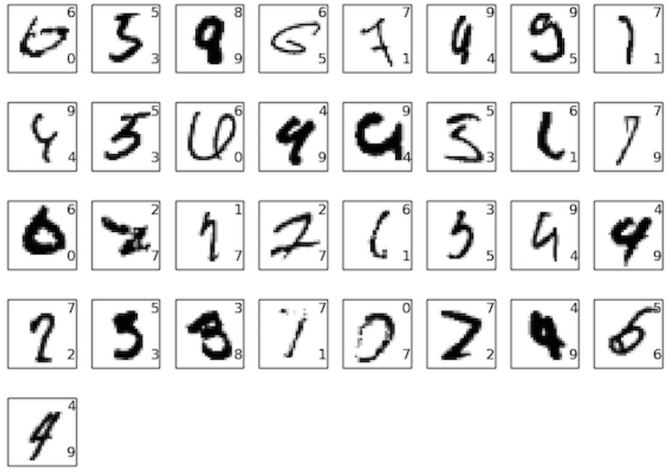
在前一章中，我们知道训练深层神经网络要比训练浅层神经网络更加困难。我们以为深层网络会比浅层网络表现的更为强大，可惜事与愿违。虽然有着种种困难，但这并不能停止我们的脚步。在本章中，我们会克服训练深层网络的困难，开发技巧并把它们应用到实际之中。我们还会了解一些最近深层网络的进展，例如它们在图像识别、语音识别等领域中的最新成果。最后，我们会讨论神经网络，以至人工智能，它们未来的发展。

本章的内容略长，为了让你有个大概的了解，我们先简单地介绍一下本章的内容。本章中不同的节的内容比较松散，你可以跳到任何你想了解的部分去阅读。

本章的主要内容是对一种广泛应用的深层网络的介绍：深层卷积网络（deep convolutional networks）。我们会用一个具体的例子，包括代码，用卷积网络去解决经典的 MNIST 手写数字识别问题：

我们首先会使用简单的浅层卷积网络来解决这个问题，然后不断地构建更加复杂的、强大的网络。同时，我们会探索许多强大的技巧：卷积（convolution）、池化（pooling）、使用 GPUs、改进的算法和 dropout（减少过拟合）、网络的整合（ensembles）等等。网络的最终表现几乎达到了人类的水平，在 $10,000$ 个 MNIST 训练图片上，我们的网络正确分类了 $9,967$ 个图片。下面展示了分类错误的 33 个图片。每个图片右上角的数字是正确的分类，右下角的数字是我们的网络给出的分类：



这些数字对于人类来说也是很难辨认的。例如上图中第三个数字，对我来说，它更像是 9 而不是所谓的 8。我们的网络也是这样认为！这种类型的“错误”也许算不上是真正的错误，我们甚至可以说它是一种值得赞赏的“错误”。我们最后对图像识别领域中一些最新的惊人进展（尤其是卷积网络）做出一些概述和总结。

对于深度学习，本章还介绍了一些其他类型的神经网络，例如循环神经网络（recurrent neural nets）和长短时记忆单元（long short-term memory units），以及这些模型是如何应用到诸如语音识别、自然语言处理等领域。我们会展望神经网络和深度学习未来的发展，包括意图驱动的用户界面（intention-driven user interfaces）和人工智能中深度学习的作用等。

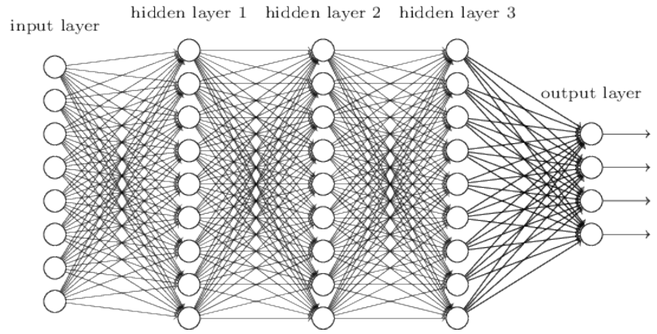
本章需要读者掌握一些例如反向传播、正则化、softmax 函数等知识。当然，阅读本章并不需要读者对前面的章节有着细致的了解。当然，如果能掌握第一章中的内容就更好了。

本章并不是最新的、最好的神经网络库函数的教学材料，我们也并不是要去训练一个超深、超强的深层网络去解决最新、最前沿的问题。本章的目的是带给读者一种隐藏在深层神经网络背后核心理念的理解，然后把它们应用到简单、容易理解的 MNIST 手写数字识别问题上。也就是说：本章并不打算介绍该领域最新的知识，而是专注于一些基础，为读者以后了解更多的内容做好准备。

介绍卷积网络

在之前的章节里，我们设计的神经网络可以在手写数字识别问题上得到一个不错的结果了：

我们所使用的网络，其相邻层之间是全部连接的。这就是说，网路中每一个神经元都与其相邻层的每一个神经元相互连接：



对于输入图片中的每一个像素，我们把其像素亮度看做是相应输入层神经元的输入值。对于一个大小为 28×28 像素的图片，这意味着我们的网络拥有 $784 (= 28 \times 28)$ 个输入神经元。我们然后训练这个网络的权值和 biases，最后网络可以对每个输入图片，输出其所代表的数字：从零到九。

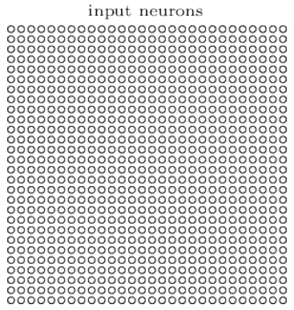
在 MNIST 数据集上，我们的网络可以达到超过98%的分类准确度。但是，使用全连接层来做数字分类，怎么说都听起来非常奇怪：这种网络结构看起来没有考虑图片的空间结构信息，无论是距离很近的像素，还是距离很远的像素，它都一视同仁。这种空间结构的信息并没有在训练数据中反应出来。有没有一种网络结构，或者一种办法，可以反映出这种空间结构呢？在本节中，我将会介绍 *卷积神经网络（convolutional neural networks)*。这种网络利用了一种特殊的结构，非常适用于分类数字。使用这种结构可以让卷积网络拥有非常快的训练速度，从而让训练一个深层的网络成为可能。现如今，深层卷积网络及其变体几乎应用在所有的图像识别问题中。

> *最早的卷积神经网络起源于上世纪70年代。但让卷积网络真正发挥实力的重要论文发表于1998年：Gradient-based learning applied to document recognition，作者是 Yann LeCun、Léon Bottou、Yoshua Bengio 和 Patrick Haffner。LeCun 对卷积网络的名称作了一个非常有趣的评论：“在卷积网络中，来自（生物）神经的灵感是很微小的，这就是我为什么把它叫做‘卷积网络’而不是‘卷积神经网络’的原因，同样地，我把节点叫做‘单元’，而不是‘神经元’。”虽然 LeCun 这样说，但卷积网络实际上使用了非常多神经网络的概念：例如反向传播、梯度下降、正则化、非线性激活

> 函数，等等。所以在本书中，我们把它看做是神经网络的一种，即”卷积神经网络“等同于”卷积网络“，”（人工）神经元“等同于”单元“。

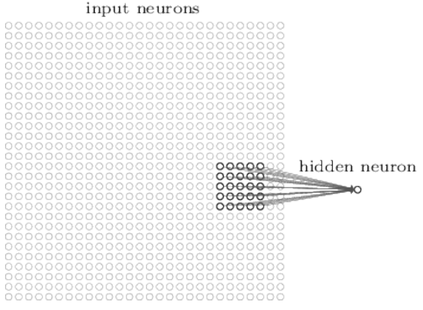
卷积神经网络有三个非常基础的概念：*局部接受域（local receptive fields）*、*共享权值（shared weights）*和 *池化（pooling）*。让我们依次介绍这三个概念。

局部接受域：在之前介绍的全连接层中，输入被表示为一个竖直排列的神经元组合。在卷积网络中，我们把输入看做是一个面积为 28×28 的神经元矩形，每个值都对应着像素亮度：



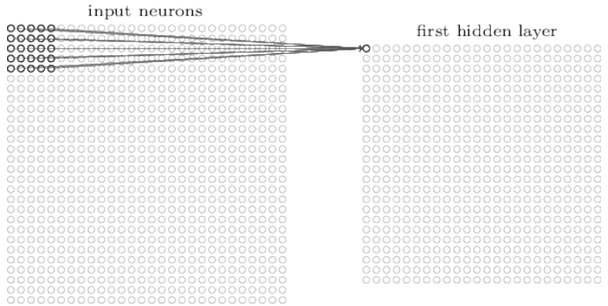
如同往常一样，我们会把输入像素连接到一个隐藏层上。但不是每个像素都去连接隐藏层中的每一个神经元，而是，我们对输入图片中一个很小的局部区域做连接。

更准确地说，第一个隐藏层中的每一个隐藏神经元，都会连接到输入神经元中的一个小的区域，例如，一个 5×5 的区域，其对应着 25 个输入像素。所以，对于某个特定的隐藏神经元，我们的连接可能会像下图展示的那样：

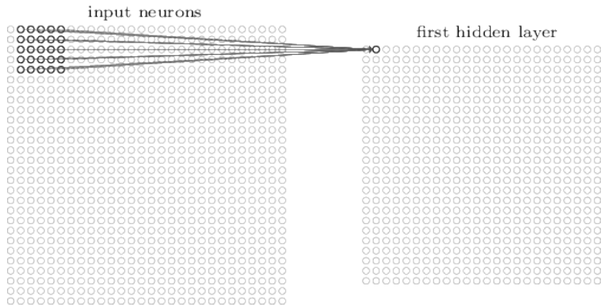


这个输入图片中的区域，称为隐藏神经元的 *局部接受域（local receptive field）*。这是输入像素中的一个小的窗口。每个连接都会学习一个权值，隐藏神经元也会学习一个整体的 bias。你可以认为这个特定的隐藏神经元在试着学习这个特定的局部接受域。

我们让局部接受域在整个输入图片上滑动。每一个局部接受域，都在第一个隐藏层中有一个对应的隐藏神经元。为了解释的具体些，让我们从输入图片的右上角开始构建一个局部接受域：



然后我们让局部接受域向右滑动一段距离（例如一个像素）：



如此反复，最终构建出整个隐藏层。如果我们的输入是像素大小为 28×28 的图片，以及大小为 5×5 的局部接受域，那么隐藏层最终会拥有 24×24 个神经元。这是因为在图片像素中的每一行（列）中，我们只能移动局部接受域 23 次。

在上面的例子中，我们每次将局部接受域移动一个像素的距离。事实上，有时候人们会使用其他大小的*步长（stride length）*。例如，我们可以让局部接受域每次向右（或者向下）移动两个像素，即步长为 2。在本章中我们会一直使用大小为 1 的步长，但是你得知道，有时候人们会使用其他大小的步长*。

> *如果我们对其他的步长设置感兴趣，我们可以使用验证集来尝试不同的步长值，最终找到效果最好的步长值。如果你对如何选择超参数感兴趣的话，可以参考本书的第三章。除了步长，你也可以使用相同的方法来设置其他的超参数，例如局部接受域的大小。一般来说，对于很大的图片来说，应该选择较大的局部接受域。

共享权值和 biases：之前我说过，每个隐藏神经元都有一个 bias，和大小为 5×5 的权值。但还有一点我没有提及：对于 24×24 个隐藏神经元，我们将会使用 *相同大小的* 权值和 bias。换句话说，对于第 j 行第 k 列的神经元，其输出是：

$$\sigma(b + \sum_{l=0}^{4} \sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}). \tag{125}$$

其中，σ 是神经激活函数——例如我们之前使用过的 sigmoid 函数。b 是 bias 的共享值。$w_{l,m}$ 是大小为 5×5 的共享权值数组。最后，我们用 $a_{x,y}$ 来表示位于 x, y 的输入激活（input activation）。

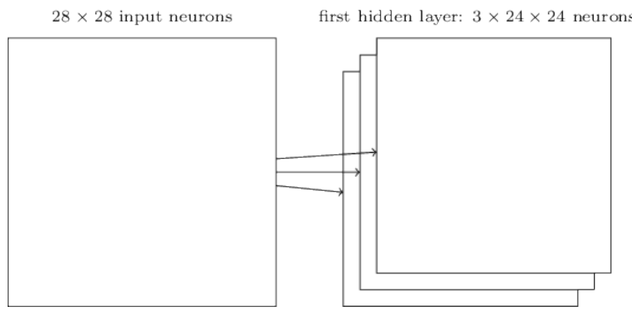
这意味着第一个隐藏层中的所有神经元都探测出相同的特征（feature）*，而它们位于输入图片中的不同位置。设想，这样的权值和 bias 代表局部接受域中某个特殊的形状。而这个形状可能在图片的其他地方也同样适用，所以使用相同的特征侦测方法有助于在图片的各个地方探测出这个形状。更抽象地说，卷积网络有着很好的转移不变性（translation invariance）：例如，将一个猫的图片进行平移、旋转，它还是一个猫的图片*。

> *我还没有准确地定义这个特征的名称，你可以把它看做是一种让神经元去激活的输入样式：例如图片的边，或者其他形状。如果探测到这样的形状，神经元就会激活，反之则不会。

