KERAS SUCCINCTLY

BY **JAMES McCAFFREY**

SUCCINCTLY EBOOK SERIES

Syncfusion®

Keras Succinctly

By

James McCaffrey

Foreword by Daniel Jebaraj

Syncfusion

Deliver innovation with ease

Technical Reviewer: Chris Lee

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.

Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

About the Author

James McCaffrey works for Microsoft Research in Redmond, WA. He holds a B.A. in psychology from the University of California at Irvine, a B.A. in applied mathematics from California State University at Fullerton, an M.S. in information systems from Hawaii Pacific University, and a doctorate in cognitive psychology and computational statistics from the University of Southern California. James enjoys exploring all forms of activity that involve human interaction and combinatorial mathematics, such as the analysis of betting behavior associated with professional sports, machine learning algorithms, and data mining.

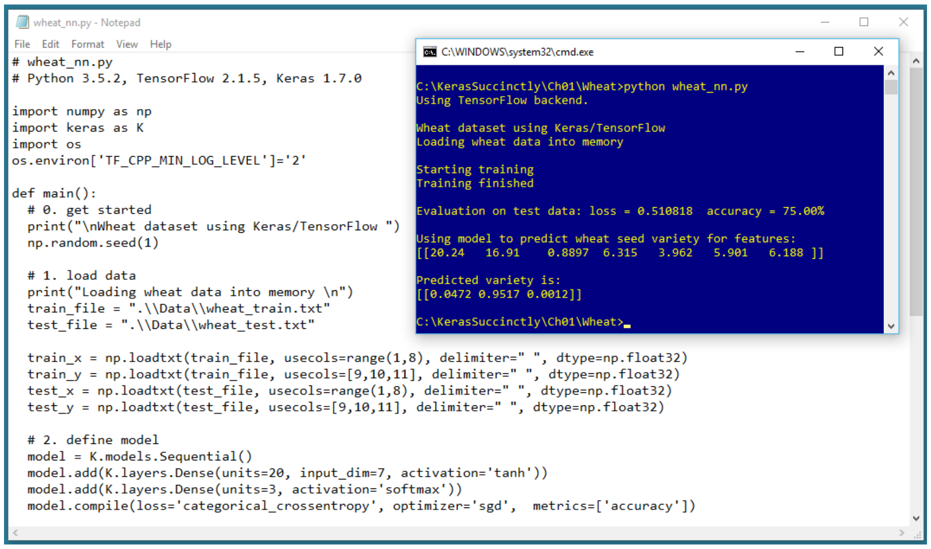
Chapter 1 Getting Started

Keras is an open-source, neural-network library written in the Python language. Keras requires a backend engine and can use TensorFlow, CNTK (Microsoft Cognitive Toolkit), Theano, or MXNet. The motivation for Keras is that, although it's possible to create deep neural systems using TensorFlow directly (or CNTK, Theano, MXNet), because TensorFlow works at a relatively low level of abstraction, coding TensorFlow directly is quite challenging. Keras adds a relatively easy-to-use layer of abstraction over TensorFlow.

Keras, which means "horn" in Greek, was first released in March 2015. This e-book is based on Keras version 2.1.5, which was released in March 2018. Because Keras is in active development, by the time you read this e-book, the latest version will certainly be different. However, any changes to Keras will likely be relatively minor and consist mainly of additional functionality rather than major architecture changes. In other words, the code presented here should work with any Keras 2.x version with few, if any, changes needed.

Keras runs on Windows, Linux, and Mac systems. This e-book focuses on Keras on Windows, but because Keras programs run in a shell, Keras on Linux or Mac systems is almost exactly the same.

The screenshot in Figure 1-1 shows a simplified Keras session on Windows. Notice that the program is just an ordinary Python script, wheat_nn.py, that references Keras as a Python package, and Keras programs run in an ordinary shell.



```
# wheat_nn.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0

import numpy as np
import keras as K
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'

def main():
  # 0. get started
  print("\nWheat dataset using Keras/TensorFlow ")
  np.random.seed(1)

  # 1. load data
  print("Loading wheat data into memory \n")
  train_file = ".\\Data\\wheat_train.txt"
  test_file = ".\\Data\\wheat_test.txt"

  train_x = np.loadtxt(train_file, usecols=range(1,8), delimiter=" ", dtype=np.float32)
  train_y = np.loadtxt(train_file, usecols=[9,10,11], delimiter=" ", dtype=np.float32)
  test_x = np.loadtxt(test_file, usecols=range(1,8), delimiter=" ", dtype=np.float32)
  test_y = np.loadtxt(test_file, usecols=[9,10,11], delimiter=" ", dtype=np.float32)

  # 2. define model
  model = K.models.Sequential()
  model.add(K.layers.Dense(units=20, input_dim=7, activation='tanh'))
  model.add(K.layers.Dense(units=3, activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='sgd',  metrics=['accuracy'])
```

```
C:\KerasSuccinctly\Ch01\Wheat>python wheat_nn.py
Using TensorFlow backend.

Wheat dataset using Keras/TensorFlow
Loading wheat data into memory

Starting training
Training finished

Evaluation on test data: loss = 0.510818  accuracy = 75.00%

Using model to predict wheat seed variety for features:
[[20.24   16.91    0.8897  6.315   3.962   5.901   6.188 ]]

Predicted variety is:
[[0.0472 0.9517 0.0012]]

C:\KerasSuccinctly\Ch01\Wheat>
```

Figure 1-1: Example Keras Session

This e-book assumes you have intermediate or better programming skill with a C-family language, but doesn't assume you know anything about Keras. Enough chit-chat already—let's get started.

Installing Keras and Anaconda

Every programmer I know, including me, learns how to program in a new language or framework by getting an example program up and running, and then experimenting with the example by making changes. So if you want to learn Keras, the first step is to install it.

Keras installation may be a bit different from other software installation you've done before. You don't install Keras directly. Instead, you install Keras as an add-on package for Python. Briefly, you first install a Python distribution (Anaconda), which contains the base Python language interpreter plus several additional packages that are required by Keras, in particular, the NumPy package. Next, you install the TensorFlow package, and then the Keras package.

It is possible to install Python, NumPy, and the other dependencies separately. But instead, I strongly recommend that you install the Anaconda distribution of Python, which has everything you need to successfully install and run Keras. Before beginning the installation process, you must carefully determine compatible versions of Keras, TensorFlow, and Anaconda. Most of the installation failures I've seen are due to incompatible versions.

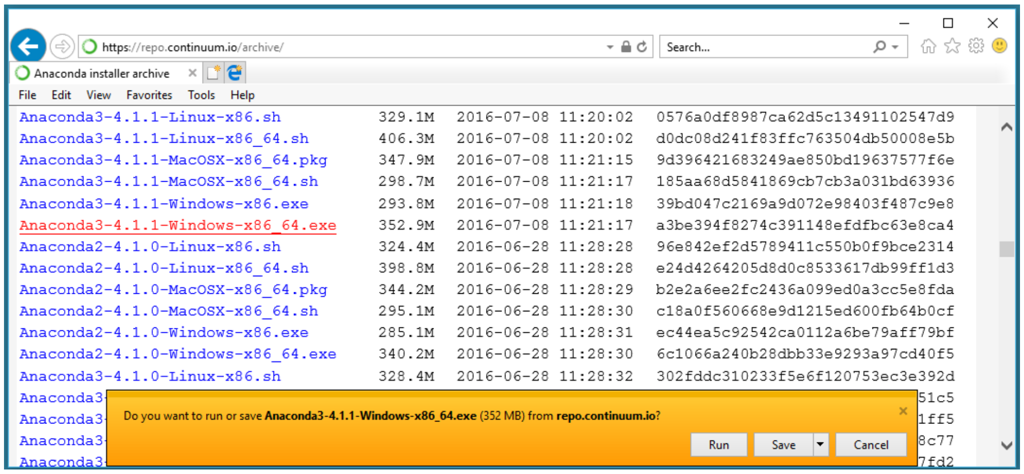
The first step is to determine which version of Keras you want to use. In general, you'll want to install the most recent stable version of Keras. However, this e-book is based on Keras version 1.7.0 rather than the most recent version, along with Anaconda3 4.1.1 (which contains Python 3.5.2) and TensorFlow 2.1.5. I'm confident the code in this e-book will work with newer versions of Keras with few, if any, modifications needed.

Before installing Anaconda/Python, you should check your machine to determine if you already have an existing Python installation. The simplest scenario is when your machine doesn't have an existing Python installed, and you can proceed. If, however, you already have one or more versions of Python installed, you should either uninstall them all (if feasible) or note their installation locations, if uninstalling them is not feasible. With multiple Python instances installed, you may run into some Python versioning issues at some point.

To install Anacoinda3 4.1.1, you can either do an internet search for "archive Anaconda install" or go directly to this location. See Figure 1-2.

On that page, you'll see many different Anaconda distributions. Be careful here; even though I've installed and uninstalled Anaconda for Keras/TensorFlow dozens of times, I've selected the incorrect version of Anaconda several times.

The Anaconda distribution contains over 500 Python packages that are compatible with each other. Some packages, such as NumPy and SciPy, are absolutely essential. Some packages are specific to a field of study, such as BioPython for molecular biology, and some are not essential, but very useful, such as MatPlotLib for creating graphs and charts. You can view the complete list of packages included with Anaconda here.



https://repo.continuum.io/archive/
Search...
Anaconda installer archive
File Edit View Favorites Tools Help

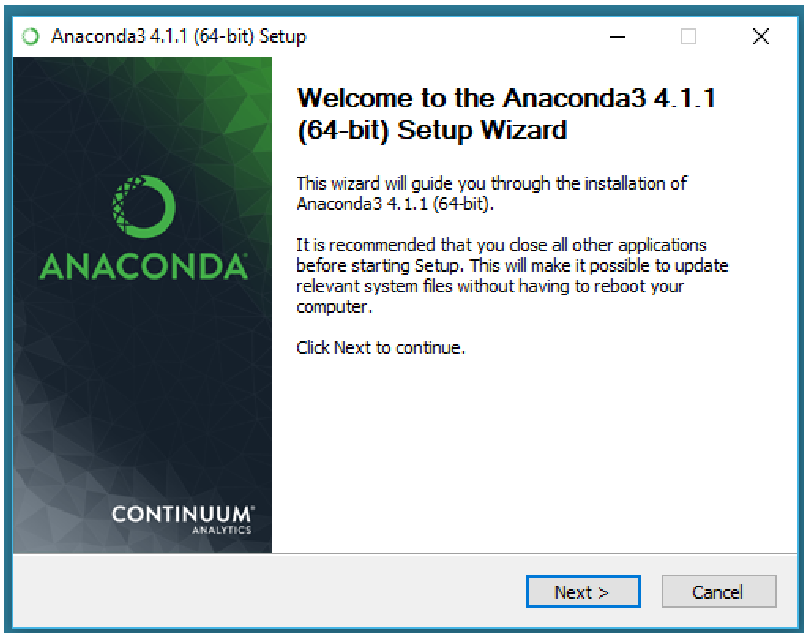
```
Anaconda3-4.1.1-Linux-x86.sh          329.1M  2016-07-08 11:20:02  0576a0df8987ca62d5c13491102547d9
Anaconda3-4.1.1-Linux-x86_64.sh       406.3M  2016-07-08 11:20:02  d0dc08d241f83ffc763504db50008e5b
Anaconda3-4.1.1-MacOSX-x86_64.pkg     347.9M  2016-07-08 11:21:15  9d396421683249ae850bd19637577f6e
Anaconda3-4.1.1-MacOSX-x86_64.sh      298.7M  2016-07-08 11:21:17  185aa68d5841869cb7cb3a031bd63936
Anaconda3-4.1.1-Windows-x86.exe       293.8M  2016-07-08 11:21:18  39bd047c2169a9d072e98403f487c9e8
Anaconda3-4.1.1-Windows-x86_64.exe    352.9M  2016-07-08 11:21:17  a3be394f8274c391148efdfbc63e8ca4
Anaconda2-4.1.0-Linux-x86.sh          324.4M  2016-06-28 11:28:28  96e842ef2d5789411c550b0f9bce2314
Anaconda2-4.1.0-Linux-x86_64.sh       398.8M  2016-06-28 11:28:28  e24d4264205d8d0c8533617db99ff1d3
Anaconda2-4.1.0-MacOSX-x86_64.pkg     344.2M  2016-06-28 11:28:29  b2e2a6ee2fc2436a099ed0a3cc5e8fda
Anaconda2-4.1.0-MacOSX-x86_64.sh      295.1M  2016-06-28 11:28:30  c18a0f560668e9d1215ed600fb64b0cf
Anaconda2-4.1.0-Windows-x86.exe       285.1M  2016-06-28 11:28:31  ec44ea5c92542ca0112a6be79aff79bf
Anaconda2-4.1.0-Windows-x86_64.exe    340.2M  2016-06-28 11:28:30  6c1066a240b28dbb33e9293a97cd40f5
Anaconda3-4.1.0-Linux-x86.sh          328.4M  2016-06-28 11:28:32  302fddc310233f5e6f120753ec3e392d
Anaconda3                                                                                      51c5
Anaconda3                                                                                      1ff5
Anaconda3                                                                                      8c77
Anaconda3                                                                                      7fd2
```

Do you want to run or save **Anaconda3-4.1.1-Windows-x86_64.exe** (352 MB) from **repo.continuum.io**?

Figure 1-2: Find Correct Anaconda Archived Install Link

Because I'm using a 64-bit Windows machine, and I want Python 3 with Anaconda version 4.1.1, I will click the link **Anaconda3-4.1.1-Windows-x86_64.exe**. This will launch a self-extracting installation program. You can select the **Run** option.

To recap, at this point you've determined which versions of Keras (1.7.0) and TensorFlow (2.1.5) you want to use, and then determined which version of Anaconda to use (Anaconda3 4.1.1 for a 64-bit Windows machine in this example), and are now beginning the Anaconda installation process.



Welcome to the Anaconda3 4.1.1 (64-bit) Setup Wizard

This wizard will guide you through the installation of Anaconda3 4.1.1 (64-bit).

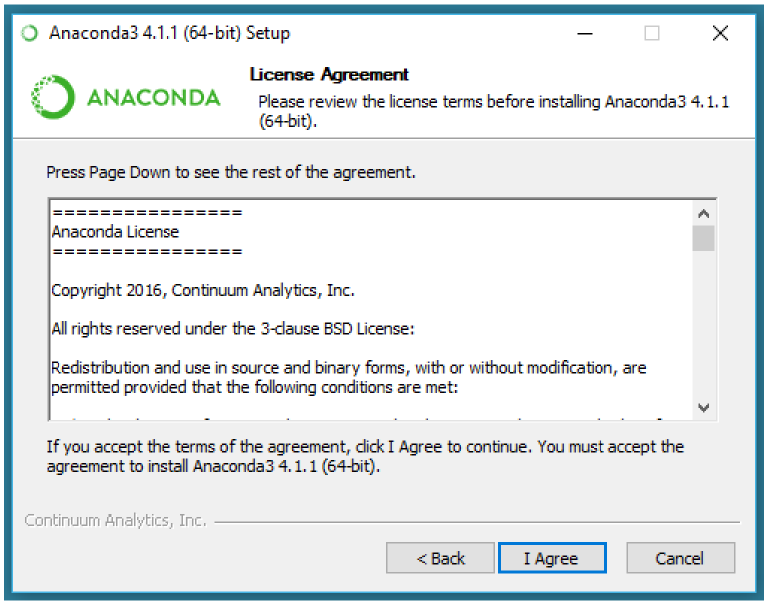
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Figure 1-3: Anaconda Installation Welcome

A few seconds after you click **Run**, the Anaconda installation Welcome window will appear, as shown in Figure 1-3. Click **Next**.

You will see the Anaconda License Agreement window, as shown in Figure 1-4. Click **I Agree**.


Anaconda3 4.1.1 (64-bit) Setup

License Agreement

Please review the license terms before installing Anaconda3 4.1.1 (64-bit).

Press Page Down to see the rest of the agreement.

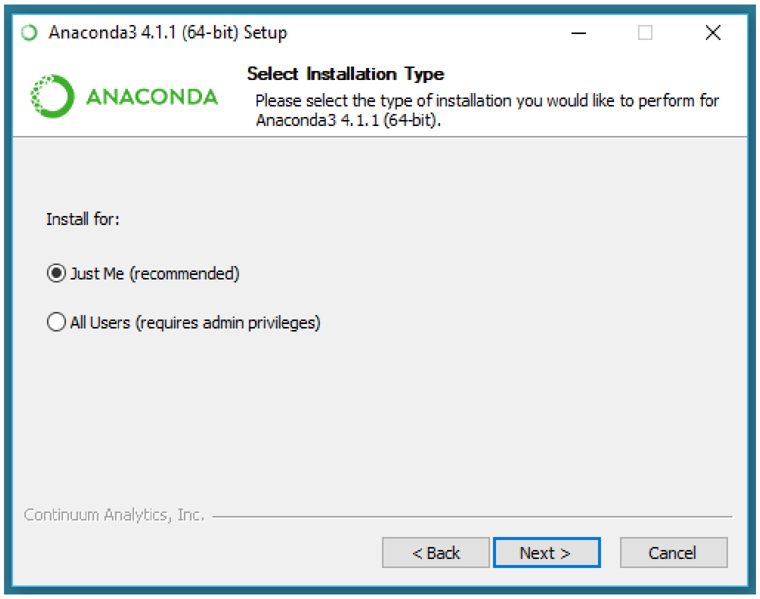
```
================
Anaconda License
================

Copyright 2016, Continuum Analytics, Inc.

All rights reserved under the 3-clause BSD License:

Redistribution and use in source and binary forms, with or without modification, are
permitted provided that the following conditions are met:
```

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Anaconda3 4.1.1 (64-bit).

Continuum Analytics, Inc.

Figure 1-4: Anaconda License Agreement

You will see the Select Installation Type window, as shown in Figure 1-5. I strongly suggest you keep the default **Just Me (recommended)** option. This will reduce the likelihood of Python versioning collisions if there are multiple user accounts on your machine. Click **Next**.


Anaconda3 4.1.1 (64-bit) Setup

Select Installation Type

Please select the type of installation you would like to perform for Anaconda3 4.1.1 (64-bit).

Install for:

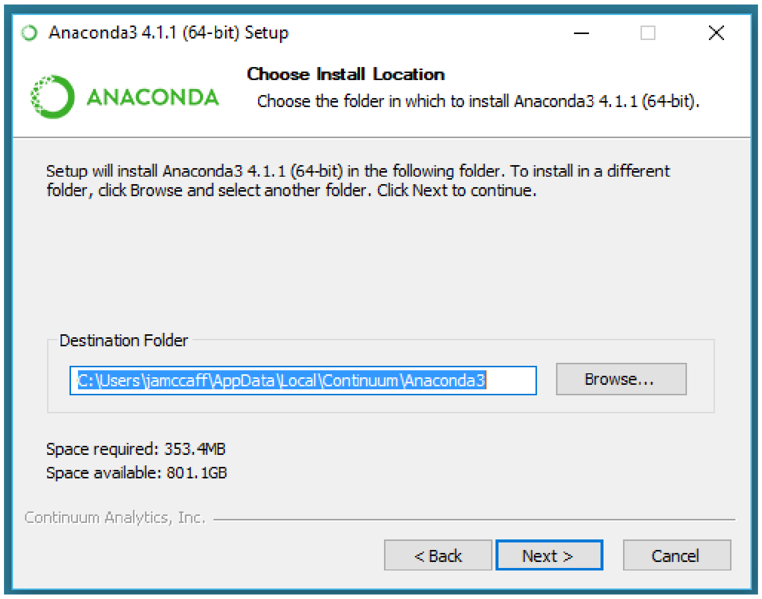
● Just Me (recommended)

○ All Users (requires admin privileges)

Continuum Analytics, Inc.

Figure 1-5: Anaconda Installation Type

Next, you'll see the Choose Install Location information. You should accept the default location (**C:\Users\<user>\AppData\Local\Continuum\Anaconda3** on Windows) if possible, because some Python packages may assume this location. Click **Next**.


Anaconda3 4.1.1 (64-bit) Setup

Choose Install Location

Choose the folder in which to install Anaconda3 4.1.1 (64-bit).

Setup will install Anaconda3 4.1.1 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder

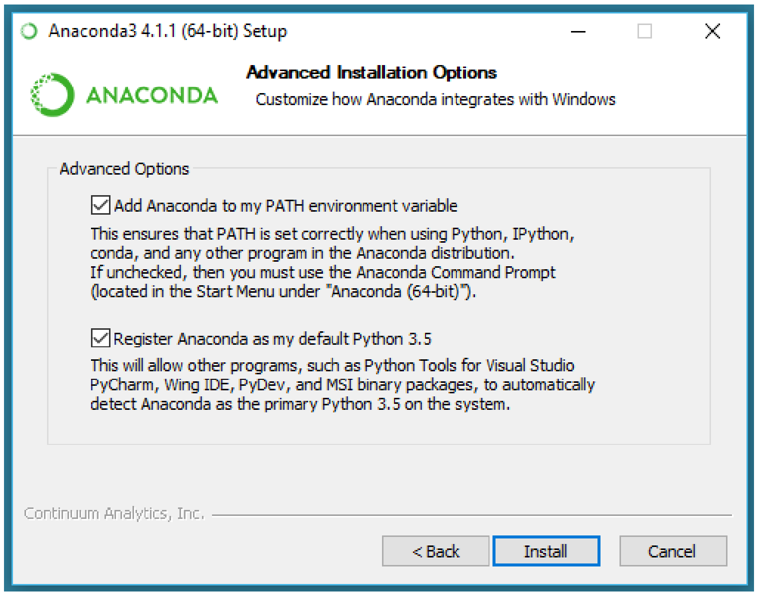
C:\Users\jamccaff\AppData\Local\Continuum\Anaconda3

Space required: 353.4MB
Space available: 801.1GB

Continuum Analytics, Inc.

