函数的最小值。不过，这个规则有的时候会失效——偶尔会有一些问题让我们不能找到全局最小值，比如只能到达局部最优，不过，这个问题我们稍后再讨论吧。

总之，通常情况下，梯度下降是很有效的，在神经网络中，我们会看到这是一个对于我们寻找代价函数最小值的有效方法，以此来帮助网络学习。

确实，对于梯度下降的梯度选择还是有可能存在更优的方案。假设我们要从当前位置开始做一个ΔV的位移，来尽可能的减小代价函数C的值。这就相当于找到最小的$\Delta C \approx \nabla C \cdot \Delta v$。我们会限制移动的距离$\|\Delta v\| = \epsilon, \epsilon$的值要大于零，并且要固定。换句话说，我们希望，每次移动一小步，并且步长一致，并且我们想要通过我们的移动，尽可能的减小C的值。可以证明，当$\Delta v = -\eta \nabla C$的时候，可以取得最小的$\nabla C \cdot \Delta v$，其中$\eta = -\epsilon / \|\Delta C\|$是由步长$\|\Delta v\| = eta$限制。所以，梯度下降算法可以被看做一种通过在C下降（坡度最大）的方向上一小步的移动来减少C的值得算法。

练习

证明上一段的断言。提示柯西施瓦兹不等式

我们已经证明了，在C有两个或者两个以上的变量时梯度下降是有效地，但是当C只有一个变量的时候呢？你能否给出一个几何证明，看他在一维空间下发生了什么。

人们给出了很多梯度下降的变种，包括模拟真实的小球行为的。这些变种有一些优点，但是，都存在一个缺点，一个主要的缺点：他们都需要去计算代价函数C的二阶偏导，这回让情况变得相当的复杂，计算的消耗变得很大。接下来我们要看一下为什么这会导致大量的计算，假设，我们想要计算二阶偏导$\partial^2 C / \partial v_j \partial v_k$，如果我们的参数V的个数有百万级别的，那么我们需要trillion（million 的三次方）次对于二阶偏导的计算（实际上应该是二分之一，因为$\partial^2 C / \partial v_j \partial v_k = \partial^2 C / \partial v_k \partial v_j$，但是，你懂得，数量级在那搁着）。这是相当大的消耗（这些计算）。所以，虽然有很多技巧可以让我们避免这样的问题，然后找到一种梯度下降的替代算法是很有意义的。但是在本书中我们还是使用梯度下降算法作为我们对神经网络学习的主要算法。

我们如何将梯度下降算法应用在神经网络中呢，方法其实就是用梯度下降找到可以使得方程（6）中的代价函数值尽可能小的权重W和偏移量b的组合。为了展示这是怎么工作的，我们要来回顾一下梯度下降算法中的一些规则，这里我们就要用权重和偏移量来替代上面的变量V。换句话说，我们的“位置”现在是通过偏移量和权重决定的，我们的梯度向量∇C相当于由$\partial C/\partial w_k, \partial C/\partial b_l$组成。这个时候我们换种方式，用这些组件来表达梯度下降的状态更新方程，我们就会得到：

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

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

通过不停地适用更新规则，我们就可以“让小球一直滚下去”，以期找到一个代价函数的最小值。换句话说，这些条件，更新规则，可以用在我们的神经网络学习中。但是，在应用这些规则的时候，还是存在很多问题和挑战的，在稍后的章节中，我们会详细介绍，说明这些问题。现在我们让我们仅仅对于一个问题进行说明。为了更好地解释这个问题，我们要回过头来看一下我们的二次代价方程（6）。 $C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$ 要注意，我们的代价函数具有这样的形式： $C = \frac{1}{n} \sum_x C_x$ 这其实就是对于所有的独立训练数据的代价$C_x \equiv \frac{\|y(x)-a\|^2}{2}$的均值。实际上，为了计算梯度$\nabla C$我们需要分别的计算 ∇C_x对于每一次独立的输入x，然后计算他们的平均值，$\nabla C = \frac{1}{n} \sum_x \nabla C_x$，不幸的是，当训练的输入数量非常大时，这就需要相当长的时间，学习就会变得很慢。

有一种叫做随机梯度下降（stochastic gradient descent）的方法可以用来加速学习过程。这个方法是通过对于部分输入样本的梯度∇C_x进行计算来估计梯度∇C。通过对这些小样本的梯度求均值，我们可以很快地对真正的梯度∇C有一个良好的估计，并且这对梯度下降的加速很有帮助，也就加速了我们的学习过程。

