

Deeplearning4J（简称 DL4J）是一个基于 Java 和 Scala 的开源的分布式深度学习库，由 Skymind 于 2014 年 6 月发布，其核心目标是创建一个即插即用的解决方案原型。埃森哲、雪弗兰、博斯咨询和 IBM 等都是 DL4J 的客户。DL4J 拥有一个多用途的 n-dimensional array 的类，可以方便地对数据进行各种操作；拥有多种后端计算核心，用以支持 CPU 及 GPU 加速，在图像识别等训练任务上的性能与 Caffe 相当；可以与 Hadoop 及 Spark 自动整合，同时可以方便地在现有集群（包括但不限于 AWS, Azure 等）上进行扩展，同时 DL4J 的并行化是根据集群的节点和连接自动优化，不像其他深度学习库那样可能需要用户手动调整。DL4J 选择 Java 作为其主要语言的原因是，目前基于 Java 的分布式计算、云计算、大数据的生态非常庞大。用户可能拥有大量的基于 Hadoop 和 Spark 的集群，因此在这类集群上搭建深度学习平台的需求便很容易被 DL4J 满足。同时 JVM 的生态圈内还有数不胜数的 Library 的支持，而 DL4J 也创建了 ND4J，可以说是 JVM 中的 NumPy，支持大规模的矩阵运算。此外，DL4J 还有商业版的支持，付费用户在出现问题时可以通过电话咨询寻求支持。

2.2.11 Chainer

官方网址：chainer.org

GitHub：github.com/pfnet/chainer

Chainer 是由日本公司 Preferred Networks 于 2015 年 6 月发布的深度学习框架。Chainer 对自己的特性描述如下。

(1) Powerful：支持 CUDA 计算，只需要几行代码就可以使用 GPU 加速，同时只需少许改动就可以运行在多 GPU 上。

(2) Flexible：支持多种前馈神经网络，包括卷积网络、循环网络、递归网络，支持运行中动态定义的网络（Define-by-Run）。

(3) Intuitive：前馈计算可以引入 Python 的各种控制流，同时反向传播时不受干扰，简化了调试错误的难度。

绝大多数的深度学习框架是基于“Define-and-Run”的，也就是说，需要首先定义一个网络，再向网络中 feed 数据（mini-batch）。因为网络是预先静态定义的，所有的控制逻辑都需要以 data 的形式插入网络中，包括像 Caffe 那样定义好网络结构文件，或者像 Theano、Torch、TensorFlow 等使用编程语言定义网络。而 Chainer 则相反，网络是在实际

运行中定义的，Chainer 存储历史运行的计算结果，而不是网络的结构逻辑，这样就可以方便地使用 Python 中的控制流，所以无须其他工作就可以直接在网络中使用条件控制和循环。

2.2.12 Leaf

官方网址：autumnai.com/leaf/book

GitHub：github.com/autumnai/leaf

Leaf 是一个基于 Rust 语言的直观的跨平台的深度学习乃至机器智能框架，它拥有一个清晰的架构，除了同属 Autumn AI 的底层计算库 Collenchyma，Leaf 没有其他依赖库。它易于维护和使用，并且拥有非常高的性能。Leaf 自身宣传的特点是为 Hackers 定制的，这里的 Hackers 是指希望用最短的时间和最少的精力实现机器学习算法的技术极客。它的可移植性非常好，可以运行在 CPU、GPU 和 FPGA 等设备上，可以支持有任何操作系统的 PC、服务器，甚至是是没有操作系统的嵌入式设备，并且同时支持 OpenCL 和 CUDA。Leaf 是 Autumn AI 计划的一个重要组件，后者的目标是让人工智能算法的效率提高 100 倍。凭借其优秀的设计，Leaf 可以用来创建各种独立的模块，比如深度强化学习、可视化监控、网络部署、自动化预处理和大规模产品部署等。

Leaf 拥有最简单的 API，希望可以最简化用户需要掌握的技术栈。虽然才刚诞生不久，Leaf 就已经跻身最快的深度学习框架之一了。图 2-9 所示为 Leaf 官网公布的各个框架在单 GPU 上训练 VGG 网络的计算时间（越小越好）的对比（这是和早期的 TensorFlow 对比，最新版的 TensorFlow 性能已经非常好了）。

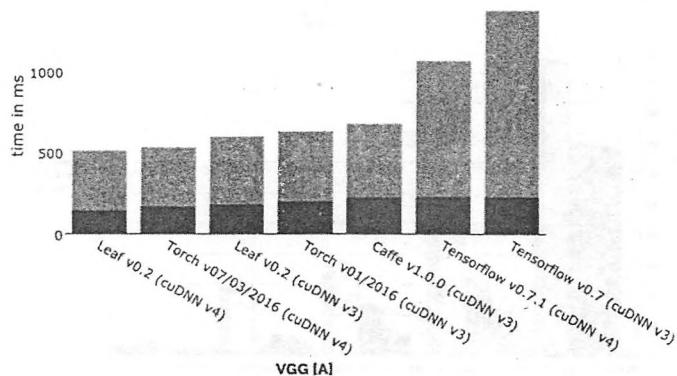


图 2-9 Leaf 和各深度学习框架的性能对比（深色为 forward，浅色为 backward）

2.2.13 DSSTNE

GitHub: github.com/amznlabs/amazon-dsstne

DSSTNE (Deep Scalable Sparse Tensor Network Engine) 是亚马逊开源的稀疏神经网络框架，在训练非常稀疏的数据时具有很大的优势。DSSTNE 目前只支持全连接的神经网络，不支持卷积网络等。和 Caffe 类似，它也是通过写一个 JSON 类型的文件定义模型结构，但是支持非常大的 Layer（输入和输出节点都非常多）；在激活函数、初始化方式及优化器方面基本都支持了 state-of-the-art 的方法，比较全面；支持大规模分布式的 GPU 训练，不像其他框架一样主要依赖数据并行，DSSTNE 支持自动的模型并行（使用数据并行需要在训练速度和模型准确度上做一定的 trade-off，模型并行没有这个问题）。

在处理特征非常多（上亿维）的稀疏训练数据时（经常在推荐、广告、自然语言处理任务中出现），即使一个简单的 3 个隐层的 MLP（Multi-Layer Perceptron）也会变成一个有非常多的参数的模型（可能高达上万亿）。以传统的稠密矩阵的方式训练方法很难处理这么多的模型参数，更不必提超大规模的数据量，而 DSSTNE 有整套的针对稀疏数据的优化，率先实现了对超大稀疏数据训练的支持，同时在性能上做了非常大的改进。

在 DSSTNE 官方公布的测试中，DSSTNE 在 MovieLens 的稀疏数据上，在单 M40 GPU 上取得了比 TensorFlow 快 14.8 倍的性能提升（注意是和老版的 TensorFlow 比较），如图 2-10 所示。一方面是因为 DSSTNE 对稀疏数据的优化；另一方面是 TensorFlow 在数据传输到 GPU 上时花费了大量时间，而 DSSTNE 则优化了数据在 GPU 内的保留；同时 DSSTNE 还拥有自动模型并行功能，而 TensorFlow 中则需要手动优化，没有自动支持。

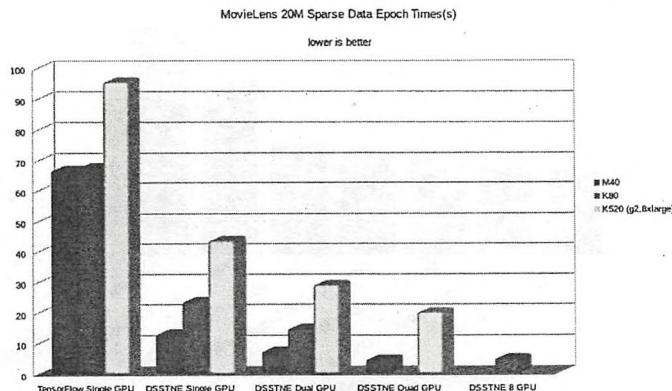
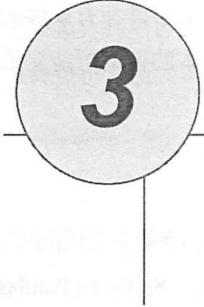


图 2-10 DSSTNE 在稀疏数据上与 TensorFlow 的性能对比



3

TensorFlow 第一步

前两章我们讲了 TensorFlow 的核心概念和编程模型，又谈了 TensorFlow 和其他深度学习框架的异同。本章将直奔主题，我们将学会安装 TensorFlow，然后使用 TensorFlow (1.0.0-rc0) 训练一个手写数字识别 (MNIST) 的模型。

3.1 TensorFlow 的编译及安装

TensorFlow 的安装方式没有 Theano 那么直接，因为它并不是全部由 Python 写成的库，底层有很多 C++ 乃至 CUDA 的代码，因此某些情况下可能需要编译安装（比如你的 gcc 版本比较新，硬件环境比较特殊，或者你使用的 CUDA 版本不是 release 版预编译的）。通常安装 TensorFlow 分为两种情况，一种是只使用 CPU，安装相对容易；另一种是使用 GPU，这种情况还需要安装 CUDA 和 cuDNN，情况相对复杂。然而不管哪种情况，我们都推荐使用 Anaconda²⁰作为 Python 环境，因为可以避免大量的兼容性问题。另外，本书默认使用 Python 3.5 作为 Python 的基础版本，相比 Python 2.7，它更代表了 Python 未来的发展趋势。TensorFlow 目前支持得比较完善的是 Linux 和 Mac（对 Windows 的支持还不太全面）。因为 Mac 系统主要使用 CPU 版本（Mac 系统很少有使用 NVIDIA 显卡的，而目前 TensorFlow 对 CUDA 支持得比较好，对 OpenCL 的支持还属于实验性质），安装方式和 Linux 的 CPU 版基本一致，而 Mac 一般没有 NVIDIA 的显卡，所以不适合使用

GPU 版本。本章将主要讲解在 Linux 下安装 TensorFlow 的过程。另外，本书基于 2017 年 1 月发布的 TensorFlow 1.0.0-rc0 版，旧版本在运行本书的代码时可能会有不兼容的情况，所以建议读者都安装这个版本或更新版本的 TensorFlow。此外，本书推荐有条件的读者使用 GPU 版本，因为在训练大型网络或者大规模数据时，CPU 版本的速度可能会很慢。

3.1.1 安装 Anaconda

Anaconda 是 Python 的一个科学计算发行版，内置了数百个 Python 经常会使用的库，也包括许多我们做机器学习或数据挖掘的库，包括 Scikit-learn、NumPy、SciPy 和 Pandas 等，其中可能有一些还是 TensorFlow 的依赖库。我们在安装这些库时，通常都需要花费不少时间编译，而且经常容易出现兼容性问题，Anaconda 提供了一个编译好的环境可以直接安装。同时 Anaconda 自动集成了最新版的 MKL（Math Kernel Library）库，这是 Intel 推出的底层数值计算库，功能上包含了 BLAS（Basic Linear Algebra Software）等矩阵运算库的功能，可以作为 NumPy、SciPy、Scikit-learn、NumExpr 等库的底层依赖，加速这些库的矩阵运算和线性代数运算。简单来说，Anaconda 是目前最好的科学计算的 Python 环境，方便了安装，也提高了性能。本书强烈推荐安装 Anaconda，接下来的章节也将默认读者使用 Anaconda 作为 TensorFlow 的 Python 环境。

(1) 我们在 Anaconda 的官网上 (www.continuum.io/downloads) 下载 Anaconda3 4.2.0 版，请读者根据自己的操作系统下载对应版本的 64 位的 Python 3.5 版。

(2) 我们在 Anaconda 的下载目录执行以下命令（请根据下载的文件替换对应的文件名）。

```
bash Anaconda3-4.2.0-Linux-x86_64.sh
```

(3) 接下来我们会看到安装提示，直接按回车键确认进入下一步。然后我们会进入 Anaconda 的 License 文档，这里直接按 q 键跳过，然后输入 yes 确认。下面的这一步会让我们输入 anaconda3 的安装路径，没有特殊情况的话，我们可以按回车键使用默认路径，然后安装就自动开始了。

(4) 安装完成后，程序提示我们是否把 anaconda3 的 binary 路径加入到.bashrc，读者可以根据自己的情况考虑，建议添加，这样以后 python 和 ipython 命令就会自动使用 Anaconda Python3.5 的环境了。

3.1.2 TensorFlow CPU 版本的安装

TensorFlow 的 CPU 版本相对容易安装，一般分为两种情况：第一种情况，安装编译好的 release 版本，推荐大部分用户安装这种版本；第二种情况，使用 1.0.0-rc0 分支源码编译安装，当用户的系统比较特殊，比如 gcc 版本比较新（gcc 6 以上），或者不支持使用编译好的 release 版本，才推荐这样安装。

第一种情况，安装编译好的 release 版本，我们可以简单地执行下面这个命令。python 的默认包管理器是 pip，直接使用 pip 来安装 TensorFlow。对于 Mac 或 Windows 系统，可在 TensorFlow 的 GitHub 仓库上的 Download and Setup 页面查看编译好的程序的地址。

```
export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-1.0.0rc0-cp35-cp35m-linux_x86_64.whl
pip install --upgrade $TF_BINARY_URL
```