Figure 1-6: Default Anaconda Python Installation Location

Next, you will see the Advanced Installation Options window. You should accept both default options. The first adds Anaconda to your System PATH variable. The second option makes Anaconda your default Python version. If you have an existing Python installation, this will usually override the existing instance, and you may want to install a Python version selector program. Click **Install**.


Anaconda3 4.1.1 (64-bit) Setup

Advanced Installation Options

Customize how Anaconda integrates with Windows

Advanced Options

☑ Add Anaconda to my PATH environment variable

This ensures that PATH is set correctly when using Python, IPython, conda, and any other program in the Anaconda distribution.
If unchecked, then you must use the Anaconda Command Prompt (located in the Start Menu under "Anaconda (64-bit)").

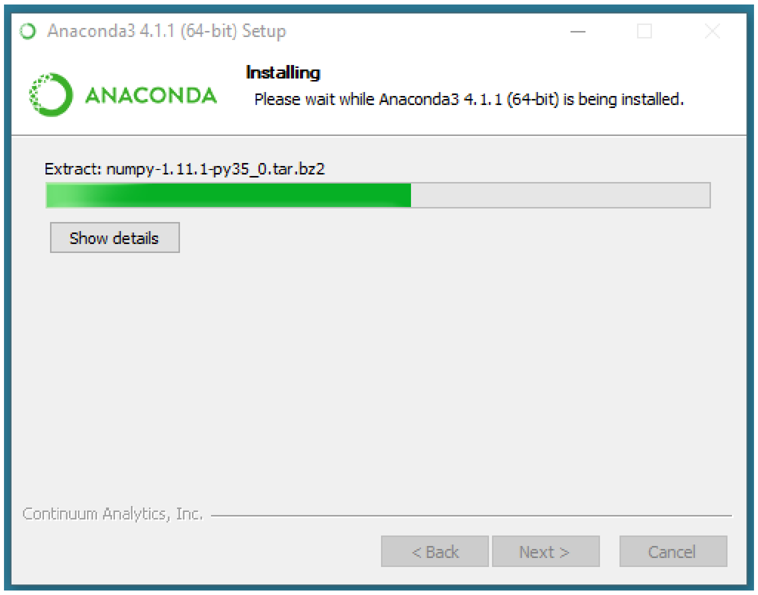
☑ Register Anaconda as my default Python 3.5

This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.5 on the system.

Continuum Analytics, Inc.

Figure 1-7: Installation PATH and Default Python Information

Installing Anaconda takes about 10 to 15 minutes. There will be no options for you to consider, so you don't need to attend to the installation. However, you may want to observe the progress bar and see which packages are installed, such as NumPy, shown in Figure 1-8.


Anaconda3 4.1.1 (64-bit) Setup

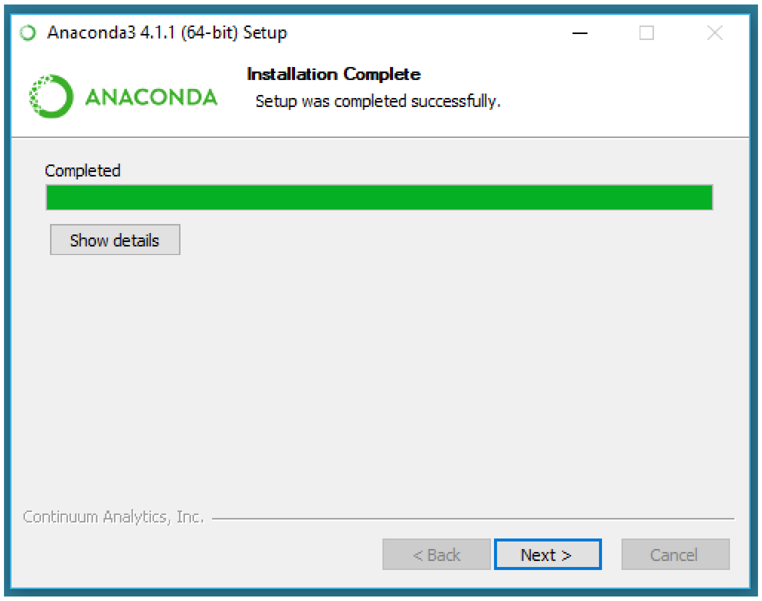
Installing

Please wait while Anaconda3 4.1.1 (64-bit) is being installed.

Extract: numpy-1.11.1-py35_0.tar.bz2

Figure 1-8: Anaconda Installation Progress

When the installation completes successfully, you'll see an Installation Complete window. Click **Next**, as shown in Figure 1-9.


Anaconda3 4.1.1 (64-bit) Setup

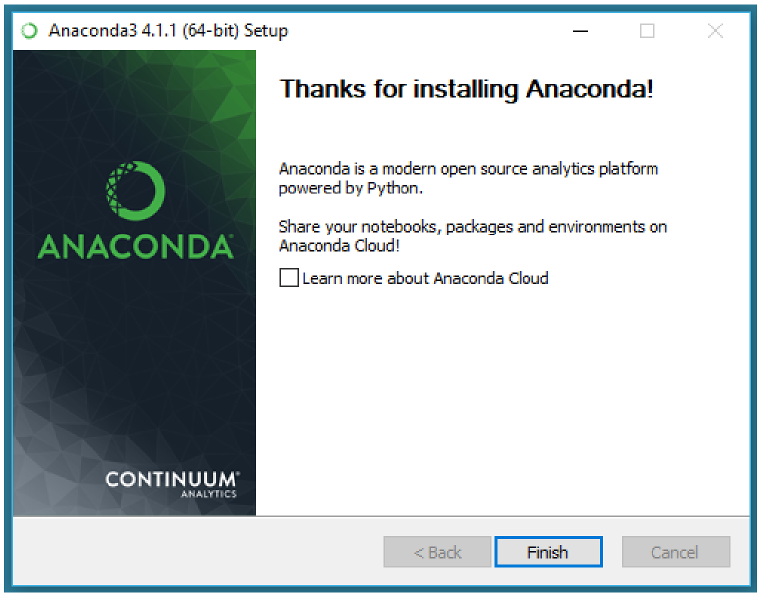
Installation Complete

Setup was completed successfully.

Completed

Figure 1-9: Successful Anaconda Installation Window

You will see a final window, with an option to view marketing information from Continuum, the company that maintains the Anaconda distribution. In Figure 1-10, I unchecked that option, and clicked **Finish**. To summarize, the Anaconda install is a self-extracting executable with a wizard-like process. You can accept all the default options.



Thanks for installing Anaconda!

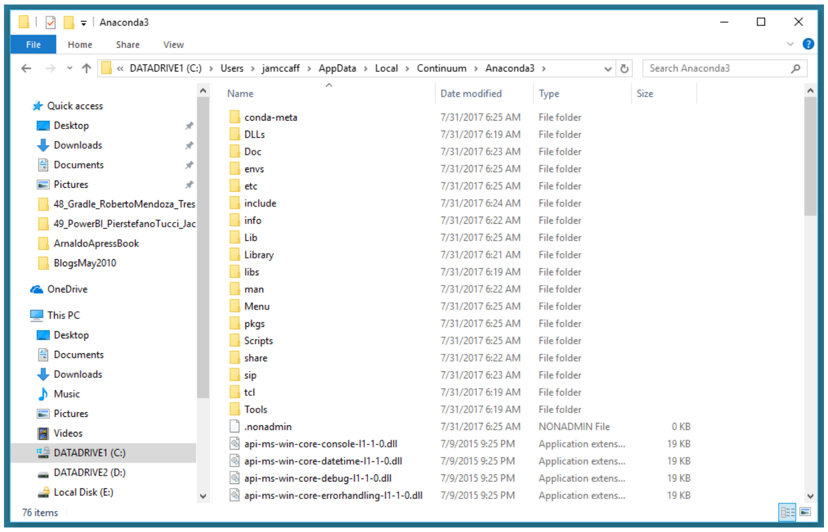
Anaconda is a modern open source analytics platform powered by Python.

Share your notebooks, packages and environments on Anaconda Cloud!

[ ] Learn more about Anaconda Cloud

Figure 1-10: Final Anaconda Installation Window

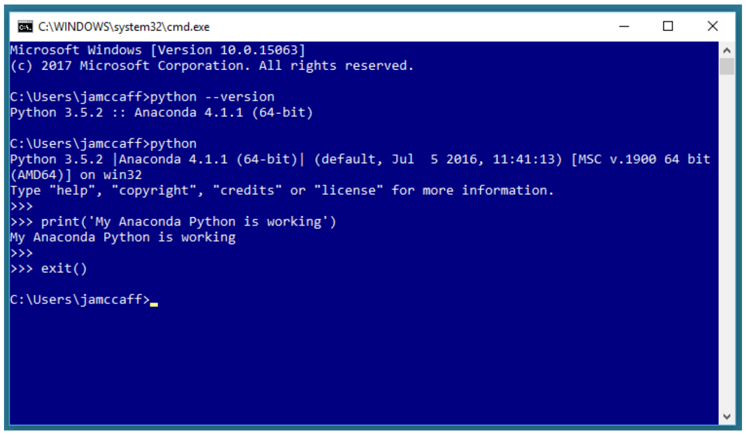
After the Anaconda installation is complete, you may want to take a look at the installation file and directory structure, as shown in Figure 1-11. Notice there are directories named Lib, Library, and libs. Just below the files shown in Figure 1-11 is the python.exe main execution engine.



Name	Date modified	Type	Size
conda-meta	7/31/2017 6:25 AM	File folder	
DLLs	7/31/2017 6:19 AM	File folder	
Doc	7/31/2017 6:23 AM	File folder	
envs	7/31/2017 6:25 AM	File folder	
etc	7/31/2017 6:25 AM	File folder	
include	7/31/2017 6:24 AM	File folder	
info	7/31/2017 6:22 AM	File folder	
Lib	7/31/2017 6:25 AM	File folder	
Library	7/31/2017 6:21 AM	File folder	
libs	7/31/2017 6:19 AM	File folder	
man	7/31/2017 6:22 AM	File folder	
Menu	7/31/2017 6:25 AM	File folder	
pkgs	7/31/2017 6:25 AM	File folder	
Scripts	7/31/2017 6:25 AM	File folder	
share	7/31/2017 6:22 AM	File folder	
sip	7/31/2017 6:23 AM	File folder	
tcl	7/31/2017 6:19 AM	File folder	
Tools	7/31/2017 6:19 AM	File folder	
.nonadmin	7/31/2017 6:25 AM	NONADMIN File	0 KB
api-ms-win-core-console-l1-1-0.dll	7/9/2015 9:25 PM	Application extens...	19 KB
api-ms-win-core-datetime-l1-1-0.dll	7/9/2015 9:25 PM	Application extens...	19 KB
api-ms-win-core-debug-l1-1-0.dll	7/9/2015 9:25 PM	Application extens...	19 KB
api-ms-win-core-errorhandling-l1-1-0.dll	7/9/2015 9:25 PM	Application extens...	19 KB

Figure 1-11: The Anaconda Installation Location

Before installing the TensorFlow and Keras add-on packages, you should verify that your Anaconda Python distribution is working. Open a command shell and enter **`python --version`** (with two hyphens). Python should respond as shown in Figure 1-12.



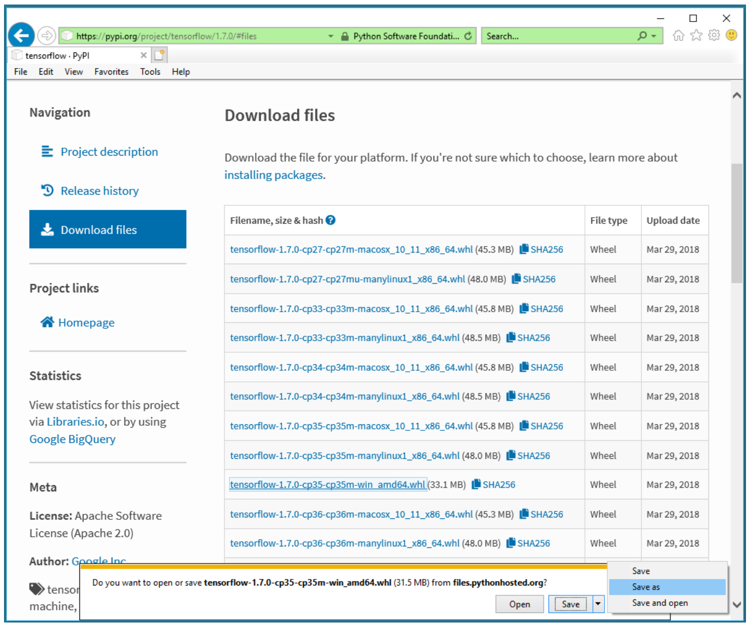
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\jamccaff>python --version
Python 3.5.2 :: Anaconda 4.1.1 (64-bit)

C:\Users\jamccaff>python
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:13) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> print('My Anaconda Python is working')
My Anaconda Python is working
>>>
>>> exit()

C:\Users\jamccaff>_
```

Figure 1-12: Verifying the Anaconda Python Installation

You can test the Python interpreter by issuing the command **`python.`** This will launch the interpreter, and you'll see the >>> prompt. Enter a **`print('hello')`** statement. You can exit the interpreter by typing the **`exit()`** command.

Installing TensorFlow and Keras

The next step is to install TensorFlow, and there are several ways to do it. I recommend using the PyPi (Python Package Index) repository.



Download files

Download the file for your platform. If you're not sure which to choose, learn more about installing packages.

Filename, size & hash	File type	Upload date
tensorflow-1.7.0-cp27-cp27m-macosx_10_11_x86_64.whl (45.3 MB) SHA256	Wheel	Mar 29, 2018
tensorflow-1.7.0-cp27-cp27mu-manylinux1_x86_64.whl (48.0 MB) SHA256	Wheel	Mar 29, 2018
tensorflow-1.7.0-cp33-cp33m-macosx_10_11_x86_64.whl (45.8 MB) SHA256	Wheel	Mar 29, 2018
tensorflow-1.7.0-cp33-cp33m-manylinux1_x86_64.whl (48.5 MB) SHA256	Wheel	Mar 29, 2018
tensorflow-1.7.0-cp34-cp34m-macosx_10_11_x86_64.whl (45.8 MB) SHA256	Wheel	Mar 29, 2018
tensorflow-1.7.0-cp34-cp34m-manylinux1_x86_64.whl (48.5 MB) SHA256	Wheel	Mar 29, 2018
tensorflow-1.7.0-cp35-cp35m-macosx_10_11_x86_64.whl (45.8 MB) SHA256	Wheel	Mar 29, 2018
tensorflow-1.7.0-cp35-cp35m-manylinux1_x86_64.whl (48.0 MB) SHA256	Wheel	Mar 29, 2018
tensorflow-1.7.0-cp35-cp35m-win_amd64.whl (33.1 MB) SHA256	Wheel	Mar 29, 2018
tensorflow-1.7.0-cp36-cp36m-macosx_10_11_x86_64.whl (45.3 MB) SHA256	Wheel	Mar 29, 2018
tensorflow-1.7.0-cp36-cp36m-manylinux1_x86_64.whl (48.0 MB) SHA256	Wheel	Mar 29, 2018

View statistics for this project via Libraries.io, or by using Google BigQuery

License: Apache Software License (Apache 2.0)

Author: Google Inc.

tensor

machine,

Do you want to open or save **tensorflow-1.7.0-cp35-cp35m-win_amd64.whl** (31.5 MB) from **files.pythonhosted.org**?

Figure 1-13: Downloading the TensorFlow .whl File

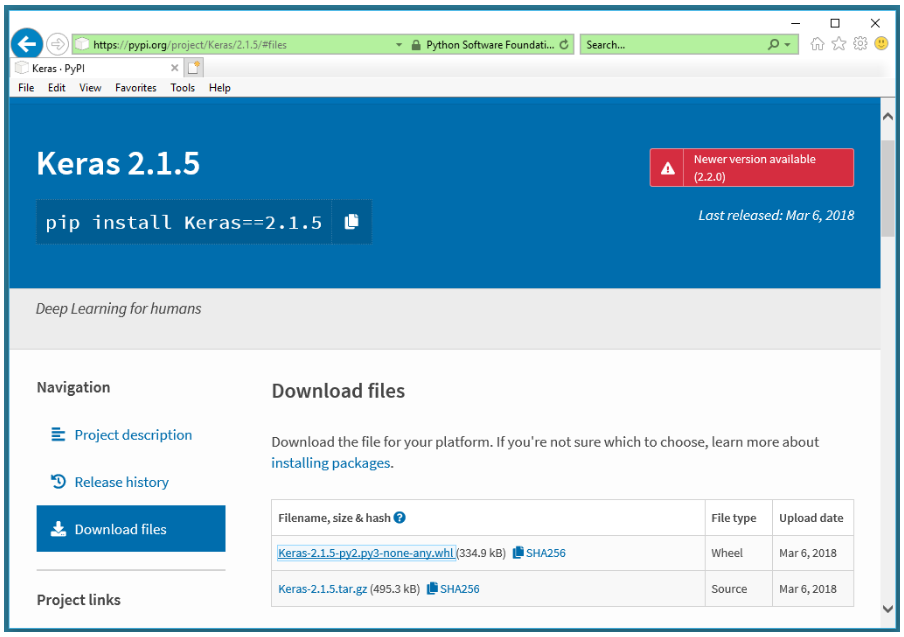
Do an Internet search for "install TensorFlow 1.7.0" or go directly to this page and click the **Download files** link. You'll go to a page that lists .whl files for different types of systems. In my case, because I'm using Python 3.5 and a Windows machine, I clicked on the link **tensorflow-1.7.0-cp35-cp35m-win_amd64.whl** (the **cp35** indicates Python 3.5). See Figure 1-13.

You'll be asked if you want to open or save the .whl file. Click the **Save as** option. You can save the wheel file in any convenient location. I saved at **C:\KerasSuccinctly\Wheels**. Now, open a command shell and navigate to the directory where you saved the TensorFlow .whl file, and enter the following command:

```
C:\KerasSuccinctly\Wheels> pip install tensorflow-1.7.0-cp35-cp35m-
win_amd64.whl
```

The `pip` utility installs Python packages using .whl files. Installation takes less than a minute, and then you'll see this message: "Successfully installed tensorflow-1.7.0".

The process for installing Keras is very much the same. Do an Internet search for "install Keras 2.1.5", or go directly to this webpage and click the link labeled **Download files**. See Figure 1-14.


https://pypi.org/project/Keras/2.1.5/#files
Python Software Foundati...
Search...
Keras - PyPI
File
Edit
View
Favorites
Tools
Help

Keras 2.1.5

pip install Keras==2.1.5

Newer version available (2.2.0)

Last released: Mar 6, 2018

Deep Learning for humans

Navigation

Project description

Release history

Download files

Project links

Download files

Download the file for your platform. If you're not sure which to choose, learn more about installing packages.

Filename, size & hash	File type	Upload date
Keras-2.1.5-py2.py3-none-any.whl (334.9 kB) SHA256	Wheel	Mar 6, 2018
Keras-2.1.5.tar.gz (495.3 kB) SHA256	Source	Mar 6, 2018

Figure 1-14: Downloading the Keras .whl File

On the next page, click the **Keras-2.1.5-py2.py3-none-any.whl** link, and you'll be prompted to open or save. Click **Save as** and save the .whl file in a convenient location, for example, **C:\KerasSuccinctly\Wheels**. Launch a command shell, navigate to the directory where you saved the Keras .whl file, and enter the following command:

```
C:\KerasSuccinctly\Wheels> pip install Keras-2.1.5-py2.py3-none-any.whl
```

Installation is very quick, and you'll see a "Successfully installed Keras-2.1.5" message. You're now ready to write Keras programs.

Editing and running Keras programs

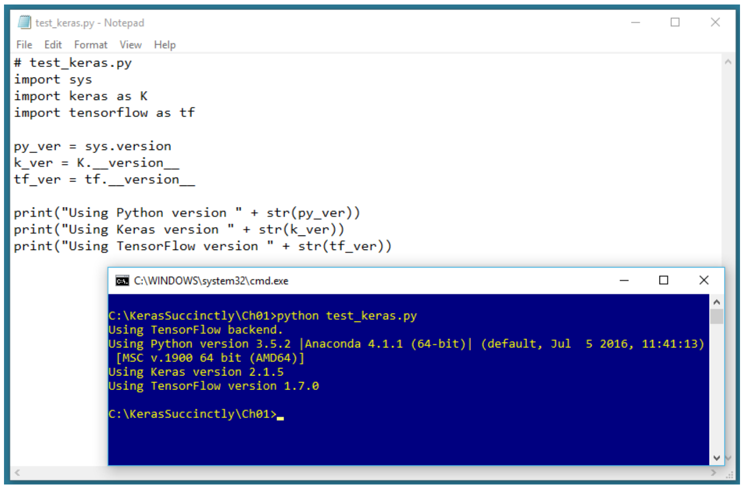
Because a Keras program is just a specialized Python program, you can use any Python editing environment. If you are relatively new to Python, selecting a Python editor or IDE (integrated development environment) can be a confusing task because there are dozens of editors and Python IDEs to choose from.

I often use plain old Notepad, or sometimes the slightly more powerful Notepad++. Neither of these give you built-in debugging functionality, so debugging means you must insert **`print()`** statements to inspect the values of variables and objects. And there's no integrated **`run`** command, so you run programs from a shell.

Code Listing 1-1: Checking the Keras and TensorFlow Versions