为了让这些想法变得更加的清晰，严谨，随机梯度下降通过产生一个小的随机数m，然后随机选取m个训练集的输入。我们会把这些随机输入标记为$X_1, X_2, \ldots, X_m$，称他们为mini-batch(一小批)。但是要保证提供的样本数m要能够满足他们的∇C_{X_j}的平均值可以大概近似于所有的∇C的平均值，用公式表示就是：

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

这里，第二项表示所有训练集的输入的梯度均值。 简化一下，我们就能得到：

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

也就意味着我们可以通过随机选择的小批数据来估算全部的梯度。

下面，我们把这个想法和神经网络的学习明确的连接起来，假设w_k和b_l表示神经网络中的权重和偏移量。然后随机梯度下降通过从所有的训练集的输入中随机选择出小批数据，并且对他们（小批数据）训练，

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

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

上式中，求和是对于当前的小批数据的输入X_j的求和。然后我们再随机的选择另一个小批数据，直到我们用尽了所有的训练集，我们称这完成了一次训练的迭代(epoch)，这时，我们开始一次新的迭代。

要说一下的是，关于调整代价函数和mini-batch来更新权重和偏移量有很多不同的想法，在方程（6）（二次代价函数的定义）中我们通过因子1/n来调节整个代价函数。人们有的时候回忽略掉这个因子，仅仅是计算独立训练的和而不是平均值。这在训练样本的大小未知的时候是很有帮助的。也就是说有时候我们的训练集会在训练的时候不断的增加。同样的我们的mini-batch的更新的规则（20），（21）有的时候也会省去1/m这一项。在概念上来看，这个会造成一些不同之处，因为这相当于重新调整了我们的学习速率η。在对于不同的工作进行分析的时候，这一点还是值得注意的。

我们可以把随机梯度下降当做一次民主投票：相对于对在所有的数据上进行梯度下降，在小数据集上进行梯度下降要简单得多，就像展开一次小范围的民意调查要比一次公开选举简单得多一样。举个栗子，如果我们的训练集的大小为n=60000，比如MNIST，选择一个小批数据的大小m=10，这就意味着我们可以通过对于梯度的估计得到6000倍的速度的提升。当然了，这个估计未必准确-这里会有一些统计误差，波动-但是也没必要那么准确：我们仅仅是关心找到一个梯度，这个梯度，可以用来帮助减少代价C，这就意味着我们不必去精确地计算梯度。在实际应用中，随机梯度下降是学习神经网络时常用的并且很有用的做法，并且，这也是我们本书中很多学习方法的基础。

练习

梯度下降中一个比较极端的做法是我们选择的小批数据的大小为1，也就是说，每得到一个训练集的输入x，我们都会通过规则$w_k \rightarrow w_k' = w_k - \eta \partial C_x/\partial w_k$和$b_l \rightarrow b_l' = b_l - \eta \partial C_x/\partial b_l$来更新我们的权重和偏移量。然后我们选择新的输入数据再来更新我们的权重和偏移量，这样不断的重复。这个过程被称为在线学习，或者增量学习(on-line learning，incremental learning)。在在线学习中，神经网络从每一个独立的训练输入中进行学习（就像人类一样）。对比在线学习和随机梯度下降（比如小批的个数为20）举出一个优点和缺点：

1）批量梯度下降---最小化所有训练样本的损失函数，使得最终求解的是全局的最优解，即求解的参数是使得风险函数最小。 2）随机梯度下降---最小化每小批样本的损失函数，虽然不是每次迭代得到的损失函数都向着全局最优方向， 但是大的整体的方向是向全局最优解的，最终的结果往往是在全局最优解附近。 3）在线学习---最小化每个样本的损失函数，比随机梯度下降的收敛更快，但是并不能保证收敛到全局最优解，往往精度较低。