第二种情况，使用 1.0.0-rc0 分支的源码编译安装。

此时，确保系统安装了 gcc（版本最好介于 4.8~5.4 之间），如果没有安装，请根据自己的系统情况先安装 gcc，本节不再赘述。此外，为了编译 TensorFlow，我们还需要有 Google 自家的编译工具 bazel（github.com/bazelbuild/bazel），根据其安装教程（www.bazel.io/versions/master/docs/install.html）直接安装它的 v0.43 release 版本即可，不需要使用最新的 dev 版本的功能。

在正确地安装完 gcc 和 bazel 之后，接下来我们正式开始编译安装 TensorFlow，首先先下载 TensorFlow 1.0.0-rc0 的源码：

```
wget https://github.com/tensorflow/tensorflow/archive/v1.0.0-rc0.tar.gz
tar -xzvf v1.0.0-rc0.tar.gz
```

完成下载之后，进入 TensorFlow 代码仓库的目录，然后执行下面的命令进行配置：

```
cd tensorflow-1.0.0-rc0
./configure
```

接下来的输出要选择 python 路径，确保是 anaconda 的 python 路径即可：

```
Please specify the location of python. [Default is /home/wenjian/anaconda3/bin/python]:
```

这里选择 CPU 编译优化选项，默认的-march=native 将选择本地 CPU 能支持的最佳配置，比如 SSE4.2、AVX 等。建议选择默认值。

```
Please specify optimization flags to use during compilation [Default is -march=native]:
```

选择是否使用 jemalloc 作为默认的 malloc 实现(仅限 Linux)，建议选择默认设置。

```
Do you wish to use jemalloc as the malloc implementation? (Linux only) [Y/n]
```

然后它会让我们选择是否开启对 Google Cloud Platform 的支持，这个在国内一般是访问不到的，有需要的用户可以选择支持，通常选 N 即可：

```
Do you wish to build TensorFlow with Google Cloud Platform support? [y/N]
```

它会询问是否需要支持 Hadoop File System，如果有读取 HDFS 数据的需求，请选 y 选项，否则就选默认的 N 即可：

```
Do you wish to build TensorFlow with Hadoop File System support? [y/N]
```

选择是否开启 XLA JIT 编译功能支持。这里 XLA 是 TensorFlow 目前实验性的 JIT (Just in Time)、AOT (Ahead of Time) 编译优化功能，还不太成熟，有探索欲望的读者可以尝试开启。

```
Do you wish to build TensorFlow with the XLA just-in-time compiler (experimental)? [y/N]
```

然后它会让我们选择 python 的 library 路径，这里依然选择 anaconda 的路径：

```
Please input the desired Python library path to use. Default is  
[/home/wenjian/anaconda3/lib/python3.5/site-packages]
```

接着选择不需要使用 GPU，即 OpenCL 和 CUDA 全部选 N：

```
Do you wish to build TensorFlow with OpenCL support? [y/N]
```

```
Do you wish to build TensorFlow with CUDA support? [y/N]
```

之后可能需要下载一些依赖库的文件，完成后 configure 就顺利结束了，接下来使用编译命令执行编译：

```
bazel build --copt=-march=native -c opt //tensorflow/tools/pip_package:build_pip_package
```

编译结束后，使用下面的命令生成 pip 安装包：

```
bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg
```

最后，使用 pip 命令安装 TensorFlow：

```
pip install /tmp/tensorflow_pkg/tensorflow-1.0.0rc0-cp35-cp35m-linux_x86_64.whl
```

3.1.3 TensorFlow GPU 版本的安装

TensorFlow 的 GPU 版本安装相对复杂。目前 TensorFlow 仅对 CUDA 支持较好，因此我们首先需要一块 NVIDIA 显卡，AMD 的显卡只能使用实验性支持的 OpenCL，效果不是很好。接下来，我们需要安装显卡驱动、CUDA 和 cuDNN。

CUDA 是 NVIDIA 推出的使用 GPU 资源进行通用计算(Genral Purpose GPU)的 SDK，CUDA 的安装包里一般集成了显卡驱动，我们直接去官网下载 NVIDIA CUDA (<https://developer.nvidia.com/cuda-toolkit>)。

在安装前，我们需要暂停当前 NVIDIA 驱动的 X server，如果是远程连接的 Linux 机器，可以使用下面这个命令关闭 X server：

```
sudo init 3
```

之后，我们将 CUDA 的安装包权限设置成可执行的，并执行安装程序：

```
chmod u+x cuda_8.0.44_linux.run
sudo ./cuda_8.0.44_linux.run
```

接下来我们正式进入 CUDA 的安装过程，先按 q 键跳过开头的 license 说明，接着输入 accept 接收协议，然后按 y 键选择安装驱动程序：

```
Install NVIDIA Accelerated Graphics Driver for Linux-x86_64 367.48?
(y)es/(n)o/(q)uit:
```

按 y 键选择安装 CUDA 并确认安装路径，一般可直接使用默认地址：

```
Install the CUDA 8.0 Toolkit?
```

```
(y)es/(n)o/(q)uit:  
Enter Toolkit Location  
[ default is /usr/local/cuda-8.0 ]:
```

按 n 键不选择安装 CUDA samples (我们只是通过 TensorFlow 调用 CUDA, 不直接写 CUDA 代码):

```
Install the CUDA 8.0 Samples?  
(y)es/(n)o/(q)uit:
```

最后等待安装程序完成。

接下来安装 cuDNN, cuDNN 是 NVIDIA 推出的深度学习中 CNN 和 RNN 的高度优化的实现。因为底层使用了很多先进技术和接口 (没有对外开源), 因此比其他 GPU 上的神经网络库性能要高不少, 目前绝大多数的深度学习框架都使用 cuDNN 来驱动 GPU 计算。我们先从官网下载 cuDNN (<https://developer.nvidia.com/rdp/cudnn-download>), 这一步可能需要先注册 NVIDIA 的账号并等待审核 (需要一段时间)。

接下来再安装 cuDNN, 我们到 cuda 的安装目录执行解压命令:

```
cd /usr/local  
sudo tar -xzvf ~/downloads/cudnn-8.0-linux-x64-v5.1.tgz
```

这样就完成了 cuDNN 的安装, 但我们可能还需要在系统环境里设置 CUDA 的路径:

```
vim ~/.bashrc  
export LD_LIBRARY_PATH=/usr/local/cuda-8.0/lib64:/usr/local/cuda-8.0/extras/  
CUPTI/lib64:$LD_LIBRARY_PATH  
export CUDA_HOME=/usr/local/cuda-8.0  
export PATH=/usr/local/cuda-8.0/bin:$PATH  
source ~/.bashrc
```

接下来, 我们开始安装 TensorFlow。对 GPU 版的 TensorFlow, 官网也提供了预编译的包, 但是这个预编译版对本地的各种依赖环境支持可能不是最佳的, 如果读者试用过没有任何兼容性问题, 可以直接安装预编译版的 TensorFlow:

```
export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow_gpu-1.0.0rc0-cp35-cp35m-linux_x86_64.whl
```

```
pip install --upgrade $TF_BINARY_URL
```

如果预编译的版本不支持当前的 CUDA、cuDNN 版本，或者存在其他兼容性问题，可以进行编译安装，和前面提到的 CPU 版本的编译安装类似，我们需要先安装 gcc 和 bazel，接下来下载 TensorFlow 1.0.0-rc0 的代码，然后使用配置程序(`./configure`)进行编译配置，前面几步和 CPU 版本的安装完全一致，直到选择是否支持 CUDA 这一步：

```
Do you wish to build TensorFlow with CUDA support? [y/N]
```

我们按 y 键选择支持 GPU，接下来选择指定的 gcc 编译器，一般选默认设置就好。

```
Please specify which gcc should be used by nvcc as the host compiler. [Default is /usr/bin/gcc]:
```

接下来选择要使用的 CUDA 版本、CUDA 安装路径、cuDNN 版本和 cuDNN 的安装路径，这里使用的是 CUDA 8.0 版本，所以 CUDA SDK Version 设置为 8.0，路径设置为`/usr/local/cuda-8.0`，cuDNN Version 设置为 5.1，cuDNN 路径也设置为`/usr/local/cuda-8.0`：

```
Please specify the Cuda SDK version you want to use, e.g. 7.0. [Leave empty to use system default]:
```

```
Please specify the location where CUDA toolkit is installed. Refer to README.md for more details. [Default is /usr/local/cuda]:
```

```
Please specify the Cudnn version you want to use. [Leave empty to use system default]:
```

```
Please specify the location where cuDNN library is installed. Refer to README.md for more details. [Default is /usr/local/cuda]:
```

最后将选择 GPU 的 compute capability (CUDA 的计算兼容性)，不同的 GPU 可能有不同的 compute capability，我们可以在官网查到具体数值，比如 GTX 1080 和新 Titan X 是 6.1，而 GTX 980 和旧版的 GTX Titan X 是 5.2。

```
Please note that each additional compute capability significantly increases your build time and binary size.
```

```
[Default is: "3.5,5.2"]:
```

至此，配置完成，配置程序可能会开始下载对应的需要其他库的代码仓库，我们耐心等待一会儿就好。

接下来，开始编译 GPU 版本的 TensorFlow，执行下面这个命令，注意和 CPU 版本的编译相比，这里多了一个--config=cuda：

```
bazel build --copt=-march=native -c opt --config=cuda //tensorflow/tools/pip_package:build_pip_package
```

编译大概需要花费一段时间，之后执行命令生成 pip 安装包并进行安装：

```
bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg  
pip install /tmp/tensorflow_pkg/tensorflow-1.0.0rc0-cp35-cp35m-linux_x86_64.whl
```

3.2 TensorFlow 实现 Softmax Regression 识别手写数字

3.1 节介绍了安装 TensorFlow，接下来我们就以一个机器学习领域的 Hello World 任务——MNIST 手写数字识别来探索 TensorFlow。MNIST²¹ (Mixed National Institute of Standards and Technology database) 是一个非常简单的机器视觉数据集，如图 3-1 所示，它由几万张 28 像素×28 像素的手写数字组成，这些图片只包含灰度值信息。我们的任务就是对这些手写数字的图片进行分类，转成 0~9 一共 10 类。

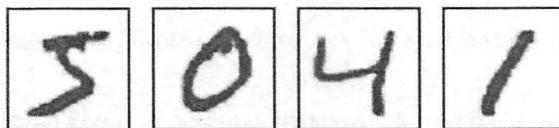


图 3-1 MNIST 手写数字图片示例

首先对 MNIST 数据进行加载，TensorFlow 为我们提供了一个方便的封装，可以直接加载 MNIST 数据成我们期望的格式，在 ipython 命令行或者 spyder 中直接运行下面的代码。本节代码主要来自 TensorFlow 的开源实现²²。

```
from tensorflow.examples.tutorials.mnist import input_data  
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

然后查看 mnist 这个数据集的情况，可以看到训练集有 55000 个样本，测试集有 10000 个样本，同时验证集有 5000 个样本。每一个样本都有它对应的标注信息，即 label。我们将在训练集上训练模型，在验证集上检验效果并决定何时完成训练，最后我们在测试集评

测模型的效果（可通过准确率、召回率、F1-score 等评测）。