```
# test_keras.py
import sys
import keras as K
import tensorflow as tf

py_ver = sys.version
k_ver = K.__version__
tf_ver = tf.__version__

print("Using Python version " + str(py_ver))
print("Using Keras version " + str(k_ver))
print("Using TensorFlow version " + str(tf_ver))
```

Launch the Notepad text editor (or any other editor you're familiar with), and copy-paste the code in Code Listing 1-1. Save the file as **test_keras.py** in any convenient directory. Open a command shell, navigate to the directory that holds your Python file, and execute by entering **`python test_keras.py`**, as shown in Figure 1-15.

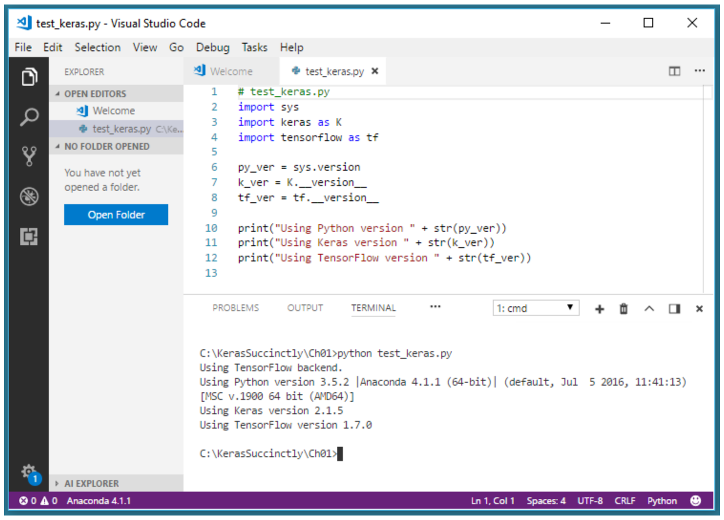


```
# test_keras.py
import sys
import keras as K
import tensorflow as tf

py_ver = sys.version
k_ver = K.__version__
tf_ver = tf.__version__

print("Using Python version " + str(py_ver))
print("Using Keras version " + str(k_ver))
print("Using TensorFlow version " + str(tf_ver))
```

```
C:\KerasSuccinctly\Ch01>python test_keras.py
Using TensorFlow backend.
Using Python version 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:13)
 [MSC v.1900 64 bit (AMD64)]
Using Keras version 2.1.5
Using TensorFlow version 1.7.0

C:\KerasSuccinctly\Ch01>_
```

Figure 1-15: Using Notepad and a Command Shell

Many of my colleagues use Visual Studio Code (VS Code), a free, open-source, cross-platform, multi-language IDE. Installing VS Code is quick and easy, and adding Python support is just a matter of a couple of clicks. See this webpage.



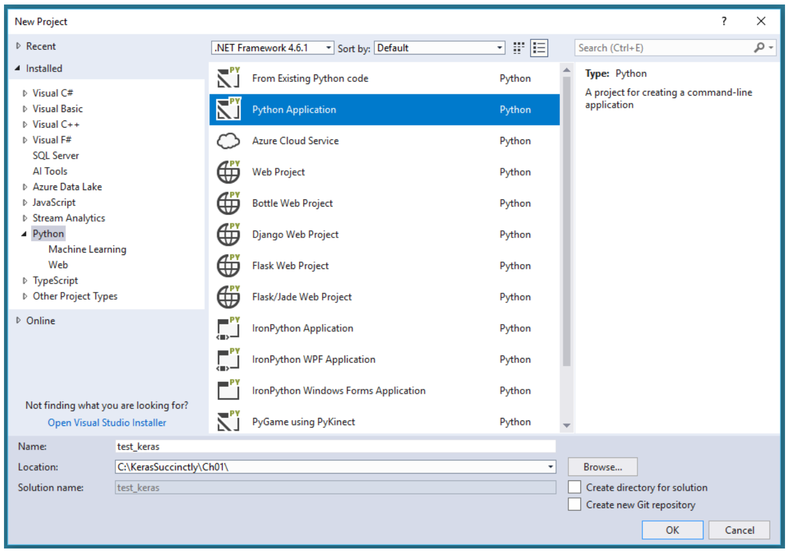
```
# test_keras.py
import sys
import keras as K
import tensorflow as tf

py_ver = sys.version
k_ver = K.__version__
tf_ver = tf.__version__

print("Using Python version " + str(py_ver))
print("Using Keras version " + str(k_ver))
print("Using TensorFlow version " + str(tf_ver))
```

```
C:\KerasSuccinctly\Ch01>python test_keras.py
Using TensorFlow backend.
Using Python version 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:13)
[MSC v.1900 64 bit (AMD64)]
Using Keras version 2.1.5
Using TensorFlow version 1.7.0

C:\KerasSuccinctly\Ch01>
```

Figure 1-16: Using Visual Studio Code

Figure 1-16 shows an example using VS Code. There are many advantages of using VS Code, including IntelliSense auto-complete, pretty formatting, and integrated debugging. However, unlike Notepad, VS Code does have a non-trivial learning curve you'll have to deal with. See this tutorial for help.

Another option for editing and running Keras programs is the heavyweight Visual Studio (VS) IDE. The default configuration of VS does not support editing Python programs, but you can install the Python Tools for Visual Studio add-in. With the add-in installed, you get full Python language support, as shown in Figures 1-17 and 1-18.



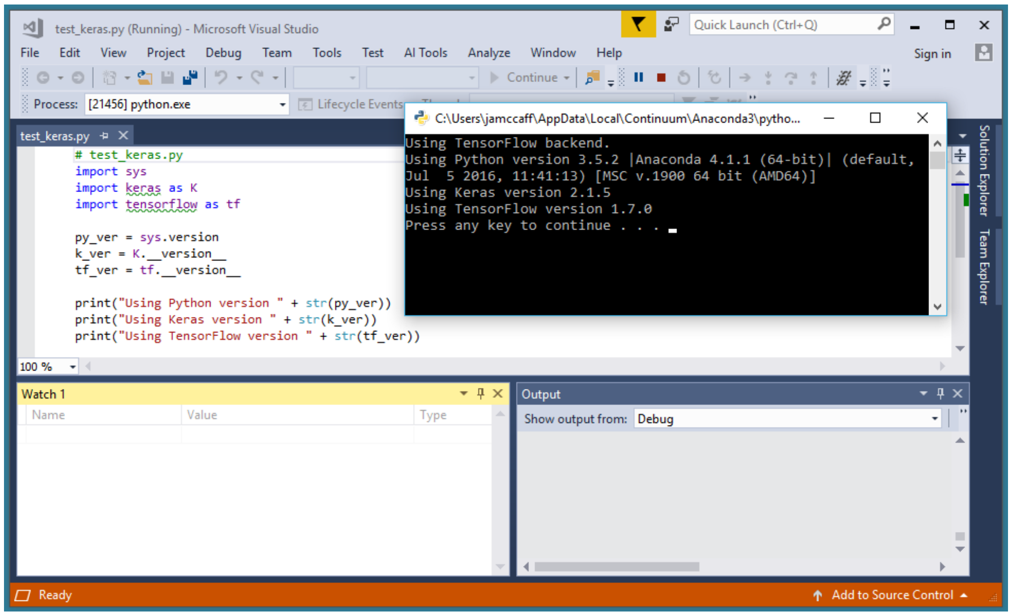
From Existing Python code	Python
Python Application	Python
Azure Cloud Service	Python
Web Project	Python
Bottle Web Project	Python
Django Web Project	Python
Flask Web Project	Python
Flask/Jade Web Project	Python
IronPython Application	Python
IronPython WPF Application	Python
IronPython Windows Forms Application	Python
PyGame using PyKinect	Python

Type: Python
A project for creating a command-line application

Name: test_keras
Location: C:\KerasSuccinctly\Ch01\
Solution name: test_keras
☐ Create directory for solution
☐ Create new Git repository

Figure 1-17: Creating a Python Project using the Visual Studio IDE

One advantage of using Visual Studio is that you get support for all kinds of additional functionality, such as data connectors to SQL Server databases and Azure data sources. The main disadvantage of Visual Studio is that it has a steep learning curve.



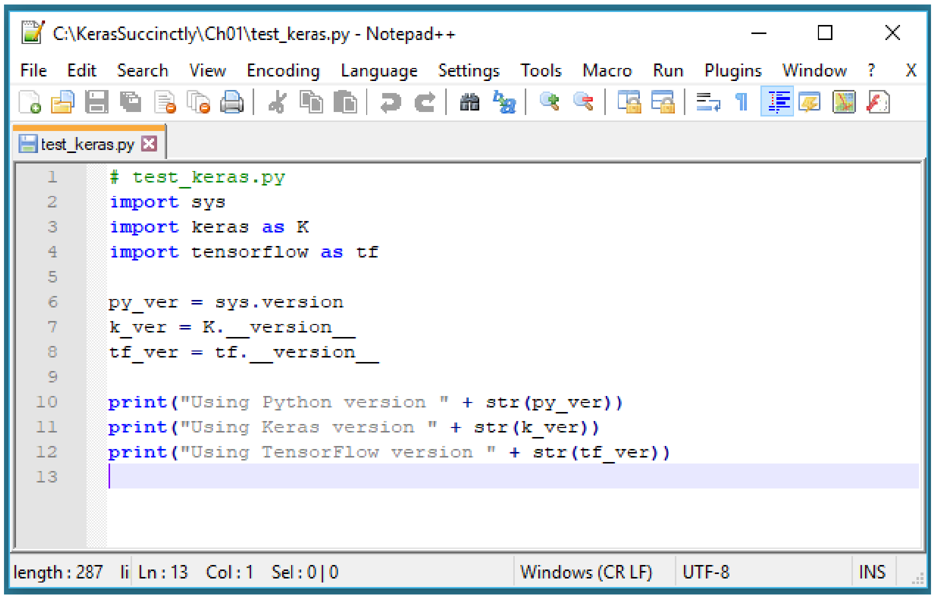
test_keras.py (Running) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Debug Team Tools Test AI Tools Analyze Window Help
Sign in
Continue
Process: [21456] python.exe
Lifecycle Events
C:\Users\jamccaff\AppData\Local\Continuum\Anaconda3\pytho...
test_keras.py
100 %
Watch 1
Name Value Type
Output
Show output from: Debug
Solution Explorer
Team Explorer
Ready
Add to Source Control

```
# test_keras.py
import sys
import keras as K
import tensorflow as tf

py_ver = sys.version
k_ver = K.__version__
tf_ver = tf.__version__

print("Using Python version " + str(py_ver))
print("Using Keras version " + str(k_ver))
print("Using TensorFlow version " + str(tf_ver))
```

```
Using TensorFlow backend.
Using Python version 3.5.2 |Anaconda 4.1.1 (64-bit)| (default,
Jul  5 2016, 11:41:13) [MSC v.1900 64 bit (AMD64)]
Using Keras version 2.1.5
Using TensorFlow version 1.7.0
Press any key to continue . . .
```

Figure 1-18: Running a Python Program from the Visual Studio IDE

If you are familiar with any Python editor or development environment, my recommendation is to continue using that system. If you are relatively new to programming, my recommendation is to start with simple Notepad, because it has essentially no learning curve. If you are an experienced developer but new to Python, my recommendation is to try VS Code.



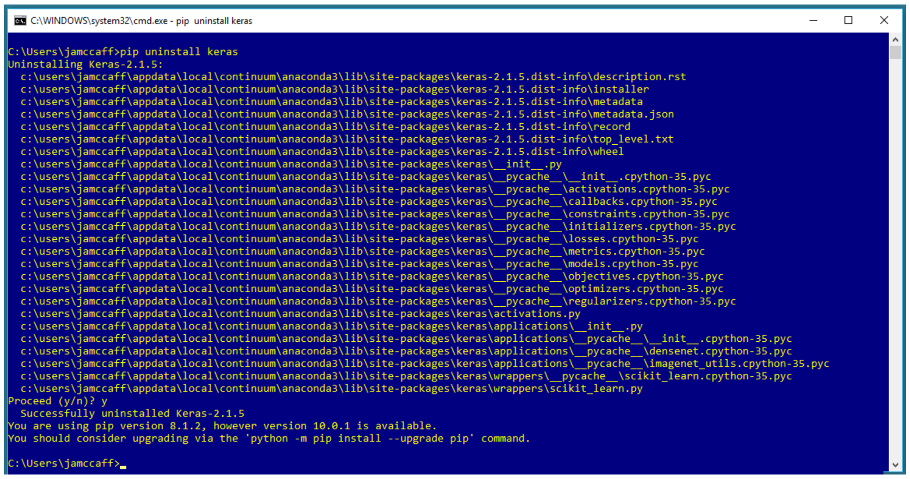
C:\KerasSuccinctly\Ch01\test_keras.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
test_keras.py
length : 287 li Ln : 13 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

```
# test_keras.py
import sys
import keras as K
import tensorflow as tf

py_ver = sys.version
k_ver = K.__version__
tf_ver = tf.__version__

print("Using Python version " + str(py_ver))
print("Using Keras version " + str(k_ver))
print("Using TensorFlow version " + str(tf_ver))
```

Figure 1-19: Using the Notepad++ Editor

Uninstalling Keras

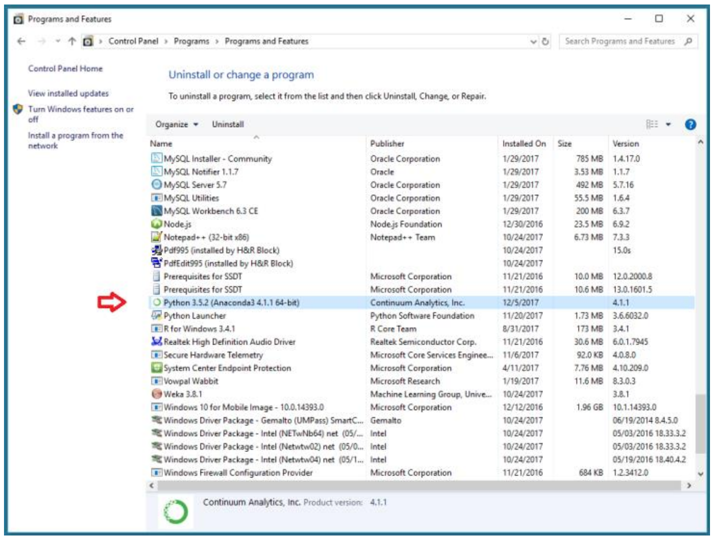
Anaconda, Keras, and TensorFlow all have quick, easy, and reliable uninstall procedures. To uninstall Keras, launch a command shell and issue the command **`pip uninstall keras`**. The Keras package will be removed from your Python system, as shown in (the slightly edited-for-size) Figure 1-20. To uninstall TensorFlow, issue **`pip uninstall tensorflow`**.



```
C:\Users\jamccaff>pip uninstall keras
Uninstalling Keras-2.1.5:
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras-2.1.5.dist-info\description.rst
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras-2.1.5.dist-info\installer
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras-2.1.5.dist-info\metadata
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras-2.1.5.dist-info\metadata.json
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras-2.1.5.dist-info\record
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras-2.1.5.dist-info\top_level.txt
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras-2.1.5.dist-info\wheel
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__init__.py
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__pycache__\__init__.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__pycache__\activations.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__pycache__\callbacks.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__pycache__\constraints.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__pycache__\initializers.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__pycache__\losses.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__pycache__\metrics.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__pycache__\models.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__pycache__\objectives.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__pycache__\optimizers.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\__pycache__\regularizers.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\activations.py
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\applications\__init__.py
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\applications\__pycache__\__init__.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\applications\__pycache__\densenet.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\applications\__pycache__\imagenet_utils.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\wrappers\__pycache__\scikit_learn.cpython-35.pyc
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\keras\wrappers\scikit_learn.py
Proceed (y/n)? y
  Successfully uninstalled Keras-2.1.5
You are using pip version 8.1.2, however version 10.0.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

C:\Users\jamccaff>_
```

Figure 1-20: Uninstalling Keras

To uninstall Python on Windows, use the **Programs and Features** section of the **Control Panel**:



Programs and Features
Control Panel › Programs › Programs and Features
Search Programs and Features
Control Panel Home
View installed updates
Turn Windows features on or off
Install a program from the network
Organize
Uninstall

Uninstall or change a program

To uninstall a program, select it from the list and then click Uninstall, Change, or Repair.

Name	Publisher	Installed On	Size	Version
MySQL Installer - Community	Oracle Corporation	1/29/2017	785 MB	1.4.17.0
MySQL Notifier 1.1.7	Oracle	1/29/2017	3.53 MB	1.1.7
MySQL Server 5.7	Oracle Corporation	1/29/2017	492 MB	5.7.16
MySQL Utilities	Oracle Corporation	1/29/2017	55.5 MB	1.6.4
MySQL Workbench 6.3 CE	Oracle Corporation	1/29/2017	200 MB	6.3.7
Node.js	Node.js Foundation	12/30/2016	23.5 MB	6.9.2
Notepad++ (32-bit x86)	Notepad++ Team	10/24/2017	6.73 MB	7.3.3
Pdf995 (installed by H&R Block)		10/24/2017		15.0s
PdfEdit995 (installed by H&R Block)		10/24/2017		
Prerequisites for SSDT	Microsoft Corporation	11/21/2016	10.0 MB	12.0.2000.8
Prerequisites for SSDT	Microsoft Corporation	11/21/2016	10.6 MB	13.0.1601.5
Python 3.5.2 (Anaconda3 4.1.1 64-bit)	Continuum Analytics, Inc.	12/5/2017		4.1.1
Python Launcher	Python Software Foundation	11/20/2017	1.73 MB	3.6.6032.0
R for Windows 3.4.1	R Core Team	8/31/2017	173 MB	3.4.1
Realtek High Definition Audio Driver	Realtek Semiconductor Corp.	11/21/2016	30.6 MB	6.0.1.7945
Secure Hardware Telemetry	Microsoft Core Services Enginee...	11/6/2017	92.0 KB	4.0.8.0
System Center Endpoint Protection	Microsoft Corporation	4/11/2017	7.76 MB	4.10.209.0
Vowpal Wabbit	Microsoft Research	1/19/2017	11.6 MB	8.3.0.3
Weka 3.8.1	Machine Learning Group, Unive...	10/24/2017		3.8.1
Windows 10 for Mobile Image - 10.0.14393.0	Microsoft Corporation	12/12/2016	1.96 GB	10.1.14393.0
Windows Driver Package - Gemalto (UMPass) SmartC...	Gemalto	10/24/2017		06/19/2014 8.4.5.0
Windows Driver Package - Intel (NETwNb64) net (05/...	Intel	10/24/2017		05/03/2016 18.33.3.2
Windows Driver Package - Intel (Netwtw02) net (05/0...	Intel	10/24/2017		05/03/2016 18.33.3.2
Windows Driver Package - Intel (Netwtw04) net (05/1...	Intel	10/24/2017		05/19/2016 18.40.4.2
Windows Firewall Configuration Provider	Microsoft Corporation	11/21/2016	684 KB	1.2.3412.0

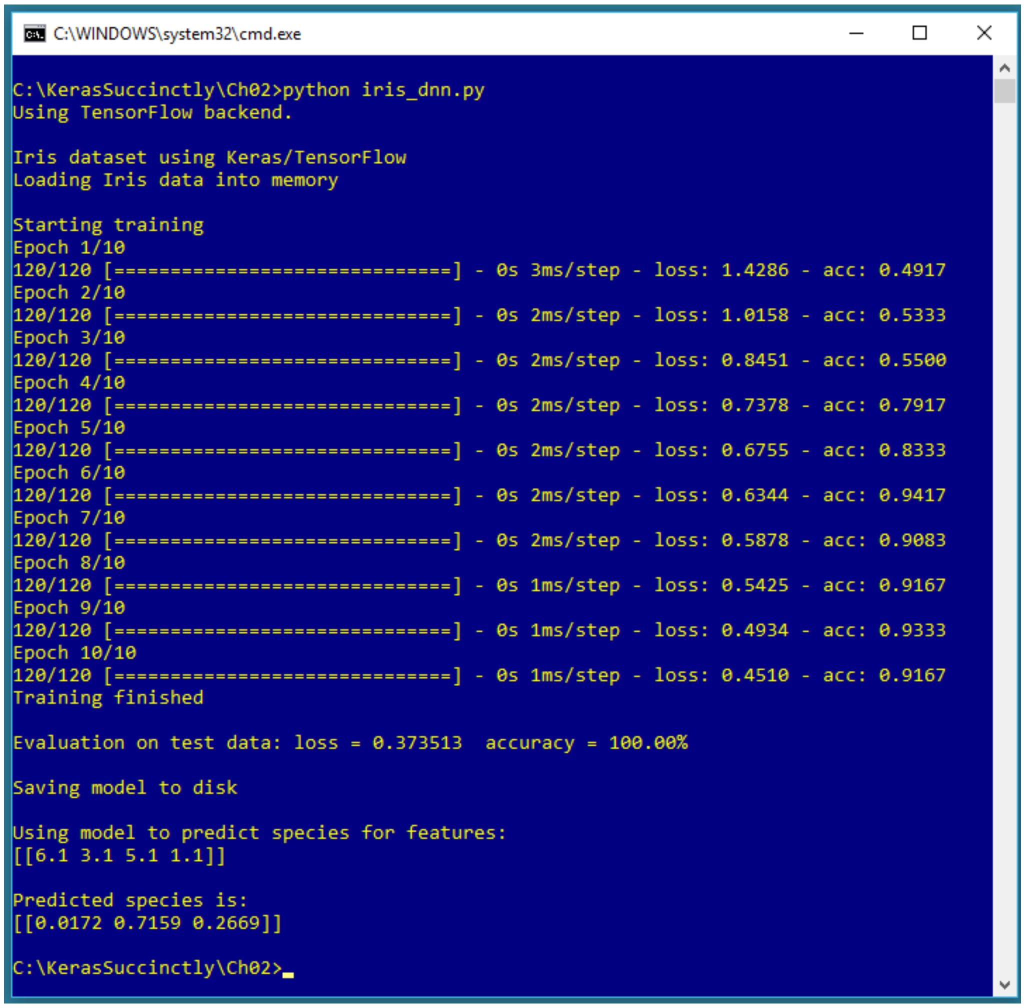
Continuum Analytics, Inc. Product version: 4.1.1

Figure 1-21: Uninstalling Anaconda Python

If you become a regular user of Keras, eventually you'll want to upgrade your version. Although the pip utility supports an upgrade command, I recommend just deleting your current version, then installing the new version.

Chapter 2 Multiclass Classification

The goal of multiclass classification is to make a prediction where the variable to predict can take on one of a set of three or more discrete values. For example, you might want to predict the political party affiliation of a person (democrat, republican, or other) based on their age, sex, annual income, and so on.