最后让我们通过讨论一个刚刚接触梯度下降的人都会有的困惑来结束这一节。在神经网络中，代价函数C是一个关于权重和偏移量的多元函数，在一些情况下，就是高维空间中的一个超平面。有的人会想“Hey，我必须要看一下这到底是什么样的”，然后，他们就开始困惑，“我都无法想象四维的空间是什么样，更别提五维（或者五百万维）了”。是因为他们缺乏一些特殊的能力？一些只有数学天才才具有的能力么？当然不是了，就连绝大多数的专业数学家也无法想象四维空间到底是什么样子。他们会用一些别的技巧，来推测高维空间下会发生什么，这也是我们在做的：使用代数的方法，而不是去看他到底什么样，通过计算ΔC来搞清楚怎么移动可以减少C。那些擅长于思考高维空间的人有着各种不同的方法，我们的代数表示仅仅是一个例子。 这些方法在我们习惯于生活在三维的空间中并不是很简单，但是当你建立起来这些感觉以后，你可以在高维空间中获得不错的体验。我们不会在这里对这些东西及逆行深入的讨论，但是如果你对这些东西感兴趣这里有一些不错的讨论你可以去看一下那些专业的数学家怎么思考高维空间下发生的事情。当然，他们中有一些想法是非常复杂的，不过大多数的还是简单直观，可以被大多数人所接受的。

实现我们用来分类手写数字的网络

现在，让我们来写程序实现识别手写数字吧，使用随机梯度下降方法和MNIST训练集。我们会用一个短小的python代码来实现仅用74行代码就可以了。首先，我们需要下载MNIST数据集，如果你是一个git用户，那么显然，你可以通过clone 本书的仓库来获得。

`git clone https://github.com/mnielsen/neural-networks-and-deep-learning.git` 如果你不使用git的话，那么你也可以下载。

要说明一下，在之前，描述MNIST数据及的时候，我们说他被切分成60000个训练图片和10000个测试图片。这个是MNIST的官方描述，实际上我们会对数据集进行小小的改变。我们将保留测试集不变，但是，对于60000个训练图片，我们把它们切分成50000个用来训练网络的图片和，10000个验证集，在这章中，我们不会使用验证集的数据，不过在往后的章节中，我们会使用这个验证集，来帮助我们设置神经网络中的超参数（hyper-parameters）--例如学习速率等一些不可以通过神经网络的训练来学习的数据时是很有用的。虽然验证集并不是原始的MNIST的划分，但是很多人都在这么用，而且验证集的存在，在很多神经网络中都是有用的。所以，稍后，当我们提到MNIST训练集的时候，我们的意思是指我们所划分出来的那50000个图片，而不是原始的60000个图片。

我们还会用到numpy这个第三方的python库来进行快速的线形代数的计算，所以，你需要去安一下numpy。

再给出完整的代码清单之前，我们要来解释一下神经网络代码中的一些核心特征。整个核心是下面的叫做 `Network` 的类我们用它来表示一个神经网络，下面是我们用来初始化这个类的对象的代码：
`py class Network(object): def __init__(self, sizes): self.num_layers = len(sizes) self.sizes = sizes self.biases = [np.random.randn(y, 1`

在这个代码中，sizes是一个list表示神经网络中各层神经元的个数，如果我们希望创建一个第一层有两个神经元，第二层有3个，最后一层有一个神经元的网络，那么我们就可以把代码写成 `net = Network([2,3,1])` Network对象中的权重和偏移量会被初始化成一个随机值，通过使用numpy中的random.randn方法会生成一个符合均值为0，标准差为1的高斯分布的数。这个随机的初始化，给了我们的随机梯度下降一个起始位置。在稍后的章节中，我们会讲到更好的初始化的方法，但是现在，我们就先这么将就着用着。注意这个Network的初始化的代码，假设第一层神经元是输入层，并且对于这一层的神经元忽略掉了他们的偏移量，因为偏移量仅仅是用来调整后面的神经元的输出的。

并且，要注意，偏移量和权重以list的形式存在，list中的元素为Numpy的矩阵。比如说net，weights[1]是一个Numpy的矩阵，存储着链接第一层和第二层神经元的权重的numpy矩阵。(不是第一层和第二层，python中的索引是从0开始的)。因为net.weights[1]显得很长，所以我们还是先用w来表示矩阵，矩阵中的W*{jk}表示连接着第二层神经网络的第j个神经元和第三层神经网络的第k个神经元。这种使用j，k的表示可能看起来有点奇怪，如果我们交换j，k会不会更好？我们来看一下，其实这个 表示第三层神经元的激活向量：*