```
print(mnist.train.images.shape, mnist.train.labels.shape)
print(mnist.test.images.shape, mnist.test.labels.shape)
print(mnist.validation.images.shape, mnist.validation.labels.shape)
```

前面提到我们的图像是 28 像素×28 像素大小的灰度图片，如图 3-2 所示。空白部分全部为 0，有笔迹的地方根据颜色深浅有 0 到 1 之间的取值。同时，我们可以发现每个样本有 784 维的特征，也就是 28×28 个点的展开成 1 维的结果 ($28 \times 28 = 784$)。因此，这里丢弃了图片的二维结构方面的信息，只是把一张图片变成一个很长的 1 维向量。读者可能会问，图片的空间结构信息不是很有价值吗，为什么我们要丢弃呢？因为这个数据集的分类任务比较简单，同时也是我们使用 TensorFlow 的第一次尝试，我们不需要建立一个太复杂的模型，所以简化了问题，丢弃空间结构的信息。后面的章节将使用卷积神经网络对空间结构信息进行利用，并取得更高的准确率。我们将图片展开成 1 维向量时，顺序并不重要，只要每一张图片都是用同样的顺序进行展开的就可以。

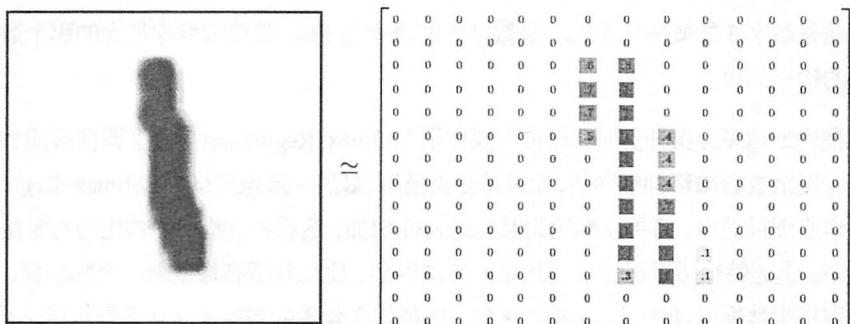


图 3-2 手写数字灰度信息示例

我们的训练数据的特征是一个 55000×784 的 Tensor，第一个维度是图片的编号，第二个维度是图片中像素点的编号，如图 3-3 所示。同时训练的数据 Label 是一个 55000×10 的 Tensor，如图 3-4 所示，这里是对 10 个种类进行了 one-hot 编码，Label 是一个 10 维的向量，只有 1 个值为 1，其余为 0。比如数字 0，对应的 Label 就是 $[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$ ，数字 5 对应的 Label 就是 $[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]$ ，数字 n 就代表对应位置的值为 1。

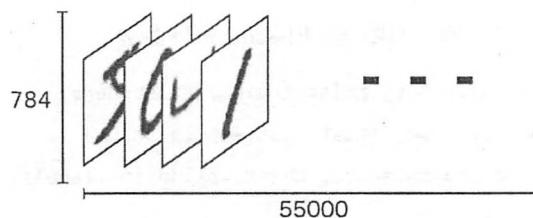


图 3-3 MNIST 训练数据的特征

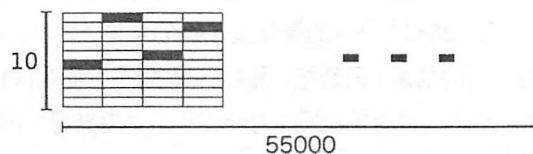


图 3-4 MNIST 训练数据的 Label

准备好数据后，接下来就要设计算法了，这里使用一个叫作 Softmax Regression 的算法训练手写数字识别的分类模型。我们的数字都是 0~9 之间的，所以一共有 10 个类别，当我们的模型对一张图片进行预测时，Softmax Regression 会对每一种类别估算一个概率：比如预测是数字 3 的概率为 80%，是数字 5 的概率为 5%，最后取概率最大的那个数字作为模型的输出结果。

当我们处理多分类任务时，通常需要使用 Softmax Regression 模型。即使后面章节的卷积神经网络或者循环神经网络，如果是分类模型，最后一层也同样是 Softmax Regression。它的工作原理很简单，将可以判定为某类的特征相加，然后将这些特征转化为判定是这一类的概率。上述特征可以通过一些简单的方法得到，比如对所有像素求一个加权和，而权重是模型根据数据自动学习、训练出来的。比如某个像素的灰度值大代表很可能是数字 n 时，这个像素的权重就很大；反之，如果某个像素的灰度值大代表不太可能是数字 n 时，这个像素的权重就可能是负的。图 3-5 所示为这样的一些特征，其中明亮区域代表负的权重，灰暗区域代表正的权重。

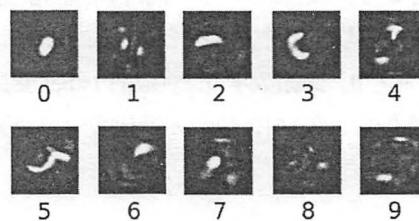


图 3-5 不同数字可能对应的特征权重

我们可以将这些特征写成如下公式： i 代表第 i 类， j 代表一张图片的第 j 个像素。 b_i 是 bias，顾名思义就是这个数据本身的一些倾向，比如大部分数字都是 0，那么 0 的特征对应的 bias 就会很大。

$$\text{feature}_i = \sum_j W_{i,j} x_j + b_i$$

接下来对所有特征计算 softmax，结果如下。简单说就是都计算一个 \exp 函数，然后在进行标准化（让所有类别输出的概率值和为 1）。

$$\text{softmax}(x) = \text{normalize}(\exp(x))$$

其中判定为第 i 类的概率就可由下面的公式得到。

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

我们先对各个类的特征求 \exp 函数，然后对它们标准化，使得和为 1，特征的值越大的类，最后输出的概率也越大；反之，特征的值越小的类，输出的概率也越小。最后的标准化操作保证所有的概率没有为 0 或者为负数的，同时它们的和为 1，也满足了概率的分布。如果将整个计算过程可视化，结果如图 3-6 所示。

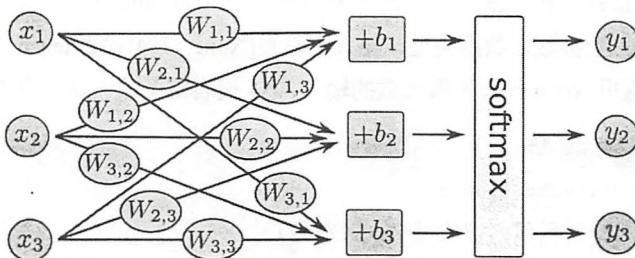


图 3-6 Softmax Regression 的流程

接着，如果将图 3-6 中的连线变成公式，结果如图 3-7 所示，最后将元素相乘变成矩阵乘法，结果如图 3-8 所示。

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \begin{bmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{bmatrix}$$

图 3-7 Softmax Regression 元素乘法示例

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

图 3-8 Softmax Regression 矩阵乘法示例

上述矩阵运算表达写成公式的话，可以用下面这样简洁的一行表达。

$$y = \text{softmax}(Wx + b)$$

接下来就使用 TensorFlow 实现一个 Softmax Regression。其实在 Python 中，当还没有 TensorFlow 时，通常使用 NumPy 做密集的运算操作。因为 NumPy 是使用 C 和一部分 fortran 语言编写的，并且调用 openblas、mkl 等矩阵运算库，因此效率很高。其中每一个运算操作的结果都要返回到 Python 中，但不同语言之间传输数据可能会带来比较大的延迟。TensorFlow 同样也把密集的复杂运算搬到 Python 外执行，不过做得更彻底。TensorFlow 通过定义一个计算图将所有的运算操作全部运行在 Python 外面，比如通过 C++ 运行在 CPU 上或者通过 CUDA 运行在 GPU 上，而不需要每次把运算完的数据传回 Python。

首先载入 TensorFlow 库，并创建一个新的 InteractiveSession，使用这个命令会将这个 session 注册为默认的 session，之后的运算也默认跑在这个 session 里，不同 session 之间的数据和运算应该都是相互独立的。接下来创建一个 Placeholder，即输入数据的地方。Placeholder 的第一个参数是数据类型，第二个参数 [None, 784] 代表 tensor 的 shape，也就是数据的尺寸，这里 None 代表不限条数的输入，784 代表每条输入是一个 784 维的向量。

```
import tensorflow as tf
sess = tf.InteractiveSession()
x = tf.placeholder(tf.float32, [None, 784])
```

接下来要给 Softmax Regression 模型中的 weights 和 biases 创建 Variable 对象，第 1 章中提到 Variable 是用来存储模型参数的。不同于存储数据的 tensor 一旦使用掉就会消失，Variable 在模型训练迭代中是持久化的（比如一直存放在显存中），它可以长期存在并且在每轮迭代中被更新。我们把 weights 和 biases 全部初始化为 0，因为模型训练时会自动学习合适的值，所以对这个简单模型来说初始值不太重要。不过对复杂的卷积网络、循环网络或者比较深的全连接网络，初始化的方法就比较重要，甚至可以说至关重要。注意这里 W 的 shape 是 [784, 10]，784 是特征的维数，而后面的 10 代表有 10 类，因为 Label 在 one-hot 编码后是 10 维的向量。

```
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
```

接下来就要实现 Softmax Regression 算法，我们回忆一下上面提到的公式：
 $y = \text{softmax}(Wx + b)$ 。改写成 TensorFlow 的语言就是下面这行代码。

```
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

Softmax 是 tf.nn 下面的一个函数，而 tf.nn 则包含了大量神经网络的组件，tf.matmul 是 TensorFlow 中的矩阵乘法函数。我们使用一行简单的代码就定义了 Softmax Regression，语法和直接写数学公式很像。然而 TensorFlow 最厉害的地方还不是定义公式，而是将 forward 和 backward 的内容都自动实现（无论 CPU 或是 GPU 上），只要接下来定义好 loss，训练时将会自动求导并进行梯度下降，完成对 Softmax Regression 模型参数的自动学习。

为了训练模型，我们需要定义一个 loss function 来描述模型对问题的分类精度。Loss 越小，代表模型的分类结果与真实值的偏差越小，也就是说模型越精确。我们一开始给模型填充了全零的参数，这样模型会有一个初始的 loss，而训练的目的是不断将这个 loss 减小，直到达到一个全局最优或者局部最优解。对多分类问题，通常使用 cross-entropy 作为 loss function。Cross-entropy 最早出自信息论（Information Theory）中的信息熵（与压缩比率等有关），然后被用到很多地方，包括通信、纠错码、博弈论、机器学习等。Cross-entropy 的定义如下，其中 y 是预测的概率分布， y' 是真实的概率分布（即 Label 的 one-hot 编码），通常可以用它来判断模型对真实概率分布估计的准确程度。

$$H_{y'}(y) = - \sum_i y'_i \log(y_i)$$

在 TensorFlow 中定义 cross-entropy 也很容易，代码如下。

```
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
                                             reduction_indices=[1]))
```

先定义一个 placeholder，输入是真实的 label，用来计算 cross-entropy。这里的 $y_ * \text{tf.log}(y)$ 也就是前面公式中的 $y'_i \log(y_i)$ ，tf.reduce_sum 也就是求和的 Σ ，而 tf.reduce_mean 则用来对每个 batch 数据结果求均值。

现在我们有了算法 Softmax Regression 的定义，又有了损失函数 cross-entropy 的定义，

只需要再定义一个优化算法即可开始训练。我们采用常见的随机梯度下降 SGD(Stochastic Gradient Descent)。定义好优化算法后，TensorFlow 就可以根据我们定义的整个计算图(我们前面定义的各个公式已经自动构成了计算图) 自动求导，并根据反向传播 (Back Propagation) 算法进行训练，在每一轮迭代时更新参数来减小 loss。在后台 TensorFlow 会自动添加许多运算操作 (Operation) 来实现刚才提到的反向传播和梯度下降，而给我们提供的就是一个封装好的优化器，只需要每轮迭代时 feed 数据给它就好。我们直接调用 `tf.train.GradientDescentOptimizer`，并设置学习速率为 0.5，优化目标设定为 cross-entropy，得到进行训练的操作 `train_step`。当然，TensorFlow 中也有很多其他的优化器，使用起来也非常方便，只需要修改函数名即可。

```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

下一步使用 TensorFlow 的全局参数初始化器 `tf.global_variables_initializer`，并直接执行它的 `run` 方法。

```
tf.global_variables_initializer().run()
```

最后一步，我们开始迭代地执行训练操作 `train_step`。这里每次都随机从训练集中抽取 100 条样本构成一个 mini-batch，并 feed 给 `placeholder`，然后调用 `train_step` 对这些样本进行训练。使用一小部分样本进行训练称为随机梯度下降，与每次使用全部样本的传统梯度下降对应。如果每次训练都使用全部样本，计算量太大，有时也不容易跳出局部最优。因此，对于大部分机器学习问题，我们都只使用一小部分数据进行随机梯度下降，这种做法绝大多数时候会比全样本训练的收敛速度快很多。

```
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    train_step.run({x: batch_xs, y_: batch_ys})
```

现在我们已经完成了训练，接下来就可以对模型的准确率进行验证。下面代码中的 `tf.argmax` 是从一个 `tensor` 中寻找最大值的序号，`tf.argmax(y, 1)` 就是求各个预测的数字中概率最大的那一个，而 `tf.argmax(y_, 1)` 则是找样本的真实数字类别。而 `tf.equal` 方法则用来判断预测的数字类别是否就是正确的类别，最后返回计算分类是否正确的操作 `correct_prediction`。

```
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
```

我们统计全部样本预测的 accuracy，这里需要先用 tf.cast 将之前 correct_prediction 输出的 bool 值转换为 float32，再求平均。

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

我们将测试数据的特征和 Label 输入评测流程 accuracy，计算模型在测试集上的准确率，再将结果打印出来。使用 Softmax Regression 对 MNIST 数据进行分类识别，在测试集上平均准确率可达 92% 左右。