```
C:\KerasSuccinctly\Ch02>python iris_dnn.py
Using TensorFlow backend.

Iris dataset using Keras/TensorFlow
Loading Iris data into memory

Starting training
Epoch 1/10
120/120 [==============================] - 0s 3ms/step - loss: 1.4286 - acc: 0.4917
Epoch 2/10
120/120 [==============================] - 0s 2ms/step - loss: 1.0158 - acc: 0.5333
Epoch 3/10
120/120 [==============================] - 0s 2ms/step - loss: 0.8451 - acc: 0.5500
Epoch 4/10
120/120 [==============================] - 0s 2ms/step - loss: 0.7378 - acc: 0.7917
Epoch 5/10
120/120 [==============================] - 0s 2ms/step - loss: 0.6755 - acc: 0.8333
Epoch 6/10
120/120 [==============================] - 0s 2ms/step - loss: 0.6344 - acc: 0.9417
Epoch 7/10
120/120 [==============================] - 0s 2ms/step - loss: 0.5878 - acc: 0.9083
Epoch 8/10
120/120 [==============================] - 0s 1ms/step - loss: 0.5425 - acc: 0.9167
Epoch 9/10
120/120 [==============================] - 0s 1ms/step - loss: 0.4934 - acc: 0.9333
Epoch 10/10
120/120 [==============================] - 0s 1ms/step - loss: 0.4510 - acc: 0.9167
Training finished

Evaluation on test data: loss = 0.373513  accuracy = 100.00%

Saving model to disk

Using model to predict species for features:
[[6.1 3.1 5.1 1.1]]

Predicted species is:
[[0.0172 0.7159 0.2669]]

C:\KerasSuccinctly\Ch02>_
```

Figure 2-1: Multiclass Classification using Keras

The screenshot in Figure 2-1 shows a demonstration of multiclass classification. The demo program begins by loading 120 training data items and 30 test data items into memory. Each item represents an iris flower where the four predictor variables are sepal length, sepal width, petal length, and petal width (a sepal is a leaf-like structure). The variable to predict is **`species`**. There are three possible species: **`setosa`**, **`versicolor`**, and **`virginica`**.

Behind the scenes, the demo program creates a 4-(5-6)-3 neural network, that is, one with four input values (one for each predictor variable), two hidden layers with five and six nodes respectively, and three output nodes (one for each possible species). The demo program trains the neural network model using 10 epochs.

After training completes, the trained model achieves a prediction accuracy of 100.00% on the test data (30 of 30 correct). The demo concludes by making a prediction for a new, previously unseen iris flower with predictor values (6.1, 3.1, 5.1, 1.1). The predicted probabilities are (0.0172, 0.7159, 0.2669), and because the second value is largest, the prediction is **`versicolor`**.

Understanding the data

Fischer's Iris dataset is one of the most well-known benchmark datasets in statistics and machine learning. There are a total of 150 items. The raw data looks like:

```
5.1, 3.5, 1.4, 0.2, setosa
7.0, 3.2, 4.7, 1.4, versicolor
6.3, 3.3, 6.0, 2.5, virginica
```

The raw data was prepared by one-hot encoding the class labels, but the feature values were not normalized as is usually done:

```
5.1, 3.5, 1.4, 0.2, 1, 0, 0
7.0, 3.2, 4.7, 1.4, 0, 1, 0
6.3, 3.3, 6.0, 2.5, 0, 0, 1
```

After encoding, the full dataset was split into a 120-item set for training, and a 30-item test set to be used after training for model evaluation. Because the data has four dimensions, it's not possible to easily visualize it in a two-dimensional graph, but you can get a rough idea of the data from the partial graph in Figure 2-2.

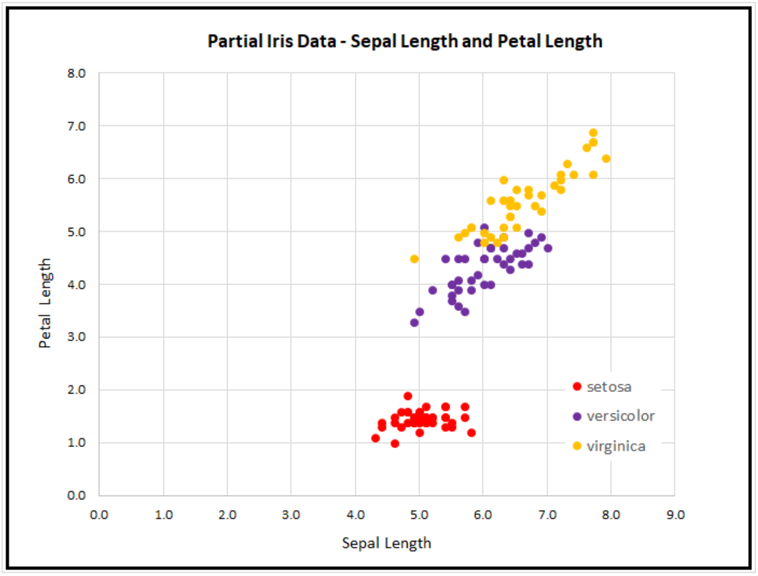


Figure 2-2: Iris Data

As the graph shows, the Iris Dataset is almost too simple. The class **setosa** can be easily distinguished from **versicolor** and **virginica**. Furthermore, the classes **versicolor** and **virginica** are nearly linearly separable. However, the Iris Dataset serves well as a simple example.

By the way, there are actually at least two different versions of Fisher's Iris Data that are in common use. The original data was collected in 1935, and then published by Fisher in 1936. However, at some point in time, a couple of the original values for **setosa** items were incorrectly copied, and years later made their way onto the Internet. This isn't serious, since the datasets are now used just for a teaching example, rather than for serious research.

The Iris program

The complete program that generated the output shown in Figure 2-1 is shown in Code Listing 2-1. The program begins with comments for the program file name and the versions of Python, TensorFlow and Keras used, and then imports the NumPy, Keras, TensorFlow, and OS packages:

```
# iris_dnn.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0
import numpy as np
import keras as K
import tensorflow as tf
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
```

In a non-demo scenario, you'd want to include additional details in the comments. Because Keras and TensorFlow are under rapid development, you should always document which versions are being used. Version incompatibilities can be a significant problem when working with Keras and other open-source software.

Code Listing 2-1: Iris Multiclass Classification Program

```
# iris_dnn.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0

#
===========================================================================
=======

import numpy as np
import keras as K
import tensorflow as tf
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
```

```
def main():
  # 0. get started
  print("\nIris dataset using Keras/TensorFlow ")
  np.random.seed(4)
  tf.set_random_seed(13)

  # 1. load data
  print("Loading Iris data into memory \n")
  train_file = ".\\Data\\iris_train.txt"
  test_file = ".\\Data\\iris_test.txt"

  train_x = np.loadtxt(train_file, usecols=[0,1,2,3],
   delimiter=",",  skiprows=0, dtype=np.float32)
  train_y = np.loadtxt(train_file, usecols=[4,5,6],
    delimiter=",", skiprows=0, dtype=np.float32)

  test_x = np.loadtxt(test_file, usecols=range(0,4),
   delimiter=",",  skiprows=0, dtype=np.float32)
  test_y = np.loadtxt(test_file, usecols=range(4,7),
    delimiter=",", skiprows=0, dtype=np.float32)

  # 2. define model
  init = K.initializers.glorot_uniform(seed=1)
  simple_adam = K.optimizers.Adam()
  model = K.models.Sequential()
  model.add(K.layers.Dense(units=5, input_dim=4, kernel_initializer=init,
    activation='relu'))
  model.add(K.layers.Dense(units=6, kernel_initializer=init,
    activation='relu'))
  model.add(K.layers.Dense(units=3, kernel_initializer=init,
    activation='softmax'))
  model.compile(loss='categorical_crossentropy',
    optimizer=simple_adam, metrics=['accuracy'])

  # 3. train model
  b_size = 1
  max_epochs = 10
  print("Starting training ")
  h = model.fit(train_x, train_y, batch_size=b_size, epochs=max_epochs,
    shuffle=True, verbose=1)
  print("Training finished \n")

  # 4. evaluate model
  eval = model.evaluate(test_x, test_y, verbose=0)
  print("Evaluation on test data: loss = %0.6f  accuracy = %0.2f%% \n" \
    % (eval[0], eval[1]*100) )

  # 5. save model
  print("Saving model to disk \n")
  mp = ".\\Models\\iris_model.h5"
  model.save(mp)

  # 6. use model
  np.set_printoptions(precision=4)
  unknown = np.array([[6.1, 3.1, 5.1, 1.1]], dtype=np.float32)
  predicted = model.predict(unknown)
  print("Using model to predict species for features: ")
  print(unknown)
  print("\nPredicted species is: ")
  print(predicted)

#
===========================================================================
=======

if __name__=="__main__":
  main()
```

The program imports the entire Keras package and assigns an alias **K**. An alternative approach is to import only the modules you need, for example:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

Even though Keras uses TensorFlow as its backend engine, you don't need to explicitly import TensorFlow, except to set its random seed. The OS package is imported only so that an annoying TensorFlow startup warning message will be suppressed.

The program structure consists of a single main function, with no helper functions. The program begins with:

```
def main():
  # 0. get started
  print("\nIris dataset using Keras/TensorFlow ")
  np.random.seed(4)
  tf.set_random_seed(13)

  # 1. load data
  print("Loading Iris data into memory \n")
  train_file = ".\\Data\\iris_train.txt"
  test_file = ".\\Data\\iris_test.txt"
. . .
```

In most situations, you want to make your results reproducible. The Keras library makes extensive use of the NumPy global random number generator, so it's good practice to set the seed value. The value used in the program, **4**, is arbitrary. Similarly, because Keras uses TensorFlow, you'll usually want to set its seed, too. However, program results typically aren't completely reproducible due to order of numeric rounding of parallelized tasks.

I indent with two spaces rather than the normal four spaces because of page-width limitations. All normal error-checking has been removed to keep the main ideas as clear as possible.

The program assumes that the training and test data files are located in a subdirectory named **Data**. The program doesn't have any information about the structure of the data files. I strongly recommend that you include program comments such as:

```
# data is comma-delimited and looks like:
# 5.1, 3.5, 1.4, 0.2, 1, 0, 0
# first four values are non-normalized features
# last three values are one-hot labels for
# setosa, versicolor, virginica
# 120 training items, 30 test items
```

This kind of information is easy to remember when you're writing your program, but difficult to remember a couple of weeks later.

The training and test data are read into memory with these statements:

```
  train_x = np.loadtxt(train_file, usecols=[0,1,2,3],
   delimiter=",",  skiprows=0, dtype=np.float32)
  train_y = np.loadtxt(train_file, usecols=[4,5,6],
    delimiter=",", skiprows=0, dtype=np.float32)

  test_x = np.loadtxt(test_file, usecols=range(0,4),
   delimiter=",",  skiprows=0, dtype=np.float32)
  test_y = np.loadtxt(test_file, usecols=range(4,7),
    delimiter=",", skiprows=0, dtype=np.float32)
```

In general, Keras needs feature data and label data stored in separate NumPy array-of-array style matrices. There are many ways to read data into memory, but the **loadtxt()** function is versatile enough to meet most problem scenarios. The NumPy **genfromtxt()** function is very similar but gives you a few additional options, such as dealing with missing data. The **loadtxt()** function has a large number of parameters, but in most cases, you only need **usecols**, **delimiter**, and **dtype**.

Notice that **usecols** can accept a list such as **[0,1,2,3]** or a Python range such as **range(0,4)**. If you use the **range()** function, be careful to remember that the first parameter is inclusive, but the second parameter is exclusive.

In addition to the comma character, common values for the **delimiter** parameter are "\t" (tab) and " " (single space) The default parameter value is **None**, which means any whitespace.

The default **dtype** parameter value is **numpy.float**, which is an alias for Python **float**, and is the exact same as **numpy.float64**. But the default data type for almost all Keras functions is **numpy.float32**, so the program specifies this type. The idea is that for the majority of machine-learning problems, the advantage in precision gained by using 64-bit values is not worth the memory and performance penalty.

An alternative approach to using static, separate training and test files is to use just a single file containing all data, read all data into memory, and then programmatically generate training and test matrices in memory. This alternative technique allows you to perform k-fold cross validation, a technique which used to be common, but which is now rarely used with deep learning and very large datasets.

Instead of using a NumPy function such as **`loadtxt()`** to read data into memory, a different approach is to use the Pandas (originally "panel data," now "Python Data Analysis Library") library, which has many advanced data manipulation features. However, Pandas has a significant learning curve.

Defining the neural network model

The program defines a 4-(5-6)-3 deep neural network using this code:

```
# 2. define model
init = K.initializers.glorot_uniform(seed=1)
simple_adam = K.optimizers.adam()
model = K.models.Sequential()
model.add(K.layers.Dense(units=5, input_dim=4, kernel_initializer=init,
  activation='relu'))
model.add(K.layers.Dense(units=6, kernel_initializer=init,
  activation='relu'))
model.add(K.layers.Dense(units=3, kernel_initializer=init,
  activation='softmax'))

model.compile(loss='categorical_crossentropy',
  optimizer=simple_adam, metrics=['accuracy'])
```

Deep neural networks are often very sensitive to the initial values of the weights and biases, so Keras has several different initialization functions. The demo uses **`glorot_uniform()`**, which assigns small, random values based on the fan-in and fan-out of the network layer in which it's used. The **`seed`** parameter is used so that program results will be reproducible. Table 2-1 lists a few of the common initialization functions in Keras.

Table 2-1: Common keras.initializers Functions

Function	Description
`Zeros()`	All `np.float32` `0.0` values
`Constant(value=0)`	All a single specified `np.float32` value
`RandomNormal(mean=0.0, stddev=0.05, seed=None)`	Gaussian, bell-shaped distribution
`RandomUniform(minval=-0.05, maxval=0.05, seed=None)`	Random, evenly distributed between `minval` and `maxval`

Function	Description
`glorot_normal(seed=None)`	Truncated `Normal` with `stddev = sqrt(2 / (fan_in + fan_out))`
`glorot_uniform(seed=None)`	Uniform random with limits `sqrt(6 / (fan_in + fan_out))`

The program prepares an **`Adam()`** optimizer object to be used by the **`fit()`** training function. Adam (adaptive moment estimation) is one of many training algorithms supported by Keras, and it's a good first choice when creating a prediction model. The program uses all **`Adam()`** default parameters, but they could have been specified explicitly as a form of documentation:

```
simple_adam = K.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999,
  epsilon=None, decay=0.0, amsgrad=False) # default parameter values
```

The program builds up the neural network architecture using the **`Sequential()`** method. The input layer is implicit, so the model begins with the first hidden layer:

```
  model = K.models.Sequential()
  model.add(K.layers.Dense(units=5, input_dim=4, kernel_initializer=init,
    activation='relu'))
```

The **`Dense()`** function is a standard, fully-connected layer. The **`units`** parameter specifies the number of hidden processing nodes in the layer, and because this is the first layer listed, you must specify how many input values there are using the **`input_dim`** parameter.

The hidden layer is configured to use **`relu`** activation (rectified linear unit). As you might expect, Keras supports many activation functions. For a **`Dense()`** hidden layer, **`relu`** is often a good first attempt. Other common activation functions are listed in Table 2-2. Keras also contains advanced, adaptive activation functions such **`LeakyReLU()`** in the **`keras.layers`** module.

Table 2-2: Common Dense Layer Activation Functions

Function	Descripton
`relu(x, alpha=0.0, max_value=None)`	if x < 0 , f(x) = 0, else f(x) = x
`tanh(x)`	hyperbolic tangent
`sigmoid(x)`	f(x) = 1.0 / (1.0 + exp(-x))
`linear(x)`	f(x) = x
`softmax(`**`x`**`, axis=-1)`	coerces vector **`x`** values to sum to 1.0 so they can be loosely interpreted as probabilities

An alternative to supplying a string value like **'relu'** to the activation parameter of the **Dense()** function, which uses default parameter values in the case of **relu()** and **softmax()**, is to use an **Activation** layer. For example:

```
model = K.models.Sequential()
model.add(K.layers.Dense(units=5, input_dim=4, kernel_initializer=init))
model.add(K.layers.Activation('relu'))
```

One of the challenges of working with Keras is that as it has evolved over time, many different techniques have been created, which can be confusing. Sometimes older examples use one approach, such as a separate **Activation** layer, and newer examples use a different approach, such as the string **activation** parameter in **Dense()**.

After setting up the implied input layer and the explicit first hidden layer, the rest of the architecture is specified like so:

```
model.add(K.layers.Dense(units=6, kernel_initializer=init,
  activation='relu'))
model.add(K.layers.Dense(units=3, kernel_initializer=init,
  activation='softmax'))
```

Because neither of these are the first layer, you don't have to specify the number of input values. If you wanted to, you could do so like this:

```
model.add(K.layers.Dense(units=6, input_dim=5, kernel_initializer=init,
  activation='relu'))
model.add(K.layers.Dense(units=3, input_dim=6, kernel_initializer=init,
 activation='softmax'))
```

Instead of using **Sequential()** and the **add()** method, you can construct a neural network by creating separate layers and then passing them to the **Model()** constructor like this:

```
init = K.initializers.glorot_uniform(seed=1)
X = K.layers.Input(shape=(4,))
net = K.layers.Dense(units=5, kernel_initializer=init,
  activation='relu')(X)
net = K.layers.Dense(units=6, kernel_initializer=init,
  activation='relu')(net)
net = K.layers.Dense(units=3, kernel_initializer=init,
  activation='softmax')(net)
model = K.models.Model(X, net)
```

The two approaches create the exact same neural network, but are quite different in terms of syntax. For multiclass classification problems, the choice is purely one of personal preference.

After a neural network model has been defined, it must be compiled before it can be trained:

```
model.compile(loss='categorical_crossentropy',
  optimizer=simple_adam, metrics=['accuracy'])
```

You can loosely think of the compilation process as translating Keras code into TensorFlow code (or CNTK code or Theano code). You must pass values to the **optimizer** and **loss** parameters so that the **fit()** method will know how to train the model. For multiclass classification, the **categorical_crossentropy** loss function is usually the best choice, but you can use the **mean_squared_error** function if needed (for example, to replicate the work of other people).

The **metrics** parameter is optional. The program passes a Python list containing just **'accuracy'** to indicate that classification accuracy (percentage correct predictions) should be computed during training.

Training and evaluating the model

After training data has been read into memory and the neural network has been created, the program trains the model using these statements:

```
  # 3. train model
  b_size = 1
  max_epochs = 10
  print("Starting training ")
  h = model.fit(train_x, train_y, batch_size=b_size, epochs=max_epochs,
    shuffle=True, verbose=1)
  print("Training finished \n")
```

The batch size is set to **1**, which is called online training. This means that the neural network weights and biases are updated for every training item. Alternatives are to set the batch size to the number of items in the training set (120), which is sometimes called full-batch training, or to set the batch size to an intermediate value such as **16**, which is called mini-batch training.

The **max_epochs** variable controls how many iterations will be used for training. The **shuffle** parameter in the **fit()** function indicates that the training items should be processed in random order. The default value is **True**, so the parameter could have been omitted. The **verbose** parameter controls how much information to display during training: **0** means display no information, **1** means display full information, and **2** means display a medium amount of information.

The **fit()** function returns a dictionary object that has the recorded training history. The demo program captures this information into object **h**, but doesn't make use of it. If you wanted to see the loss values, you could do so like this:

```
print(h.history['loss'])
```

After training, the demo program evaluates the model on the test data:

```
  # 4. evaluate model
  eval = model.evaluate(test_x, test_y, verbose=0)
  print("Evaluation on test data: loss = %0.6f  accuracy = %0.2f%% \n" \
    % (eval[0], eval[1]*100) )
```

The **`evaluate()`** function returns a list of values. The first value at index **`[0]`** is the always value of the required loss function specified in the **`compile()`** function. Other values in the list are any optional **`metrics`** from the **`compile()`** function. In this example, **`'accuracy'`** was passed, so the value at index **`[1]`** holds the classification accuracy. The program multiples by 100 to convert accuracy from a proportion (like 0.9123) to a percentage (like 91.23%).

Saving and using the model

In most situations you'll want to save a trained model, especially if the training took hours or even longer. The demo program saves the trained model like so:

```
  # 5. save model
  print("Saving model to disk \n")
  mp = ".\\Models\\iris_model.h5"
  model.save(mp)
```

Somewhat unusually, compared to other neural network libraries, Keras saves a trained model using the hierarchical data format (HDF) version 5. It is a binary format, so saved models can't be inspected with a text editor. In addition to saving an entire model, you can save just the model weights and biases, which is sometimes useful. You can also save the just model architecture but not the weights.

A saved Keras model can be loaded from a different program like this:

```
print("Loading a saved model")
mp = ".\\Models\\iris_model.h5"
model = K.models.load_model(mp)
```

The whole point of creating and training a model is so that it can be used to make predictions for new, previously unseen data:

```
 # 6. use model
  np.set_printoptions(precision=4)
  unknown = np.array([[6.1, 3.1, 5.1, 1.1]], dtype=np.float32)
  predicted = model.predict(unknown)
  print("Using model to predict species for features: ")
  print(unknown)
  print("\nPredicted species is: ")
  print(predicted)
```

The **`predict()`** method accepts input and computes output based on the values of the model's current weights and biases. Notice that variable **`unknown`** is an array-of-arrays (indicated by the double square brackets) rather than a simple vector.

The output is raw in the sense that it's a set of probabilities. It's up to you to interpret the meaning. You can do so programmatically along the lines of:

```
  labels = ["setosa", "versicolor", "virginica"]
  idx = np.argmax(predicted)
  species = labels[idx]
  print(species)
```

The **`argmax(v)`** function returns the index of the largest value in vector or list **`v`**. It's a useful function for many classification problems.

Summary and resources

To perform multiclass classification, you encode the target labels using one-hot (also called 1-of-N) encoding. The activation function on the output layer should be set to **`softmax`** so the node values sum to 1.0, and can be loosely interpreted as probabilities.

The loss function in most cases should be set to **`categorical_crossentropy`**, but you can use **`mean_squared_error`** if you wish. In general, you should pass **`accuracy`** to the optional **`metrics`** list of the **`compile()`** function.

Free parameters for multiclass classification include the number of hidden layers and the number of nodes in each hidden layer, the **`optimizer`** algorithm (but **`Adam`** is often a good choice), batch size, and the maximum number of training epochs to use.

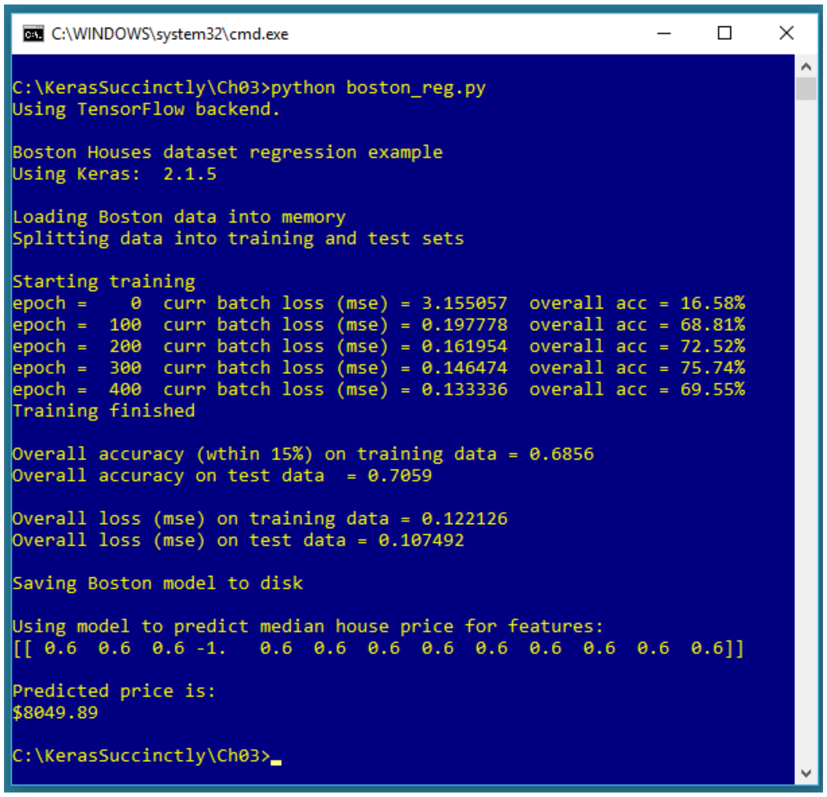
The training and test data used by the demo program can be found here.

The demo program uses the **`glorot_uniform()`** function for initialization. See additional information and alternatives here.

The demo program uses the **`relu()`** function for hidden-layer activation. See additional information and alternatives here.

Chapter 3 Regression

The goal of a regression problem is to make a prediction where the variable to predict is a single numeric value. For example, you might want to predict the annual income of a person based on their age, sex, political-party affiliation, and so on.



```
C:\KerasSuccinctly\Ch03>python boston_reg.py
Using TensorFlow backend.

Boston Houses dataset regression example
Using Keras:  2.1.5

Loading Boston data into memory
Splitting data into training and test sets

Starting training
epoch =    0  curr batch loss (mse) = 3.155057  overall acc = 16.58%
epoch =  100  curr batch loss (mse) = 0.197778  overall acc = 68.81%
epoch =  200  curr batch loss (mse) = 0.161954  overall acc = 72.52%
epoch =  300  curr batch loss (mse) = 0.146474  overall acc = 75.74%
epoch =  400  curr batch loss (mse) = 0.133336  overall acc = 69.55%
Training finished

Overall accuracy (wthin 15%) on training data = 0.6856
Overall accuracy on test data  = 0.7059

Overall loss (mse) on training data = 0.122126
Overall loss (mse) on test data = 0.107492

Saving Boston model to disk

Using model to predict median house price for features:
[[ 0.6  0.6  0.6 -1.   0.6  0.6  0.6  0.6  0.6  0.6  0.6  0.6  0.6]]

Predicted price is:
$8049.89

C:\KerasSuccinctly\Ch03>_
```

Figure 3-1: Regression using Keras

The screenshot in Figure 3-1 shows a demonstration of regression using a deep neural network. The demo program begins by loading 506 data items into memory. Each item represents the median house price in one of 506 towns near Boston. After loading, the demo programmatically splits the dataset into a training set and a test set.

Behind the scenes, the demo program creates a 13-(10-10)-1 deep neural network. Then, the network is trained using 500 epochs. After training, the regression model has 68.56 percent accuracy on the training data, and 70.59 percent accuracy on the held-out test data.

The demo program concludes by making a prediction for a hypothetical, previously unseen town near Boston. The predicted median house price is $8,049.89 (the data comes from the 1970s when house prices were much lower than they are today).

Understanding the data

The Boston Housing dataset is a fairly well-known benchmark for regression problems. There are a total of 506 items. The raw data looks like this:

```
0.00632  18  2.31  0  0.538  6.575  65.2  4.09    1  296  15.3  396.9  4.98  24
0.02731   0  7.07  0  0.469  6.421  78.9  4.9671  2  242  17.8  396.9  9.14  21.6
```

Each line has 14 fields. The first 13 are the predictor values. The last value is the median home price in the town, divided by 1,000, so the first town shown has a median house price of $24,000.

The first three predictors, [0] to [2], are per capita crime rate, proportion of land zoned for large residential lots, and proportion of non-retail acres. The predictor [3] is a Boolean if the town borders the Charles River (0 = no, 1 = yes).

Briefly, the remaining predictors are: [4] = air pollution metric, [5] = average number rooms per house, [6] = proportion of old houses, [7] = weighted distance to Boston, [8] = index of accessibility to highways, [9] = tax rate, [10] = pupil-teacher ratio, [11] = measure of proportion of Black residents, and [12] = percentage lower socio-economic status residents.

Because the data has 14 dimensions, it's not possible to easily visualize it. However, you can get a rough idea of the data by looking at the partial graph in Figure 3-2.

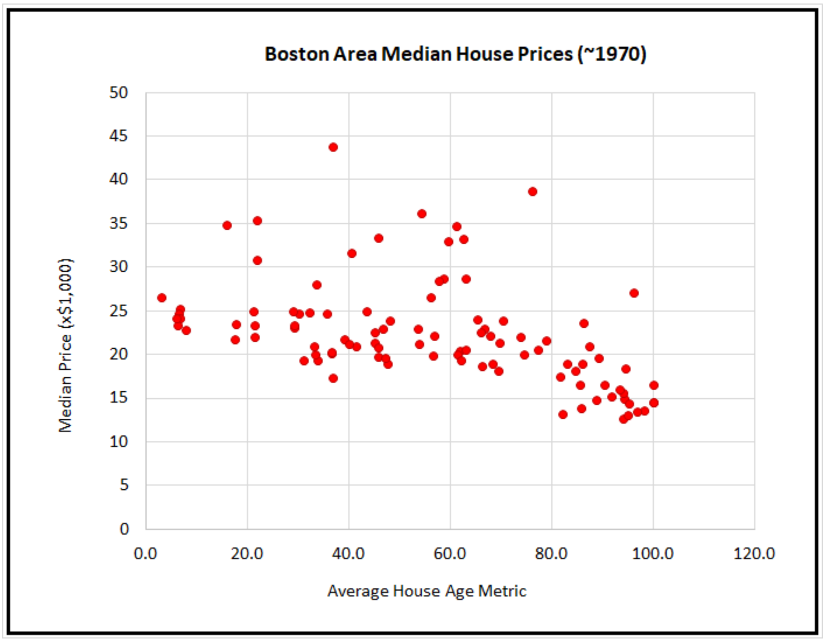


Figure 3-2: Partial Boston Housing Data

The graph plots just median price as a function of predictor [6], proportion of old houses, for the first 100 of the 506 data items. You can see from the graph that it's not possible to create an accurate prediction model based on just the old-houses proportion variable.

The raw data was preprocessed. The Boolean predictor [3] was converted from (0, 1) dummy encoding to (-1, +1) encoding. The 12 other (numeric) predictor variables were min-max normalized, resulting in all values being between 0.0 and 1.0. The target median house price variable, which was already divided by 1,000 was further divided by 10 so that all values are between 1.0 and 5.0. The resulting data looks like:

```
0.000000  0.180000  0.067815  -1  . . . 2.400000
0.000236  0.000000  0.242302  -1  . . . 2.160000
```

Somewhat fortunately, other than predictor [3], all the predictor variables are numeric, so there's no need for 1-of-(N-1) or possibly one-hot encoding of categorical variables.

The Boston program

The complete program that generated the output shown in Figure 3-1 is shown in Code Listing 3-1. The program begins with comments the program file name and versions of Python, TensorFlow, and Keras used, and then imports the NumPy, Keras, TensorFlow, and OS packages:

```
# boston_reg.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0
import numpy as np
import keras as K
import tensorflow as tf
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
```

In a non-demo scenario, you'd want to include additional details in the comments. Because Keras and TensorFlow are under rapid development, you should always document which versions are being used. Version incompatibilities can be a significant problem when working with Keras and other open-source software.

Code Listing 3-1: Boston Housing Regression Program

```
# boston_reg.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0

#
===========================================================================
=======

import numpy as np
import keras as K
import tensorflow as tf
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'

class MyLogger(K.callbacks.Callback):
  def __init__(self, n, data_x, data_y, pct_close):
    self.n = n
    self.data_x = data_x
    self.data_y = data_y
    self.pct_close = pct_close

  def on_epoch_end(self, epoch, logs={}):
    if epoch % self.n == 0:
      curr_loss = logs.get('loss')
      total_acc = my_accuracy(self.model, self.data_x,
        self.data_y, self.pct_close)
      print("epoch = %4d  curr batch loss (mse) = %0.6f  overall acc = 
%0.2f%%" % \
        (epoch, curr_loss, total_acc * 100))

def my_accuracy(model, data_x, data_y, pct_close):
  num_correct = 0; num_wrong = 0
  n = len(data_x)
  for i in range(n):
    predicted = model.predict(np.array([data_x[i]], dtype=np.float32))  # 
[[x]]
    actual = data_y[i]
    if np.abs(predicted[0][0] - actual) < np.abs(pct_close * actual):
      num_correct += 1
    else:
      num_wrong += 1
  return (num_correct * 1.0) / (num_correct + num_wrong)

# 
===========================================================================
=======

def main():
  # 0. get started
  print("\nBoston Houses dataset regression example ")
  np.random.seed(2)
  tf.set_random_seed(3)

  kv = K.__version__
  print("Using Keras: ", kv, "\n")

  # 1. load data
  print("Loading Boston data into memory ")
  data_file = ".\\Data\\boston_mm_tab.txt"
  all_data = np.loadtxt(data_file, delimiter="\t", skiprows=0, 
dtype=np.float32)

  N = len(all_data)
  indices = np.arange(N)
  np.random.shuffle(indices)
  n_train = int(0.80 * N)

  print("Splitting data into training and test sets \n")
  data_x = all_data[indices,:-1]
  data_y = all_data[indices,-1]
  train_x = data_x[0:n_train,:]
  train_y = data_y[0:n_train]
  test_x = data_x[n_train:N,:]
  test_y = data_y[n_train:N]

  # 2. define model
  init = K.initializers.RandomUniform(seed=1)
  simple_sgd = K.optimizers.SGD(lr=0.010)
  model = K.models.Sequential()
  model.add(K.layers.Dense(units=10, input_dim=13, kernel_initializer=init,
    activation='tanh'))  # hidden layer
  model.add(K.layers.Dense(units=10, kernel_initializer=init,
    activation='tanh'))  # hidden layer
  model.add(K.layers.Dense(units=1, kernel_initializer=init,
    activation=None))

  model.compile(loss='mean_squared_error', optimizer=simple_sgd,
metrics=['mse'])

  # 3. train model
  batch_size= 8
  max_epochs = 500
  my_logger = MyLogger(int(max_epochs/5), train_x, train_y, 0.15)
  print("Starting training ")
  h = model.fit(train_x, train_y, batch_size=batch_size, epochs=max_epochs,
    verbose=0, callbacks=[my_logger])
  print("Training finished \n")

  # 4. evaluate model
  acc = my_accuracy(model, train_x, train_y, 0.15)
  print("Overall accuracy (wthin 15%%) on training data = %0.4f" % acc)

  acc = my_accuracy(model, test_x, test_y, 0.15)
  print("Overall accuracy on test data  = %0.4f \n" % acc)

  eval = model.evaluate(train_x, train_y, verbose=0)
  print("Overall loss (mse) on training data = %0.6f" % eval[0])

  eval = model.evaluate(test_x, test_y, verbose=0)
  print("Overall loss (mse) on test data = %0.6f" % eval[0])

  # 5. save model
  print("\nSaving Boston model to disk \n")
  mp = ".\\Models\\boston_model.h5"
  model.save(mp)

  # 6. use model
  np.set_printoptions(precision=1)
  unknown = np.full(shape=(1,13), fill_value=0.6, dtype=np.float32)
  unknown[0][3] = -1.0  # binary feature
  predicted = model.predict(unknown)
  print("Using model to predict median house price for features: ")
  print(unknown)
  print("\nPredicted price is: ")
  print("$%0.2f" % (predicted * 10000))

#
===========================================================================
=======

if __name__=="__main__":
  main()
```

The program imports the entire Keras package and assigns an alias **K**. An alternative approach is to import just the modules you need, for example:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

Even though Keras uses TensorFlow as its backend engine, you don't need to explicitly import TensorFlow, except in order to set its random seed. The OS package is imported only so that an annoying TensorFlow startup warning message will be suppressed.

The program structure consists of a single **main** function, plus the helper class **MyLogger**, and the helper function **my_accuracy()**. These are needed because in a regression problem, there's no inherent definition of accuracy—you must define how close a predicted value must be to a correct training value in order to be considered a correct prediction.

The **MyLogger** class initializer is defined:

```
class MyLogger(K.callbacks.Callback):
  def __init__(self, n, data_x, data_y, pct_close):
    self.n = n
    self.data_x = data_x
    self.data_y = data_y
    self.pct_close = pct_close
. . .
```

The class inherits from the Keras **Callback** base class. As you'll see shortly, a **Callback** object is something that can be invoked automatically during training via the **fit()** function. The **MyLogger** initializer function (similar to a constructor in other programming languages) accepts a **pct_close** parameter, which specifies how close a predicted value must be to a target value.

The **MyLogger** class functionality is defined like so:

```
  def on_epoch_end(self, epoch, logs={}):
    if epoch % self.n == 0:
      curr_loss = logs.get('loss')
      total_acc = my_accuracy(self.model, self.data_x,
        self.data_y, self.pct_close)
      print("epoch = %4d  curr batch loss (mse) = %0.6f  overall acc = 
%0.2f%%" % \
        (epoch, curr_loss, total_acc * 100))
```

Note that Python uses the backslash character for line continuation. The **on_epoch_end()** method is inherited from the base **Callback** class. It will trigger automatically after each training epoch finishes. The program restricts output to every **n** epochs using the modulus (**%**) operator.

The method fetches the built-in **loss** metric value from the **logs** dictionary collection. Then, the method calls the program-defined **my_accuracy()** function to compute the prediction accuracy over the entire training data (not, just the current batch). The logger object displays the accuracy metric as a percentage (such as 85.12%) rather than a proportion (such as 0.8512), but this is a subjective matter of personal preference.

The program-defined accuracy function is:

```
def my_accuracy(model, data_x, data_y, pct_close):
  num_correct = 0; num_wrong = 0
  n = len(data_x)


  for i in range(n):
    predicted = model.predict(np.array([data_x[i]], dtype=np.float32))  # 
[[x]]
    actual = data_y[i]
    if np.abs(predicted[0][0] - actual) < np.abs(pct_close * actual):
      num_correct += 1
    else:
      num_wrong += 1
  return (num_correct * 1.0) / (num_correct + num_wrong)
```

There are a couple of tricky syntax issues. The **predict()** method expects a matrix, but **data_x[i]** is a vector, so the values are passed as **[data_x[i]]**. When using Keras, you shouldn't underestimate the frequency of running into problems with the shape of various objects.

The predicted house median price return value from **predict()** is a single value stored in a NumPy array-of-arrays matrix, so the scalar value itself is at **[0][0]**.

Reading and splitting the data

The demo program begins execution like so:

```
def main():
  # 0. get started
  print("\nBoston Houses dataset regression example ")
  np.random.seed(2)
  tf.set_random_seed(3)
  kv = K.__version
  print("Using Keras: ", kv, "\n")
. . .
```

Setting the NumPy and TensorFlow global random seeds is an attempt to get reproducible results. The seed values, **`2`** and **`3`**, are arbitrary. The demo prints the Keras version just to show how it's done. The entire 506-item normalized data is read into memory using these statements:

```
  # 1. load data
  print("Loading Boston data into memory ")
  data_file = ".\\Data\\boston_mm_tab.txt"
  all_data = np.loadtxt(data_file, delimiter="\t", skiprows=0,
dtype=np.float32)
```

The NumPy **`loadtxt()`** function is simple and versatile. The strategy used by the demo program is to read all data into memory, and then split it into training and test matrices. The primary alternative is to split the dataset before loading into memory. Reading then splitting has the advantage of keeping your data file(s) simpler at the expense of slightly more program code.

The demo prepares the data split:

```
  N = len(all_data)
  indices = np.arange(N)
  np.random.shuffle(indices)
  n_train = int(0.80 * N)
```

The **`len()`** function, applied to an **`n`**-dimensional NumPy array, returns the number of items in the first dimension. In this case, that is the number of rows in the training data. An alternative is to use the **`size()`** function and explicitly specify the dimension:

```
N = np.size(all_data, 0)  # 0 = rows, 1 = cols
```

The call to **`arange(N)`** ("array-range," not "arrange") returns an array of integers from **`0`** to **`N-1`** inclusive. The **`shuffle()`** functions rearranges the values in its argument by reference, so you don't need to assign a return value. The number of training items is computed as 80 percent of the total number, in this case 0.80 * 506 = 404 items, leaving 102 items for test purposes.

Splitting the data is done by these statements:

```
  data_x = all_data[indices,:-1]
  data_y = all_data[indices,-1]
  train_x = data_x[0:n_train,:]
  train_y = data_y[0:n_train]
  test_x = data_x[n_train:N,:]
  test_y = data_y[n_train:N]
```

The indexing **[indices,:-1]** means all rows in scrambled order, and all columns except the last column. Indexing **[indices,-1]** means all rows in scrambled order, and just the last column. Indexing **[0:n_train,:]** means rows **0** to **n_train-1** inclusive, and all columns. Indexing **[0:n_train]** is an alternative syntax that means rows **0** to **n_train-1** inclusive, and all columns. Indexing **[n_train:N,:]** means rows **n_train** to **N-1** inclusive. Indexing **[n_train,N]** is an alternative syntax that means rows **n_train** to **N-1** inclusive. Whew!

Personally, I don't find NumPy-array indexing anywhere near intuitive, and when I work with indexing, I always have to look up the syntax rules in online documentation.

Defining the model

The deep neural regression model is defined by this code:

```
  init = K.initializers.RandomUniform(seed=1)
  simple_sgd = K.optimizers.SGD(lr=0.010)
  model = K.models.Sequential()
  model.add(K.layers.Dense(units=10, input_dim=13, kernel_initializer=init,
    activation='tanh'))
  model.add(K.layers.Dense(units=10, kernel_initializer=init,
    activation='tanh'))
  model.add(K.layers.Dense(units=1, kernel_initializer=init,
    activation=None))
  model.compile(loss='mean_squared_error', optimizer=simple_sgd,
metrics=['mse'])
```

The network model has 13 input nodes, two hidden layers, each with 10 nodes, and one output node. Therefore, the network has (13 * 10) + 10 + (10 * 10) + 10 + (10 * 1) + 1 = 261 weights and biases. You can get this information programmatically by calling the **model.summary()** function without any parameters.

The demo program prepares the model by setting up a weight initializer using the **RandomUniform()** function with a seed value of 1, plus default parameter values of **minval = -0.05** and **maxval = +0.05**. The demo uses a stochastic gradient descent optimizer. The default learning rate is 0.01, so the demo code could have omitted the explicit assignment. Other default parameter values are **momentum = 0.0** (no momentum), **decay = 0.0** (no decay), and **nesterov = False** (no Nesterov momentum used).

The model is defined using the **Sequential()** layers syntax. There is no explicit input layer, so the first **Dense()** layer added is the first hidden layer. The activation on both hidden layers is **tanh**, which is often used for regression networks, but **relu** sometimes works better. Both hidden layers have 10 nodes. It's common practice, but not required, to specify the same number of nodes for all hidden layers in a regression network.

The output layer has a single node. Because the output node represents the median house price, it can take any value, so no activation is applied. An activation value of **None** is the default, so the demo code could have omitted the **activation** parameter.

After the model definition statements, the model is compiled. A **loss** parameter value is required, and the demo passes **mean_squared_error**, which is the most common choice for a regression network. A rare alternative scenario is when the output node value has been coerced to the range (0.0, 1.0), typically via sigmoid activation on the output node, and the training target values are also in (0.0, 1.0), typically via min-max normalization on the target, dependent variable values.

The **metrics** parameter is optional; when used, it accepts a list of quantities to compute during training. The demo passes '**mse'**, which is a shortcut alias for **mean_squared_error**. Unlike a classification problem where you usually pass **accuracy**, in regression there's no inherent definition of accuracy.

Instead of using the **Sequential()** syntax, you can define layers individually and chain them together:

```
  init = K.initializers.RandomUniform(seed=1)
  simple_sgd = K.optimizers.SGD(lr=0.010)
  X = K.layers.Input(shape=(13,))
  net = K.layers.Dense(units=10, kernel_initializer=init,
    activation='tanh')(X)
  net = K.layers.Dense(units=10, kernel_initializer=init,
    activation='tanh')(net)
  net = K.layers.Dense(units=1, kernel_initializer=init,
    activation=None)(net)
  model = K.models.Model(X, net)
  model.compile(loss='mean_squared_error', optimizer=simple_sgd,
metrics=['mse'])
```

The two approaches create identical models and yield identical results, so the choice is purely one of personal coding style preference.

Training and evaluating the model

The demo program prepares training with these statements:

```
  # 3. train model
  batch_size= 8
  max_epochs = 500
  my_logger = MyLogger(int(max_epochs/5), train_x, train_y, 0.15)
```

The batch size and the maximum number of epochs to train are hyperparameters, and good value must be determined by trial and error. The **my_logger** object is instantiated to fire once every max_epochs / 5 = 500 / 5 = 100 epochs. Because of how the **on_epoch_end()** method is defined, this means that the current mean squared error loss and the prediction accuracy on all 404 training items, will be displayed every 100 epochs. Recall that when accuracy is computed, a prediction is correct if it is plus or minus 15 percent of the correct target value.

Training is performed by calling the **fit()** function:

```
  print("Starting training ")
  h = model.fit(train_x, train_y, batch_size=batch_size, epochs=max_epochs,
    verbose=0, callbacks=[my_logger])
  print("Training finished \n")
```

Notice that the demo uses the same argument value name, **batch_size**, as the parameter name. Some people use this style consistently, while others go out of their way to avoid using the same name.

Setting **verbose = 0** suppresses all built-in progress messages during training, but because the **callbacks** list has the **my_logger** object, the demo will display the custom messages.

The **fit()** function returns an object that holds a **History** object containing metrics computed during training. The demo doesn't use the return value, but could have done so like this:

```
  loss_list = h.history['loss']  # loss of last batch every epoch
  print(loss_list)
```

After training, the demo program computes and prints the model's prediction accuracy:

```
  # 4. evaluate model
  acc = my_accuracy(model, train_x, train_y, 0.15)
  print("Overall accuracy (within 15%%) on training data = %0.4f" % acc)
  acc = my_accuracy(model, test_x, test_y, 0.15)
  print("Overall accuracy on test data  = %0.4f \n" % acc)
```

In general, the prediction accuracy on the test data should be roughly similar to the prediction accuracy on the training data. If accuracy on the test data is significantly less than accuracy on the training data, there's a good chance that you trained your model too aggressively and your model is overfitted.

In addition to displaying prediction accuracy, the demo program displays the loss values:

```
  eval = model.evaluate(train_x, train_y, verbose=0)
  print("Overall loss (mse) on training data = %0.6f" % eval[0])
  eval = model.evaluate(test_x, test_y, verbose=0)
  print("Overall loss (mse) on test data = %0.6f" % eval[0])
```

The **evaluate()** function returns a list of values. The first value at index **[0]** is the always the value of the (required) **loss** function specified in the **compile()** function, mean squared error in this case. Other values in the list are any optional **metrics** from the **compile()** function. For this example, and for regression in general, optional metrics usually are not specified.

Metrics include **loss** functions such as **mean_squared_error** and **categorical_crossentropy**, as well as five additional accuracy metrics for classification problems, shown in Table 3-1.

Table 3-1: Accuracy Metrics Functions

Function	Description
`binary_accuracy(y_true, y_pred)`	For binary classification
`categorical_accuracy(y_true, y_pred)`	For multiclass classification
`sparse_categorical_accuracy(y_true, y_pred)`	Rarely used (see documentation)
`top_k_categorical_accuracy(y_true, y_pred, k=5)`	Rarely used (see documentation)
`sparse_top_k_categorical_accuracy(y_true, y_pred, k=5)`	Rarely used (see documentation))

It's also possible to write custom, program-defined metric functions. Note that the shortcut alias **`acc`** can be used for either binary classification or multiclass classification.

Saving and using the model

The demo program saves the trained model like so:

```
  # 5. save model
  print("\nSaving Boston model to disk \n")
  mp = ".\\Models\\boston_model.h5"
  model.save(mp)
```

Keras saves models using the hierarchical data format (HDF) version 5. It's a binary format, so saved models can't be inspected with a text editor. In addition to saving an entire model, you can save just the model weights and biases, which is sometimes useful. You can also save the model architecture, but not the weights. Keras does not support saving models with the ONNX (open neural network exchange) format.

A saved Keras model can be loaded like so:

```
print("Loading a saved model")
mp = ".\\Models\\boston_model.h5"
model = K.models.load_model(mp)
```

The demo program uses the trained model to predict the median house price for a hypothetical, previously unseen town near Boston:

```
# 6. use model
np.set_printoptions(precision=1)
unknown = np.full(shape=(1,13), fill_value=0.6, dtype=np.float32)
unknown[0][3] = -1.0  # binary feature
predicted = model.predict(unknown)
print("Using model to predict median house price for features: ")
print(unknown)
print("\nPredicted price is: ")
print("$%0.2f" % (predicted * 10000))
```

The code sets up 13 predictor variables. Recall that 12 of the 13 predictors were min-max normalized to values between 0.0 and 1.0, so when predicting, you must use min-max normalized values too. Predictor variable [3] is the Boolean next-to-river value, so it must be encoded as either -1 or +1.

The demo program uses 0.6 for all min-max normalized predictor values. In a non-demo scenario, you'd have to actually perform normalization on raw input data, which means you need the minimum an maximum values for each normalized variable in the training data. This creates a tightly coupled connection between training data and trained model. The point is that you must retain your training data.

Summary and resources

When performing neural regression, you usually want to normalize your data. The number of hidden layers, and the number of nodes in each hidden layer, are hyperparameters that must be determined by trial and error. The two most common hidden layer activation functions are **`tanh`** and **`relu`**. The output layer should have a single node, and its activation function should be set to **`None`**, except in unusual scenarios.

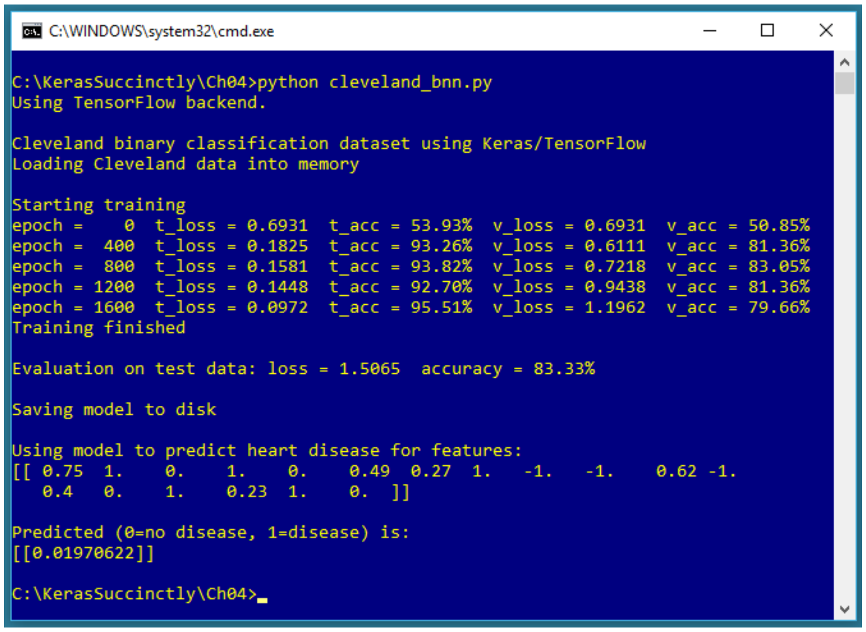
Because there is no inherent definition of accuracy for a regression problem, you must define your own accuracy function. In order to monitor prediction accuracy during training, you can implement a custom callback class and pass it as an argument to the **`fit()`** function. Common training optimizer functions for regression problems are stochastic gradient descent and Adam, but other optimizers often perform better.

The 506-item normalized data used by the demo program can be found here.

The demo program uses a custom accuracy metric (for regression). You can find information about built-in accuracy metrics (for classification) here.

Chapter 4 Binary Classification

The goal of binary classification is to make a prediction where the variable to predict can take on one of just two discrete values. For example, you might want to predict the sex (male or female) of a person based on their age, political party affiliation, annual income, and so on. Binary classification works somewhat differently than multiclass classification, where the variable to predict can be one of three or more possible discrete values.



```
C:\KerasSuccinctly\Ch04>python cleveland_bnn.py
Using TensorFlow backend.

Cleveland binary classification dataset using Keras/TensorFlow
Loading Cleveland data into memory

Starting training
epoch =    0  t_loss = 0.6931  t_acc = 53.93%  v_loss = 0.6931  v_acc = 50.85%
epoch =  400  t_loss = 0.1825  t_acc = 93.26%  v_loss = 0.6111  v_acc = 81.36%
epoch =  800  t_loss = 0.1581  t_acc = 93.82%  v_loss = 0.7218  v_acc = 83.05%
epoch = 1200  t_loss = 0.1448  t_acc = 92.70%  v_loss = 0.9438  v_acc = 81.36%
epoch = 1600  t_loss = 0.0972  t_acc = 95.51%  v_loss = 1.1962  v_acc = 79.66%
Training finished

Evaluation on test data: loss = 1.5065  accuracy = 83.33%

Saving model to disk

Using model to predict heart disease for features:
[[ 0.75  1.    0.    1.    0.    0.49  0.27  1.   -1.   -1.    0.62 -1.
   0.4   0.    1.    0.23  1.    0.  ]]

Predicted (0=no disease, 1=disease) is:
[[0.01970622]]

C:\KerasSuccinctly\Ch04>
```

Figure 4-1: Binary Classification using Keras

The screenshot in Figure 4-1 shows a demonstration of binary classification. The demo program begins by loading 178 training data items, 59 validation data items, and 60 test data items into memory. Each item represents a patient who has heart disease (**`1`**) or not (**`0`**). There are 13 predictor variables in the raw data. After normalization and encoding, there are 18 input variables.

Behind the scenes, the demo program creates an 18-(10-10)-1 deep neural network, that is, one with 18 input values (one for each predictor value), two hidden layers both with 10 nodes, and a single output node. The demo program trains the neural network model using 2,000 epochs. During training, the loss and accuracy values for both the training data and the validation data are displayed.

After training completes, the trained model achieves a prediction accuracy of 83.33 percent on the test data (50 of 60 correct, 10 incorrect). The demo concludes by making a prediction for a new, hypothetical, previously unseen patient. The predicted probability is 0.0197, and because the value is less than 0.5, the output maps to **`0`**, which in turn maps to a prediction of "**`no heart disease`**."

Understanding the data

The demo program uses the Cleveland Heart Disease dataset, a well-known classification benchmark dataset for statistics and machine learning. There are a total of 303 items. The raw data looks like this:

```
56.0,  1,  2,  120.0,  236.0,  0,  0,  178.0,  0,  0.8,  1,   3,   3,   0
62.0,  0,  4,  140.0,  268.0,  0,  2,  160.0,  0,  3.6,  3,   1,   6,   3
63.0,  1,  4,  130.0,  254.0,  0,  1,  147.0,  0,  1.4,  2,   2,   ?,   2
53.0,  1,  1,  140.0,  203.0,  1,  2,  155.0,  1,  3.1,  3,   0,   7,   1
[0]   [1] [2]  [3]     [4]    [5] [6]  [7]    [8]  [9]  [10] [11] [12] HD
```

The first 13 values on each line are the predictor values. The last value is 0 to 4, where 0 indicates no heart disease and 1 to 4 indicate heart disease of some kind. Predictor [0] is patient age. Predictor [1] is a Boolean sex (0 = female, 1 = male). Predictor [2] is categorical chest pain type encoded as 1 to 4.

Predictor [3] is blood pressure. Predictor [4] is cholesterol. Predictor [5] is a Boolean related to blood sugar (0 = low, 1 = high). Predictor [6] is categorical electrocardiographic result encoded as (0, 1, 2). Predictor [7] is maximum heart rate. Predictor [8] is a Boolean for angina (0 = no, 1 = yes). Predictor [9] is ST ("S-wave, T-wave") graph depression.

Predictor [10] is a categorical ST metric encoded as (1, 2, 3). Predictor [11] is a categorical count of colored fluoroscopy vessels encoded as (0, 1, 2, 3). Predictor [12] is a categorical value related to thalassemia encoded as (3, 6, 7).

The first step in data preparation is to deal with six data items that have one or more missing values. I took the simplest approach, which is to just delete any rows with missing data, leaving 297 data items. In my opinion, alternatives such as supplying an average column value, are usually not a good idea.

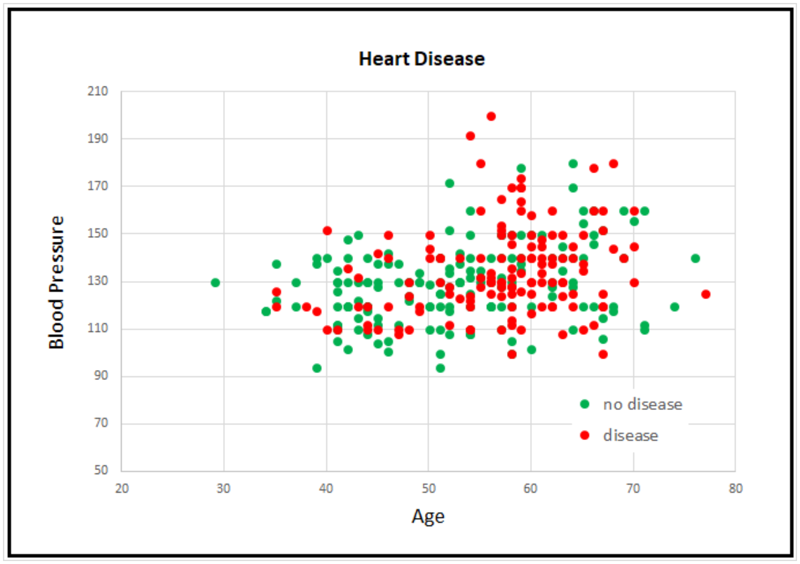


Figure 4-2: Partial Cleveland Heart Disease Data

The raw data was prepared by min-max normalizing the five numeric predictor variable values, by (-1, +1) encoding the three Boolean predictors, and by 1-of-(N-1) encoding the five categorical predictors. The class values-to-predict were encoded so that 0 means no indication of heart disease, and 1 means indication of some form of disease. I replaced the comma delimiters with tab characters.

After dealing with missing values, normalization, and encoding, the 297-item dataset was randomly split into three files: a 178-item (60 percent) set for training, and a 59-item (20 percent) set for validation, and a 60-item (20 percent) set for testing.

Because the Cleveland Heart Disease dataset has 13 dimensions, it's not possible to easily visualize it in a two-dimensional graph. But you can get a rough idea of the data from the partial graph in Figure 4-2. The graph shows only patient age and blood pressure for the first 160 items of the full dataset. As you can see, it's not possible to get a good prediction model using a simple linear technique like logistic regression or a base support vector machine linear model.

The Cleveland program

The complete program that generated the output shown in Figure 4-1 is shown in Code Listing 4-1. The program begins with comments the program file name (the **`_bnn`** is not a standard convention and just stands for binary neural network) and versions of Python, TensorFlow, and Keras used, and then imports the NumPy, Keras, TensorFlow, and OS packages:

```
# iris_dnn.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0
import numpy as np
import keras as K
import tensorflow as tf
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
```

In a non-demo scenario, you'd want to include additional details in the comments. Because Keras and TensorFlow are under rapid development, it's a good idea to document which versions are being used. Version incompatibilities can be a significant problem when working with Keras and open-source software.

Code Listing 4-1: Cleveland Heart Disease Binary Classification Program

```
# cleveland_bnn.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0

#
===========================================================================
=======

import numpy as np
import keras as K
import tensorflow as tf
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'

class MyLogger(K.callbacks.Callback):
  def __init__(self, n):
    self.n = n

  def on_epoch_end(self, epoch, logs={}):
    if epoch % self.n == 0:
      t_loss = logs.get('loss')
      t_accu = logs.get('acc')
      v_loss = logs.get('val_loss')
      v_accu = logs.get('val_acc')
      print("epoch = %4d  t_loss = %0.4f  t_acc = %0.2f%%  v_loss = %0.4f
\
v_acc = %0.2f%%" % (epoch, t_loss, t_accu*100, v_loss, v_accu*100))

#
===========================================================================
=======

def main():
  # 0. get started
  print("\nCleveland binary classification dataset using Keras/TensorFlow
")
  np.random.seed(1)
  tf.set_random_seed(2)

  # 1. load data
  print("Loading Cleveland data into memory \n")
  train_file = ".\\Data\\cleveland_train.txt"
  valid_file = ".\\Data\\cleveland_validate.txt"
  test_file = ".\\Data\\cleveland_test.txt"

  train_x = np.loadtxt(train_file, usecols=range(0,18),
   delimiter="\t",  skiprows=0, dtype=np.float32)
  train_y = np.loadtxt(train_file, usecols=[18],
    delimiter="\t", skiprows=0, dtype=np.float32)

  valid_x = np.loadtxt(valid_file, usecols=range(0,18),
   delimiter="\t",  skiprows=0, dtype=np.float32)
  valid_y = np.loadtxt(valid_file, usecols=[18],
    delimiter="\t", skiprows=0, dtype=np.float32)

  test_x = np.loadtxt(test_file, usecols=range(0,18),
   delimiter="\t",  skiprows=0, dtype=np.float32)
  test_y = np.loadtxt(test_file, usecols=[18],
    delimiter="\t", skiprows=0, dtype=np.float32)
  # 2. define model
  init = K.initializers.RandomNormal(mean=0.0, stddev=0.01, seed=1)
  simple_adadelta = K.optimizers.Adadelta()
  X = K.layers.Input(shape=(18,))
  net = K.layers.Dense(units=10, kernel_initializer=init,
    activation='relu')(X)
  net = K.layers.Dropout(0.25)(net)  # dropout for layer above
  net = K.layers.Dense(units=10, kernel_initializer=init,
    activation='relu')(net)
  net = K.layers.Dropout(0.25)(net)  # dropout for layer above
  net = K.layers.Dense(units=1, kernel_initializer=init,
    activation='sigmoid')(net)
  model = K.models.Model(X, net)

  model.compile(loss='binary_crossentropy', optimizer=simple_adadelta,
    metrics=['acc'])