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

这么看有点复杂，分解一下，a表示第二层神经网络的的神经元的激活向量，为了得到a'我们可以通过把第二层的激活向量a乘以权重矩阵w，再加上偏移量向量b，然后将σ 函数应用在每一个$wa + b$上（这叫做σ函数的向量化）。我们可以很轻易的验证上面的方程其实是符合我们之前的那个sigmoid神经元的激活函数 $\begin{eqnarray} \frac{1}{1+\exp(-\sum*j w*j x*j-b)} \nonumber\end{eqnarray}$ 。他们的结果是一样的。

练习

写出方程22的组成形式，并证明他和方程4--sigmoid神经元的输出函数是一样的。

掌握了这些知识，计算神经网络中的sigmoid函数就很简单了： `python def sigmoid(z): return 1.0/(1.0+np.exp(-z))` 要注意输入z是一个向量，或者说是一个Numpy数组，Numpy会自动的对于这个输入向量化。

然后，我们给我们的network类增加前馈(feedforward)方法，也就是当我们给一个定输入a会的带对应的输出。（在这里，我们假设输入a是一个（n,1）的numpy数组，而不是一个（n,）的向量，这里，n是网络的输入数，如果你尝试用一个（n，）的向量作为输入那么你有可能得到一些奇怪的结果，虽然使用一个（n,）的向量会显得更加的自然，但是在这里当我们使用（n,1）的numpy的ndarray会让代码变得更加好写）

```
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
```

当然了，我们主要还是希望我们的network对象做的事情是学习。所以，我们会实现一个SGD方法，来实现随机梯度下降，下面是他的代码，可能会有一些不太好理解的地方，但是我们在列出代码后会有详细的解释。

```
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 ecaluated against the test data after each
  epoch,and partial progress printed out. This is useful for
  tracking progress,but slows things down substatially"""
  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是一个元组的列表（x，y）表示训练的输入以及期望的输出，变量epochs和mini*batch*size是你期望的训练的迭代次数和mini-batches的大小，eta表示学习速率也就是η。如果可选的参数test_data被提供的话，那么程序会在每一次的迭代完成后对网络进行评估，并且将处理进度和结果输出，这对于跟踪程序的进度是很有帮助的，但是这回降低执

行效率。

代码的执行顺序如下。在每一次迭代中，最开始，会随机的打乱训练集数据，然后把它们划分成合适的大小，这是一种简单的对于训练数据的随机采样。之后对于每一个mini－batch我们会进行一步梯度下降。这个是通过`self.update_mini_batch(mini,eta)`来完成的，这个方法会进行一次梯度下降来更新网络的权重和偏移量，不过每次梯度下降的时候仅仅使用训练集中的`mini_batch`。下面是delta*nabla*w = self.backprop(x, y)``

这里引用了一个叫做back propagation（反向传播）的算法，这是一个计算代价函数的梯度的快速的方法。所以这个update*mini*batch的工作其实很简单，就是计算mini-batch中的每一个训练样本的梯度，然后更新self.weights和self.biases。

现在，我们不会展示反向传播方法的代码，我们会在下一章中学习反向传播算法，以及反向传播算法的代码。现在，我们仅仅假设他的行为已经被我们知道了，就是返回对于训练样本x的相应的梯度。

让我们看看整个代码的样子。包括一些在上文中我们忽略的文档，注释，除了反向传播部分的代码，其他的是不需要解释说明的最重要的部分都在self.SGD和self.update*mini*batch中实现了，而这些，我们都在上文中讨论过了。Self.backprop方法会使用一些额外的函数，来帮助计算梯度，有一个sigmoid*prime,是用来计算σ函数的导数，self.cost*derivate,在这里我们不会介绍，你可以猜到大概的意思(也有可能知道详细的信息)通过看代码和文档注释，在下一章中，我们会详细的讲解这个。注意到代码其实仅仅有74行有效代码，下面的大多数都是文档注释。