```
print(accuracy.eval({x: mnist.test.images, y_: mnist.test.labels}))
```

通过上面的这个简单例子，我们使用 TensorFlow 实现了一个简单的机器学习算法 Softmax Regression，这可以算作是一个没有隐含层的最浅的神经网络。我们来回忆一下整个流程，我们做的事情可以分为 4 个部分。

- (1) 定义算法公式，也就是神经网络 forward 时的计算。
- (2) 定义 loss，选定优化器，并指定优化器优化 loss。
- (3) 迭代地对数据进行训练。
- (4) 在测试集或验证集上对准确率进行评测。

这几个步骤是我们使用 TensorFlow 进行算法设计、训练的核心步骤，也将会贯穿之后其他类型神经网络的章节。需要注意的是，TensorFlow 和 Spark 类似，我们定义的各个公式其实只是 Computation Graph，在执行这行代码时，计算还没有实际发生，只有等调用 run 方法，并 feed 数据时计算才真正执行。比如 cross_entropy、train_step、accuracy 等都是计算图中的节点，而并不是数据结果，我可以通过调用 run 方法执行这些节点或者说运算操作来获取结果。

我们再来看看 Softmax Regression 达到的效果，准确率为 92%，虽然是一个还不错的数字，但是还达不到实用的程度。手写数字识别的主要应用场景是识别银行支票，如果准确率不够高，可能会引起严重的后果。后面我们将讲解使用多层感知机和卷积网络，来解决 MNIST 手写数字识别问题的方法。事实上，MNIST 数字识别也算是卷积神经网络的首个经典应用，LeCun 的 LeNet5 在 20 世纪 90 年代就已经提出，而且可以达到 99% 的准确率，可以说是领先时代的重大突破。可惜后面因为计算能力制约，卷积神经网络的研究一直没有太大突破，神经网络也一度被 SVM 等超越而陷入低谷。在 20 世纪初的很多年

里，神经网络几乎被大家遗忘，相关研究一直不受重视，这一段是深度学习的一次冰期（神经网络的研究一共有三次大起大落）。2006 年，Hinton 等人提出逐层预训练来初始化权重的方法及利用多层 RBM 堆叠的神经网络 DBN，神经网络才逐渐重回大家视野。Hinton 揭示了神经网络的最大价值在于对特征的自动提取和抽象，它免去了人工提取特征的繁琐，可以自动找出复杂且有效的高阶特征。这一点类似人的学习过程，先理解简单概念，再逐渐递进到复杂概念，神经网络每加深一层，可以提取的特征就更抽象。随着 2012 年 Hinton 学生的研究成果 AlexNet 以巨大优势摘得了当年 ImageNet ILSVRC 比赛的第一名，深度学习的热潮被再次点燃。ImageNet 是一个非常著名的图片数据集，大致有几百万张图片和 1000 类（大部分是动物，约有几百类的动物）。官方会每年举办一次大型的比赛，有图片分类、目标位置检测、视频检测、图像分割等任务。在此之前，参赛读物都是做特征工程，然后使用 SVM 等模型进行分类。而 AlexNet 夺冠后，每一年 ImageNet ILSVRC 的冠军都是依靠深度学习、卷积神经网络，而且趋势是层数越深，效果越好。2015 年，微软研究院提出的 ResNet 甚至达到惊人的 152 层深，并在分类准确率上有了突破性的进展。至此，深度学习在复杂机器学习任务上的巨大优势正式确立，现在基本在任何问题上，仔细设计的神经网络都可以取得比其他算法更好的准确率和泛化性，前提是足够多的数据。

接下来的章节，我们会继续使用其他算法在 MNIST 数据集上进行训练，事实上，现在的 Softmax Regression 加入隐含层变成一个正统的神经网络后，再结合 Dropout、Adagrad、ReLU 等技术准确率就可以达到 98%。引入卷积层、池化层后，也可以达到 99% 的正确率。而目前基于卷积神经网络的 state-of-the-art 的方法已经可以达到 99.8% 的正确率。

4

TensorFlow 实现自编码器及多层感知机

4.1 自编码器简介

传统机器学习任务很大程度上依赖于好的特征工程，比如对数值型、日期时间型、种类型等特征的提取。特征工程往往是非常耗时耗力的，在图像、语音和视频中提取到有效的特征就更难了，工程师必须在这些领域有非常深入的理解，并且使用专业算法提取这些数据的特征。深度学习则可以解决人工难以提取有效特征的问题，它可以大大缓解机器学习模型对特征工程的依赖。深度学习在早期一度被认为是一种无监督的特征学习（Unsupervised Feature Learning），模仿了人脑的对特征逐层抽象提取的过程。这其中有一点很重要：一是无监督学习，即我们不需要标注数据就可以对数据进行一定程度的学习，这种学习是对数据内容的组织形式的学习，提取的是频繁出现的特征；二是逐层抽象，特征是需要不断抽象的，就像人总是从简单基础的概念开始学习，再到复杂的概念。学生们要从加减乘除开始学起，再到简单函数，然后到微积分，深度学习也是一样，它从简单的微观的特征开始，不断抽象特征的层级，逐渐往复杂的宏观特征转变。

例如在图像识别问题中，假定我们有许多汽车的图片，要如何判定这些图片是汽车呢？

如果我们从像素级特征开始进行训练分类器，那么绝大多数算法很难有效地工作。如果我们提取出高阶的特征，比如汽车的车轮、汽车的车窗、汽车的车身，那么使用这些高阶特征便可以非常准确地对图片进行分类，这就是高阶特征的效果。不过任何高阶特征都是由底层特征组合而成的，比如车轮由橡胶轮胎、车轴、轮辐等组成。而其中每一个组件都是由更小单位的特征组合而成的，比如橡胶轮胎由许多黑色的同心圆组成，而这些同心圆也都由许多圆弧曲线组成，圆弧曲线都由像素组成。我们将前面的过程反过来，将一张图片的原始像素慢慢抽象，从像素组成点、线，再将点、线组合成小零件，再将小零件组成车轮、车窗、车身等高阶特征，这便是深度学习在训练过程中所做的特征学习。

早年由学者们研究稀疏编码（Sparse Coding）时，他们收集了大量黑白风景照，并从中提取了许多 16×16 像素的图像碎片。他们发现几乎所有的图像碎片都可以由 64 种正交的边组合得到，如图 4-1 所示，并且组合出一张图像碎片需要的边的数量是很少的，即稀疏的。学者同时发现声音也存在这种情况，他们从大量的未标注音频中发现了 20 种基本结构，绝大多数声音可以由这些基本结构线性组合得到。这其实就是特征的稀疏表达，使用少量的基本特征组合拼装得到更高层抽象的特征。通常我们也需要多层的神经网络，对每一层神经网络来说，前一层的输出都是未加工的像素，而这一层则是对像素进行加工组织成更高阶的特征（即前面提到的将边组合成图像碎片）。

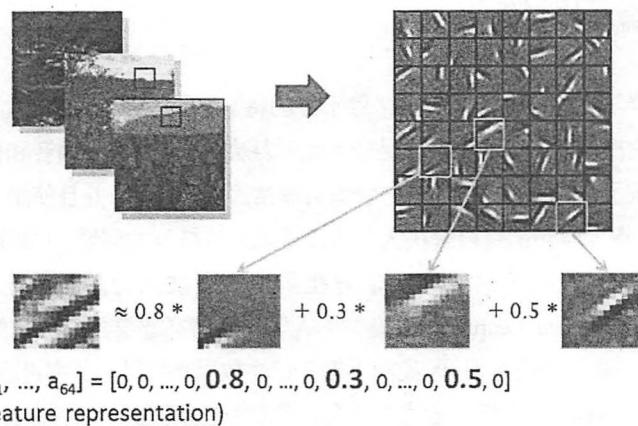


图 4-1 图像碎片可由少量的基本结构稀疏表达

我们来看一下实际的例子。假如我们有许多基本结构，比如指向各个方向的边、白块、黑块等，如图 4-2 所示，我们可以通过不同方式组合出不同的高阶特征，并最终拼出不同的目标物体。这些基本结构就是 *basis*，在人脸识别任务中，我们可以使用它们拼出人脸

的不同器官，比如鼻子、嘴、眼睛、眉毛、脸颊等，这些器官又可以向上一层拼出不同样式的脸，最后模型通过在图片中匹配这些不同样式的脸（即高阶特征）来进行识别。同样，basis 可以拼出汽车上不同的组件，最终拼出各式各样的车型；也可以拼出大象身体的不同部位，最后组成各种尺寸、品种、颜色的大象；还可以拼出椅子的凳、座、靠背等，最后组成不同款式的椅子。特征是可以不断抽象转为高一级的特征的，那我们如何找到这些基本结构，然后如何抽象呢？如果我们有很多标注的数据，则可以训练一个深层的神经网络。如果没有标注的数据呢？这种情况下，我们依然可以使用无监督的自编码器来提取特征。自编码器（AutoEncoder），顾名思义，即可以使用自身的高阶特征编码自己。自编码器其实也是一种神经网络，它的输入和输出是一致的，它借助稀疏编码的思想，目标是使用稀疏的一些高阶特征重新组合来重构自己。因此，它的特点非常明显：第一，期望输入/输出一致；第二，希望使用高阶特征来重构自己，而不仅是复制像素点。

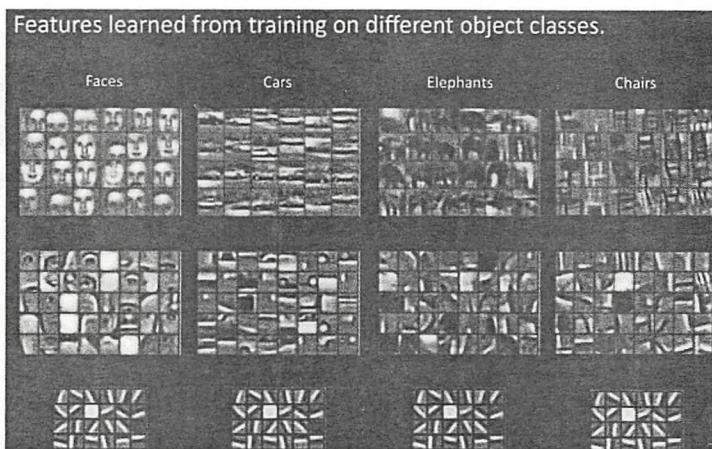


图 4-2 由基本结构不断抽象为高阶特征

Hinton 教授在 *Science* 发表文章 *Reducing the dimensionality of data with neural networks*²³，讲解了使用自编码器对数据进行降维的方法。Hinton 还提出了基于深度信念网络（Deep Belief Networks²⁴，DBN，由多层 RBM 堆叠而成）可使用无监督的逐层训练的贪心算法，为训练很深的网络提供了一个可行方案：我们可能很难直接训练极深的网络，但是可以用无监督的逐层训练提取特征，将网络的权重初始化到一个比较好的位置，辅助后面的监督训练。无监督的逐层训练，其思想和自编码器（AutoEncoder）非常相似。后者的目标是让神经网络的输出能和原始输入一致，相当于学习一个恒等式 $y = x$ ，如图 4-3 所示。自编码器的输入节点和输出节点的数量是一致的，但如果只是单纯地逐个复制输入

节点则没有意义，像前面提到的，自编码器通常希望使用少量稀疏的高阶特征来重构输入，所以我们可以加入几种限制。

(1) 如果限制中间隐含层节点的数量，比如让中间隐含层节点的数量小于输入/输出节点的数量，就相当于一个降维的过程。此时已经不可能出现复制所有节点的情况，因为中间节点数小于输入节点数，那只能学习数据中最重要的特征复原，将可能不太相关的內容去除。此时，如果再给中间隐含层的权重加一个 L1 的正则，则可以根据惩罚系数控制隐含节点的稀疏程度，惩罚系数越大，学到的特征组合越稀疏，实际使用（非零权重）的特征数量越少。

(2) 如果给数据加入噪声，那么就是 Denoising AutoEncoder (去噪自编码器)，我们将从噪声中学习出数据的特征。同样，我们也不可能完全复制节点，完全复制并不能去除我们添加的噪声，无法完全复原数据。所以唯有学习数据频繁出现的模式和结构，将无规律的噪声略去，才可以复原数据。

去噪自编码器中最常使用的噪声是加性高斯噪声(Additive Gaussian Noise, AGN)，其结构如图 4-3 所示。当然也可以使用 Masking Noise，即有随机遮挡的噪声，这种情况下，图像中的一部分像素被置为 0，模型需要从其他像素的结构推测出这些被遮挡的像素是什么，因此模型依然需要学习图像中抽象的高阶特征。

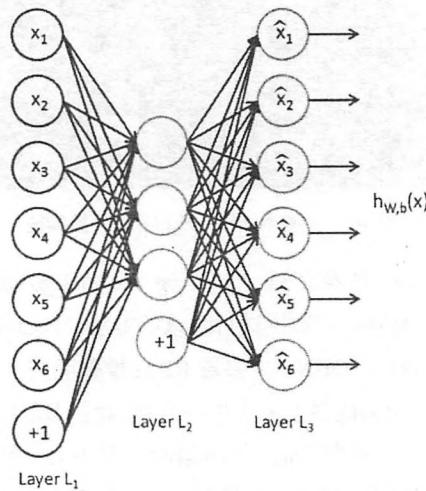


图 4-3 自编码器结构图，学习目标是使用少量高阶特征重构输入

如果自编码器的隐含层只有一层，那么其原理类似于主成分分析 (PCA)。Hinton 提

出的 DBN 模型有多个隐含层，每个隐含层都是限制性玻尔兹曼机 RBM (Restricted Boltzman Machine，一种具有特殊连接分布的神经网络)。DBN 训练时，需要先对每两层间进行无监督的预训练 (pre-training)，这个过程其实就相当于一个多层的自编码器，可以将整个网络的权重初始化到一个理想的分布。最后，通过反向传播算法调整模型权重，这个步骤会使用经过标注的信息来做监督性的分类训练。当年 DBN 给训练深层的神经网络提供了可能性，它能解决网络过深带来的梯度弥散 (Gradient Vanishment) 问题，让训练变得容易。简单地说，Hinton 的思路就是先用自编码器的方法进行无监督的预训练，提取特征并初始化权重，然后使用标注信息进行监督式的训练。当然自编码器的作用不仅局限于给监督训练做预训练，直接使用自编码器进行特征提取和分析也是可以的。现实中数据最多的还是未标注的数据，因此自编码器拥有许多用武之地。

4.2 TensorFlow 实现自编码器

下面我们就开始实现最具代表性的去噪自编码器。去噪自编码器的使用范围最广也最通用。而其他几种自编码器，读者可以对代码加以修改自行实现，其中无噪声的自编码器只需要去掉噪声，并保证隐含层节点小于输入层节点；Masking Noise 的自编码器只需要将高斯噪声改为随机遮挡噪声；Variational AutoEncoder (VAE) 则相对复杂，VAE 对中间节点的分布有强假设，拥有额外的损失项，且会使用特殊的 SGVB (Stochastic Gradient Variational Bayes) 算法进行训练。目前 VAE 还在生成模型中发挥了很大的作用。

这里我们依然是先导入常用库 NumPy，还有 Scikit-learn 中的 preprocessing 模块，这是一个对数据进行预处理的常用模块，之后我们会使用其中的数据标准化功能。同时本节依然使用 MNIST 数据集，因此也导入 TensorFlow 中 MNIST 数据的加载模块。本节代码主要来自 TensorFlow 的开源实现²⁵。