> *事实上，MNIST 数据集的转移不变性并不好，图片中的数字都是居中的，大小也经过调整。在图片中，边和角的特征有时候也十分重要。

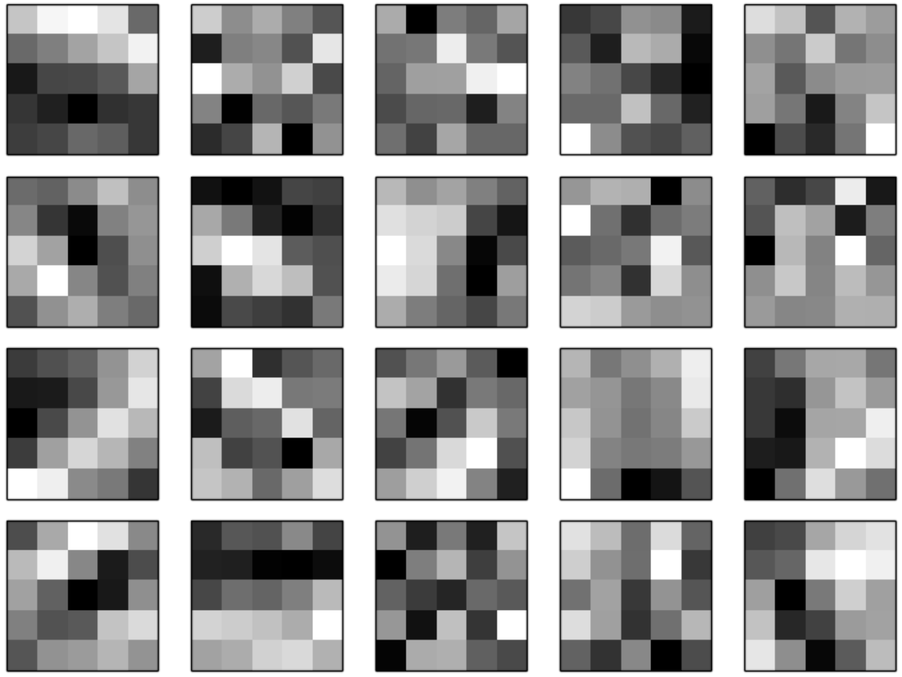
因为这个原因，我们有时候也把从输入层到隐藏层的映射（map）称为 *特征映射*。我们把定义特征映射的权值称为 *共享权值（shared weights）*，把定义特征映射的 bias 称为 *共享 bias（shared bias）*。这些共享权值和共享 bias 又被称作是 *核（kernel）* 或者是 *过滤器（filter）*。对于这些术语，不同的场合下有着一些细微的不同用法，在这里我并不会给出一个精确的定义，而是给出几个具体的例子。

上面我描述的网络只能探测（detect）出一种局部特征（localized feature），而在图像识别中，我们需要更多的特征映射，所以对于一个完整的卷积层，它由许多不同的特征映射组成：



在上面的例子中有三个特征映射（feature map）。每个特征映射由一大小为 5×5 的共享权值和单个 bias 定义。这样，我们的网络就可以探测出三种不同类型的特征，每种特征都可以在整张图片上探测出来。

三个特征映射并不多，在实际的卷积网络中，可能拥有更多（多的多的多）的特征映射。之前一个经典的卷积网络，LeNet-5，使用了六种特征映射，每个映射由大小为 5×5 的局部接受域构建，然后用来识别 MNIST 手写数字。所以上面的例子与 LeNet-5 非常相似。在后面的例子中，我们构建的卷积网络将拥有 20 个、40 个特征映射。让我们简单看一看那些学习到的特征*：



> *这些特征映射来自于本章后面所训练的卷积网络。

上面的20个图片对应着20个不同的特征映射（或者叫过滤器，或者叫核，随您的便！）每个映射的大小是 5×5，对应着局部接受域中的 5×5 的权值。图片中浅色的方块代表小的（或者说是负面的）权值，深色的方块代表大的权值。我们可以粗略地说，上面的这些图片代表着卷积层所对应的特征类型。

所以，从这些特征映射之中，我们可以得到什么呢？图片中深深浅浅的方块表现了某种结构信息，这说明我们的网络确实学习了一些东西，但是我们很难直接看出到底学习了什么。如果你对此有兴趣的话，我推荐你阅读 Matthew Zeiler 和 Rob Fergus 在2013年发表的论文 Visualizing and Understanding Convolutional Networks。

共享权值和 biases 一大好处是，这可以极大地减少卷积网络需要学习的参数数量。对于每个特征映射，我们需要 $25 = 5 \times 5$ 个共享权值，再加上一个共享的 bias，总共需要 26 个参数。如果我们有20个特征映射，那么，定义这个卷积层共需要 $20 \times 26 = 520$ 个参数。作为对比，如果是一个完整的全连接卷积层，对应 $784 = 28 \times 28$ 个输入神经元，以及 30 个的隐藏神经元，我们最终需要

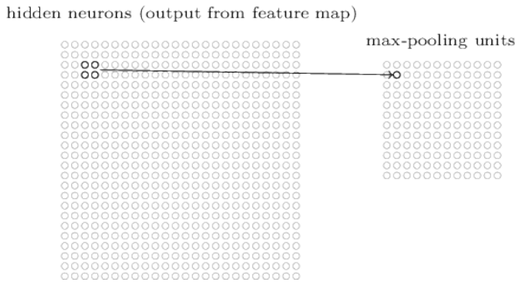
784×30 个权值，加上 30 个 biases，相当于 $23,550$ 个参数。也就是说，全连接的卷积层将比共享权值的卷积层要多出 40 倍的参数。

当然，这是两个不同的模型，做直接的对比其实并不恰当。但是，利用这种转移不变形似乎可以让我们的模型在很少的参数下就可以获得与全连接层等同的效果。这样，我们的卷积网络就可以训练的更快，同时也让训练的更深成为可能。

顺便提一句，*卷积* 这个词来自于公式（125）中的有时候被称为 *卷积* 的操作。更准确地说，人们有时候把这个公式写为 $a^1 = \sigma(b + w * a^0)$，其中 a^1 表示来自于一个特征映射的输出激活，a^0 表示输入激活，$*$ 是卷积操作。当然，我们并不是要去深究卷积在数学上的意义，所以你不需要担心太多，权当了解一下卷积这个名字的来源。

池化层（Pooling layers）：除了我们刚刚介绍的卷积层，卷积神经网络还有一种层，叫做 *池化层*。池化层一般直接用在卷积层之后，它的作用是去简化卷积层输出的信息。

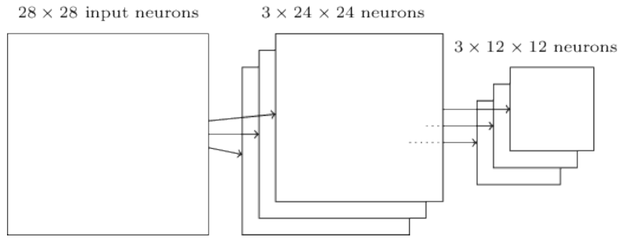
池化层接收每个特征映射*在卷积层上的输出，生成一个压缩的特征映射。例如，池化层中的一个单元也许代表着上一层中一块 2×2 的神经元区域。一种常见的池化操作是 *最大池化（max-pooling）*，在最大池化中，一个池化单元的值就是这个 2×2 区域中最大的激活，如下图所示：



> *这里使用的术语并不严谨。特别是，这里的“特征映射”表示的并不是卷积层所计算的函数，而是卷积层中隐藏神经元输出的激活。这种后果并不严重的术语滥用其实在学术界还挺常见的。

在池化之前，卷积层中有 24×24 个神经元，在池化之后，我们有 12×12 个神经元。

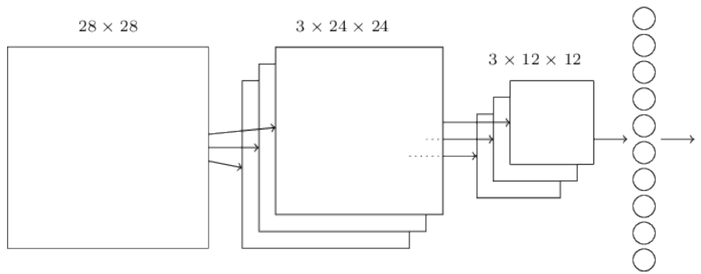
我们之前提过，卷积层中有着不止一个特征映射。我们对每一个特征映射都做一次最大池化。所以假如有三个特征映射的话，卷积层和池化层是这样的：



我们可以这样去想象最大池化的作用：它去询问网络，图片中某个区域上，有没有这种特征。然后它丢弃了具体的位置信息。这其中的道理在于，一旦在某个区域内找到了这种特征，那么这个特征在这个区域内的具体位置并不如这个特征与其他特征的相对位置重要。一个巨大的好处是，图片中这样的特征可能很少，所以这样做可以显著地减少之后的层所需要的参数。

最大池化并不是池化的唯一选择，还有一些其他的技巧，例如 *L2 池化（L2 pooling）*。除了选择 2×2 区域中的最大值，我们还可以选择这样的值：2×2 区域中所有的值的和的平方根。虽然做法不同，但 L2 池化内在的思想与最大池化是一样的：L2 池化是一种压缩卷积层中信息的技巧。在实际中，这两种技巧都被人们广泛地使用。有时候人们还会使用其他的池化方法，如果你打算优化模型的效果的话，你可以使用验证集来尝试不同的池化技巧，然后选择效果最好的那种。在这里我们并不去做这样的尝试。

整合：我们现在把所有的这些想法整合在一起，构建出一个完整的卷积神经网络。它与我们刚刚看过的结构很相似，但是多了一个输出层：



这个网络首先从 28×28 的输入神经元开始，它们对 MNIST 图片中的像素强度进行编码。然后是一个卷积层，局部接受域的大小为 5×5，拥有 3 个特征映射，总共包含 $3 \times 24 \times 24$ 个隐藏特征神经元。下一步是最大池化层，分别应用于 3 个特征映射中大小为 2×2 的区域，总共包含 $3 \times 12 \times 12$ 个隐藏特征神经元。

网络中最后一层是一个全连接层，这也就是说这一层中 10 个输出神经元连接到上一池化层中的 *每一个* 神经元。这种全连接的结构与我们在之前章节中使用的结构并没有什么不同。为了简化表示，图中我只用一个箭头表示了这些连接。

卷积结构与我们之前使用过的结构有着很大的不同。但是总体来看它们是相似的：一个有着很多单元的网络，其表现由权值和 biases 决定；它们的目标是相似的：使用训练数据去训练网络的权值和 biases，从而让网络可以准确地分类输入数字。

特别是，与以往相同，我们依旧会使用随机梯度下降和反向传播算法来训练我们的网络。这与之前的章节几乎一模一样。当然，我们需要对反向传播算法中的一些细节进行修改，因为在之前，反向传播是相对于全连接层网络来推导的。幸运的是，对卷积层和池化层的推导是很简单的。如果你想去了解细节的话，那么我推荐你去试着解开下面的难题。解决这个难题可能要花一些时间，除非，你对反向传播算法的推导已经了解的非常透彻了（其实很简单）。

难题

- **卷积网络中的反向传播**：全连接层的反向传播算法的核心公式是（BP1-BP4）（详见第二章）。设想我们有一个网络，它包含一个卷积层，一个最大池化层，以及一个全连接的输出层。请写出修正后的反向传播算法的公式。

实战卷积神经网络

我们已经学习了卷积神经网络的核心思想，现在来看看它们在实际中效果如何。我们将会创建一些卷积网络，然后试着去解决 MNIST 手写数字识别问题。我们将要使用的程序叫做`network3.py`，它是我们之前使用的`network.py`和`network2.py`程序的改进版本。程序的代码可以在 GitHub 上找到。在本节中，我们将会使用`network3.py`来构建卷积网络。

程序`network.py`和程序`network2.py`使用 Python 和矩阵库 Numpy 写成的。这些程序实现了诸如反向传播算法、随机梯度下降等功能。现在我们已经了解了这些功能背后的细节，所以对于`network3.py`，我们用到了一个机器学习库 Theano*，使用 Theano 可以非常容易地实现卷积神经网络的反向传播算法，它可以自动地计算所有的映射操作。Theano 在运行速度上也要比我们之前的代码快一些（之前的代码的目的是易于理解，而不是快），这可以让我们更有效率地去训练复杂的网络。Theano 一个巨大的优点是，它不仅仅可以在 CPU 上进行计算，还可以使用 GPU 来进行计算。使用 GPU 可以极大地提升程序运行的速度，从而帮助我们训练复杂的网络。

> *这里有一个写于2010年的教程可供参考：Theano: A CPU and GPU Math Expression Compiler in Python，作者是 James Bergstra、Olivier Breuleux、Frederic Bastien、Pascal Lambilin、Ravzan Pascanu、Guillaume Desjardins、Joseph Turian、David Warde-Farley 和 Yoshua Bengio。Theano 也是流行的神经网络库 Pylearn2 和 Keras 的基础。还有一些其他流行的神经网络库，例如 Caffe 和 Torch。（译者注：不要忘了 TensorFlow）