```
"""
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))
```

程序究竟可以得到什么样的结果呢？让我们从载入MNIST数据集开始吧。首先我们会使用一些小程序来帮助我们完成这个任务，mnist_loader.py， 像下面写的那样，我们在python shell中运行下面的命令

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

当然，这个也可以在单独的Python程序中完成，但是在shell里面应该更加简单一些。加载完数据以后，我们会使用下面的代码来建立一个有三十个隐藏神经元的网络。 ```python

> import network net = network.Network([784, 30, 10])
> 最后，我们会使用随机梯度下降进行学习，相应的参数为30次迭代，mini-batch大小为10，学习速率$\eta = 3.0$ python net.SGD(training*data, 30, 10, 3.0, test*data=test_data)
> 要注意，这个程序会需要一些时间去运行。我们建议你可以先让程序跑着，然后回来继续阅读此书，定期的去检查结果，但是如果你比较着急的话你可以通过减少迭代次数(epo
> Epoch 0: 9129 / 10000 Epoch 1: 9295 / 10000 Epoch 2: 9348 / 10000 ... Epoch 27: 9528 / 10000 Epoch 28: 9542 / 10000 Epoch 29: 9534 / 10000 ``` 可以看到，我们的训练好的神经网络可以得到95%的正确率—95.42%在第28次迭代完成后达到最大值。作为第一次的尝试这个结果还是很激动人心的。但是我们还是要警告一下你，我们得到的结果还是会有偏差的，因为我们在初始化的时候权重和偏移量是随机的。为了得到这个结果我们是在三次试验中取得最好的那次。

让我们回到实验中来，在建立网络的时候，我们把隐藏层的节点个数改成100个，结果会怎么样呢.他会把结果提升到96%左右。至少，在这个例子里，使用更多的隐藏神经元会帮助我们改善结果。(*有人可能会得到一些很差的结果，在后面第三章有一些方法是可以避免，改善这种情况的)

当然了，为了提高准确率，我们必须去手动的去改变迭代次数，mini-batch的大小，以及学习速率。就像我们在上面提到的一样这些被称为我们的神经网络中的超参数，以此来将他们和我们的代码中可以通过学习算法得到的参数区分开来如果，我们选择了一些很糟糕的超参数，那么我们会得到一些非常糟糕的结果。比如，当我们把学习速度改成0.01那么你会得到下面这样的结果:
`Epoch 0: 1139 / 10000 Epoch 1: 1136 / 10000 Epoch 2: 1135 / 10000 ... Epoch 27: 2101 / 10000 Epoch 28: 2123 / 10000 Epoch 29: 2142 / 100`
你可以看到，我们的准确率的提升非常的缓慢，这个时候，其实我们就应该尝试着增加我们的学习速率。如果我们那么做了，那么我们的结果会得到改善，那么我们就应该尝试再次增加学习速率。(If making a change improves things, try doing more!)，当我们尝试了多次以后，比如，当我们把学习速率增加到了1.0以后(可能更好的结果是3.0),这个时候其实已经和我们之前的实验挺接近的了，这样，就算在最初的时候我们没有选择出合适的超参数，那么我们也可以通过慢慢的调整来达到好的效果。

通常而言，对于神经网络的调试是有挑战性的，尤其是有那么多的超参数需要自己去设置。假设我们用最开始的那个30个隐藏神经元的结构，但是我们的学习速度是100的话，这个时候我们有可能就走的太远了，因为学习速度太快了。
`Epoch 0: 1009 / 10000 Epoch 1: 1009 / 10000 Epoch 2: 1009 / 10000 Epoch 3: 1009 / 10000 ... Epoch 27: 982 / 10000 Epoch 28: 982 / 10000`
现在，假设我们是第一次遇到这个问题。当然通过之前的实验我们知道了是因为我们错误的将学习速率设置的太大了，正确的做法是去减少学习速度。但是如果我们是第一次遇到这个情况呢？从输出的结果中我们并不能得到太多的有帮助的信息。

我们可能不仅仅会怀疑学习速度，还有关于网络的结构啊之类的情况。我们会怀疑是不是权重和偏移量没有得到好的初始化，是不是训练集太小了，是不是迭代次数不够，或者这个结构根本不对，或者我们的学习速度太低了，太高了等等，而且当第一次遇到这种问题时，可能我们无法去确定问题出现在哪里。

我们终于意识到，调试神经网络是很难的，但是在通常情况下，其实还是有技巧去完场这件事的。你需要去学习这些技巧来获得合适的，好用的神经网络。解决的办法是构建一个启发式的程序，来选择好的超参数，以及网络结构。在本书中，我们会一直讨论这个问题，包括之前的试验中，我们对于超参数的选择。

练习

尝试创建一个仅有输入层和输出层的神经网络—拥有784个输入神经元，和10个输出神经元。使用SGD来训练神经网络。看看你得到了什么结果。

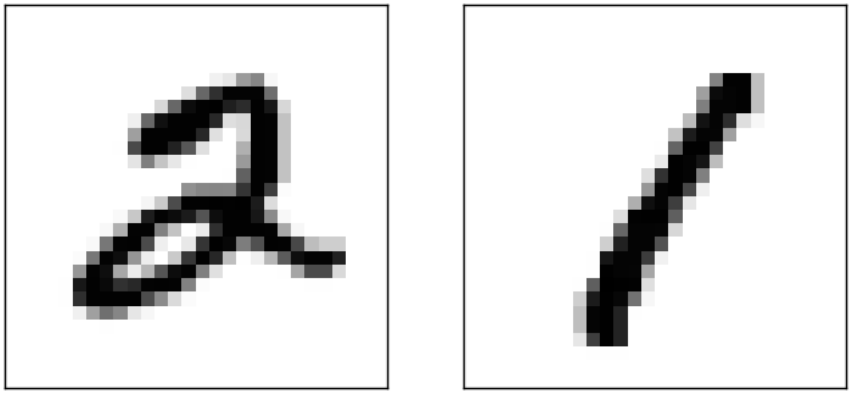
之前，我们略过了数据是如何被载入的。这个其实很直接，为了保证我们代码的完整，下面是我们的代码。被用来存储文档中描述的MNIST数据的数据结构是--是元组，和Numpy中ndarray对象的列表(如果你不是很了解ndarray，你可以把它们当做向量):