```
import numpy as np
import sklearn.preprocessing as prep
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

我们的自编码器中会使用到一种参数初始化方法 xavier initialization²⁶，需要先定义好它。Xavier 初始化器在 Caffe 的早期版本中被频繁使用，它的特点是会根据某一层网络的输入、输出节点数量自动调整最合适的分布。Xavier Glorot 和深度学习三巨头之一的

Yoshua Bengio 在一篇论文中指出，如果深度学习模型的权重初始化得太小，那信号将在每层间传递时逐渐缩小而难以产生作用，但如果权重初始化得太大，那信号将在每层间传递时逐渐放大并导致发散和失效。而 Xavier 初始化器做的事情就是让权重被初始化得不大不小，正好合适。从数学的角度分析，Xavier 就是让权重满足 0 均值，同时方差为 $\frac{2}{n_{in}+n_{out}}$ ，分布可以用均匀分布或者高斯分布。如下代码所示，我们通过 `tf.random_uniform` 创建了一个 $(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}})$ 范围内的均匀分布，而它的方差根据公式 $D(x) = (max - min)^2 / 12$ 刚好等于 $\frac{2}{n_{in}+n_{out}}$ 。因此，这里实现的就是标准的均匀分布的 Xavier 初始化器，其中 `fan_in` 是输入节点的数量，`fan_out` 是输出节点的数量。

```
def xavier_init(fan_in, fan_out, constant = 1):
    low = -constant * np.sqrt(6.0 / (fan_in + fan_out))
    high = constant * np.sqrt(6.0 / (fan_in + fan_out))
    return tf.random_uniform((fan_in, fan_out),
                            minval = low, maxval = high,
                            dtype = tf.float32)
```

下面我们就开始定义一个去噪自编码的 `class`，方便以后使用。这个类会包含一个构建函数 `__init__()`，还有一些常用的成员函数，因此会比较长，下面会分为几个代码段讲解，我们先来看构建函数。

`__init__` 函数包含这样几个输入：`n_input`（输入变量数）、`n_hidden`（隐含层节点数）、`transfer_function`（隐含层激活函数，默認為 `softplus`）、`optimizer`（优化器，默認為 `Adam`）、`scale`（高斯噪声系数，默認為 0.1）。其中，`class` 内的 `scale` 参数做成了一个 `placeholder`，参数初始化则使用了接下来定义的 `_initialize_weights` 函数。这里需要注意的是，我们只使用了一个隐含层，有需要的读者可以自行尝试多添加几个隐含层。

```
class AdditiveGaussianNoiseAutoencoder(object):
    def __init__(self, n_input, n_hidden, transfer_function=tf.nn.softplus,
                 optimizer = tf.train.AdamOptimizer(), scale=0.1):
        self.n_input = n_input
        self.n_hidden = n_hidden
        self.transfer = transfer_function
        self.scale = tf.placeholder(tf.float32)
        self.training_scale = scale
```

```
network_weights = self._initialize_weights()
self.weights = network_weights
```

接下来开始定义网络结构，我们为输入 x 创建一个维度为 n_input 的 placeholder。然后建立一个能提取特征的隐含层，我们先将输入 x 加上噪声，即 $self.x + scale * tf.random_normal((n_input,))$ ，然后用 $tf.matmul$ 将加了噪声的输入与隐含层的权重 $w1$ 相乘，并使用 $tf.add$ 加上隐含层的偏置 $b1$ ，最后使用 $self.transfer$ 对结果进行激活函数处理。经过隐含层后，我们需要在输出层进行数据复原、重建操作（即建立 $reconstruction$ 层），这里我们就不需要激活函数了，直接将隐含层的输出 $self.hidden$ 乘上输出层的权重 $w2$ ，再加上输出层的偏置 $b2$ 即可。

```
self.x = tf.placeholder(tf.float32, [None, self.n_input])
self.hidden = self.transfer(tf.add(tf.matmul(
    self.x + scale * tf.random_normal((n_input,)),
    self.weights['w1']), self.weights['b1']))
self.reconstruction = tf.add(tf.matmul(self.hidden,
    self.weights['w2']), self.weights['b2'])
```

接下来定义自编码器的损失函数，这里直接使用平方误差 (Squared Error) 作为 cost，即用 $tf.subtract$ 计算输出 ($self.reconstruction$) 与输入 ($self.x$) 之差，再使用 $tf.pow$ 求差的平方，最后使用 $tf.reduce_sum$ 求和即可得到平方误差。再定义训练操作为优化器 $self.optimizer$ 对损失 $self.cost$ 进行优化。最后创建 Session，并初始化自编码器的全部模型参数。

```
self.cost = 0.5 * tf.reduce_sum(tf.pow(tf.subtract(
    self.reconstruction, self.x), 2.0))
self.optimizer = optimizer.minimize(self.cost)

init = tf.global_variables_initializer()
self.sess = tf.Session()
self.sess.run(init)
```

下面来看一下参数初始化函数 $_initialize_weights$ ，先创建一个名为 $all_weights$ 的字典 $dict$ ，然后将 $w1$ 、 $b1$ 、 $w2$ 、 $b2$ 全部存入其中，最后返回 $all_weights$ 。其中 $w1$ 需要使用前面定义的 $xavier_init$ 函数初始化，我们直接传入输入节点数和隐含层节点数，然后 $xavier$

即可返回一个比较适合于 softplus 等激活函数的权重初始分布，而偏置 b1 只需要使用 tf.zeros 全部置为 0 即可。对于输出层 self.reconstruction，因为没有使用激活函数，这里将 w2、b2 全部初始化为 0 即可。

```
def _initialize_weights(self):
    all_weights = dict()
    all_weights['w1'] = tf.Variable(xavier_init(self.n_input,
                                                self.n_hidden))
    all_weights['b1'] = tf.Variable(tf.zeros([self.n_hidden]),
                                    dtype = tf.float32)
    all_weights['w2'] = tf.Variable(tf.zeros([self.n_hidden,
                                              self.n_input]), dtype = tf.float32)
    all_weights['b2'] = tf.Variable(tf.zeros([self.n_input]),
                                    dtype = tf.float32)

    return all_weights
```

我们定义计算损失 cost 及执行一步训练的函数 partial_fit。函数里只需让 Session 执行两个计算图的节点，分别是损失 cost 和训练过程 optimizer，输入的 feed_dict 包括输入数据 x，以及噪声的系数 scale。函数 partial_fit 做的就是用一个 batch 数据进行训练并返回当前的损失 cost。

```
def partial_fit(self, X):
    cost, opt = self.sess.run((self.cost, self.optimizer),
                             feed_dict = {self.x: X, self.scale: self.training_scale})
    return cost
```

我们也需要一个只求损失 cost 的函数 calc_total_cost，这里就只让 Session 执行一个计算图节点 self.cost，传入的参数和前面的 partial_fit 一致。这个函数是在自编码器训练完毕后，在测试集上对模型性能进行评测时会用到的，它不会像 partial_fit 那样触发训练操作。

```
def calc_total_cost(self, X):
    return self.sess.run(self.cost, feed_dict = {self.x: X,
                                                self.scale: self.training_scale
                                               })
```

我们还定义了 transform 函数，它返回自编码器隐含层的输出结果。它的目的是提供

一个接口来获取抽象后的特征，自编码器的隐含层的最主要功能就是学习出数据中的高阶特征。

```
def transform(self, X):
    return self.sess.run(self.hidden, feed_dict = {self.x: X,
                                                    self.scale: self.training_scale
                                                   })
```

我们再定义 generate 函数，它将隐含层的输出结果作为输入，通过之后的重建层将提取到的高阶特征复原为原始数据。这个接口和前面的 transform 正好将整个自编码器拆分为两部分，这里的 generate 接口是后半部分，将高阶特征复原为原始数据的步骤。

```
def generate(self, hidden = None):
    if hidden is None:
        hidden = np.random.normal(size = self.weights["b1"])
    return self.sess.run(self.reconstruction,
                        feed_dict = {self.hidden: hidden})
```

接下来定义 reconstruct 函数，它整体运行一遍复原过程，包括提取高阶特征和通过高阶特征复原数据，即包括 transform 和 generate 两块。输入数据是原数据，输出数据是复原后的数据。

```
def reconstruct(self, X):
    return self.sess.run(self.reconstruction, feed_dict = {self.x: X,
                                                            self.scale: self.training_scale
                                                           })
```

这里的 getWeights 函数作用是获取隐含层的权重 w1。

```
def getWeights(self):
    return self.sess.run(self.weights['w1'])
```

而 getBiases 函数则是获取隐含层的偏置系数 b1。

```
def getBiases(self):
    return self.sess.run(self.weights['b1'])
```

至此，去噪自编码器的 class 就全部定义完了，包括神经网络的设计、权重的初始化，

以及几个常用的成员函数（`transform`、`generate` 等，他们属于计算图中的子图）。接下来使用定义好的 AGN 自编码器在 MNIST 数据集上进行一些简单的性能测试，看看模型对数据的复原效果究竟如何。

接下来依然使用 TensorFlow 提供的读取示例数据的函数载入 MNIST 数据集。

```
mnist = input_data.read_data_sets('MNIST_data', one_hot = True)
```

先定义一个对训练、测试数据进行标准化处理的函数。标准化即让数据变成 0 均值，且标准差为 1 的分布。方法就是先减去均值，再除以标准差。我们直接使用 `sklearn.preprocessing` 的 `StandardScaler` 这个类，先在训练集上进行 `fit`，再将这个 `Scaler` 用到训练数据和测试数据上。这里需要注意的是，必须保证训练、测试数据都使用完全相同的 `Scaler`，这样才能保证后面模型处理数据时的一致性，这也就是为什么先在训练数据上 `fit` 出一个共用的 `Scaler` 的原因。

```
def standard_scale(X_train, X_test):
    preprocessor = prep.StandardScaler().fit(X_train)
    X_train = preprocessor.transform(X_train)
    X_test = preprocessor.transform(X_test)
    return X_train, X_test
```

再定义一个获取随机 block 数据的函数：取一个从 0 到 `len(data) - batch_size` 之间的随机整数，再以这个随机数作为 block 的起始位置，然后顺序取到一个 `batch_size` 的数据。需要注意的是，这属于不放回抽样，可以提高数据的利用效率。

```
def get_random_block_from_data(data, batch_size):
    start_index = np.random.randint(0, len(data) - batch_size)
    return data[start_index:(start_index + batch_size)]
```

使用之前定义的 `standard_scale` 函数对训练集、测试集进行标准化变换。

```
X_train, X_test = standard_scale(mnist.train.images, mnist.test.images)
```

接下来定义几个常用参数，总训练样本数，最大训练的轮数(`epoch`)设为 20，`batch_size` 设为 128，并设置每隔一轮 (`epoch`) 就显示一次损失 `cost`。

```
n_samples = int(mnist.train.num_examples)
training_epochs = 20
```

```
batch_size = 128
display_step = 1
```

创建一个 AGN 自编码器的实例，定义模型输入节点数 n_input 为 784，自编码器的隐含层节点数 n_hidden 为 200，隐含层的激活函数 transfer_function 为 softplus，优化器 optimizer 为 Adam 且学习速率为 0.001，同时将噪声的系数 scale 设为 0.01。

```
autoencoder = AdditiveGaussianNoiseAutoencoder(n_input = 784,
                                               n_hidden = 200,
                                               transfer_function = tf.nn.softplus,
                                               optimizer = tf.train.AdamOptimizer(learning_rate = 0.001),
                                               scale = 0.01)
```