为了运行这个程序，你首先得建立起能使 Theano 运行的系统环境。你可以参考官方的教程来安装 Theano。下面的示例运行的 Theano 版本是 0.6*。这些实验有的是在 Mac OS X Yosemite 系统上运行的，没有利用 GPU，有些实验室在 Ubuntu 14.04 上运行的，用到了 NVIDIA 的 GPU。而有些实验在上述两个系统上都进行了测试。为了运行`network3.py`，你需要在源码中把`GPU`标志设置为`True`或者是`False`。以及，为了让程序在 GPU 上成功运行，这篇教程可能会对你有些帮助。当然，网上还有很多攻略，你可以用 Google 轻松地找到它们（译者注：算了译者不注了）。如果你的机器没有 GPU，你可以使用 Amazon Web Services EC2 G2 实例（译者注：Google 的 Colab 也非常好用）。需要注意的是，就算使用 GPU 进行加速计算，程序也要花费很多的时间去执行，可能需要数小时之久。而在 CPU 上运行的话，可能要花费数天的时间才能得到满意的结果。所以，在程序执行的过程中，你可以继续阅读本书。如果你使用 CPU 来运行那些非常复杂的程序的话，那么减少一些训练的 epochs 会是一个比较好的选择，当然，不亲自做这些实验也是可以的。

> *在我发布本章的时候，最新的 Theano 版本已经更新到了 0.7，我在 0.7 的版本上运行了这些示例，得到了几乎一模一样的结果。（译者注：在本翻译发布之时即2019年4月9日，Theano 已经更新到了 1.0 版本。）

我们首先设计一个对比的模型：一个浅层的、只有一个隐藏层的、包含 100 个隐藏神经元的网络。我们首先训练 60 个 epochs，学习率设置为 $\eta = 0.1$，mini-batch 的大小为 10，不使用正则化技巧。代码如下*：

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

> *本实验的代码可以在这里找到。代码的内容即是本章中所讲的内容。
>
> 在本章的实验中，我明确地定义了训练所需的 epochs。这样做是为了阐述清楚我们的所进行的训练。在实际中，使用早停（early stop）策略是种更好的选择，也就是说，我们跟踪验证集上的准确率，在准确率不再提升的时候停止训练。

实验最好的分类准确率大约是 97.80%，这是在`test_data`上得到的分类准确率。使用验证集可以帮助我们确定什么时候去验证测试集上的准确率，从而避免在测试集上发生过拟合（第三章中有讲到验证集的使用）。在后面的实验中我们会一直使用验证集。你实际得到的结果可能会有一些出入，因为网络的权值和 biases 是随机初始化的*。

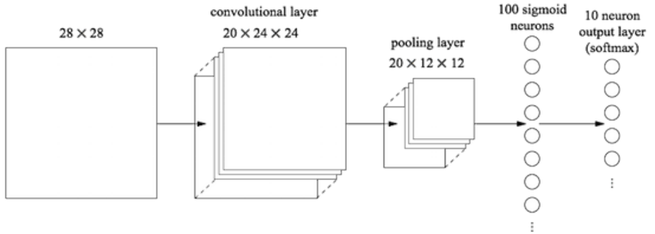
> *事实上，在做这个实验的时候，我分别运行了三次独立的实验。然后使用三次实验中效果最好的那次作为实验的最终结果。多次运行实验可以减少结果的波动，这在与其他网络结构进行比较时会比较有用。所以在之后的实验中，我都会多次运行实验，然后选择最好的那次作为结果。实际上，这样做对结果的影响其实微乎其微。

97.80% 的准确率与我们在第三章中得到的 98.04% 非常接近。这两个实验都使用了单层的包含 100 个隐藏神经元的网络，都训练了 60 轮 epochs，mini-batch 的大小为 10，学习率为 $\eta = 0.1$。

当然，与之前的方法相比，这次实验有两个不同之处。第一，之前的实验使用了正则化技巧，它可以帮助减少过拟合。如果给上面的实验加上正则化的话，确实会提升准确率，不过提升的效果非常少，所以我们先不考虑正则化。第二，之前实验模型中最后一层使用的是 sigmoid 激活函数和交叉熵代价函数，而本次实验使用的是 softmax 层和对数似然代价函数。当然，我们在第三章中进行过解释，这并不是一个很大的变化。做这样的改变并没有什么特别的考虑，只是因为在图像分类网络中，使用 softmax 加对数似然代价是一种非常常见的用法。

如果使用深层网络的话，我们能不能得到更好的结果?

让我们试着在网络的开始加上一个卷积层。我们使用 5×5 大小的局部接受域，步长为 1，以及 20 个特征映射。我们还会插入一个最大池化层，它使用一个大小为 2×2 的池化窗口来对特征进行整合。这个网络的整体结构如下图所示：



在这个结构中，我们可以认为卷积层和池化层在训练图片中对局部的空间结构信息进行了学习，然后，全连接层对其进行了更为抽象地学习，把遍布整张图片上的全局信息进行了整合。这便是卷积神经网络工作的模式。

让我们训练这样一个网络，看看其效果如何*：

```
>>> net = Network([
      ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                    filter_shape=(20, 1, 5, 5),
                    poolsize=(2, 2)),
      FullyConnectedLayer(n_in=20*12*12, n_out=100),
      SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.SGD(training_data, 60, mini_batch_size, 0.1,
            validation_data, test_data)
```

> *我依旧使用了大小为 10 的mini-batch，我们在之前讨论过，使用更大的 mini-batch 可以加速训练。不过，在这里为了前后的一致性，我们不对 mini-batch 的大小进行改变。

最终的结果是 98.78%，多么惊人的效果提升！比我们之前的任何一次实验都要好。事实上，我们减少了超过三分之一的错误率，这个提升非常的大了！

在设计网络的结构的时候，我把卷积层和池化层看做是单个的层。当然，不管把它们放在一起看，还是分开看，并没有什么实际的影响。`network3.py`把它们看做是一个层，这样代码看起来更紧凑一些。当然，把它们看做两个层是很容易的，只需要对`network3.py`做一点点修正即可，如果你想的话。

练习

- 如果我们不考虑全连接层，只使用卷积-池化层和一个 softmax 层，分类准确率会有什么变化？使用全连接层会有帮助吗？

尽管 98.78% 已经够高了，我们还有没有进一步提升的可能?

让我们试着再添加一个卷积-池化层，在第一个卷积池化层和全连接层之间。同样地，我们使用大小为 5×5 的局部接受域，池化区域为 2×2。让我们看看这样的网络结构会得到什么样的结果：

```
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2)),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2)),
        FullyConnectedLayer(n_in=40*4*4, n_out=100),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.SGD(training_data, 60, mini_batch_size, 0.1,
            validation_data, test_data)
```

我们又一次得到了提升：现在的分类准确率是 99.06%!

我们会想到两个问题。第一个问题是：添加第二个卷积-池化层的意义是什么？你可以把它想作是一个输入为 12×12 的“图片”，它的“像素”表示着某个特定的特征存在，或者不存在于原始的那张图片上。所以你可以认为，这一层的输入是原始图片的另外一个版本。这个版本既抽象又浓缩，但是还保留着很多结构特征。这样说的话，我们添加第二个卷积-池化层是有意义的。

如果这样解释的话，似乎是可行的，但是还有第二个问题。之前一层的输出包含 20 个独立的特征映射，所以第二个卷积-池化层的输入大小为 $20 \times 20 \times 12$。这相当于我们输入了 20 个独立的图片，这显然与第一个卷积-池化层的输入不同，后者的输入是 1 个图片。第二个卷积-池化层中的神经元是如何对这多个输入图片进行学习的呢？事实上，我们允许该层中每个神经元从 *所有* 的在其局部接受域中的 $20 \times 5 \times 5$ 个输入神经元中进行学习。不正式地说：第二个卷积-池化层中的特征探测器探测了之前层中 *所有* 的特征，但限定在它们特定的局部接受域之中*。

> *如果输入图片是有颜色的话，这种情况就会发生。在这种情况下，对于每个像素，我们都有 3 个输入特征，分别代表着红色、绿色和蓝色。所以我们允许特征探测器去访问所有的颜色信息，但是是在一个特定的局部接受域之中。

难题

- **使用 tanh 激活函数**：在本书中我不止一次地提到，tanh 函数也许要比 sigmoid 激活函数更为强大。但是我们一直没有验证这一点，因为我们使用 sigmoid 也获得了不少进步。但是现在让我们试一试 tanh 激活函数。试着使用 tanh 激活函数来在卷积层中和全连接层中训练网络*。

  > *你可以在`ConvPoolLayer`类和`FullyConnectedLayer`类中设定`activation_fn=tanh`来做到这一点。

  我们使用与 sigmoid 网络相同的参数，但是只训练 20 轮 epochs 而不是 60 轮。我们的网络表现的如何呢？如果训练 60 轮的话，结果会如何？用 tanh 网络和 sigmoid 网络作为对比，试着画出验证准确率在每一轮次上的变化。如果你的结果与我的相似，那么你会发现，tanh 网络会训练的稍微快一些，但是最终的准确率基本都差不多。你能解释一下，为什么 tanh 网络的训练会快一些呢？可不可以通过改变学习率，或者使用一些 缩放*（rescaling），让 sigmoid 网络也能得到相同的训练速度呢？

  > *你也许可以从下面的式子里获得一些灵感：$\sigma(z) = (1+\tanh(z/2))/2$。

  你可以在超参数上，或者网络结构上进行一系列实验，来证明 tanh 也许比 sigmoid 要更优越。*注意：这是一个开放式的难题。对于我自己来说，我没有找到任何 tanh 要更优越的证据，虽然我没有进行非常多的实验，所以也许你可以证明这一点。不过，待会我们会发现使用 ReLU 激活函数的优点，所以我们不会对 tanh 激活函数讨论太多。*

使用纠正线性单元（ReLU）：我们开发的这个网络其实是一篇1998年发表的首次介绍 MNIST 问题的论文*中使用的网络 LeNet-5 的一个变种。从这篇论文中我们可以得到更多关于实验的理解和灵感。特别是，其中介绍了很多可以提升效果的技巧。

> *Gradient-based learning applied to document recognition，作者是 Yann LeCun、Leon Bottou、Yoshua Bengio 和 Patrick Haffner，发表于1998年。当然，在实现的细节上他们的网络和我们的网络有着很多的不同，但是从本质上说，其实是非常相似的。

我们首先把 sigmoid 激活函数替换为纠正线性单元。也就是说，我们使用这个激活函数：$f(z) \equiv \max(0, z)$。我们将训练 60 个 epochs，学习率为 $\eta = 0.03$。我发现使用 l2 正则化的话，效果会有一些提升，在这里正则化系数设置为 $\lambda = 0.1$：

```
>>> from network3 import ReLU
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
```

```
                    activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(n_in=40*4*4, n_out=100, activation_fn=ReLU),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.SGD(training_data, 60, mini_batch_size, 0.03,
            validation_data, test_data, lmbda=0.1)
```

我们现在的分类准确率是 99.23%。我们获得了比使用 sigmoid 激活函数（99.06%）更好的结果。我发现，在所有的实验中，ReLU 都要比 sigmoid 激活函数表现的更好。所以，对于这个问题来说，使用 ReLU 是个更好的选择。

为什么纠正线性单元要比 sigmoid 激活函数或者 tanh 激活函数表现的更好呢？到目前为止，我们对这个问题知道的甚少。事实上，ReLU 被广泛地使用才不过短短几年时间。人们开始广泛使用 ReLU 的原因非常简单*：一些人尝试了 ReLU，并且发现其效果非常出众。在理想的情况下，对于不同的问题，应该有一个理论告诉我们，使用什么样的激活函数是更好的选择。但我们离这样的理想情况还很远。也许使用其他某个未知的激活函数可以获得更好的效果呢。我希望在接下来的几十年内，科学家们可以研究出一个理论，来告诉我们使用什么样的激活函数。如今，我们依旧依赖于实际经验来选择激活函数。

> *一些比较流行的说法是 $\max(0, z)$ 在限制大 z 的时候，其并不会像 sigmoid 函数一样趋于饱和，这可以让 ReLU 继续进行学习。虽然这个解释有一定道理，但并不是多么严谨的一个解释。（我们在第二章里对神经元饱和问题进行过讨论）

扩大训练数据：另一种提升效果的方法是扩大训练数据。我们可以使用一些简单的算法来做到这一点，例如对输入图片，移动其中的某个像素。我们可以用程序`expand_mnist.py`来扩大训练数据*：

```
$ python expand_mnist.py
```

> *代码可以从这里找到。

在 $50,000$ 张 MNIST 训练图片上运行这个程序，我们可以获得一个扩大的训练集，其中包含 $250,000$ 张训练图片。我们然后使用这些训练图片来训练我们的网络，网络结构与之前的实验相同，使用 ReLU 激活函数。在我的前几次尝试之中，我减少了训练的轮次，因为我们的训练数据比之前多了5倍，但是，扩大训练数据似乎可以很好地减少过拟合，所以，在经过几次试验之后，我依旧选择了大小为 60 的训练 epochs。不管怎么样，让我们先试一试：

```
>>> expanded_training_data, _, _ = network3.load_data_shared(
        "../data/mnist_expanded.pkl.gz")
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(n_in=40*4*4, n_out=100, activation_fn=ReLU),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.SGD(expanded_training_data, 60, mini_batch_size, 0.03,
            validation_data, test_data, lmbda=0.1)
```