  # 3. train model
  bat_size = 8
  max_epochs = 2000
  my_logger = MyLogger(int(max_epochs/5))

  print("Starting training ")
  h = model.fit(train_x, train_y, batch_size=bat_size, verbose=0,
    epochs=max_epochs, validation_data=(valid_x,valid_y),
    callbacks=[my_logger])
  print("Training finished \n")

  # 4. evaluate model
  eval = model.evaluate(test_x, test_y, verbose=0)
  print("Evaluation on test data: loss = %0.4f  accuracy = %0.2f%% \n" \
    % (eval[0], eval[1]*100) )

  # 5. save model
  print("Saving model to disk \n")
  mp = ".\\Models\\cleveland_model.h5"
  model.save(mp)

  # 6. use model
  unknown = np.array([[0.75, 1, 0, 1, 0, 0.49, 0.27, 1, -1, -1, 0.62, -1,
0.40,
    0, 1, 0.23, 1, 0]], dtype=np.float32) # .0197
  predicted = model.predict(unknown)
  print("Using model to predict heart disease for features: ")
  print(unknown)
  print("\nPredicted (0=no disease, 1=disease) is: ")
  print(predicted)
```

```
# 
==========================================================================
=======

if __name__=="__main__":
  main()
```

The program imports the entire Keras package and assigns an alias **`K`**. An alternative approach is to import only the modules you need, for example:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

Even though Keras uses TensorFlow as its backend engine, you don't need to explicitly import TensorFlow, except in order to set its random seed. The OS package is imported only so that an annoying TensorFlow startup warning message will be suppressed.

The program structure consists of a single **`main`** function, with a program-defined helper class, **`MyLogger`**, for custom logging. The class definition is:

```
class MyLogger(K.callbacks.Callback):
  def __init__(self, n):
    self.n = n


  def on_epoch_end(self, epoch, logs={}):
    if epoch % self.n == 0:
      t_loss = logs.get('loss')
      t_accu = logs.get('acc')
      v_loss = logs.get('val_loss')
      v_accu = logs.get('val_acc')
      print("epoch = %4d  t_loss = %0.4f  t_acc = %0.2f%%  v_loss = %0.4f  \
v_acc = %0.2f%%" % (epoch, t_loss, t_accu*100, v_loss, v_accu*100))
```

The **`MyLogger`** class is used only to display loss and accuracy metrics that are computed automatically, so the class initializer doesn't need to accept references to the training data. Most program-defined callbacks would pass that information like this:

```
def __init__(self, n, data_x, data_y):
  self.n = n
  self.data_x = data_x
  self.data_y = data_y
```

The **`on_epoch_end()`** method pulls the current loss and accuracy metrics from the built-in **`logs`** dictionary and displays them. The demo program does this only so that the logging display messages can be reduced to once every 400 epochs instead of every epoch. Keras computes loss and accuracy for every training batch and averages the values over all batches at the end of each epoch. If you want more granular information, you can use the **`on_batch_end()`** method.

The **main()** function begins with:

```
def main():
  # 0. get started
  print("\nCleveland binary classification dataset using Keras/TensorFlow ")
  np.random.seed(1)
  tf.set_random_seed(2)

  # 1. load data
  print("Loading Cleveland data into memory \n")
  train_file = ".\\Data\\cleveland_train.txt"
  valid_file = ".\\Data\\cleveland_validate.txt"
  test_file = ".\\Data\\cleveland_test.txt"
. . .
```

In most situations you want your results to be reproducible. The Keras library uses the NumPy and TensorFlow global random-number generators, so it's good practice to set the **seed** values. The values used in the program, **1** and **2**, are arbitrary. However, be aware that Keras program results typically aren't completely reproducible, due to order of numeric rounding of parallelized tasks.

The program assumes that the training, validation, and test data files are located in a subdirectory named **Data**. The purpose of the validation data is to monitor its loss and accuracy during training to prevent training the model too much, which could result in an overfitted model.

The basic idea is illustrated in the graph in Figure 4.3. The graph indicates that over time, loss/error on the training data will decline steadily. If you measure the loss/error on a hold-out set of validation data, you may be able to identify when model overfitting is starting to occur, and then you can stop training (known as "early stopping").

Although the train-validate-test idea is good in principle, in practice it usually doesn't work so well. The problem is that the loss/error values rarely drop in the nice, smooth way shown on the graph. Instead, the values tend to jump erratically, which makes identifying the start of model-overfitting very difficult.

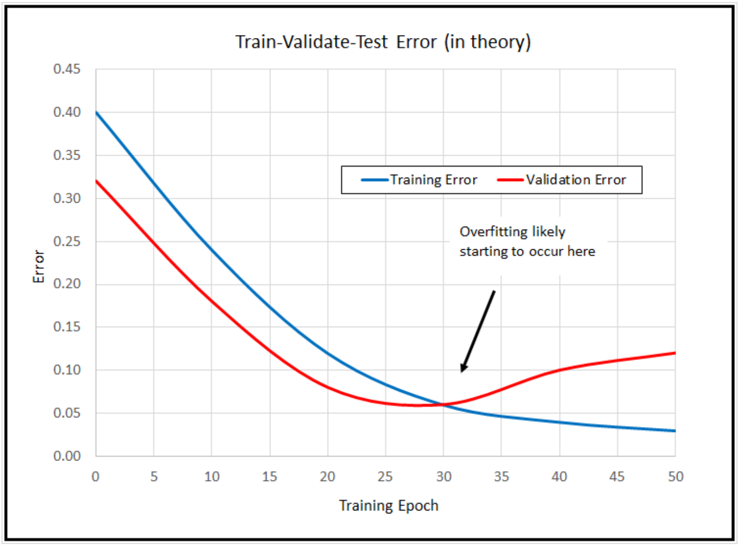


Figure 4-3: Train-Validate-Test in Theory

Additionally, holding out data for validation purposes reduces the amount of data you have for training. For these reasons, train-validate-test isn't very common. The demo program shows you how to use the technique because there are times when its useful, and so that you can understand it if you see it used.

The demo program doesn't have any information about the structure of the data files. I recommend that you include comments in your program that explain things such as how many predictor variables there are, types of encoding and normalization used, and so on. This kind of information is easy to remember when you're writing your program, but difficult to remember a couple of weeks later.

The training, validation, and test data is read into memory with these statements:

```
train_x = np.loadtxt(train_file, usecols=range(0,18),
 delimiter="\t",  skiprows=0, dtype=np.float32)
train_y = np.loadtxt(train_file, usecols=[18],
  delimiter="\t", skiprows=0, dtype=np.float32)



valid_x = np.loadtxt(valid_file, usecols=range(0,18),
 delimiter="\t",  skiprows=0, dtype=np.float32)
valid_y = np.loadtxt(valid_file, usecols=[18],
  delimiter="\t", skiprows=0, dtype=np.float32)

test_x = np.loadtxt(test_file, usecols=range(0,18),
 delimiter="\t",  skiprows=0, dtype=np.float32)
test_y = np.loadtxt(test_file, usecols=[18],
  delimiter="\t", skiprows=0, dtype=np.float32)
```

In general, Keras needs feature data and label data stored in separate NumPy array-of-array style matrices. There are many ways to read data into memory, but the **`loadtxt()`** function is versatile enough to meet most problem scenarios. A common alternative approach is to use the **`read_csv()`** function from the Pandas ("Python Data Analysis Library") package. For example:

```
import pandas as pd
train_x = pd.read_csv(train_file, usecols=range(0,18),
  delimiter="\t", header=None, dtype=np.float32).values
train_y = pd.read_csv(train_file, usecols=[18],
  delimiter="\t", header=None, dtype=np.float32).values
```

Notice that **`usecols`** can accept a list such as **`[0,1,2,3]`** or a Python range such as **`range(0,4)`**. If you use the **`range()`** function, be careful to remember that the first parameter is inclusive, but the second parameter is exclusive.

In addition to the comma character, common values for the **`delimiter`** parameter are "\t" (tab) and " " (single space) The default parameter value is **`None`** which means any whitespace.

The default **`dtype`** parameter value is **`numpy.float`**, which is an alias for Python **`float`**, and is the exact same as **`numpy.float64`**. The default data type for almost all Keras functions is **`numpy.float32`**, so the program specifies this type. The idea is that for the majority of machine learning problems, the advantage in precision gained by using 64-bit values is not worth the memory and performance penalty.

Defining the neural network model

The demo program defines an 18-(10-10)-1 deep neural network using this code:

```
# 2. define model
init = K.initializers.RandomNormal(mean=0.0, stddev=0.01, seed=1)
simple_adadelta = K.optimizers.Adadelta()
X = K.layers.Input(shape=(18,))
net = K.layers.Dense(units=10, kernel_initializer=init,
  activation='relu')(X)
net = K.layers.Dropout(0.25)(net)  # dropout for layer above
net = K.layers.Dense(units=10, kernel_initializer=init,
  activation='relu')(net)
net = K.layers.Dropout(0.25)(net)  # dropout for layer above
net = K.layers.Dense(units=1, kernel_initializer=init,
  activation='sigmoid')(net)


model = K.models.Model(X, net)
model.compile(loss='binary_crossentropy', optimizer=simple_adadelta,
  metrics=['acc'])
```

The demo sets up random Gaussian initial weights. Deep neural networks are often very sensitive to the initial values of the weights and biases, so Keras has several different initialization functions you can use.

The training optimizer object is **`Adadelta()`** ("adaptive delta"), one of many advanced variations of basic stochastic gradient descent. Selecting a Keras optimizer can be a bit intimidating. Table 4-1 lists five of the most commonly used optimizers.

Table 4-1: Five Common Keras Optimizers

Optimizer	Description
`SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)`	Basic optimizer for simple neural networks
`RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)`	Often used with recurrent neural networks, very similar to Adadelta
`Adagrad(lr=0.01, epsilon=None, decay=0.0)`	General purpose adaptive algorithm

Optimizer	Description
`Adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0)`	Advanced version of Adagrad, similar to RMSprop
`Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)`	Excellent general-purpose, adaptive algorithm

One of the strengths of the Keras optimizers is that they all have sensible default parameter values, so you can try different optimizers very easily.

The demo program specifies each layer separately, and then combines them using the **`Model()`** method. An alternative approach is to use the **`Sequntial()`** method. The two approaches create the exact same neural network, but are quite a bit different in terms of syntax. The choice is one of personal preference.

The demo uses **`Dropout()`** layers. The purpose of **`Dropout`** is to reduce the likelihood of model overfitting. From a syntax point of view, you place a **`Dropout()`** layer immediately *after* the layer you wish to apply it to. The single parameter is the percentage of nodes in the affected layer to randomly drop on each training iteration. The advantage of using **`Dropout()`** is that it's often effective in combating overfitting. The disadvantage is that you have to deal with additional hyperparameters to define where to apply dropout, and what dropout rate to use..

Note that it's possible to apply dropout to a neural network input layer; this is sometimes called jittering. However, using dropout on a neural network input layer is quite rare, and you should use it somewhat cautiously.

The demo program uses **`relu`** (rectified linear unit) activation for the hidden nodes. The **`relu`** activation function is often more resistant to the vanishing gradient problem, which causes training to stall out, than **`tanh`** or **`sigmoid`** activation.

The output layer, with its single node, uses **`sigmoid`** activation. This coerces the output node to a single value between 0.0 and 1.0, which can be interpreted as the probability that the target class is 1 (presence of heart disease in this problem). Put another way, if the output node value is less than 0.5, the prediction is class = 0 (no heart disease); otherwise, the prediction is class = 1 (heart disease).

The model is compiled with **`binary_crossentropy`** as the loss function. For multiclass classification, you can use **`categorical_crossentropy`** or **`mean_squared_error`**, but for binary classification problems, you can use **`binary_crossentropy`** or **`mean_squared_error`**.

The **`metrics`** parameter to **`compile()`** is optional. The demo program passes a Python list containing just **`'acc'`** to indicate that classification accuracy (percentage correct predictions) should be computed for each batch during training.

Training and evaluating the model

After training data has been read into memory and the neural network has been created, the program trains the network using these statements:

```
# 3. train model
bat_size = 8
max_epochs = 2000
my_logger = MyLogger(int(max_epochs/5))
print("Starting training ")
h = model.fit(train_x, train_y, batch_size=bat_size, verbose=0,
  epochs=max_epochs, validation_data=(valid_x,valid_y),
  callbacks=[my_logger])
print("Training finished \n")
```

The batch size is a hyperparameter, and a good value must be determined by trial and error. The **`max_epochs`** argument is also a hyperparameter. Larger values typically lead to lower loss and higher accuracy on the training data, at the risk of overfitting on the test data.

Training is configured to display loss and accuracy on the training data and the validation data every 2000 / 5 = 400 epochs. In a non-demo scenario, you'd want to see information displayed much more often.

The **`fit()`** function returns an object that holds complete logging information. This is sometimes useful for analysis of a model that refuses to train. The demo program does not use the **`h`** object, so it could have been omitted.

After training, the model is evaluated:

```
# 4. evaluate model
eval = model.evaluate(test_x, test_y, verbose=0)
print("Evaluation on test data: loss = %0.4f  accuracy = %0.2f%% \n" \
  % (eval[0], eval[1]*100) )
```

The **`evaluate()`** function returns a Python list. The first value at index [0] is the always value of the required loss function specified in the **`compile()`** function, **`binary_crossentropy`** in this case. Other values in the list are any optional **`metrics`** from the **`compile()`** function. In this example, the shortcut string **`'acc'`** was passed so the value at index [1] holds the classification accuracy. The program multiples by 100 to convert accuracy from a proportion (like 0.8234) to a percentage (like 82.34 percent).

Saving and using the model

In most situations you'll want to save a trained model, especially if the training took hours or even longer. The demo program saves the trained model like so:

```
  # 5. save model
  print("Saving model to disk \n")
  mp = ".\\Models\\cleveland_model.h5"
  model.save(mp)
```

Keras saves a trained model using the hierarchical data format (HDF) version 5. It's a binary format, so saved models can't be inspected with a text editor. In addition to saving an entire model, you can save just the model weights and biases, which is sometimes useful. You can also save the just model architecture without the weights.

A saved Keras model can be loaded from a different program like this:

```
print("Loading a saved model")
mp = ".\\Models\\cleveland_model.h5"
model = K.models.load_model(mp)
```

An alternative to saving the fully trained model is to save different versions of the model as they're trained. You could add the save code to the **`on_epoch_end()`** method of the program-defined **`MyLogger`** object, for example:

```
def on_epoch_end(self, epoch, logs={}):
  if epoch % self.n == 0:
    m_name = ".\\Models\\cleveland_" + str(epoch) + "_model.h5"
    self.model.save(m_name)
```

Keras also has a built-in **`ModelCheckpoint`** callback, which has parameters that allow you to do things such as saving only if a specified metric improves (lower loss, higher accuracy).

The demo program concludes by making a prediction:

```
  # 6. use model
  unknown = np.array([[0.75, 1, 0, 1, 0, 0.49, 0.27, 1, -1, -1, 0.62, -1,
0.40,
    0, 1, 0.23, 1, 0]], dtype=np.float32)
  predicted = model.predict(unknown)
  print("Using model to predict heart disease for features: ")
  print(unknown)
  print("\nPredicted (0=no disease, 1=disease) is: ")
  print(predicted)
```

Because the model was trained using normalized and encoded data, you must pass input values that have been normalized and encoded in the same way. Notice that the feature predictor values are passed as a NumPy array-of-arrays object.

The output prediction is raw in the sense that it's just a value between 0.0 and 1.0, and therefore, it's up to you to interpret the meaning. You can do so programmatically along the lines of:

```
labels = ["no indication of heart disease", "indication of heart disease"]
if predicted < 0.50:
  result = labels[0]
else:
  result = labels[1]
print(result)
```

Note that it is possible to perform binary classification by encoding the two classes-to-predict as (1, 0) and (0, 1), and then treating the problem as multiclass classification (softmax output layer activation and categorical cross entropy loss).

Summary and resources

To perform binary classification, you encode the target labels using 0-or-1 encoding. The number of nodes in the input layer is determined by the structure of your normalized and encoded data. The number of output nodes should be one, and the activation function on the output layer should be set to **`sigmoid`** so the node value is between 0.0 and 1.0, where a value less than 0.5 indicates a prediction of class = 0; otherwise, the prediction is class = 1.

The **`loss`** function should be set to **`binary_crossentropy`**, but you can use **`mean_squared_error`** if you wish. In general you should pass **`accuracy`** to the optional **`metrics`** list of the **`compile()`** function.

Free parameters for binary classification include the number of hidden layers and the number of nodes in each hidden layer, optimizer algorithm (Adagrad, Adadelta, and Adam are often good choices), batch size, and the maximum number of training epochs to use.

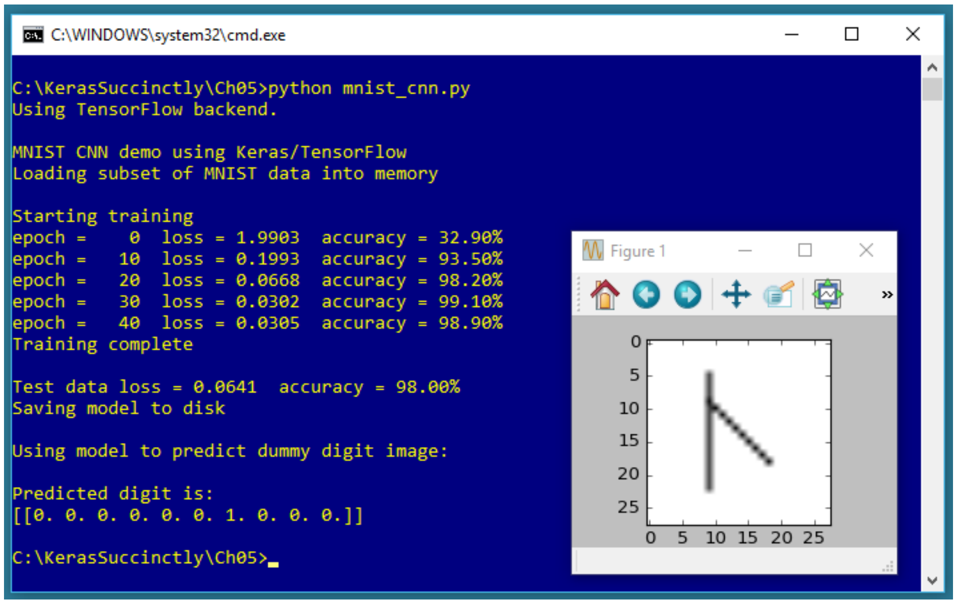
You can find the training, validation, and test data used by the demo program here.

The demo program uses a custom, program-defined callback class. See information about Keras built-in callbacks here.

This chapter described five of the most commonly used training optimizer algorithms. See information about all optimizers here.

Chapter 5 Image Classification

The goal of image classification is to make a prediction where the variable to predict is a label that's associated with an image. For example, you might want to predict whether a photograph contains an "apple," "banana," or "orange." The most common way to perform image classification is to use what's called a convolutional neural network (CNN).



```
C:\KerasSuccinctly\Ch05>python mnist_cnn.py
Using TensorFlow backend.

MNIST CNN demo using Keras/TensorFlow
Loading subset of MNIST data into memory

Starting training
epoch =    0  loss = 1.9903  accuracy = 32.90%
epoch =   10  loss = 0.1993  accuracy = 93.50%
epoch =   20  loss = 0.0668  accuracy = 98.20%
epoch =   30  loss = 0.0302  accuracy = 99.10%
epoch =   40  loss = 0.0305  accuracy = 98.90%
Training complete

Test data loss = 0.0641  accuracy = 98.00%
Saving model to disk

Using model to predict dummy digit image:

Predicted digit is:
[[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]]

C:\KerasSuccinctly\Ch05>_
```

Figure 5-1: Image Classification on the MNIST Dataset Using Keras

The screenshot in Figure 5-1 shows a demonstration of image classification. The demo program begins by loading 1,000 training images and 100 test images into memory. Each image is a handwritten digit from '0' to '9' and is 28 pixels wide by 28 pixels tall, for a total of 784 pixels. The data is a subset of the MNIST (modified National Institute of Standards and Technology) benchmark dataset.

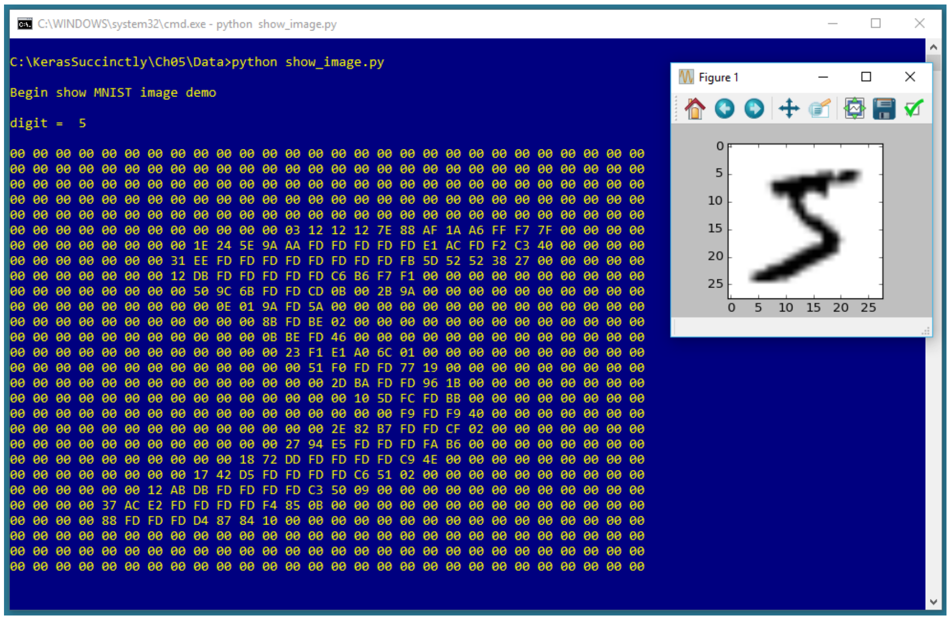
Behind the scenes, the demo program creates a 784-32-64-100-10 convolutional neural network. The network has 784 input nodes and 10 output nodes, one for each possible digit or label. The meaning of the 32-64-100 part of the architecture will be explained shortly. The demo program trains the neural network model using 50 epochs. During training, the loss and accuracy values for the training data are displayed to make sure that training is making progress.

After training completes, the trained model achieves a prediction accuracy of 98.00 percent on the test data (98 of 100 images correct, 2 incorrect). The demo concludes by making a prediction for a dummy, hypothetical, previously unseen image that vaguely resembles a handwritten digit. The predicted digit is a '6' because the largest value (1.0) of the output probabilities vector is at index [6].

Understanding the data

The MNIST dataset is essentially the "Hello World" dataset for deep learning. The full dataset consists of 60,000 training images and 10,000 test images. The demo program uses a subset of MNIST (1,000 training images and 100 test images) for simplicity.

Each of the 28x28 pixels is a grayscale integer value between 0 (white) and 255 (black). Figure 5-2 shows the first training image, displayed as 784 pixel values in hexadecimal format, and also as an image.