下面开始训练过程，在每一轮（epoch）循环开始时，我们将平均损失 avg_cost 设为 0，并计算总共需要的 batch 数（通过样本总数除以 batch 大小），注意这里使用的是不放回抽样，所以并不能保证每个样本都被抽到并参与训练。然后在每一个 batch 的循环中，先使用 get_random_block_from_data 函数随机抽取一个 block 的数据，然后使用成员函数 partial_fit 训练这个 batch 的数据并计算当前的 cost，最后将当前的 cost 整合到 avg_cost 中。在每一轮迭代后，显示当前的迭代数和这一轮迭代的平均 cost。我们在第一轮迭代时，cost 大约为 19000，在最后一轮迭代时，cost 大约为 7000，再接着训练 cost 也很难继续降低了。读者如果感兴趣，可以通过调整 batch_size、epoch 数、优化器、自编码器的隐含层数、隐含节点数等，来尝试获得更低的 cost。

```
for epoch in range(training_epochs):
    avg_cost = 0.
    total_batch = int(n_samples / batch_size)
    for i in range(total_batch):
        batch_xs = get_random_block_from_data(X_train, batch_size)

        cost = autoencoder.partial_fit(batch_xs)
        avg_cost += cost / n_samples * batch_size

    if epoch % display_step == 0:
        print("Epoch:", '%04d' % (epoch + 1), "cost=", "%{:.9f}" .format(avg_cost))
```

最后对训练完的模型进行性能测试，这里使用之前定义的成员函数 `cal_total_cost` 对测试集 `X_test` 进行测试，评价指标依然是平方误差，如果使用示例中的参数，损失值约为 60 万。

```
print("Total cost: " + str(autoencoder.calc_total_cost(X_test)))
```

至此，去噪自编码器的 TensorFlow 实现就全部结束了。读者可以发现，实现自编码器和实现一个单隐含层的神经网络差不多，只不过是在数据输入时做了标准化，并加上了一个高斯噪声，同时我们的输出结果不是数字分类结果，而是复原的数据，因此不需要用标注过的数据进行监督训练。自编码器作为一种无监督学习的方法，它与其他无监督学习的主要不同在于，它不是对数据进行聚类，而是提取其中最有用、最频繁出现的高阶特征，根据这些高阶特征重构数据。在深度学习发展早期非常流行的 DBN，也是依靠这种思想，先对数据进行无监督的学习，提取到一些有用的特征，将神经网络权重初始化到一个较好的分布，然后再使用有标注的数据进行监督训练，即对权重进行 fine-tune。

现在，无监督式预训练的使用场景比以前少了许多，训练全连接的 MLP 或者 CNN、RNN 时，我们都不需要先使用无监督训练提取特征。但是无监督学习乃至 AutoEncoder 依然非常有用。现实生活中，大部分的数据都是没有标注信息的，但人脑就很擅长处理这些数据，我们会提取其中的高阶抽象特征，并使用在其他地方。自编码器作为深度学习在无监督领域的尝试是非常成功的，同时无监督学习也将是深度学习接下来的一个重要发展方向。

4.3 多层感知机简介

在第 3 章中我们使用 TensorFlow 实现了一个简单的 Softmax Regression 模型，这个线性模型最大的特点就是简单易用，但是拟合能力不强。Softmax Regression 可以算是多分类问题 logistic regression，它和传统意义上的神经网络的最大区别是没有隐含层。隐含层是神经网络的一个重要概念，它是指除输入、输出层外，中间的那些层。输入层和输出层是对外可见的，因此也被称作可视层，而中间层不直接暴露出来，是模型的黑箱部分，通常也比较难具有可解释性，所以一般被称作隐含层。有了隐含层，神经网络就具有了一些特殊的属性，比如引入非线性的隐含层后，理论上只要隐含节点足够多，即使只有一个隐含层的神经网络也可以拟合任意函数。同时隐含层越多，越容易拟合复杂函数。有理论研究表明，为了拟合复杂函数需要的隐含节点的数目，基本上随着隐含层的数量增多呈指

数下降趋势。也就是说层数越多，神经网络所需要的隐含节点可以越少。这也是深度学习的特点之一，层数越深，概念越抽象，需要背诵的知识点（神经网络隐含节点）就越少。不过实际使用中，使用层数较深的神经网络会遇到许多困难，比如容易过拟合、参数难以调试、梯度弥散，等等。对这些问题我们需要很多 Trick 来解决，在最近几年的研究中，越来越多的方法，比如 Dropout²⁷、Adagrad²⁸、ReLU²⁹ 等，逐渐帮助我们解决了一部分问题。

过拟合是机器学习中一个常见的问题，它是指模型预测准确率在训练集上升高，但是在测试集上反而下降了，这通常意味着泛化性不好，模型只是记忆了当前数据的特征，不具备推广能力。尤其在神经网络中，因为参数众多，经常出现参数比数据还要多的情况，这就非常容易出现只是记忆了训练集特征的情况。为了解决这个问题，Hinton 教授团队提出了一个思路简单但是非常有效的方法，Dropout。在使用复杂的卷积神经网络训练图像数据时尤其有效，它的大致思路是在训练时，将神经网络某一层的输出节点数据随机丢弃一部分。我们可以理解成随机把一张图片 50% 的点删除掉（即随机将 50% 的点变成黑点），此时人还是很可能识别出这张图片的类别，当时机器也是可以的。这种做法实质上等于创造出了很多新的随机样本，通过增大样本量、减少特征数量来防止过拟合。Dropout 其实也算是一种 bagging 方法，我们可以理解成每次丢弃节点数据是对特征的一种采样。相当于我们训练了一个 ensemble 的神经网络模型，对每个样本都做特征采样，只不过没有训练多个神经网络模型，只有一个融合的神经网络。

参数难以调试是神经网络的另一大痛点，尤其是 SGD 的参数，对 SGD 设置不同的学习速率，最后得到的结果可能差异巨大。神经网络通常不是一个凸优化的问题，它处处充满了局部最优。SGD 本身也不是一个比较稳定的算法，结果可能会在最优解附近波动，而不同的学习速率可能导致神经网络落入截然不同的局部最优之中。不过，通常我们也并不指望能达到全局最优，有理论表示，神经网络可能有很多个局部最优解都可以达到比较好的分类效果，而全局最优反而容易是过拟合的解。我们也可以从人来类推，不同的人有各自迥异的脑神经连接，没有两个人的神经连接方式能完全一致，就像没有两个人的见解能完全相同，但是每个人的脑神经网络（局部最优解）对识别图片中物体类别都有很不错的效果。对 SGD，一开始我们可能希望学习速率大一些，可以加速收敛，但是训练的后期又希望学习速率可以小一些，这样可以比较稳定地落入一个局部最优解。不同的机器学习问题所需要的学习速率也不太好设置，需要反复调试，因此就有像 Adagrad、Adam³⁰、Adadelta³¹ 等自适应的方法可以减轻调试参数的负担。对于这些优化算法，通常我们使用

它默认的参数设置就可以取得一个比较好的效果。而 SGD 则需要对学习速率、Momentum³²、Nesterov³³等参数进行比较复杂的调试，当调试的参数较为适合问题时，才能达到比较好的效果。

梯度弥散（Gradient Vanishment）是另一个影响深层神经网络训练的问题，在 ReLU 激活函数出现之前，神经网络训练全部都是用 Sigmoid 作为激活函数。这可能是因为 Sigmoid 函数具有限制性，输出数值在 0~1，最符合概率输出的定义。非线性的 Sigmoid 函数在信号的特征空间映射上，对中央区的信号增益较大，对两侧区的信号增益小。从生物神经科学的角度来看，中央区酷似神经元的兴奋态，两侧区酷似神经元的抑制态。因而在神经网络训练时，可以将重要特征置于中央区，将非重要特征置于两侧区。可以说，Sigmoid 比最初期的线性激活函数 $y = x$ ，阶梯激活函数 $y = \begin{cases} -1 & (x < 0) \\ 1 & (x \geq 0) \end{cases}$ 和 $y = \begin{cases} 0 & (x < 0) \\ 1 & (x \geq 0) \end{cases}$ 好了不少。但是当神经网络层数较多时，Sigmoid 函数在反向传播中梯度值会逐渐减小，经过多层的传递后会呈指数级急剧减小，因此梯度值在传递到前面几层时就变得非常小了。这种情况下，根据训练数据的反馈来更新神经网络的参数将会非常缓慢，基本起不到训练的作用。直到 ReLU 的出现，才比较完美地解决了梯度弥散的问题。ReLU 是一个简单的非线性函数 $y = \max(0, x)$ ，它在坐标轴上是一条折线（如图 4-4(B) 所示），当 $x \leq 0$ 时， $y = 0$ ；当 $x > 0$ 时， $y = x$ ，非常类似于人脑的阈值响应机制（如图 4-4(A) 所示）。信号在超过某个阈值时，神经元才会进入兴奋和激活的状态，平时则处于抑制状态。ReLU 可以很好地传递梯度，经过多层的反向传播，梯度依旧不会大幅缩小，因此非常适合训练很深的神经网络。ReLU 从正面解决了梯度弥散的问题，而不需要通过无监督的逐层训练初始化权重来绕行。ReLU 对比 Sigmoid 的主要变化有如下 3 点。

- (1) 单侧抑制。
- (2) 相对宽阔的兴奋边界。
- (3) 稀疏激活性。

神经科学家在进行大脑能量消耗的研究中发现，神经元编码的工作方式具有稀疏性，推测大脑同时被激活的神经元只有 1%~4%。神经元只会对输入信号有少部分的选择性响应，大量不相关的信号被屏蔽，这样可以更高效地提取重要特征。传统的 Sigmoid 函数则有接近一半的神经元被激活，不符合神经科学的研究。Softplus 虽然有单侧抑制，却没有稀疏激活性，因而 ReLU 函数 $\max(0, x)$ 成为了最符合实际神经元的模型。目前，ReLU 及其变种（ELU³⁴，PReLU³⁵，RReLU³⁶）已经成为了最主流的激活函数。实践中大部分情况下

(包括 MLP 和 CNN, RNN 内部主要还是使用 Sigmoid、Tanh、Hard Sigmoid) 将隐含层的激活函数从 Sigmoid 替换为 ReLU 都可以带来训练速度及模型准确率的提升。当然神经网络的输出层一般都还是 Sigmoid 函数, 因为它最接近概率输出分布。

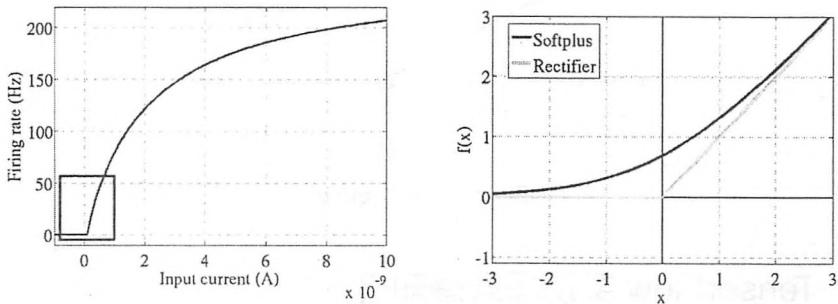


图 4-4(A) 神经科学家提出的神经元激活模型 图 4-4(B) ReLU 激活函数及 Softplus 激活函数

上面三段分别提到了可以解决多层神经网络问题的 Dropout、Adagrad、ReLU 等, 那么多层神经网络到底有什么显著的能力值得大家探索呢? 或者说神经网络的隐含层到底有什么用呢? 隐含层的一个代表性的功能是可以解决 XOR 问题。在早期神经网络的研究中, 有学者提出一个尖锐的问题, 当时(没有隐含层)的神经网络无法解决 XOR 的问题。如图 4-5 所示, 假设我们有两个维度的特征, 并且有两类样本, $(0, 0), (1, 1)$ 是灰色, $(0, 1), (1, 0)$ 是黑色, 在这个特征空间中这两类样本是线性不可分的, 也就是说, 我们无法用一条直线把灰、黑两类分开。没有隐含层的神经网络是线性的, 所以不可能对这两类样本进行正确地区分。这是早期神经网络的致命缺点, 也直接导致了当时神经网络研究的低谷。当引入了隐含层并使用非线性的激活函数(如 Sigmoid、ReLU)后, 我们可以使用曲线划分两类样本, 可以轻松解决 XOR 异或函数的分类问题。神经网络的隐含层越多, 就可以对原有特征进行越抽象的变换, 模型的拟合能力就越强。这就是多层神经网络(或多层感知机, Multi-Layer Perceptron; MLP)的功能所在。

接下来, 我们通过例子展示在仅加入一个隐含层的情况下, 神经网络对 MNIST 数据集的分类性能就有显著提升, 可以达到 98% 的准确率。当然, 其中使用了 Dropout、Adagrad、ReLU 等辅助性组件。