使用扩大的训练数据，我们获得了 99.37% 的训练准确率这个并不很复杂的技巧，使我们得到了更好的分类准确率，这与我们在第三章中讨论的结果相似，扩大训练数据可以获得更好的结果。在之前的2003年，Simard、Steinkraus 和 Platt 等人*在 MNIST 数据集上获得了高达 99.6% 的分类准确率，他们使用的网络与我们非常类似，两个卷积-池化层，每层中包含 100 个隐藏神经元，当然，在一些细节上有一些改变，例如他们没有使用 ReLU 激活函数。他们能达到这个准确率的主要贡献来自于扩大训练数据，扩大训练数据的方法包括在 MNIST 数据上旋转、变换、扭曲图片等。他们还开发了一个名为“弹性扭曲（elastic distortion）”的过程，这个过程模仿了人类在书写的时候手掌上肌肉抖动的过程。在结合了这些技巧之后，他们扩大了训练数据，最终得到了 99.6% 的惊人结果。

> *Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis，作者是 Patrice Simard、Dave Steinkraus 和 John Platt，发表于2003年。

难题

- 卷积层背后的想法是在学习图片时保持一种不变性。所以，当我们对训练图片进行变换之后，我们的网络还能学到更多的东西，这个结果令人感到惊奇。你能解释为什么吗？

再增加一个全连接层：我们能不能得到更好的效果？也许增加全连接层的规模可以帮助我们做到这一点。我做了一些尝试，分别使用 300 个神经元和 $1,000$ 个神经元，得到了 99.46% 和 99.43% 的分类准确率。这个结果很有意思，不过提升相比之前（99.37%）并不算很大。

那么，除了增加隐藏神经元的数目，我们增加额外的一层全连接层，效果会如何呢？让我们来试一试，增加一个大小为 100 的全连接层：

```
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(n_in=40*4*4, n_out=100, activation_fn=ReLU),
        FullyConnectedLayer(n_in=100, n_out=100, activation_fn=ReLU),
        SoftmaxLayer(n_in=100, n_out=10)], mini_batch_size)
>>> net.SGD(expanded_training_data, 60, mini_batch_size, 0.03,
            validation_data, test_data, lmbda=0.1)
```

结果是 99.43%，看来扩大网络结构并不能帮助很多。使用大小为 300 和 $1,000$ 的神经元数目得到的结果是 99.48% 和 99.47，虽然是有一些提升，总之是不很大了。

问题出在哪呢？更多，或者更大的全连接层并不能真正帮助我们获得更好的效果吗？也许我们的网络有这样的潜力，但是学习过程发生了一些问题。例如，我们可以用更强的正则化技巧来帮助减少过拟合。也许使用 dropout 可以得到更好的结果。回想一下，dropout 技巧是在训练网络的时候，随机地移除一些个体激活。这可以让我们的模型表现的更为稳定，尤其是在信息不足的时候；此外，它也可以减少对训练数据的特定特征的依赖。让我们来试一试添加 dropout 技巧的全连接层：

```
>>> net = Network([
        ConvPoolLayer(image_shape=(mini_batch_size, 1, 28, 28),
                      filter_shape=(20, 1, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        ConvPoolLayer(image_shape=(mini_batch_size, 20, 12, 12),
                      filter_shape=(40, 20, 5, 5),
                      poolsize=(2, 2),
                      activation_fn=ReLU),
        FullyConnectedLayer(
            n_in=40*4*4, n_out=1000, activation_fn=ReLU, p_dropout=0.5),
```

```
        FullyConnectedLayer(
            n_in=1000, n_out=1000, activation_fn=ReLU, p_dropout=0.5),
        SoftmaxLayer(n_in=1000, n_out=10, p_dropout=0.5)],
        mini_batch_size)
>>> net.SGD(expanded_training_data, 40, mini_batch_size, 0.03,
            validation_data, test_data)
```

在使用 dropout 之后，我们获得了 99.60% 的准确率，相比之前的结果，这是一个非常可观的提升。

这里面有两个变化值得关注。

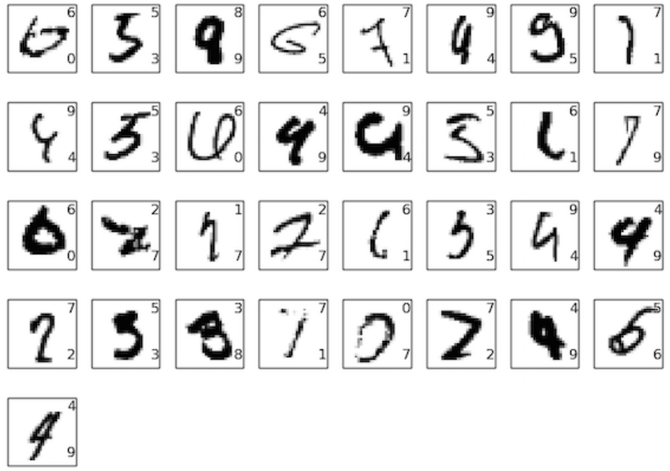
第一，我减少了训练的轮次，即大小为 40 的 epochs：dropout 可以减少过拟合，所以我们可以学习的更快。

第二，全连接层中的每一层都有 $1,000$ 个神经元，而不是之前使用的 100 个神经元。因为 dropout 在训练的时候会随机忽略很多的神经元，所以扩大神经元的数目是一个比较合理的选择。事实上，我尝试使用 300 个或者 $1,000$ 个神经元，在后者上获得了（虽然很小）更好的验证表现。

使用网络的集成：一种简单的，还可以提升更多的效果的方法是创建更多的神经网络，然后使用投票的方式来决定最终的分类。设想，我们训练了 5 个不同的神经网络，每个网络都取得了大约为 99.6% 的准确率，虽然这些网络的准确率都比较相似，但是它们犯错误的方式可能不同，这来自于我们的随机初始化操作。所以，对于这些不同的网络的预测结果，我们可以使用投票的方式来决定最终的分类。

这听起来似乎好的令人难以置信，但是这类集成技巧已经广泛的应用于神经网络和其他的机器学习任务中了。事实上，集成学习确实可以提升效果：我们得到了 99.67% 的分类准确率。换句话说，我们的集成网络分类器在 $10,000$ 张测试图片上只犯了 33 次错误。

分类错误的测试图片如下图所示。右上角的数字是 MNIST 数据集中正确的分类，而右下角是我们的集成网络给出的分类：



让我们看看具体的情况。前两个数字，一个是 6 一个是 5，我们的分类器确实分类错了。这些错误是可以理解的错误，对于人类来说，也可能犯这样的错误，因为图片中的 6 确实和 0 比较像，而那个 5 看起来和 3 差不多。第三个图片，虽然标签是 8，但它不管怎么看，都更像是数字 9。所以这一次我站在了分类器的这一边：我认为它做的比当初写这个数字的人要好。不过，另一方面，例如第四个图片 6，我们的网络确实分类错了。

这样的例子还有很多，在大多数情况下，我们的网络表现的都合乎情理，在某些情况下它们比当初写这个数字的人做的更好。总的来说，我们的网络表现的非常优秀，特别是那些没有展现出来的 $9,967$ 张图片，我们的网络全部分类正确。在这种情况下，犯一些失误似乎也是可以理解的。就算是一个非常谨慎细心的人类来分类这些数字，他们也可能会犯错误。我相信，只有那些极度细心、有条不紊的人才会表现的更好。我们网络的表现已经接近于人类了。

为什么我们只在全连接层上使用 dropout：如果你仔细地看了上面的代码的话，你可能会好奇为什么 dropout 技巧只在全连接层上得到了应用，而没有在卷积层上使用 dropout。当然，在卷积层上应用 dropout 是非常简单的，但是我们没有必要这样做：因为卷积层天然地就有一种对抗过拟合的机制。共享权值意味着卷积过滤器被强迫地在整张图片上进行学习，这让它们有着更小的可能性去只选择训练数据中的局部特征。所以在卷积层中没有必要使用类似的对抗过拟合的技巧，例如 dropout 等。

走的更远：我们还可能走的更远，在 MNIST 问题上获得更好的结果。Rodrigo Benenson 写了一个很不错的总结文章，展示了过去几年间人们的各种成果，配有相应的论文链接。大多数的论文都使用了深层卷积网络，与我们使用的结构相似。如果你仔细阅读这些论文的话，还会发现很多有趣的技巧。

虽然我并不打算介绍这些论文，不过有一篇精彩的论文不容错过。这是一篇2010年发表的论文，作者是 Ciresan、Meier、Gambardella 和 Schmidhuber*。我喜欢这篇论文的原因是其简单性，它使用的是很多层的神经网络，只有全连接层（没有卷积层）。他们最好的网络，隐藏层中分别包含 $2,500$ 个、$2,000$ 个、$1,500$ 个、$1,000$ 个，和 500 个神经元。他们还使用了类似于 Simard 等人使用的扩大训练数据的技巧。但是除了这些，他们使用的技巧更少，没有包含任何卷积层：一个非常普通的网络，如果你有耐心的话，在上个世纪八十年代就可以训练这样的网络（如果 MNIST 数据集存在的话）。这个网络最终可以得到 99.65% 的分类准确率，几乎和我们差不多。其中的秘诀在于，使用一个非常大的、非常深的、以及用 GPU 加速训练的网络，训练的轮数也很大。他们还使用了递减的学习率，从 10^{-3} 到 10^{-6}。你可以试着去实现这样一个网络，应该会比较有趣。

> *Deep, Big, Simple Neural Nets Excel on Handwritten Digit Recognition，作者是 Dan Claudiu Ciresan、Ueli Meier、Luca Maria Gambardella 和 Jurgen Schmidhuber，发表于2010年。

为什么我们能够去训练？在上一章中，我们了解到在训练深层网络的时候有着很多的障碍。特别是，我们看到梯度会变得非常不稳定：从输出层到之前的层，梯度会趋向于消失（即梯度消失问题）或者爆炸（即梯度爆炸问题）。因为梯度是我们训练时的信号，如果不够稳定，那么会发生一些期望之外的问题。

该怎么样做才能避免这些问题呢？

答案是，我们还不能完全避免这些问题。不过，我们还是做了一些改变来解决这些问题。特别是：（1）使用卷积层可以极大地减少网络需要的参数，让训练变得更为容易；（2）使用更多更强大的正则化技巧（特别是 dropout 和卷积层）去减轻过拟合，过拟合也是复杂网络中一个非常棘手的问题；（3）使用 ReLU 激活函数而不是 sigmoid 激活函数，可以加快训练速度，在实际中，这个提升大约是五分之一到三分之一左右；（4）使用 GPUs 来加速训练。特别是，在我们最后的实验中，我们用比原始 MNIST 训练集大五倍的数据来训练 40 个 epochs。在本书的之前部分，我们最多使用原始的 MNIST 数据集来训练 30 个 epochs。就算结合因素（3）和因素（4），我们耗费的训练时间也要比之前多上 30 倍。

你也许会问：“就是这样？这就是我们训练深层网络的方式？我们到底在瞎忙什么？”

当然，我们还用到了其他的想法：例如使用更大的训练集（去避免过拟合）；使用正确的代价函数（去避免学习缓慢）；使用更好的权值初始化方法（从神经元饱和的角度看，这也可以避免学习缓慢）；使用算法去扩大训练数据等。我们在之前的章节中讨论了这些技巧，并且已经实现了其中的大多数。

虽然这些想法都是简单的想法，简单，但却强大。入门深度学习也相应变得简单起来。

这些网络到底有多深？如果把卷积-池化层算作单层的话，我们最终的网络结构拥有 4 个隐藏层。这样的网络可以被称作是 *深层* 网络吗？当然，4 个隐藏层已经要比我们之前使用的浅层网络的层数要多了，后者大多数只有一层或者是两层隐藏层。另一方面，在2015年里那些最强大的深层网络中，有的已经使用了几十个隐藏层了。我有时候听到人们说，网络越深越好。这似乎在说如果你网络的深度看起来不像别人那么深的时候，你并不是在做深度学习。我并不同意这种观点，这似乎是把深度学习定义为某种依赖于层数的东西。其实，深度学习真正的突破在于人们意识到使用超过一层或者两层的网络在实际中是可行的，而这种浅层网络在2005年左右之前支配着神经网络的使用。探索更深、更有表现力的网络确实是一种有意义的突破。但是除了这个，谁的网络层数多，谁的网络层数少，这并不是关键，使用深层网络是为了得到更好的结果。

关于过程的一些话：在本节中，我们从单层的浅层网络，逐渐过渡到了多层的卷积网络。这看起来多么简单！我们不单单做出了改变，我们还获得了效果的提升。在你自己做其他实验的时候，我可以保证过程不会这么轻松。因为我在这里给出的都是一些清理过的陈述，忽略了那些失败的实验。这样的陈述可以帮助你更好地理解基础的知识，但是可能带给你一种不完整的印象。为了获得一个好的网络，你可能会遇到许许多多的错误，你需要不断地进行实验，有时候会让你感到抓狂。阅读本书第三章的一些内容，例如关于如何选择超参数的讨论等，也许可以帮助你做的更好。

卷积网络的代码

现在让我们来看一看程序`network3.py`的代码。从结构上看，它与我们在第三章中使用的`network2.py`相似，不过在细节上有一些不同，因为前者使用了 Theano。我们首先来看`FullyConnectedLayer`类，它与我们之前描述的也非常类似。下面是代码*：