```
"""
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
```

在上面有说过我们的程序有非常好的结果。这意味着什么呢？和什么相比我们的结果是好的呢？有一些别的（非神经网络）例子作为baseline来和我们的结果比较，对于理解什么叫表现良好是很有用的。最简单的baseline其实就是随机的去猜那个数。这估计能有个十分之一的正确率，我们做的好多了。

那么有没有一些比较有意义，有难度的baseline？让我们试试下面这个简单的想法:我们看这个图片有多黑。你看，一个表示2的图片要明显的比1黑的地方多一点，仅为有更多的像素



是黑色的，就像下面那样：

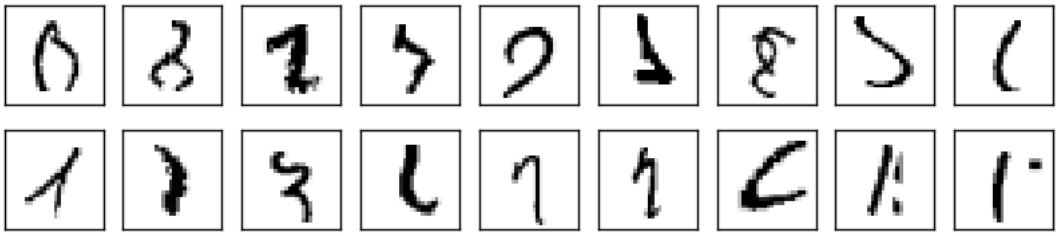
现在，我们使用训练集的数据去计算每个数的平均的黑度（黑色像素所占的比例）。当遇到新的图面，我们会计算有多黑，然后去看他和哪个数的黑度平均值最为接近。这是一个很简单的想法，而且很容易去实现。如果你感兴趣的话你可以看一下这个。虽然它很简单，但是却相对于我们瞎猜有了很大的提升，正确率在百分之二十左右。

还有很多很简单的方法可以让我们的准确率在百分之20到50之间。如果你更努力一些你也可以达到百分之五十以上。但是，如果想要达到更高的准确率，那么你就需要一些机器学习的方法来进行了。让我们试试用一个广为人知的算法支持向量机来做这个事情，如果你不熟悉svm的话，不要紧，我们并不需要深入的理解svm，我们使用一个python的第三方库，叫做scikit-learn来所这个事情，它提供了一个调用LIVSVM(c语言的svm库)的接口来实现svm。

如果我们使用scikit-learn的svm分类器并且使用它的默认参数的话，我们可以得到94%的正确率。代码这对于我们的那些简单的方法而言已经是一个巨大的提升了。确实，这意味着svm和我们的神经网络其实差不太多，就那么一点点。在后面的章节中我们会介绍一些新的方法来大幅度提升我们的神经网络的效果。

现在还没结束。我们之前的百分之94的正确率是使用的默认的svm的参数。svm还是有很多我们可以去调整的参数的，通过对这些参数的调整我们是可以改善svm的表现的，我们不会在这里细说，如果你有兴趣的话你可以去参考这里，这里有一些对于svm参数的优化可以让准确率提升到98.5%。换言之，一个调好参数的svm的识别结果是70个图片中出一个错。神经网络可以达到更好的效果么？