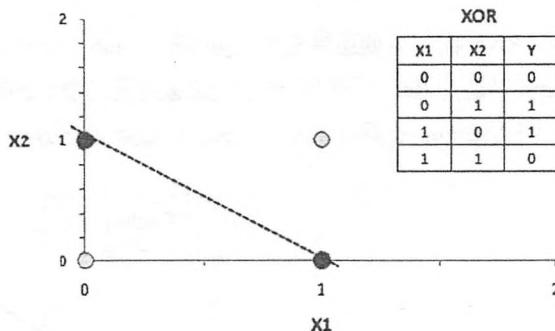


图 4-5 XOR 分类问题

4.4 TensorFlow 实现多层感知机

在第 3 章中我们讲解了使用 TensorFlow 实现一个完整的 Softmax Regression(无隐含层)，并在 MNIST 数据集上取得了大约 92% 的正确率。现在，我们要给神经网络加上隐含层，并使用 4.3 节提到的减轻轻过拟合的 Dropout、自适应学习速率的 Adagrad，以及可以解决梯度弥散的激活函数 ReLU。在 TensorFlow 中实现这些都是非常方便的，通常只需要调用相应的类或者函数即可。

首先，载入 TensorFlow 并加载 MNIST 数据集，创建一个 TensorFlow 默认的 Interactive Session，这样后面执行各项操作就无须指定 Session 了。

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
sess = tf.InteractiveSession()
```

接下来我们要给隐含层的参数设置 Variable 并进行初始化，这里 in_units 是输入节点数，h1_units 即隐含层的输出节点数设为 300 (在此模型中隐含节点数设在 200~1000 范围内的结果区别都不大)。W1、b1 是隐含层的权重和偏置，我们将偏置全部赋值为 0，并将权重初始化为截断的正态分布，其标准差为 0.1，这一步可以通过 tf.truncated_normal 方便地实现。因为模型使用的激活函数是 ReLU，所以需要使用正态分布给参数加一点噪声，来打破完全对称并且避免 0 梯度。在其他一些模型中，有时还需要给偏置赋上一些小的非零值来避免 dead neuron (死亡神经元)，不过在这里作用不太明显。而对最后输出层的

Softmax，直接将权重 W_2 和偏置 b_2 全部初始化为 0 即可（上面提到过，对于 Sigmoid，在 0 附近最敏感、梯度最大）。

```
in_units = 784
h1_units = 300
W1 = tf.Variable(tf.truncated_normal([in_units, h1_units], stddev=0.1))
b1 = tf.Variable(tf.zeros([h1_units]))
W2 = tf.Variable(tf.zeros([h1_units, 10]))
b2 = tf.Variable(tf.zeros([10]))
```

接下来定义输入 x 的 placeholder。另外因为在训练和预测时，Dropout 的比率 `keep_prob`（即保留节点的概率）是不一样的，通常在训练时小于 1，而预测时则等于 1，所以也把 Dropout 的比率作为计算图的输入，并定义成一个 placeholder。

```
x = tf.placeholder(tf.float32, [None, in_units])
keep_prob = tf.placeholder(tf.float32)
```

下面定义模型结构。首先需要一个隐含层，命名为 `hidden1`，可以通过 `tf.nn.relu(tf.matmul(x, W1) + b1)` 实现一个激活函数为 ReLU 的隐含层，这个隐含层的计算公式就是 $y = \text{relu}(W_1x + b_1)$ 。接下来，调用 `tf.nn.dropout` 实现 Dropout 的功能，即随机将一部分节点置为 0，这里的 `keep_prob` 参数即为保留数据而不置为 0 的比例，在训练时应该是小于 1 的，用以制造随机性，防止过拟合；在预测时应该等于 1，即使用全部特征来预测样本的类别。最后是输出层，也就是第 3 章介绍的 Softmax，这一行代码的功能和之前是一致的。

```
hidden1 = tf.nn.relu(tf.matmul(x, W1) + b1)
hidden1_drop = tf.nn.dropout(hidden1, keep_prob)
y = tf.nn.softmax(tf.matmul(hidden1_drop, W2) + b2)
```

第 3 章提到的使用 TensorFlow 训练神经网络的 4 个步骤，到目前为止，我们已经完成了 4 步中的第 1 步：定义算法公式，即神经网络 forward 时的计算。接下来继续第 2 步，定义损失函数和选择优化器来优化 loss，这里的损失函数继续使用交叉信息熵，和之前一致，但是优化器选择自适应的优化器 Adagrad，并把学习速率设为 0.3，这里我们直接使用 `tf.train.AdagradOptimizer` 就可以了。类似地，还有 Adadelta 及 Adam 等优化器，读者可以自行尝试，不过学习速率可能需要调整。

```

y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
                                             reduction_indices=[1]))
train_step = tf.train.AdagradOptimizer(0.3).minimize(cross_entropy)

```

然后进行第 3 步，训练步骤，这里有一点和之前不同，我们加入了 `keep_prob` 作为计算图的输入，并且在训练时设为 0.75，即保留 75% 的节点，其余的 25% 置为 0。一般来说，对越复杂越大规模的神经网络，Dropout 的效果越显著。另外，因为加入了隐含层，我们需要更多的训练迭代来优化模型参数以达到一个比较好的效果。所以一共采用了 3000 个 batch，每个 batch 包含 100 条样本，一共 30 万的样本，相当于是对全数据集进行了 5 轮（epoch）迭代。读者也可以尝试增大循环次数，准确率会略有提高。

```

tf.global_variables_initializer().run()
for i in range(3000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    train_step.run({x: batch_xs, y_: batch_ys, keep_prob: 0.75})

```

最后我们进行第 4 步，对模型进行准确率评测，这里的代码和第 3 章讲解的评测代码基本一致，但还是需要加入一个 `keep_prob` 作为输入。因为是预测部分，所以我们直接令 `keep_prob` 等于 1 即可，这样可以达到模型最好的预测效果。