> *该注释添加于2016年11月：许多读者指出在初始化`self.w`的那行代码中，`scale=np.sqrt(1.0/n_out)`，而我在第三章中写的是`scale=np.sqrt(1.0/n_in)`。这是我犯的一个错误。在理想的情况下，我应该使用修正后的代码重新把本章中所有的例子再运行一遍。不过，最近我忙于其他的项目，所以没有做这样的事情。

```
class FullyConnectedLayer(object):

    def __init__(self, n_in, n_out, activation_fn=sigmoid, p_dropout=0.0):
        self.n_in = n_in
        self.n_out = n_out
        self.activation_fn = activation_fn
        self.p_dropout = p_dropout
        # Initialize weights and biases
        self.w = theano.shared(
```

```
            np.asarray(
                np.random.normal(
                    loc=0.0, scale=np.sqrt(1.0/n_out), size=(n_in, n_out)),
                dtype=theano.config.floatX),
            name='w', borrow=True)
        self.b = theano.shared(
            np.asarray(np.random.normal(loc=0.0, scale=1.0, size=(n_out,)),
                       dtype=theano.config.floatX),
            name='b', borrow=True)
        self.params = [self.w, self.b]

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

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

`__init__`方法中的代码应该比较浅显易懂，不过我还是多说几句。像往常一样，我们使用标准的正态分布对权值和 biases 进行随机初始化。初始化的代码看起来可能有些复杂，其实大部分的工作只是把权值和 biases 载入到被 Theano 称作是共享变量的东西里。这些操作是为了保证变量可以被 GPU 所处理，如果 GPU 是可用的话。我们不会详细讨论这一点，如果你感兴趣的话，可以看一看 Theano 的官方文档。需要注意的是，我们初始化权值和 biases 的方法是基于 sigmoid 激活函数来设计的（参见第三章）。在理想的情况下，我们应该为诸如 tanh 激活函数或者是 ReLU 激活函数设计不同的初始化方法。我们在后面会讨论这一点。在`__init__`方法中，最后一行是`self.params = [self.w, self.b]`，这是一个简便的方法，把所有需要训练（学习）的变量组合在一起。之后，`Network.SGD`方法会利用到`params`属性来确定`Network`实例中，哪些变量时可以进行训练（学习）的。

`set_inpt`方法被用来设置层的输入，并且计算相应的输出。我使用了`inpt`而不是`input`，因为`input`是 Python 中的一个内部函数，如果混乱命名的话可能会发生一些难以察觉的 Bugs。在设置输入的时候，我们使用了两种不同的方法：一种是`self.inpt`，另外一种是`self.inpt_dropout`。这样做的原因是，在训练的时候，我们也许希望去使用 dropout 技巧。在使用 dropout 的时候，我们希望以`self.p_dropout`的概率去移除神经元。这也是`set_inpt`方法中倒数第二行的函数`dropout_layer`所做的事情。所以，`self.inpt_dropout`和`self.output_dropout`在训练的时候使用，而`self.inpt`和`self.output`在其他的情况下使用，例如评估验证集和测试集上准确率的时候。

`ConvPoolLayer`和`SoftmaxLayer`类的定义与`FullyConnectedLayer`相似。事实上，它们相似到我不打算贴出它们的代码。如果你感兴趣的话，可以看看待会展示的`network3.py`的完整代码。

还有一些细微的改动，我觉得应该提及一下。最明显的改动是，在`ConvPoolLayer`和`SoftmaxLayer`有着不同的输出激活的计算方法。幸运的是，Theano 提供了一些内部操作去计算卷积、最大池化、以及 Softmax 函数。

不太明显的有，在第三章中我们介绍 softmax 层的时候，我们从来没有讨论过如何去初始化它们的权值和 biases。我们讨论过对于 sigmoid 层，我们应该使用正态随机变量来初始化权值，但是这样的想法是针对于 sigmoid 神经元的（经过一些修改后，也适用于 tanh 神经元）。不管怎么样，我们还不知道如何去初始化一个 softmax 层。所以我们还没有一个 *先验* 的原因去使用相同的初始化方法。在这里，我把所有的权值和 biases 都初始化为 0。这看起来像是一种临时的做法，不过在实践中表现的很好。

Okay，我们已经讨论过所有的层的类了，下面讨论`Network`类，首先是`__init__`方法：

```
class Network(object):

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

        """
        self.layers = layers
        self.mini_batch_size = mini_batch_size
        self.params = [param for layer in self.layers for param in layer.params]
        self.x = T.matrix("x")
        self.y = T.ivector("y")
        init_layer = self.layers[0]
```

```
        init_layer.set_inpt(self.x, self.x, self.mini_batch_size)
        for j in xrange(1, len(self.layers)):
            prev_layer, layer  = self.layers[j-1], self.layers[j]
            layer.set_inpt(
                prev_layer.output, prev_layer.output_dropout,
self.mini_batch_size)
        self.output = self.layers[-1].output
        self.output_dropout = self.layers[-1].output_dropout
```

大多数的代码应该看起来浅显易懂。`self.params = [param for layer in ...]`把每一层中的参数聚集到一个列表中。如同前面刚刚提过的，`Network.SGD`方法会使用`self.params`去确定哪些变量是`Network`可以训练的。`self.x = T.matrix("x")`和`self.y = T.ivector("y")`定义了 Theano 的符号变量`x`和`y`，它们表示了网络的输入和期望输出。

当然，本文不是 Theano 的教程，所以我不会介绍的太详细*。大致来说，它们表示的是一种数学意义上的变量，而 *不是* 显式的数值。我们可以对它们应用许多普通的操作，例如加、减、乘，套用函数等等。事实上，Theano 提供了非常多的操纵符号变量的方法，可以实现例如卷积、最大池化等操作。但其最大的好处是，使用一种一般形式的反向传播算法，来快速地进行符号微分计算（symbolic differentiation）。这在应用随机梯度下降的时候非常有用，适用于不同的网络结构。例如，下面的代码定义了网络的符号输出。我们首先设置初始层的输入：

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

> *Theano 的官方文档提供了很多有用的信息。如果你遇到了困难的话，你也可以在网上找到很多其他的教程。例如，这篇教程覆盖了许多基础知识。

你可能注意到代码中使用了`self.mini_batch_size`，这是因为每一次设置一个 mini-batch 大小的输入。值得注意的是，我们传输了两次输入`self.x`：这是因为我们会有两种不同的方法来使用网络（使用 dropout 或者不使用 dropout）。然后，`for`循环让符号变量`self.x`在`Network`的层中传播。然后，我们定义了`output`和`output_dropout`属性，它们象征性地表示了`Network`的输出。

我们理解了`Network`是如何初始化的，现在看看它如何使用`SGD`方法来进行训练。下面的代码看起来很长，但是它的结构其实很简单。代码之后附有一些注释。

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

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

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

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

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

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

```
        print("Corresponding test accuracy of {0:.2%}".format(test_accuracy))
```

前面几行代码比较直接，把数据集分为 x 和 y，然后计算每个数据集中所需要 mini-batch 数。接下来的几行代码更有趣一些：

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

上面的代码象征性地设置了正则化过后的对数似然代价函数，计算了梯度函数中所有对应的导数，以及所有对应的参数更新。Theano 实现这些功能只需要短短几行代码。不过其中如何计算`cost`所使用的`cost`方法在`network3.py`的其他地方，这里没有显示出来，总之它也很简单就是了。在完成定义之后，接下来的事就是定义`train_mb`函数，一种 Theano 符号函数，在给定 mini-batch 的索引之后，它使用`updates`来更新`Network`的参数。相似地，`validate_mb_accuracy`和`test_mb_accuracy`计算某个给定的验证集或者测试集的 mini-batch 上的准确率。把所有的 mini-batch 的准确率进行平均，我们就可以得到最终整个验证集，或者测试集上的准确率。

剩余的`SGD`代码简单地迭代了所有的 epochs，重复地在训练数据的 mini-batch 上训练网络，计算验证准确率和测试准确率。

Okay，到现在，我们已经了解了`network3.py`大多数代码的原理。让我们来看看整个程序的代码。你不需要仔细地阅读整个程序，浏览一遍即可。如果你有什么地方不懂的话，可以直接阅读相应的代码。当然，最好的理解代码的方式是，修改它！增加更多的功能，重构代码，等等。在浏览完代码之后，我会给出一些难题。下面是代码：

> *在 GPU 上使用 Theano 可能会有一些困难，特别是，在从 GPU 中提取数据的是，很多用户会犯一些错误，这会导致程序运行的非常慢。我已经努力去避免这些问题。这也就是说，在仔细地优化 Theano 的设置之后，这份代码确实可以加快训练的速度。更多的细节参考 Theano 的官方文档。

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

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

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

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

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

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

Written for Theano 0.6 and 0.7, needs some changes for more recent
versions of Theano.

"""

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

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

```
from theano.tensor import shared_randomstreams
from theano.tensor.signal import downsample

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


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

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

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

```
#### Main class used to construct and train networks
class Network(object):

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

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

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

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

        # define the (regularized) cost function, symbolic gradients, and updates
```

```
        l2_norm_squared = sum([(layer.w**2).sum() for layer in self.layers])
        cost = self.layers[-1].cost(self)+\
               0.5*lmbda*l2_norm_squared/num_training_batches
        grads = T.grad(cost, self.params)
        updates = [(param, param-eta*grad)
                   for param, grad in zip(self.params, grads)]

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

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

#### Define layer types

class ConvPoolLayer(object):
    """Used to create a combination of a convolutional and a max-pooling
    layer.  A more sophisticated implementation would separate the
```

```
    two, but for our purposes we'll always use them together, and it
    simplifies the code, so it makes sense to combine them.

    """

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

        `image_shape` is a tuple of length 4, whose entries are the
        mini-batch size, the number of input feature maps, the image
        height, and the image width.

        `poolsize` is a tuple of length 2, whose entries are the y and
        x pooling sizes.

        """
        self.filter_shape = filter_shape
        self.image_shape = image_shape
        self.poolsize = poolsize
        self.activation_fn=activation_fn
        # initialize weights and biases
        n_out = (filter_shape[0]*np.prod(filter_shape[2:])/np.prod(poolsize))
        self.w = theano.shared(
            np.asarray(
                np.random.normal(loc=0, scale=np.sqrt(1.0/n_out),
size=filter_shape),
                dtype=theano.config.floatX),
            borrow=True)
        self.b = theano.shared(
            np.asarray(
                np.random.normal(loc=0, scale=1.0, size=(filter_shape[0],)),
                dtype=theano.config.floatX),
            borrow=True)
        self.params = [self.w, self.b]
```

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

class FullyConnectedLayer(object):

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

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

```
            inpt_dropout.reshape((mini_batch_size, self.n_in)), self.p_dropout)
        self.output_dropout = self.activation_fn(
            T.dot(self.inpt_dropout, self.w) + self.b)

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

class SoftmaxLayer(object):

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

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

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

    def accuracy(self, y):
```

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


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

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

难题

- 到目前为止，`SGD`方法要求用户去手动选择训练的轮次。在之前的章节里我们讨论了一种自动选择训练轮次的方法，即早停（early stopping）。修改`network3.py`使其支持早停。
- 使`Network`方法可以在任意一个数据集上返回准确率。
- 修改`SGD`方法，从而允许学习率 η 是训练轮次的一个函数。*提示：如果你实在不会做的话，可以看看这个链接。*
- 在第一章中，我介绍了一种扩大训练数据的方法，即小幅度地旋转、扭曲、变换数据。修改`network3.py`去增加这样的功能。*提示：除非你的内存非常非常大，直接生成全部的扩大后的数据集是不现实的。你也许应该考虑其他的方法。*
- 在`network3.py`中增加保存和载入网络的功能。
- 代码的一个缺点是它提供了很少的诊断工具。你能添加一个诊断功能，从而可以让用户可以轻松地判断过拟合的程度吗？
- 对于 ReLU 激活函数，我们使用的初始化方法与 sigmoid 或者 tanh 激活函数没什么区别。这个初始化方法是特定于 sigmoid 函数来的。证明如果用一个常数因子 $c > 0$ 来调节网络所有的权值，那么网络的输出会以因子 c^{L-1} 调整，其中 L 是网络的层数。如果最终层是 softmax 的话，这会有什么影响？你如何看待对 ReLU 使用 sigmoid 初始化方法？你能想到一个更好的初始化方法吗？*提示：这是一个开放式的问题，没有绝对的答案。思考这个问题有利于你对包含 ReLU 激活函数的网络的理解。*

- 在上一章中，我们讨论了 sigmoid 神经元的不稳定梯度问题。如果网络使用 ReLU 激活函数的话，会有什么改变？你能想到一种好的方法，可以让网络避免遇到不稳定梯度问题吗？*提示：这是一个研究性的问题，改进方法虽然很多，但是我还没有遇见一个真正好的技巧，虽然我也没有非常深入地研究这个问题。*

图像识别领域的最新进展

（译者注：可能不够新了，仅供参考。）

1998年，诞生了 MNIST 数据集，在当时，使用最好的方法，训练好几周得到的结果，比我们使用 GPU 训练不到一个小时得到的结果都差。现在看来，MNIST 数据集已经不那么具有挑战性了；在如今各种高性能设备的加持之下，容易训练的 MNIST 问题作为机器学习的教学数据倒是比较合适。同时，图像识别领域研究的重点转移到了其他更具有挑战的问题上。在本节中，我会简单介绍一下最近的进展。

本节与本书的其他部分不太一样。我一般描述的是一些常见、一直使用的方法，例如反向传播算法、正则化、以及卷积网络等。对于一些新奇的技术，我刻意没有去写它们，因为这些新技术的长期价值还没有被验证。在科学领域中，这些技术可能并不会长久地存在下去，而是慢慢消失，被其他更新的技术所替代。所以，有人可能会有疑惑：“是的，这些图像识别领域的最新技术确实可能只是暂时的，过几年它们可能就会过时，在现在看来，只有那些专门研究这些的专家才会对这些技术感兴趣吧，我们为什么要费力介绍它们呢？”