实际上，神经网络可以的。实际上一个良好设计的神经网络可以在mnist数据集上的识别准确率远超过其他算法，2013年的正确率记录是99.79%。在书的后面，我们会介绍很多技术。在这个水平上神经网络的表现已经很接近甚至超过人类的识别了，因为其实MNIST数据集中还是存在一部分人类都很难识别的例子的：



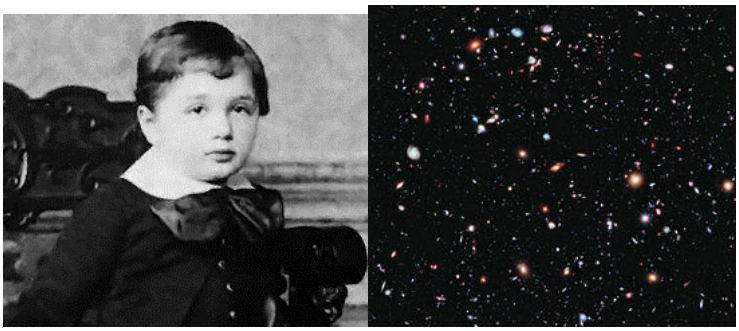
相信你会理解这有多难认。有上面一样的数据存在，神经网络能够达到这样的准确率其实是一件很出众的事情了。通常情况下呢，人们会觉得要去解决这样的复杂的问题，需要很多极为高深的算法。但是世界上神经网络仅仅用了一些很简单的算法，通过对这些算法进行小小的改变就可以解决问题了。这些错综复杂的参数，设置都被自动的通过训练数据学习到了。其实有的时候很精巧的算法取得的结果还不如简单的算法和良好的训练数据集。

转向深度学习

现在，我们的神经网络有着一个令人惊奇的表现，迷之表现。网络中的权重和偏移量可以被自动的学习。这意味着，我们并不需要去了解网络怎么做到这样的效果。可是，我们能理解神经网络分类的原则么？如果知道这些原则，我们会做得更好么？

或者把我们的问题放大来看，假设十年以后有一个神经网络领导了全世界的AI，我们能知道我们的神经网络可以有多么智能么？很有可能如果网络是一个黑盒的，我们可能无法去了解权重和偏移量到底是什么样子的，因为，我们的神经网络是自动的去学习这些东西的。在AI探索的初期，人们原本是希望可以努力建立出来一个AI，来帮助我们去理解智能背后的原则，有可能的话，甚至是可以学习到人类大脑的工作方式。但是最后的结果可能是我们根本不能理解人脑或者人工智能是如何工作的！

为了认清这些问题，让我们回到对于在本章开始时对于人工神经元的说明。假设我们想要识别下面的图片中哪一个是人脸：

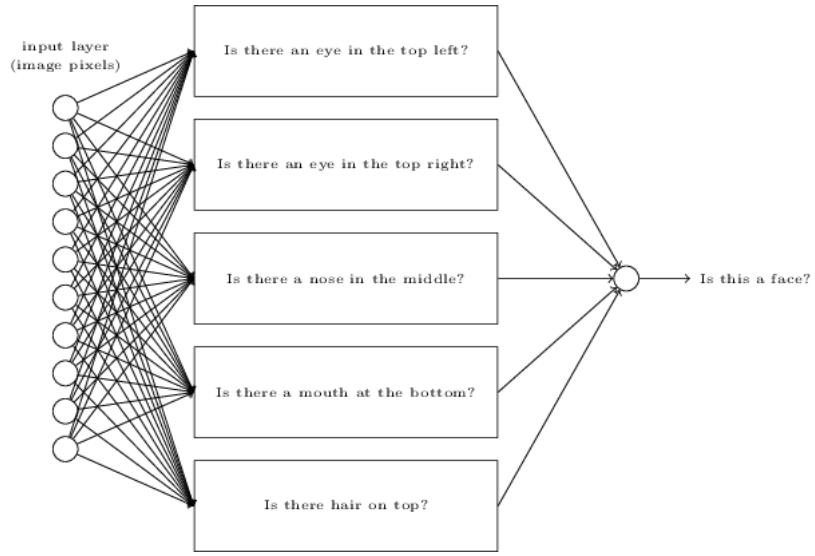


我们可以用解决手写数字分类的想法来解决这个问题——通过将图片中的像素点作为神经网络的输入，神经网络的输出是一个信号表示“是的，这是一个脸”或者不是这不是一个脸。

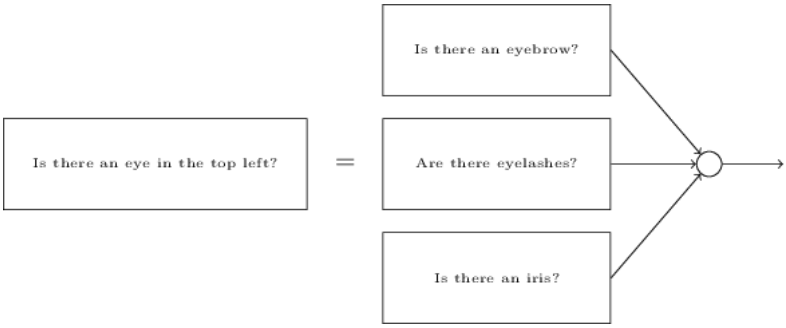
假设我们这么做了，但是我们并不使用我们上文说到的什么SGD啊什么的算法，而是我们手动的去设计这个网络，并且想办法找到权重和偏移量，我们能做到么？让我们先把整个神经网络给忘掉吧，我们将问题分解一下：图片的左上方是否有一只眼睛？右上方有么？中间有鼻子么？下方有嘴么？上面有头发么？等等的子问题。

如果我们这些问题的答案都是“yes”或者甚至是“probably yes”，那么我们就可以把这个图片当做是一个人脸了，相反的，如果问题的回答大多数都是no，那么很可能，结果就是这不是一个脸。