```

correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print(accuracy.eval({x: mnist.test.images, y_: mnist.test.labels,
                     keep_prob: 1.0}))

```

最终，我们在测试集上可以达到 98% 的准确率。相比之前的 Softmax，我们的误差率由 8% 下降到 2%，对识别银行账单这种精确度要求很高的场景，可以说是飞跃性的提高。而这个提升仅靠增加一个隐含层就实现了，可见多层神经网络的效果有多显著。当然，其中我们也使用了一些 Trick 进行辅助，比如 Dropout、Adagrad、ReLU 等，但是起决定性作用的还是隐含层本身，它能对特征进行抽象和转化。

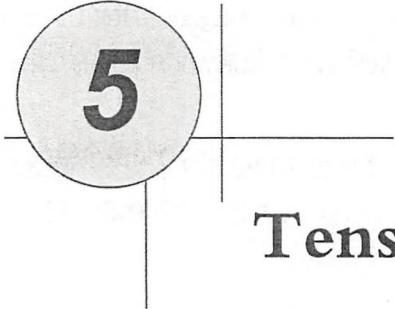
没有隐含层的 Softmax Regression 只能直接从图像的像素点推断是哪个数字，而没有特征抽象的过程。多层神经网络依靠隐含层，则可以组合出高阶特征，比如横线、竖线、圆圈等，之后可以将这些高阶特征或者说组件再组合成数字，就能实现精准的匹配和分类。隐含层输出的高阶特征（组件）经常是可以复用的，所以每一类的判别、概率输出都共享

这些高阶特征，而不是各自连接独立的高阶特征。

同时我们可以发现，新加了一个隐含层，并使用了 Dropout、Adagrad 和 ReLU，而代码没有增加很多，这就是 TensorFlow 的优势之一。它的代码非常简洁，没有太多的冗余，可以方便地将有用的模块拼装在一起。

总结一下，本节我们介绍了如何实现包含一个隐含层的 MLP，对于有更多个隐含层的 MLP，读者可以如法炮制。我们讲解了 Dropout、Adagrad、ReLU 的原理和作用，以及如何在 TensorFlow 中使用它们。

不过，使用全连接神经网络（Fully Connected Network, FCN, MLP 的另一种说法）也是有局限的，即使我们使用很深的网络、很多的隐藏节点、很大的迭代轮数，也很难在 MNIST 数据集上达到 99%以上的准确率。因此第 5 章我们将介绍卷积神经网络，以及如何在 MNIST 数据集上使用 CNN 达到 99%以上的准确率，真正满足识别银行支票这种高精度系统的需求。



5

TensorFlow 实现卷积 神经网络

5.1 卷积神经网络简介

卷积神经网络（Convolutional Neural Network，CNN）最初是为解决图像识别等问题设计的，当然其现在的应用不仅限于图像和视频，也可用于时间序列信号，比如音频信号、文本数据等。在早期的图像识别研究中，最大的挑战是如何组织特征，因为图像数据不像其他类型的数据那样可以通过人工理解来提取特征。在股票预测等模型中，我们可以从原始数据中提取过往的交易价格波动、市盈率、市净率、盈利增长等金融因子，这即是特征工程。但是在图像中，我们很难根据人为理解提取出有效而丰富的特征。在深度学习出现之前，我们必须借助 SIFT、HoG 等算法提取具有良好区分性的特征，再集合 SVM 等机器学习算法进行图像识别。如图 5-1 所示，SIFT 对一定程度内的缩放、平移、旋转、视角改变、亮度调整等畸变，都具有不变性，是当时最重要的图像特征提取方法之一。可以说，在之前只能依靠 SIFT 等特征提取算法才能勉强进行可靠的图像识别。

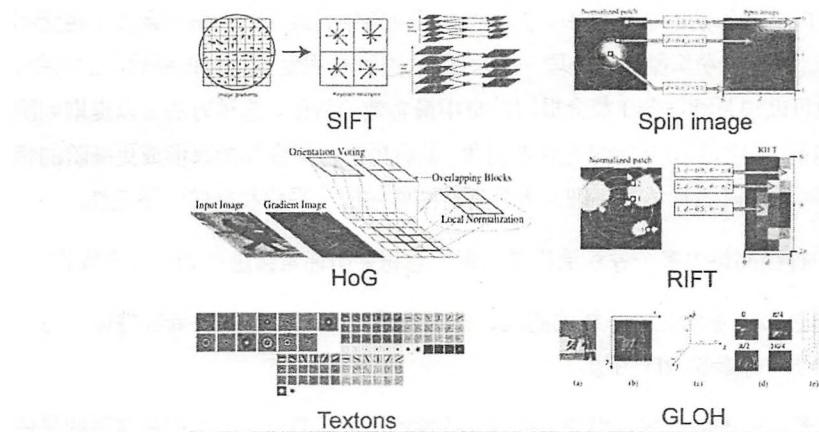


图 5-1 SIFT、HoG 等图像特征提取方法

然而 SIFT 这类算法提取的特征还是有局限性的，在 ImageNet ILSVRC 比赛的最好结果的错误率也有 26% 以上，而且常年难以产生突破。卷积神经网络提取的特征则可以达到更好的效果，同时它不需要将特征提取和分类训练两个过程分开，它在训练时就自动提取了最有效的特征。CNN 作为一个深度学习架构被提出的最初诉求，是降低对图像数据预处理的要求，以及避免复杂的特征工程。CNN 可以直接使用图像的原始像素作为输入，而不必先使用 SIFT 等算法提取特征，减轻了使用传统算法如 SVM 时必须需要做的大量重复、繁琐的数据预处理工作。和 SIFT 等算法类似，CNN 训练的模型同样对缩放、平移、旋转等畸变具有不变性，有着很强的泛化性。CNN 的最大特点在于卷积的权值共享结构，可以大幅减少神经网络的参数量，防止过拟合的同时又降低了神经网络模型的复杂度。CNN 的权值共享其实也很像早期的延时神经网络（TDNN），只不过后者是在时间这一个维度上进行权值共享，降低了学习时间序列信号的复杂度。

卷积神经网络的概念最早出自 19 世纪 60 年代科学家提出的感受野（Receptive Field³⁷）。当时科学家通过对猫的视觉皮层细胞研究发现，每一个视觉神经元只会处理一小块区域的视觉图像，即感受野。到了 20 世纪 80 年代，日本科学家提出神经认知机（Neocognitron³⁸）的概念，可以算作是卷积网络最初的实现原型。神经认知机中包含两类神经元，用来抽取特征的 S-cells，还有用来抗形变的 C-cells，其中 S-cells 对应我们现在主流卷积神经网络中的卷积核滤波操作，而 C-cells 则对应激活函数、最大池化（Max-Pooling）等操作。同时，CNN 也是首个成功地进行多层训练的网络结构，即前面章节提到的 LeCun 的 LeNet³⁹，而全连接的网络因为参数过多及梯度弥散等问题，在早期很难顺利地进行多层的训练。卷积神经网络可以利用空间结构关系减少需要学习的参数

量，从而提高反向传播算法的训练效率。在卷积神经网络中，第一个卷积层会直接接受图像像素级的输入，每一个卷积操作只处理一小块图像，进行卷积变化后再传到后面的网络，每一层卷积（也可以说是滤波器）都会提取数据中最有效的特征。这种方法可以提取到图像中最基础的特征，比如不同方向的边或者拐角，而后再进行组合和抽象形成更高阶的特征，因此 CNN 可以应对各种情况，理论上具有对图像缩放、平移和旋转的不变性。

一般的卷积神经网络由多个卷积层构成，每个卷积层中通常会进行如下几个操作。

(1) 图像通过多个不同的卷积核的滤波，并加偏置 (bias)，提取出局部特征，每一个卷积核会映射出一个新的 2D 图像。

(2) 将前面卷积核的滤波输出结果，进行非线性的激活函数处理。目前最常见的是使用 ReLU 函数，而以前 Sigmoid 函数用得比较多。

(3) 对激活函数的结果再进行池化操作（即降采样，比如将 2×2 的图片降为 1×1 的图片），目前一般是使用最大池化，保留最显著的特征，并提升模型的畸变容忍能力。

这几个步骤就构成了最常见的卷积层，当然也可以再加上一个 LRN⁴⁰ (Local Response Normalization，局部响应归一化层) 层，目前非常流行的 Trick 还有 Batch Normalization 等。

一个卷积层中可以有多个不同的卷积核，而每一个卷积核都对应一个滤波后映射出的新图像，同一个新图像中每一个像素都来自完全相同的卷积核，这就是卷积核的权值共享。那我们为什么要共享卷积核的权值参数呢？答案很简单，降低模型复杂度，减轻过拟合并降低计算量。举个例子，如图 5-2 所示，如果我们的图像尺寸是 1000 像素 \times 1000 像素，并且假定是黑白图像，即只有一个颜色通道，那么一张图片就有 100 万个像素点，输入数据的维度也是 100 万。接下来，如果连接一个相同大小的隐含层（100 万个隐含节点），那么将产生 $100 \times 100 = 10000$ 万个连接。仅仅一个全连接层 (Fully Connected Layer)，就有一万亿连接的权重去训练，这已经超出了普通硬件的计算能力。我们必须减少需要训练的权重数量，一是降低计算的复杂度，二是过多的连接会导致严重的过拟合，减少连接数可以提升模型的泛化性。

图像在空间上是有组织结构的，每一个像素点在空间上和周围的像素点实际上是有紧密联系的，但是和太遥远的像素点就不一定有什么关联了。这就是前面提到的人的视觉感受野的概念，每一个感受野只接受一小块区域的信号。这一小块区域内的像素是互相关联

的，每一个神经元不需要接收全部像素点的信息，只需要接收局部的像素点作为输入，而后将所有这些神经元收到的局部信息综合起来就可以得到全局的信息。这样就可以将之前的全连接的模式修改为局部连接，之前隐含层的每一个隐含节点都和全部像素相连，现在我们只需要将每一个隐含节点连接到局部的像素节点。假设局部感受野大小是 10×10 ，即每个隐含节点只与 10×10 个像素点相连，那么现在就只需要 $10 \times 10 \times 100$ 万=1 亿个连接，相比之前的 1 万亿缩小了 10000 倍。

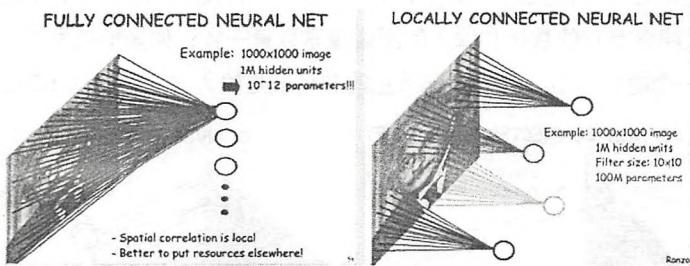


图 5-2 全连接（左）和局部连接（右）

上面我们通过局部连接（Locally Connect）的方法，将连接数从 1 万亿降低到 1 亿，但仍然偏多，需要继续降低参数量。现在隐含层每一个节点都与 10×10 的像素相连，也就是每一个隐含节点都拥有 100 个参数。假设我们的局部连接方式是卷积操作，即默认每一个隐含节点的参数都完全一样，那我们的参数不再是 1 亿，而是 100。不论图像有多大，都是这 $10 \times 10 = 100$ 个参数，即卷积核的尺寸，这就是卷积对缩小参数量的贡献。我们不需要再担心有多少隐含节点或者图片有多大，参数量只跟卷积核的大小有关，这也就是所谓的权值共享。但是如果我们只有一个卷积核，我们就只能提取一种卷积核滤波的结果，即只能提取一种图片特征，这不是我们期望的结果。好在图像中最基本的特征很少，我们可以增加卷积核的数量来多提取一些特征。图像中的基本特征无非就是点和边，无论多么复杂的图像都是点和边组合而成的。人眼识别物体的方式也是从点和边开始的，视觉神经元接受光信号后，每一个神经元只接受一个区域的信号，并提取出点和边的特征，然后将点和边的信号传递给后面一层的神经元，再接着组合成高阶特征，比如三角形、正方形、直线、拐角等，再继续抽象组合，得到眼睛、鼻子和嘴等五官，最后再将五官组合成一张脸，完成匹配识别。因此我们的问题就很好解决了，只要我们提供的卷积核数量足够多，能提取出各种方向的边或各种形态的点，就可以让卷积层抽象出有效而丰富的高阶特征。每一个卷积核滤波得到的图像就是一类特征的映射，即一个 Feature Map。一般来说，我们使用 100 个卷积核放在第一个卷积层就已经很充足了。那这样的话，如图 5-3 所示，我

们的参数量就是 $100 \times 100 = 1$ 万个，相比之前的 1 亿又缩小了 10000 倍。因此，依靠卷积，我们就可以高效地训练局部连接的神经网络了。卷积的好处是，不管图片尺寸如何，我们需要训练的权值数量只跟卷积核大小、卷积核数量有关，我们可以使用非常少的参数量处理任意大小的图片。每一个卷积层提取的特征，在后面的层中都会抽象组合成更高阶的特征。而且多层抽象的卷积网络表达能力更强，效率更高，相比只使用一个隐含层提取全部高阶特征，反而可以节省大量的参数。当然，我们需要注意的是，虽然需要训练的参数量下降了，但是隐含节点的数量并没有下降，隐含节点的数量只跟卷积的步长有关。如果步长为 1，那么隐含节点的数量和输入的图像像素数量一致；如果步长为 5，那么每 5×5 的像素才需要一个隐含节点，我们隐含节点的数量就是输入像素数量的 $1/25$ 。

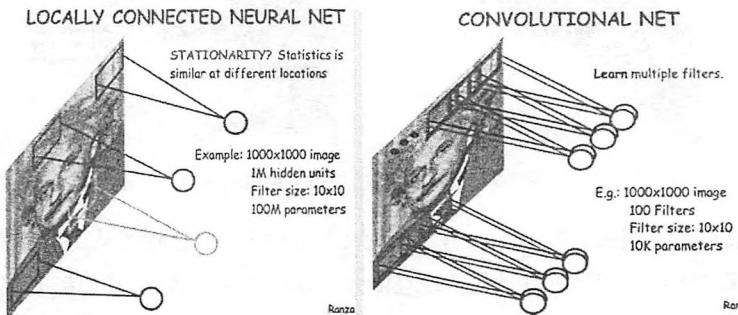


图 5-3 局部连接（左）和卷积操作（右）

我们再总结一下，卷积神经网络的要点就是局部连接（Local Connection）、权值共享（Weight Sharing）和池化层（Pooling）中的降采样（Down-Sampling）。其中，局部连接和权值共享降低了参数量，使训练复杂度大大下降，并减轻了过拟合。同时权值共享还赋予了卷积网络对平移的容忍性，而池化层降采样则进一步降低了输出参数量，并赋予模型对轻度形变的容忍性，提高了模型的泛化能力。卷积神经网络相比传统的机器学习算法，无须手工提取特征，也不需要使用诸如 SIFT 之类的特征提取算法，可以在训练中自动完成特征的提取和抽象，并同时进行模式分类，大大降低了应用图像识别的难度；相比一般的神经网络，CNN 在结构上和图片的空间结构更为贴近，都是 2D 的有联系的结构，并且 CNN 的卷积连接方式和人的视觉神经处理光信号的方式类似。

大名鼎鼎的 LeNet5 诞生于 1994 年，是最早的深层卷积神经网络之一，并且推动了深度学习的发展。从 1988 年开始，在多次成功的迭代后，这项由 Yann LeCun 完成的开拓性成果被命名为 LeNet5。LeCun 认为，可训练参数的卷积层是一种用少量参数在图像的多个位置上提取相似特征的有效方式，这和直接把每个像素作为多层神经网络的输入不

同。像素不应该被使用在输入层，因为图像具有很强的空间相关性，而使用图像中独立的像素直接作为输入则利用不到这些相关性。

LeNet5 当时的特性有如下几点。

- 每个卷积层包含三个部分：卷积、池化和非线性激活函数
- 使用卷积提取空间特征
- 降采样（Subsample）的平均池化层（Average Pooling）
- 双曲正切（Tanh）或 S 型（Sigmoid）的激活函数
- MLP 作为最后的分类器
- 层与层之间的稀疏连接减少计算复杂度

LeNet5 中的诸多特性现在依然在 state-of-the-art 卷积神经网络中使用，可以说 LeNet5 是奠定了现代卷积神经网络的基石之作。Lenet-5 的结构如图 5-4 所示。它的输入图像为 32×32 的灰度值图像，后面有三个卷积层，一个全连接层和一个高斯连接层。它的第一个卷积层 C1 包含 6 个卷积核，卷积核尺寸为 5×5 ，即总共 $(5 \times 5 + 1) \times 6 = 156$ 个参数，括号中的 1 代表 1 个 bias，后面是一个 2×2 的平均池化层 S2 用来进行降采样，再之后是一个 Sigmoid 激活函数用来进行非线性处理。而后是第二个卷积层 C3，同样卷积核尺寸是 5×5 ，这里使用了 16 个卷积核，对应 16 个 Feature Map。需要注意的是，这里的 16 个 Feature Map 不是全部连接到前面的 6 个 Feature Map 的输出的，有些只连接了其中的几个 Feature Map，这样增加了模型的多样性。下面的第二个池化层 S4 和第一个池化层 S2 一致，都是 2×2 的降采样。接下来的第三个卷积层 C5 有 120 个卷积核，卷积大小同样为 5×5 ，因为输入图像的大小刚好也是 5×5 ，因此构成了全连接，也可以算作全连接层。F6 层是一个全连接层，拥有 84 个隐含节点，激活函数为 Sigmoid。LeNet-5 最后一层由欧式径向基函数（Euclidean Radial Basis Function）单元组成，它输出最后的分类结果。

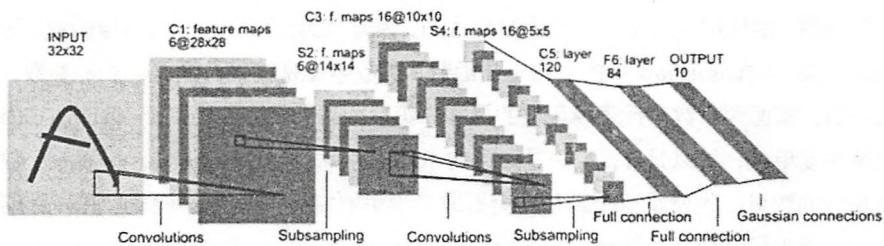


图 5-4 LeNet-5 结构示意图

5.2 TensorFlow 实现简单的卷积网络

本节将讲解如何使用 TensorFlow 实现一个简单的卷积神经网络，使用的数据集依然是 MNIST，预期可以达到 99.2% 左右的准确率。本节将使用两个卷积层加一个全连接层构建一个简单但是非常有代表性的卷积神经网络，读者应该能通过这个例子掌握设计卷积神经网络的要点。

首先载入 MNIST 数据集，并创建默认的 Interactive Session。本节代码主要来自 TensorFlow 的开源实现⁴¹。

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
sess = tf.InteractiveSession()
```

接下来要实现的这个卷积神经网络会有很多的权重和偏置需要创建，因此我们先定义好初始化函数以便重复使用。我们需要给权重制造一些随机的噪声来打破完全对称，比如截断的正态分布噪声，标准差设为 0.1。同时因为我们使用 ReLU，也给偏置增加一些小的正值（0.1）用来避免死亡节点（dead neurons）。

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

卷积层、池化层也是接下来要重复使用的，因此也为他们分别定义创建函数。这里的 `tf.nn.conv2d` 是 TensorFlow 中的 2 维卷积函数，参数中 `x` 是输入，`W` 是卷积的参数，比如 `[5,5,1,32]`：前面两个数字代表卷积核的尺寸；第三个数字代表有多少个 channel。因为我们只有灰度单色，所以是 1，如果是彩色的 RGB 图片，这里应该是 3。最后一个数字代表卷积核的数量，也就是这个卷积层会提取多少类的特征。`Strides` 代表卷积模板移动的步长，都是 1 代表会不遗漏地划过图片的每一个点。`Padding` 代表边界的处理方式，这里的 `SAME` 代表给边界加上 `Padding` 让卷积的输出和输入保持同样（`SAME`）的尺寸。