这样的说话在某些程度上是对的，最近几年发表的各种论文的影响力会逐渐减弱，被更新的论文所替代。而在过去的几年里，得益于深度学习，人们在许多极度困难的图像识别任务上获得了惊人的效果。设想在2100年，一位历史学家打算写一本计算机视觉的发展史。他会把2011年到2015年（也许会向后推迟几年）这几年定义为由深层卷积网络所推动的产生各种重大突破的一段时期。这并不意味着我们在2100年仍然还在使用深层卷积网络，或者 dropout，或者 ReLU 等等。但这意味着在历史的长河中，一个巨大的转变已经在发生了。这有一点类似于原子的发现，或者抗生素的发明：在一种历史的视角下的发现和发明。所以，虽然我们不会对这些进展深入地去做研究，但是了解最近各种激动人心的技术突破是非常有益的。

2012年的 LRMD 论文：让我们从2012年来自斯坦福大学和谷歌的研究人员所发表的论文*开始。我把这篇论文称为 LRMD，即前四位作者的姓的首字母。LRMD 使用了神经网络去分类 ImageNet 中的图片，这是一个非常有挑战性的图像识别问题。他们使用的2011年的 ImageNet 数据包含了一千六百万张完整的带有色彩的图片，这些图片总共有着大约两万种分类。这些图片是从网上使用爬虫获得的，由 Amazon 的 Mechanical Turk service 的员工对其进行分类。下面展示了一些 ImageNet 中的图片*：

> *Building high-level features using large scale unsupervised learning，作者是 Quoc Le、Marc`Aurelio Ranzato、Rajat Monga、Matthieu Devin、Kai Chen、Greg Corrado、Jeff Dean 和 Andrew Ng，发表于2012年。该论文中使用的网络结构与我们所学习的卷积网络的结构在细节上有着非常多的不同。不过大概来说，LRMD 的想法与我们是类似。
>
> *这些图片来自于2014年的数据集，与2011年的数据集有一些不停。在质量上看，它们是非常相似的。有关 ImageNet 数据集的详细信息可以查阅原始的 ImageNet 论文：ImageNet: a large-scale hierarchical image databse，作者是 Jia Deng、Wei dong、Richard Socher、Li-Jia Li、Kai Li 和 Li Fei-Fei，发表于2009年。

上面的这些图片，它们各自的分类为 beading plane、brown root rot fungus、煮牛奶和一种常见的线虫。如果你打算做一些挑战的话，我建议你去访问一下 ImageNet 的手工器具列表，它们可以分辨出 beading planes、block planes、chamfer planes，还有好多好多各种各样千奇百怪的 planes。你能不能分辨的出我不太清楚，反正我是分不清咯。相比 MNIST 来说，ImageNet 显然要难的多的多！LRMD 的网络的最终结果是 15.8% 的分类准确率。这个结果似乎，看起来，不太高，不过相比于之前最好的结果 9.3%，这已经是个巨大的提升了。这个提升似乎告诉我们，神经网络在解决类似于 ImageNet 这样非常困难的任务时确实有着很好的效果。

2012年的 KSH 论文：LRMD 的工作其实是基于2012年由 Krizhevsky、Sutskever 和 Hinton（合称 KSH）*等人发表的论文。KSH 对 ImageNet 数据集的一个限制后的子集进行了训练和测试，使用的方法是卷积神经网络。这个子集来自于一个流行的机器学习竞赛：ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)。使用竞赛数据集可以很好的将他们的方法与其他领先的方法进行比较。ILSVRC-2012 数据中的训练集包含超过一百二十万张 ImageNet 图片，有着一千种分类。验证集和测试集分别包含 $50,000$ 张和 $150,000$ 张图片，同样有着一千种分类。

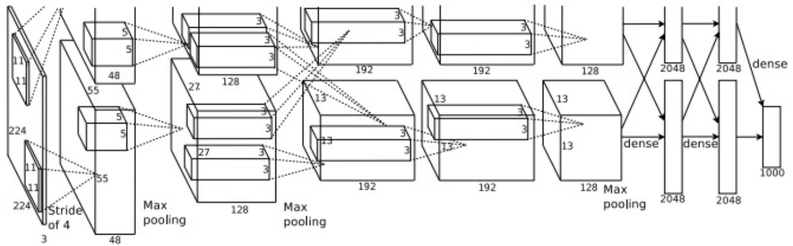
> *ImageNet classification with deep convolutional neural networks，作者是 Alex Krizhevsky、Ilya Sutskever 和 Geoffrey E. Hinton，发表于2012年。

ILSVRC 竞赛的一个难点在于，许多图片中包含多个物体。设想某一张图片，其中有一只拉布拉多猎犬在追逐一颗足球。也许这张图片的"正确"分类是"拉布拉多猎犬"，但是如果某种算法把这张图片分类为"足球"，这算是分类错误吗？因为存在着这种模棱两可的不确定性，竞赛规定，预测器给出的前五个最有可能分类中，包含了正确的分类，那么就算做是分类成功。在这种 top-5 的标准下，KSH 的深层卷积

网络可以得到高达 84.7% 的准确率，比之前最好的方法要高不少（73.8%）。如果要求预测的分类要完全正确，KSH 网络的准确率是 63.3%。

KSH 激励了许多之后的工作，所以在这里值得介绍一下它的网络。其网络结构与我们之前使用的网络结构非常相似，不过它更为复杂一些。KSH 使用了深层卷积网络，因为他们使用的 GPU （即 NVIDIA GeForce GTX 580）的显存不太够，所以在两块 GPU 上进行了训练。首先把网络分为两个部分，不同的部分在不同的 GPU 上训练。

KSH 的网络总共拥有 7 个隐藏层。前 5 个隐藏层是卷积层（有些配有池化层），后 2 个是全连接层，输出层是大小为 $1,000$ 个单元的 softmax 层，对应着 $1,000$ 种图片分类。下图展示了网络的结构，该图来自于 KSH* 论文。图中许多层都分为了两部分，对应着两块 GPU。



> *感谢 Ilya Sutskever 供图。（译者注：此图并非来自于视觉中国，^_^）

输入层包含了 $3 \times 224 \times 224$ 个神经元，表示着图片大小为 224×224 的 RGB 值。因为 ImageNet 原始图片的分辨率大小不一，所以不能直接输入到网络中，神经网络的输入层一般拥有一个固定的大小。KSH 调整了每张图片的大小，使其短边的长度为 256，然后从中间切割出一块大小为 256×256 的子图片，再从这个 256×256 的图片中随机抽取出大小为 224×224 的多张图片。这样裁剪图片是扩大训练数据的一种方式，从而减少过拟合，对于像 KSH 这样复杂的大网络来说，这样做尤其有效。这些大小为 224×224 的图片然后被输入到网络之中，在大多数的情况下，修剪过后的图片依然包含着原始图片中的主要物体。

KSH 的第一个隐藏层是卷积层，之后是最大池化层，其局部接受域的大小为 11×11，步长为 4 个像素，总共产生 96 张特征映射。这些特征映射被分为两部分，每部分 48 个，一部分存放在一个 GPU 中，另一部分存放在另一个 GPU 中。随后的最大池化层的池化区域为 3×3，池化区域可以允许重叠，这样每个区域相隔 2 个像素。

第二个隐藏层依旧是卷积层，之后是最大池化层，其局部接受域的大小变成了 5×5，所以总共有 256 张特征映射，分为各自大小为 128 的两个部分，送往不同的 GPU 中。需要注意的是，特征映射中只包含 48 个输入通道（input channel），而不是之前层输出的完整的 96，这是因为单个特征映射只使用了相同 GPU 的输出。在这种角度下，这个卷积网络的结构与我们之前使用的并不相同，不过其基本的想法仍旧是相似的。

第三、第四、第五个隐藏层都是卷积层，不过与前两层不一样，它们之后没有包含池化层。它们各自的参数为：（3）384 张特征映射，局部接受域的大小为 3×3，以及 192 个输入通道；（4）384 张特征映射，局部接受域的大小为 3×3，以及 192 个输入通道；（5）256 张特征映射，局部接受域的大小为 3×3，以及 192 个输入通道。需要注意的是，第三层中的 GPU 会相互通信（在图中可以看出这一点），故特征映射使用了所有的 256 个输入通道。

第六层和第七层是全连接层，每层中包含 $4,096$ 个神经元。

输出层是包含 $1,000$ 个单元的 softmax 层。

KSH 网络利用了非常多的技巧。首先，它没有使用 sigmoid 或者 tanh 激活函数，使用的是 ReLU 激活函数，其可以显著地增快训练速度。KSH 网络大约含有六千万个可供学习的参数，就算训练集很大，其也不可避免地会容易发生过拟合。为了克服这一点，他们使用了之前所述的扩大训练数据的方法，以及 l2 正则化和 dropout。网络本身是由基于动量的 mini-batch 随机梯度下降算法（momentum-based mini-batch stochastic gradient descent）来训练的。

以上便是 KSH 网络中所使用的主要方法。我忽略了一些细节，你感兴趣的话可以去阅读该论文。你也可以去看看 Alex Krizhevsky 写的 cuda-convnet，里面包含了很多代码实现。还有一个基于 Theano 的代码实现*，你可以从这里下载。这些代码与本章中开发的代码很类似，不过前者使用了多 GPU 训练，会稍微复杂些。Caffe 神经网络框架也包含了 KSH 网络的一个实现，可以从这里了解详情。

> *Theano-based large-scale visual recognition with multiple GPUs，作者是 Weiguang Ding、Ruoyan Wang、Fei Mao 和 Graham Taylor，发表于 2014 年。

2014年 ILSVRC 竞赛：从2012年开始，许许多多的进展如雨后春笋般接踵而至，例如2014年的 ILSVRC 竞赛，竞赛数据中的训练集包含一千二百万张图片，一千种图片分类，评估方法是 top-5。最后获胜的队伍来自谷歌*，他们使用了一种包含 22 个隐藏层的深层卷积网络，其被称为 GoogLeNet，显然是对经典网络 LeNet-5 的致敬。GoogLetNet 实现了 93.33% 的 top-5 准确率，比2013年的冠军（Clarifai，88.3%），以及2012年的冠军（KSH，84.7%）都要高很多。

> *Going deeper with convolutions，作者是 Christian Szegedy、Wei Liu、Yangqing Jia、Pierre Sermanet、Scott Reed、Dragomir Angueloy、Dumitru Erhan、Vincent Vanhoucke 和 Andrew Rabinovich，发表于2014年。

为什么 GoogLetNet 可以取得高达 93.3% 的准确率？在2014年一个研究者小组写了一篇关于 ILSVRC 竞赛的综述论文*。他们讨论了一个有趣的问题，即人类能在 ILSVRC 竞赛上取得什么样的结果。为了做到这一点，他们构建了一个系统，允许人类来对 ILSVRC 图片做分类。作者 Andrej Karpathy 在一篇博文中作了一些非常有意思的解释，让人类达到 GoogLeNet 的表现是非常困难的：

> **...就算对于那些非常熟悉 ILSVRC 数据集的人，从 $1,000$ 个类别中挑出 5 个来对图片做分类，都变得非常的有挑战性。一开始我们想把它放在 [Amazon Mechanical Turk] 上，然后我们想可以雇佣几个没有毕业的大学生。我还在实验室里组织了一个标记小组。我还开发了一个程序，把分类的类别从 $1,000$ 降到了 100。这个挑战仍旧十分困难，人们不断地错误分类，比例大约是 $13-15\%$。最后，我发现如果想要达到和 GoogLetNet 一样的效果的话，最有效率的方法可能是我亲自坐下来痛苦地学习各种分类，然后再对图片进行标记...标记一开始需要花费很长时间，大概一分钟一张图，但是之后速度变快了...有些图片很容易分辨，而有些图片很难分辨（例如那些种类繁多的狗、鸟、或者猴子等）。我最后变得非常擅长于分辨犬类...最后，GoogLetNet 数据集（不完全的）上的分类错误为 6.8%...而我自己的分类错误率是 5.1%，大约要高了 1.7%。**

> *ImageNet large scale visual recognition challenge，作者是 Olga Russakovsky、Jia Deng、Hao Su、Jonathan Krause、Sanjeev Satheesh、Sean Ma、Zhiheng Huang、Andrej Karpathy、Aditya Khosla、Michael Bernstein、Alexander C. Berg 和 Li Fei-Fei，发表于2014年。

换句话说，对于一个专家，极力细心地分类，才可能稍微超过深层神经网络。事实上，Karpathy 的报道里说有一位专家，在一些图片上进行过训练之后，仅仅有 12.0% 的 top-5 错误率，比 GoogLeNet 的表现要低不少。大概有一半的错误是因为该专家“没有把正确分类当做一种选项”。