当然了，这仅仅是一个简单的想法，并且有着各种缺点。有可能这个人没有头发，有可能我们的图片仅仅是半张脸，所以有一些人脸的特征并不能被我们观察到，不过这个想法还是有意义的，如果我们可以通过神经网络来解决这些子问题，那么我们就可以通过组合这些解决子问题的神经网络来构建一个可以识别人脸的神经网络。下面就是一个可能的网络结构，我们用长方形来表示子网络，要注意的是这可能不是一个真正能用的神经网络，而是我们根据直觉来建立的神经网络：



对于这些子网络，其实，我们也可以进行进一步的拆分，比如那个左上角有眼睛的问题，我们可以把它分解成有眉毛么，由睫毛么，有虹膜么，有眼睑么等等等等，当然了，这些问题也可以包含位置信息，，在左上角么，不过我们简化一下表达。所以我们的子网络会有下面的结构了：



剩余的问题也可以被拆分，这样我们就有了更多的层，最后我们可以得到一系列功能非常简单的子网络，他们仅仅对于像素级别的数据进行处理，比如图片中的某个位置是否存在一个特别的形状，这些问题就可以被让某一个神经元，通过对图片上的原始像素处理来回答，

最后的结果就是一个网络被拆分成很多很多的小问题，从这个图片是一个人脸么，变成了对于像素级别的数据的问题的回答。我们就有了一个层级的网络结构，最底层是像素级别的问题，然后越往上越复杂。拥有着这样的，多层结构的(两层或者两层以上的隐藏层的)神经网络就被称之为deep neural networks。

当然了，我们没有说如何去递归的分解成子网络。因为我们会发现这样手动的去设置偏移量和权重是很不现实的。相对而言，我们还是让机器从训练数据中自动的去学习这些参数比较好。80到90年代的时候人们就曾经尝试利用随机梯度下降和反向传播来训练深度网络。不幸的是，除了几个特别的网络结构，人们并没有取得什么突破。网络可以学习，但是却非常的慢，慢到没有实用价值。

直到2006年，一系列的新的技术被建立，才使得建立一个深度神经网络变得可行。这些新的技术建立在随机梯度下降和反向传播的基础上，不过被人们引入了一些新的想法来改善。有了这些帮助，我们通常往往可以建立一个5到10层的神经网络。事实也证明，这些技术在很多情况下都比浅层的神经网络（只有一个隐藏层的）表现的好很多。原因当然是因为深度神经网络可以建立一个更复杂的概念，层次结构。就像传统的程序一样，通过抽象，模块化的设计来完成一个复杂的程序。把浅层的神经网络和深度神经网络进行比较，就像是将

编程语言和脚本进行比较一样。当然，他们之间的抽象还是有一点不同的，不过都一样的重要。