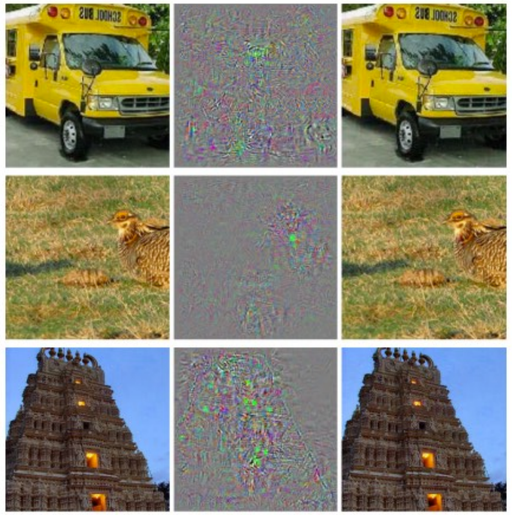
这些结果令人惊奇。事实上，在这篇论文之后，有许多团队都获得了 *小于* 5.1% 的 top-5 错误率。有些媒体甚至报道了这些结果：神经网络的表现超过了人类视觉，等等。虽然这些结果确实令人兴奋，但这并不真正意味着计算机超过了人类视觉。ILSVRC 竞赛在某种意义上说只是一个局限的问题，在网上爬去的图片并不能真正代表现实世界中真实的场景！当然，top-5 这个评价指标也过于简单化了。在图像识别领域上，或者说是在计算机视觉领域上，我们要走的路还很长。但是这几年来，有着这么多鼓舞人心的惊人进展，我们应有信心做到这一点。

其他的问题：前面我主要介绍的是 ImageNet，其实还有非常多其他的图像识别问题。让我简要地介绍一下最近的一些有趣的成果。

其中一个有趣的问题来自于 Google，他们使用深层卷积网络来识别谷歌街景上的门牌号（street number）*。在论文中，他们以大相近与人类的准确率检测和记录了大约一亿张门牌号。这个系统运行起来很快：记录法国街景图片中所有的门牌号只花了不到一个小时！他们说：“使用这个新的数据集可以显著地增加谷歌地图中地理编码的质量，特别是在那些质量还很低的国家的数据上。”他们还做了更乐观的估计：“我们相信在拥有这个模型之后，我们已经解决了许多应用中的 [短字符] 的 [视觉字符识别] 问题。”

> *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks，作者是 Ian J. Goodfellow、Yaroslav Bulatov、Julian Ibarz、Sacha Arnoud 和 Vinay Shet，发表于2013年。

最近的这些进展和成果看起来都非常出色，不过，有一些有趣的工作指出，这里面还有一些基础的东西我们目前还不能理解。例如，在2013年发表的一篇论文*里指出，深层网络可能会遇到有效盲点问题（effectively blind spots）。考虑下面的图片，左边的三张来自于 ImageNet，网络成功地对它们进行了分类。而在右边的一些经过扰动的图片（扰动显示在中间的三张图片），这右边的三张图片全部分类*失败*。作者发现对于每张图片，都存在着被称为“对抗（adversarial）图片”的东西，而不是某些图片会有这个问题。



> *Intriguing properties of neural networks，作者是 Christian Szegedy、Wojciech Zaremba、Ilya Sutskever、Joan Bruna、Dumitru Erhan、Ian Goodfellow 和 Rob Fergus，发表于2013年。

这个现象让人感到疑惑。论文中使用的网络与已经被广泛使用的 KSH 相同。这些网络计算的函数，大体上都是连续函数。而这个现象似乎是在说，它们计算的都是不连续的函数。最糟糕的是，这种现象打破了我们对网络的某种幻想，即网络的行为是合理的、可解释的。这确实是一个非常必要的担忧。现在，我们还没有理解为什么会造成这种不连续性：是代价函数的问题吗？是激活函数的问题吗？或者一些其他的东西？我们还不知道。

不过，在实际中，结果可能并不会那么糟。尽管这样的对抗图片是存在的，但是它们在实际中可能并不会发生。就像论文里说的：

> **这种对抗的负面图片的存在，似乎在否定网络的泛化性能。确实，如果网络的泛化性能足够好的话，它们怎么会被这样的对抗图片所迷惑呢？一种解释是，这样的对抗图片出现的概率是非常非常低的，虽然这样的负面图片很多（就像有理数一样），但它们可能从未出现在测试集中。**

此外，这个最近才发现的现象也说明我们对神经网络知之甚少。当然，这个现象也激励了很多随后的工作。例如，最近的一篇论文*展示了对于一个训练过后的网络，输入一个没有任何意义的像白噪声那样的图片，网络可能以非常高的置信度将其分类为某个已知的类别，例如把一个空白图片分类成猫一样。这也是我们对神经网络知之甚少的另外一个例证。

> *Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images， 作者是 Anh Nguyen、Jason Yosinski 和 Jeff Clune，发表于2014年。

除了这些糟糕的结果，总体来看，大多数结果还是好的。我们已经看到了就算是在某些高难度的问题上例如 ImageNet，我们也可以得到相对不错的结果。在现实生活中的实际问题上，我们也解决了不少问题，例如街景图片中的门牌号识别。但是这远远不够，我们对那些非常基础的现象还了解的很少。像这样那样的缺陷还在不断地被发现，虽然说完全解决图像识别问题还为时过早，不过这些新的发现都在激励着新的工作、新的技术、新的解决方案，*士不可以不弘毅，任重而道远*。

其他类型的深层神经网络

本书中我们主要讨论了一个问题：识别 MNIST 手写数字。在解决这个问题的过程中，我们了解了非常多的技术：随机梯度下降、反向传播、卷积网络、正则化等等。其实，还有很多其他类型的神经网络：循环神经网络（recurrent neural networks, RNN）、波兹曼机（Boltzmann machines）、生成模型（generative models）、迁移学习（transfer learning）、强化学习（reinforcement learning）等，等，等，咚咚隆咚齐咚，呛！神经网络是一个发展非常快的领域，虽然有着许多重要的技术，不过我们

已经了解了很多基础的技术，再去了解这些新的技术就容易多了。在本节中，我会介绍一些值得了解的技术，当然是很简要的，不然本书的篇幅就会很长很长了。我还会附上很多链接，如果你对某个领域感兴趣的话，可以自己去深入了解。当然，这些链接可能会失效，内容也可能不够新了，你也许需要去搜索引擎上找更多更新的相关资料。

循环神经网络（Recurrent Neural Networks, RNNs）：在前向传播的网络中，单个的输入完全决定着后面层中神经元的激活的值。这是一个非常固定的过程：网络中的所有东西都是固定的。如果，我们允许网络可以动态的变化呢？例如，隐藏神经元的行为不仅由前一层的输出激活所确定，它也可以由之前的输入所确定。当然，一个神经元的激活确实可能部分由之前时间的自己的激活所确定，但这并不是前向传播网络中所发生的事情。也许隐藏神经元和输出神经元的行为并不只由最近网络的输入所确定，还可能受到更早的输入的影响。

拥有这种时间变化行为的神经网络被称为 *循环神经网络* 或者是 *RNNs*。现在有着很多种实现循环神经网络的数学方法，如果你感兴趣的话，可以看看维基百科上面循环神经网络的页面，里面描述了不下十三种模型。除却那些数学上的细节，循环神经网络是一种带有时间动态变化的神经网络。所以，在处理那些随着时间变化的数据或者过程的时候，循环神经网络表现的很好，例如语音，或者是自然语言领域。

近几年 RNNs 的兴起是人们想把神经网络与传统的例如图灵机或者是传统的编程语言建立起联系。一篇2014年的论文开发了一种 RNN 模型，它可以把一个非常非常简单的 Python 程序的字符作为输入，然后来预测输出。这个模型试着去“理解”某个特定的 Python 程序。还有一篇论文，也发表于2014年，使用 RNNs 开发了一种被称为 neural Turing machine （NTM）的东西。这个电脑可以使用梯度下降来进行训练。作者们训练 NTM，使其可以推断一些非常简单的问题的算法，例如排序和复制等。

但这终归是一个极度简单的模型。学习执行一个 Python 程序，例如`print(398345+42598)`，并不能让这个网络编程一个完整的 Python 解释器！这个想法到底能走多远，现在还不好说。从现在的角度上看，神经网络在解决模式识别问题上做的很不错，但是在传统的算法问题上无法展现其威力。传统的算法方法可以很好地解决神经网络所不能解决的问题，两者形成了鲜明的对比和互补。所以，我们希望有一个模型能够聚合神经网络和算法方法的优点。RNNs 及那些类似于 RNNs 的模型也许可以帮助我们做到这一点。

RNNs 也被应用到其他类型的问题上，例如，人们发现 RNNs 在语音识别上表现的非常不错。一些基于 RNNs 的模型在音素识别（phoneme recognition）问题上创造了最好的记录，以及被用来开发识别人类说话的语言的模型，一个好的模型可以用来分辨那些发音非常相近的单词，例如，识别为“to infinity and beyond”，而不是"two infinity and beyond"，就算"to"和"two"在发音上完全相同。RNNs 在这些问题上创造了新的记录。

上面列举的只是深层神经网络可以做到的一部分事情，除了 RNNs，除了语音识别，深层神经网络还可以应用在很多问题上，例如最近一个基于深层网络的模型在 [large vocabulary continuous speech recognition] 问题上达到了非常高的结果。在谷歌的安卓操作系统中也应用了深层网络（相关的技术工作，可以参考 Vincent Vanhoucke 在2012年到2015年发表的论文）。

虽然我说了一些 RNNs 可以做到的事情，但没有说 RNNs 是怎么做到这些事情的。其实，我们之前在前馈网络中所学到的许多技术都可以应用在 RNNs 里。特别是，我们可以用稍加修改的梯度下降和反向传播算法来训练 RNNs。前馈网络中应用的其他技术，从正则化到卷积，从激活函数到代价函数，都可以用在循环网络之中。我们在本书中所开发的各种技巧也都适用于 RNNs。

长短期记忆网络（Long short-term memory units, LSTMs）：早期的 RNNs 的一个缺点是它们比传统的深层前馈网络要难以训练，原因就是我们在第五章中所讨论过的不稳定梯度问题，即网络中的梯度在反向传播的过程中容易变得非常小，或者变得非常大，这使得网络中前面层的学习速度会非常之慢。在 RNNs 中，这个问题变得更加严重了，因为梯度不仅仅会在层之间反向传播，它们在 RNNs 中还会随着时间反向传播。如果网络运行的时间过久的话，会让梯度变得非常不稳定，进而变得难以训练。幸运的是，我们可以用长短期记忆网络来克服这一点。该模型由 Hochreiter 和 Schmidhuber 在1997年提出，目的即是为了解决梯度不稳定问题。LSTMs 在训练的时候可以非常容易地得到更好的结果，最近的许多论文（包括我上面提到的一些）都利用到了 LSTMs 或者与之类似的想法。

深层信念网络、生成模型、以及波兹曼机（Deep belief nets, generative models, and Boltzmann machines）：业界对深度学习的兴趣始于2006年发表的一系列解释如何训练一种被称为 *深度信念网络（deep belief network, DBN）** 的论文。DBNs 流行了很多年，不过随着前馈网络和循环神经网络的风行，DBNs 热度下降了不少，不过其仍然有着许多非常有趣的特点。

> *参见 A fast learning algorithm for deep belief nets，作者是 Geoffrey Hinton、Simon Osindero 和 Yee-Whye Teh，发表于2006年，以及另外一篇相关的论文 Reducing the dimensionality of data with neural networks，作者是 Geoffrey Hinto 和 Ruslan Salakhutdinov，发表于2006年。

DBNs 有趣的一个点是它们是 *生成模型（generative model）* 的一种。在前馈网络中，之前的神经元激活决定着后面的神经元激活。而像 DBN 这样的生成模型，虽然也可以像前馈网络那样工作，但它可以指定某些神经元的值，然后“顺着网络反向传播”，生成输入激活的值。更具体地说，一个在手写数字图片上经过训练的 DBN，它可以（如果可以的话）去生成一些图片，这些图片看起来就像是真的手写数字一样。在这种思路下，生成模型就像人类的大脑一样：不仅可以读数字，还可以写数字。Geoffrey Hinto 对此有评论称，在识别形状之前，先学着生成图片。

DBNs 第二个有趣的点在于，它可以做无监督学习（unsupervised learning）和半监督学习（semi-supervised learning）。例如，在训练图片数据的时候，就算没有图片的标签，它也可以学习到非常多有用的特征，进而理解图片。无监督学习是一个非常非常有价值的科学问题，我们期冀它可以在实际问题中大放异彩。

既然 DBNs 有着这么多吸引人的优点，为什么这些年来它的热度逐渐减退了呢？一部分原因是诸如前馈网络和循环网络在图像和语音领域上的强大表现，所以人们的眼光被这些模型所吸引去也是可以理解的。不过令人遗憾的是，现在的现象就是赢者通吃，人们在哪些不再流行的模型和想法上工作是十分困难的，就算这些模型和想法还有开发和利用的价值。我个人的意见是，像 DNBs 这样的生成模型应该得到更多的关注。这里有一个 DBNs 的概述论文，你可以参考，还有，我发现这篇文章也写的非常的好。*本质上来说*，上面的讨论不仅包含深度信念网络，还包括受限波兹曼机等，后者是 DBNs 的一个关键组件。

其他想法：有没有其他的关于神经网络和深度学习的东西呢？当然有啦，最近热门的研究领域还包括使用神经网络来解决自然语言处理（natural language processing）（具体参考这篇综述论文），机器翻译，以及其他的一些应用例如音乐信息（music informatics）。在阅读本书之后，你应该有能力去独自了解最近的新进展了，当然，学习一些背景知识是必要的。

让我介绍一篇非常有趣的论文来结束本节的内容。这篇论文结合了深层卷积网络和强化学习，其目的是，玩电子游戏（还有一篇相关论文）！其想法是利用卷积网络来简化游戏屏幕上的像素，转化为一系列更简单的特征，利用这些特征，可以用来下达指令，例如：”向左“、”向右“、”攻击“等。这个网络在七个电子游戏上经过训练后，在其中三个游戏上甚至超过了人类玩家的表现。现在看来，这更像是一种噱头，这篇名为”Playing Atari with reinforcement learning“的论文毫无疑问经过了很好的市场营销。如果我们越过它外表的光鲜，探究其本质的话，会发现这个模型只考虑了原始的像素数据，它甚至不知道游戏规则！从数据中学习高质量的决策，在每个非常不同和对立的环境中，它都有着一系列复杂的规则。这听起来很不错。

神经网络的未来

意图驱动的用户交互（Intention-driven user interfaces）：这里有一个很老的笑话：一个没有耐心的教授告诉一个困惑的学生：“不要光听我说了什么；要理解我说的*意思*（don't listen to what I say; listen to what I *mean*）”。（译者注：非常好笑。）计算机经常被比喻为那个困惑的学生，不知道用户的真实意图。我还非常清楚地记得，有一次我在谷歌上进行搜索的时候，拼错了几个单词，但是页面上显示：“你搜索的是 [...] 吗？”，然后给出了正确的搜索结果。谷歌的 CEO Larry Page 有一次这样说到：“完美的搜索引擎应该准确地理解用户搜索的意图，然后返回用户想要看到的结果。”

这是一种 *意图驱动的用户交互* 方式。在这种方式下，引擎并不完全按照用户绝对的搜索关键词进行搜索，而是使用机器学习技术，输入用户模糊的、不确定的请求，确认其真实目的，然后输出用户想要的结果。

这种意图驱动的用户交互不仅可以用在搜索之中。在未来的几十年中，会有成千上万家公司利用机器学习技术来构建用户交互方式，捕获用户不准确的行为，给出用户真正想要的结果。其实我们已经看到了很多这样的例子：苹果的 Sir；Wolfram Alpha；IBM 的 Watson；可以注释图片和视频的系统；等等等等。

不过，大多数这样的产品将会失败。优秀的用户交互设计是非常困难的，我希望这些公司可以利用机器学习技术的强大之处来构建优秀的用户交互。如果你的产品的用户交互臭名昭著，就算是世界上最好的机器学习方法也不会奏效的。当然，会有一些产品最终获得成功。随着时间的增长，在人机交互中，我们会看到巨大的改变。在不久之前，例如2005年，用户们认为与计算机交互的话，他们需要输入完全正确的指令。确实，计算机执行的指令就是用户所发出的指令。我希望在接下来的日子里，人们可以开发出成功的意图驱动的用户交互方式，这可以显著地改变我们当前与计算机进行交互的方式。

机器学习、数据科学、以及创新的良性循环：机器学习并不只应用于意图驱动的用户交互之上，另外一个显著的例子是，在数据科学中机器学习被用来寻找隐藏在数据中的"已知的未知"信息。这是一个很流行的领域，已经有很多的论文对其进行了研究，在这里我不会对其介绍过多，但是有一点经常被人们忽略的问题我想提及一下：这些年来在机器学习上最大的突破不会是任何一个单独的概念突破，而是机器学习它本身变得有利可图，机器学习的盛行所带来的在各种数据科学及其他领域上的各种实际应用。如果一家公司在机器学习研究上投资 1 美元，它可以获得 1.1 美元的回报，那么，大量的资金将会涌入到机器学习研究中。换句话说，机器学习驱动了各种新领域的出现，各种新技术的发展。不断出现的各种专家、各种资源，将会推动机器学习走的更远，创造更多新的市场、新的领域、新的机会，这是一种创新的良性循环。

神经网络和深度学习所扮演的角色：我把机器学习描述为技术上新的机会的创造者。那么，神经网络和深度学习，它们在之中扮演着什么样的角色呢?

为了回答这个问题，让我们从历史中寻找答案。在上个世纪八十年代，人们对神经网络的前景具有非常大的信心，尤其是反向传播算法在广为人知之后。而在九十年代，这种热情逐渐消退了，机器学习的热点转移到了其他技术上，例如支持向量机等。如今，神经网络又一次地引起广泛地关注，创造了几乎所有的实验记录，在许多问题上几乎打败了所有的对手。但是谁又敢说，在未来不会有那么一个新的模型，重新抢过神经网络的风头呢?

基于这个原因，想象机器学习的未来要比想象神经网络的未来要容易的多了。现在我们对于神经网络的行为知之甚少，为什么神经网络的泛化性能会这么好呢？在给定这么多的参数的情况下，为什么神经网络还可以有效地避免过拟合呢？为什么随机梯度下降算法工作的那么好呢？如果数据集经过调整，神经网络还会表现的那么好吗？例如，如果 ImageNet 数据集扩大 10 倍，神经网络的表现相比于其他机器学习方法，会提升还是会下降？这些都是很简单，但是很重要的问题，但是到目前为止，我们了解的还很少。基于这些原因，神经网络在机器学习的未来中能扮演怎样的角色，目前还难以预测。

不过我作了这样一个预测：我相信深度学习会经久不衰，其学习层级概念（hierarchies of concepts）的能力，其构建多层抽象（multiple layers of abstraction）的能力，足以发挥十分重要的作用。这不是说未来的深度学习和今天的深度学习不会产生什么变化，无论是单元组件、结构框架、还是学习算法，我们已经看到了很多变化。也许，未来的焦点模型不再是神经网络，但它们仍将处于深度学习的范畴之内。

神经网络和深度学习会引领人工智能的发展吗？在本书中，我们在特定的问题上学习了神经网络，例如识别数字。让我们拓宽视野，问这样一个问题：计算机可以独立思考吗？（what about general-purpose thinking computers?）神经网络和深度学习可以帮助我们实现真正的人工智能吗？在现如今深度学习飞速发展的现状下，我们马上就要实现真正的人工智能了吗？

如果真的要回答这个问题，可能要写另外一本书了。相反，请允许我用一个现象来回答这个问题，这个想法又被称为康威定律（Conway's law）：

> **不管什么组织所设计的什么系统...将会不可避免地产生一种设计，这种设计的结构与这个组织的通信结构完全相同。**

（译者注：通俗的翻译就是说，组织结构等同于系统设计。）康威定律告诉我们，例如，波音747飞机的设计与波音公司的组织结构和其承包商的结构相同。或者，另外一个简单的例子，设想一个公司设计了一个复杂的软件应用。如果这个软件想增加某种机器学习算法的功能，那么这个功能的设计者就会和公司的机器学习专家进行讨论，来决定这个功能的设计。康威定律就是这样一个现象。

在第一次听到康威定律的时候，大多数人都感到惊奇：“啊？康威定律就是这么无聊和简单？”，或者说：”这可能不对吧？“让我从反对意见开始做一些解释，考虑下面的问题：波音公司的会计部门会影响747飞机的设计吗？卫生部门会影响吗？餐饮部门会影响吗？确实，这些部门可能不会直接影响波音747飞机的设计。我们应该把康威定律定义在那些直接会影响设计和工程的部门。

让我们看看另外一个反对意见：康威定律显然易见，而且无关紧要。对于那些经常违背康威定律的组织来说，这也许是真的，但我并不这样认为。那些构建新产品的部门，经常挤满了无关的人员，或者缺少某些重要的专家。想想那些充斥着复杂无用的功能的软件，或者那些有着明显的重大缺陷的软件，例如，极度糟糕的用户界面等。导致这些问题的原因是缺少能设计出好的产品的团队。康威定律或许显然

易见，但这并不意味着人们可以随随便便不把它当回事。

康威定律适用于系统的设计和工程，我们对这样的设计和工程通常都有着很好的理解，它们需要什么样的组件，要如何去构建这样的组件等。但是定律并不能直接应用到人工智能的开发之上，因为人工智能并不是这样的一个问题：我们不知道用什么样的东西可以构建一个人工智能系统。甚至，我们都不知道要回答什么问题。可以这样说，相比于工程问题，人工智能更像是一个科学问题。在设计747飞机的时候，我们需要知道飞机引擎的知识，或者空气动力学的知识。而对于人工智能，你该雇佣什么样的专家呢？就像 Wernher von Braun 说的那样："基础的研究工作是我不知道我在做什么。"有没有一种康威定律，它更适用于科学问题而不是工程问题呢？

为了理解这个问题，设想医学的历史。在早先的时候，医学的从业者，就像 Galen 和 Hippocrates 那样，用自己的身体来做试验。随着人类知识的增长，人们的工作开始趋向于专业化。我们发现了许多深层次（deep*）的新想法：例如疾病领域中的微生物理论，人体中抗体的工作方式，或者是对心脏、肺、血管、动脉等构成了人体完整的心血管系统的认识等。这样深层次的发现是其他某些子领域的基础，例如流行病学、免疫学等等。所以我们的知识结构影响着医学科学的结构。这个现象在免疫学上尤其显著：认识到免疫系统的存在，认识到免疫系统的重要性，绝非是什么不足挂齿的事情。自此我们拥有了完整的医学领域：各类专家、各类学术会议、各类奖项等等，它们又某种所看不见的现象所组织，这个现象并不是一个可以描述出来的事情。

> *抱歉，我可能过于滥用 deep 这个词语了。"深层次的想法"并不是一个准确的定义，大致来说，我的意思是这样的想法是那些非常重要非常基础的想法。反向传播算法，或者微生物理论等，都是很好的例子。

这样的现象已经在许多科学领域中不断的重现了：物理学、数学、化学等。这些领域在一开始是一个整体，早期的专家可以掌握所有的领域知识。随着时间的不断增长，这种整体性（monolithic）随之瓦解，我们发现了更多深层次的想法，多到一个人根本没有足够的时间和能力去全部掌握。所以，这些学科开始重新组织、分化，最终得到一个又一个的子领域，复杂，循环，自我参照的结构，它们的结构恰恰是我们的这些深层次想法结构的重现。*所以我们的知识结构影响着科学的组织结构。这些结构进一步约束和帮助我们的发现*。这即是康威定律在科学领域上的类比。

那么，康威定律会如何影响深度学习或人工智能呢？

在人工智能的早期阶段，人们已经开始了部分争论，一些人说："实现真正的人工智能并不困难，我们已经拥有了[秘密武器]"，另外一些人说："只有[秘密武器]是不够的"。在他们所说的[秘密武器]中，最新的是深度学习*；以前还有 logic，或者 Prolog，或者专家系统，或者当时那个年代最先进的什么东西。这些争论的问题在于，他们并没有告诉你这些秘密武器到底有多强大。虽然我们刚刚结束了一章介绍深度学习的文章，明白了深度学习可以解决很多非常复杂的问题。深度学习看起来令人激动，也很有前

途，但当年的那些技术，不管是 Prolog，还是 Eurisko，还是专家系统，它们在它们那个年代看起来同样令人激动，同样前途无限。我们可以说深度学习确实与上面的那些方法有着本质上的不同吗？有没有某种手段，来衡量这些想法和技术呢？康威定律似乎可以帮助我们粗略地评估想法和技术的结构复杂性。

> *有趣的是，深度学习的专家一般比较保守，外行人倒是信心十足。例如，Yann LeCun 的一篇唱反调的评论。

所以，这里有两个问题需要得到回答。第一，根据这种复杂性衡量手段，深度学习中的那些技术到底有多强大呢？第二，为了构建出真正的人工智能，我们到底需要多么强大的一个理论？

当我们在今天审视深度学习的时候，它是一个激动人心、飞速发展，但相对来说也是一个非常整体（monolithic）的领域。深度学习中确实已经有了一些深层次的想法，也有了一些虽然与其他领域交叉很大的学术会议。一篇又一篇基于相同想法的论文不断地被发表出来：使用随机梯度下降去优化代价函数。这些想法确实很成功。但是我们还没有看到深度学习中有更多的子领域被发掘出来，这些子领域会各自研究一个深层次的问题，从而推动深度学习向不同的方向发展。所以，根据这种对学科复杂性的评估方法，深度学习，还只是一个相对来说，很单一、浅薄的领域。对于一个深度学习领域的专家来说，掌握深度学习中所有的方法并非不可能。

对于第二个问题：我们到底需要多么强大的理论和想法，才能实现真正的人工智能？当然，没有人能回答这个问题。在本书的附录中，我会总结一些关于这个问题的已有的评论。我认为，就算现在人工智能已经成了炙手可热的领域，它还有着非常非常长的路要走。就如康威定律所昭示的那样，为了达到我们所期望的那个程度，除了神经网络，或者深度学习，人工智能还需要更多更复杂、更强大的子学科。我相信，如果使用深度学习来构建真正的人工智能的话，我们至少还有几十年的路要走。

为了作出这些试验性的评论，我已经遇到了不少困难。这些评论也许看起来很明显，也没有一个确切的结论。这对于追求确定性的人来说并不是一个好消息。我在网上看了很多评论，我看到许许多多的人在谈论着人工智能，他们看起来对于自己的观点都非常自信，甚至有点武断，其观点都是基于某种不存在或者劣质的证据之上。坦白地说：现在抛出定论还为时过早。如同那个老笑话一样，如果你问一个科学家他的进展如何，还需要多少年的时候，他会说：“还要十年！”（或者更多），他的意思其实是说：“我根本还没有想法呢！”对于人工智能，不止是十年，人们已经研究超过六十年了。在另一方面，我们对深度学习知之甚少，还有很多基础的、重要的问题没有解决，这既是挑战，又是机遇。

Next:
附录：有没有一个简单的人工智能算法？