```
C:\KerasSuccinctly\Ch05\Data>python show_image.py

Begin show MNIST image demo

digit =  5

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 03 12 12 12 7E 88 AF 1A A6 FF F7 7F 00 00 00 00
00 00 00 00 00 00 00 00 1E 24 5E 9A AA FD FD FD FD FD E1 AC FD F2 C3 40 00 00 00 00
00 00 00 00 00 00 00 31 EE FD FD FD FD FD FD FD FD FB 5D 52 52 38 27 00 00 00 00 00
00 00 00 00 00 00 00 12 DB FD FD FD FD FD C6 B6 F7 F1 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 50 9C 6B FD FD CD 0B 00 2B 9A 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 0E 01 9A FD 5A 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 8B FD BE 02 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 0B BE FD 46 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 23 F1 E1 A0 6C 01 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 51 F0 FD FD 77 19 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 2D BA FD FD 96 1B 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 10 5D FC FD BB 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 F9 FD F9 40 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 82 B7 FD FD CF 02 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 27 94 E5 FD FD FD FA B6 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 18 72 DD FD FD FD FD C9 4E 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 17 42 D5 FD FD FD FD C6 51 02 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 12 AB DB FD FD FD FD C3 50 09 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 37 AC E2 FD FD FD FD F4 85 0B 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 88 FD FD FD D4 87 84 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 5-2: A Typical MNIST Image

The raw data is stored in an unusual format, and before coding the demo program, the raw data has to be preprocessed. Both the raw training and raw test datasets are stored in two files each. The first file contains just the pixel values, 784 values per line (60,000 lines for the training file, 10,000 lines for the test file). The second file contains just the labels, '0' through '9', one per line.

Additionally, all four files are stored in a proprietary binary format, and in big endian format, rather than in the far more common little endian format used by Intel and similar processors. And the four source files are compressed in .gz format.

First, the four raw source files are unzipped. For the training data, the preprocessing in high level pseudo-code is:

```
open (binary) pixel file for reading
open (binary) labels file for reading
open (text) result file for writing

read and discard header bytes in pixels and labels file
```

```
loop 1000 times
  read label from label file
  write label to result file
  write "##" separator
  loop 784 times
    read a pixel byte
    write pixel to result file
  end-loop
  write a newline to result file
end-loop
close all files
```

The result is a training file with 1000 lines that looks like this:

```
2 ** 0 0 152 27 .. 0
5 ** 0 0 38 122 .. 0
```

The first value is the lass label, '0' through '9'. Next is a double-asterisk separator, just for readability. Next are the 784 pixel values. The 100-image test file has the same structure. The preprocessing does not encode the class labels, and does not normalize the pixel values. As you'll see shortly, encoding and normalization are done programmatically.

When working with machine learning, getting your data ready is often time-consuming, annoying, and difficult. It's not uncommon for data preprocessing to require 90 percent (or more) of your total time and effort.

Note that the Keras library comes with a pre-processed MNIST dataset that can be accessed like this:

```
from keras.datasets import mnist
(train_x, train_y), (test_x, test_y) = mnist.load_data()
```

However, this is a bit of a cheat because in a non-demo scenario, you won't have a nice way like this to access your data.

The MNIST program

The complete program that generated the output shown in Figure 5-1 is shown in Code Listing 5-1. The program begins with comments for the program file name (the **`_cnn`** is not a standard convention, and just stands for convolutional neural network) and versions of Python, TensorFlow, and Keras used, and then imports the NumPy, Keras, TensorFlow, and OS packages. The PyPlot module is also imported so that the dummy input image can be displayed:

```
# mnist_cnn.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0
import numpy as np
import keras as K
import tensorflow as tf
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
import matplotlib.pyplot as plt
```

In a non-demo scenario, you'd want to include lots of additional details in the comments. Because Keras and TensorFlow are under rapid development, it's a good idea to document which versions are being used. Version incompatibilities can be a significant problem when working with Keras and open-source software.

Code Listing 5-1: MNIST Image Classification Program

```
# mnist_cnn.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0

#
===========================================================================
=======

import numpy as np
import keras as K
import tensorflow as tf
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
import matplotlib.pyplot as plt

class MyLogger(K.callbacks.Callback):
  def __init__(self, n):
    self.n = n

  def on_epoch_end(self, epoch, logs={}):
    if epoch % self.n == 0:
      t_loss = logs.get('loss')
      t_accu = logs.get('acc')
      print("epoch = %4d  loss = %0.4f  accuracy = %0.2f%%" % \
(epoch, t_loss, t_accu*100))

def encode_y(y_mat, y_dim):
  # convert to one-hot
  n = len(y_mat)  # rows
  result = np.zeros(shape=(n, y_dim), dtype=np.float32)
  for i in range(n):  # each row
    val = int(y_mat[i])    # like 5
    result[i][val] = 1
  return result

#
===========================================================================
=======

def main():
  # 0. get started
  print("\nMNIST CNN demo using Keras/TensorFlow ")
  np.random.seed(1)
  tf.set_random_seed(2)

  # 1. load data
  print("Loading subset of MNIST data into memory \n")
  train_file = ".\\Data\\mnist_train_keras_1000.txt"
  test_file = ".\\Data\\mnist_test_keras_100.txt"

  train_x = np.loadtxt(train_file, usecols=range(2,786),
    delimiter=" ",  skiprows=0, dtype=np.float32)
  train_y = np.loadtxt(train_file, usecols=[0],
    delimiter=" ", skiprows=0, dtype=np.float32)

  train_x = train_x.reshape(train_x.shape[0], 28, 28, 1)
  train_x /= 255
  train_y = encode_y(train_y, 10)  # one-hot

  test_x = np.loadtxt(test_file, usecols=range(2,786),
    delimiter=" ",  skiprows=0, dtype=np.float32)
  test_y = np.loadtxt(test_file, usecols=[0],
    delimiter=" ", skiprows=0, dtype=np.float32)

  test_x = test_x.reshape(test_x.shape[0], 28, 28, 1)
  test_x /= 255
  test_y = encode_y(test_y, 10)  # one-hot

  # 2. define model
  init = K.initializers.glorot_uniform()
  simple_adadelta = K.optimizers.Adadelta()

  model = K.models.Sequential()
  model.add(K.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1,1),
    padding='same', kernel_initializer=init, activation='relu',
    input_shape=(28,28,1)))
  model.add(K.layers.Conv2D(filters=64, kernel_size=(3, 3), strides=(1,1),
    padding='same', kernel_initializer=init, activation='relu'))
  model.add(K.layers.MaxPooling2D(pool_size=(2, 2)))
  model.add(K.layers.Dropout(0.25))
  model.add(K.layers.Flatten())
  model.add(K.layers.Dense(units=100, kernel_initializer=init,
activation='relu'))
  model.add(K.layers.Dropout(0.5))
  model.add(K.layers.Dense(units=10, kernel_initializer=init,
activation='softmax'))

  model.compile(loss='categorical_crossentropy', optimizer='adadelta',
    metrics=['acc'])

  # 3. train model
  bat_size = 128
  max_epochs = 50
  my_logger = MyLogger(int(max_epochs/5))

  print("Starting training ")
  model.fit(train_x, train_y, batch_size=bat_size, epochs=max_epochs,
verbose=0,
    callbacks=[my_logger])
  print("Training complete")

  # 4. evaluate model
  loss_acc = model.evaluate(test_x, test_y, verbose=0)
  print("\nTest data loss = %0.4f  accuracy = %0.2f%%" % \
(loss_acc[0], loss_acc[1]*100) )

  # 5. save model
  print("Saving model to disk \n")
  mp = ".\\Models\\mnist_model.h5"
  model.save(mp)

  # 6. use model
  print("Using model to predict dummy digit image: ")
  unknown = np.zeros(shape=(28,28), dtype=np.float32)
  for row in range(5,23): unknown[row][9] = 180  # vertical line
  for rc in range(9,19): unknown[rc][rc] = 250   # diagonal line
  plt.imshow(unknown, cmap=plt.get_cmap('gray_r'))
  plt.show()

  unknown = unknown.reshape(1, 28,28,1)
  predicted = model.predict(unknown)
  print("\nPredicted digit is: ")
  print(predicted)

#
===========================================================================
=======

if __name__=="__main__":
  main()
```

The program imports the entire Keras package and assigns an alias **K**. An alternative approach is to import only the modules you need, for example:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

Even though Keras uses TensorFlow as its backend engine, you don't need to explicitly import TensorFlow, except in order to set its random seed. Instead of importing the entire TensorFlow package, you could import only the module need to set the random seed. The OS package is imported only so that an annoying TensorFlow startup warning message will be suppressed.

The program structure consists of a single **main** function, plus a program-defined helper class, **MyLogger**, for custom logging, and a program-defined helper function to programmatically encode the labels. The logging class definition is:

```
class MyLogger(K.callbacks.Callback):
  def __init__(self, n):
    self.n = n

  def on_epoch_end(self, epoch, logs={}):
    if epoch % self.n == 0:
      t_loss = logs.get('loss')
      t_accu = logs.get('acc')
      print("epoch = %4d  loss = %0.4f  accuracy = %0.2f%%" % \
(epoch, t_loss, t_accu*100))
```

The **MyLogger** class is used only to display loss and accuracy metrics that are computed automatically, so the **class** initializer doesn't need to accept references to the training data. Most program-defined callbacks would pass that information like this:

```
def __init__(self, n, data_x, data_y):
  self.n = n
  self.data_x = data_x
  self.data_y = data_y
```

The **on_epoch_end()** method pulls the current loss and accuracy metrics from the built-in **logs** dictionary and displays them. The demo program does this only so that the logging display messages can be reduced to once every 10 epochs instead of every epoch. Keras computes loss and accuracy for every training batch and averages the values over all batches at the end of each epoch. If you want more granular information, you can use the **on_batch_end()** method.

The **main()** function begins with:

```
def main():
  # 0. get started
  print("\nMNIST CNN demo using Keras/TensorFlow ")
  np.random.seed(1)
  tf.set_random_seed(2)

  # 1. load data
  print("Loading subset of MNIST data into memory \n")
  train_file = ".\\Data\\mnist_train_keras_1000.txt"
  test_file = ".\\Data\\mnist_test_keras_100.txt"
. . .
```

In most situations, you want your results to be reproducible. The Keras library uses the NumPy and TensorFlow global random-number generators, so it's good practice to set the seed values. The values used in the program, **1** and **2**, are arbitrary. However, be aware that the Keras program results typically aren't completely reproducible due, in part, to order of numeric rounding of parallelized tasks.

The program assumes that the training and test data files are located in a subdirectory named **Data**. The demo program doesn't have any information about the structure of the data files. I recommend that you include comments in your program that explain things such as how many predictor variables there are, types of encoding and normalization used, and so on. This kind of information is easy to remember when your writing your program, but difficult to remember a couple of weeks later.

The training data is read into memory by these two statements:

```
  train_x = np.loadtxt(train_file, usecols=range(2,786),
    delimiter=" ", skiprows=0, dtype=np.float32)
  train_y = np.loadtxt(train_file, usecols=[0],
    delimiter=" ", skiprows=0, dtype=np.float32)
```

In general, Keras needs feature data and label data stored in separate NumPy array-of-array style matrices. There are many ways to read data into memory, but the **loadtxt()** function is versatile enough to meet most problem scenarios. An alternative approach is to use the **read_csv()** function from the Pandas (Python Data Analysis Library) package.

The default **dtype** parameter value is **numpy.float**, which is an alias for Python **float**, and is the exact same as **numpy.float64**. But the default data type for almost all Keras functions is **numpy.float32**, so the program specifies this type. The idea is that for the majority of machine learning problems, the advantage in precision gained by using 64-bit values is not worth the memory and performance penalty.

After the training data is in memory, it is encoded and normalized like so:

```
  train_x = train_x.reshape(train_x.shape[0], 28, 28, 1)
  train_x /= 255
  train_y = encode_y(train_y, 10)  # one-hot
```

A Keras CNN expects image input data as a NumPy array with four dimensions: number of items, width of image, height of image, and number of channels (**1** for grayscale, **3** for an RGB image).

Because all pixel values are between 0 and 255, dividing by 255 results in all pixel values being normalized to 0.0 to 1.0, which is in effect min-max normalization. The y-values are one-hot encoded, for example, the first training image label is a '5' digit, so it is encoded as (0, 0, 0, 0, 0, 1, 0, 0, 0, 0).

The test data is read, reshaped, normalized, and encoded in the same way as the training data. The two main advantages of programmatically encoding and normalizing data are that you can easily experiment with different approaches, and that when you use the trained model to make predictions, you can use raw, unnormalized or encoded input values. The main disadvantage of programmatically encoding and normalizing is that it adds complexity to your program.

Defining the Convolutional Neural Network model

The demo program prepares to create the CNN model with these statements:

```
  init = K.initializers.glorot_uniform()
  simple_adadelta = K.optimizers.Adadelta()
```

The demo sets up initial weights using the Glorot uniform algorithm, which, because it is the default, could have been omitted. Deep neural networks are often very sensitive to the initial values of the weights and biases, so Keras has several different initialization functions you can use.

The training optimizer object is **Adadelta()** ("adaptive delta"), one of many advanced variations of basic stochastic gradient descent. Reasonable alternatives include **RMSprop()**, **Adagrad()**, and **Adam()**; however, **SGD()** typically does not work well for CNN image classification.

The CNN network is defined like so:

```
  model = K.models.Sequential()
  model.add(K.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1,1),
    padding='same', kernel_initializer=init, activation='relu',
    input_shape=(28,28,1)))
  model.add(K.layers.Conv2D(filters=64, kernel_size=(3, 3), strides=(1,1),
    padding='same', kernel_initializer=init, activation='relu'))
  model.add(K.layers.MaxPooling2D(pool_size=(2, 2)))
  model.add(K.layers.Dropout(0.25))
  model.add(K.layers.Flatten())
  model.add(K.layers.Dense(units=100, kernel_initializer=init,
activation='relu'))
  model.add(K.layers.Dropout(0.5))
  model.add(K.layers.Dense(units=10, kernel_initializer=init,
activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='adadelta',
    metrics=['acc'])
```

There's a lot going on here. The key to a CNN is the **`Conv2D()`** layer. The idea of convolution is best explained by a diagram, such as the one in Figure 5-3. The figure shows a simplified 5x5 image in blue. The image is padded on top and bottom and on left and right by a single row/column of 0-values, shown in gray.

Convolution uses a filter (sometimes called a kernel). In Figure 5-3, there's a 3x3 filter, shown in orange. The convolution filter values are essentially the same as weight values in a regular neural network.

You can see that the filter overlays the padded image, starting in the upper left. The output result, shown in yellow, is a 5x5 matrix where the values are calculated as shown. After the filter is applied, the filter is shifted to the right by one pixel—the shift distance is called the stride.

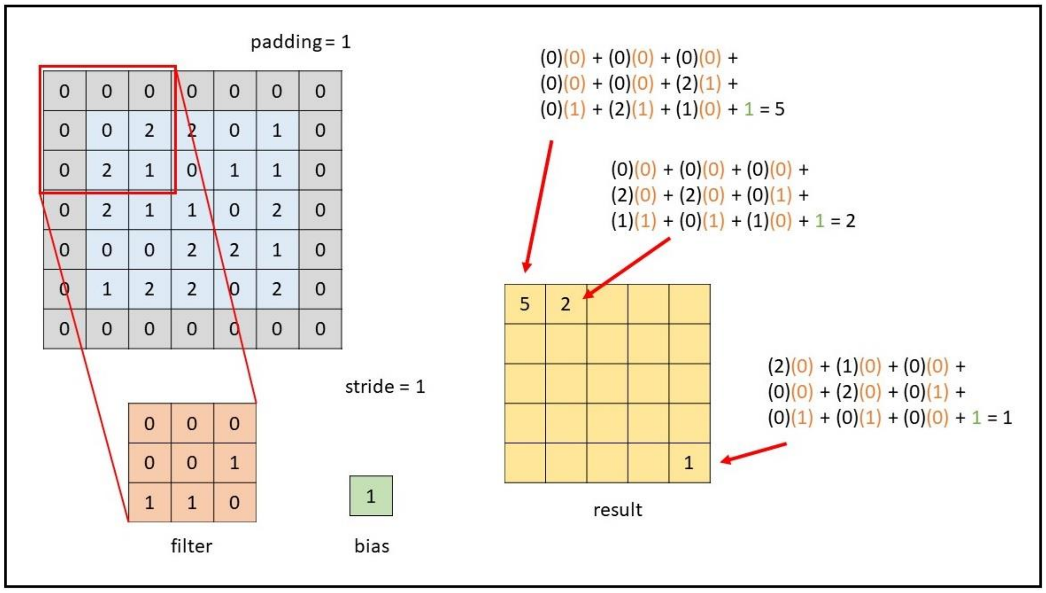


Figure 5-3: Convolution

The ideas underlying convolution are very deep. Briefly, using convolution dramatically decreases the number of weights in a CNN, making training feasible for large images. Additionally, convolution allows the model to handle mages that are shifted a few pixels up or down.

The Keras **`Conv2D()`** function accepts 15 parameters, but only two are required: **`filters`** (the number of filters) and **`kernel_size`**. The strides default is (1,1), so it could have been omitted in the demo code. The padding parameter can be **`'valid'`** or **`'same'`**, where **`'valid'`** is the default value. Using **`'same'`** means that **`Conv2D()`** will try to have even padding all around the image, as closely as possible. Using **`'valid'`** means that padding will be added only to the right and to the bottom of the image.

The demo adds a second convolutional layer with 64 filters. Additional layers, and larger number of filters, increase the predictive power of a CNN, at the expense of increasing the numbers of weights (filter values) and therefore, increasing the training time.

After the two convolutional layers, the demo program adds a **MaxPooling2D()** layer. CNN pooling is optional, but is usually employed. A 2x2 pooling layer scans through its input matrix, looking at each possible 2x2 set of cells. The four values in each 2x2 grid are replaced by a single value—the largest of the current four values. Pooling reduces the number of parameters, and therefore speeds up training. Pooling also smooths out images, which often leads to a better model in terms of accuracy.

Ultimately, the CNN is a classifier, so it needs a final layer that isn't multidimensional. The **Flatten()** layer reshapes the current matrix, which began as (28, 28, 1), into a single dimension so that one or more **Dense()** layers can be applied and categorical cross entropy can be used. The demo program also adds two **Dropout()** layers to control overfitting.

Working with CNN models can be intimidating at first, but after working with a few examples, the basic ideas start to become clear. At a high level of abstraction, the demo model accepts 784 pixel values (all between 0.0 and 1.0), and outputs a single vector of 10 values that sum to 1.0 and can be interpreted as the probability of each of the 10 possible digits. The connecting plumbing is complicated to be sure, but that plumbing is just variations of basic neural network input-output.

The model is compiled with **categorical_crossentropy** as the loss function. You could use **mean_squared_error** instead.

Training and evaluating the model

After training data has been read into memory and the CNN model has been defined, the model is trained by these statements:

```
# 3. train model
bat_size = 128
max_epochs = 50
my_logger = MyLogger(int(max_epochs/5))
print("Starting training ")
model.fit(train_x, train_y, batch_size=bat_size, epochs=max_epochs,
  verbose=0, callbacks=[my_logger])
print("Training complete")
```

The batch size is a hyperparameter, and a good value must be determined by trial and error. The **max_epochs** argument is also a hyperparameter. Larger values typically lead to lower loss and higher accuracy on the training data, at the risk of overfitting on the test data.

Training is configured to display loss and accuracy on the training data every 50 / 5 = 10 epochs. In a non-demo scenario, you'd want to see information displayed much more often.

The **fit()** function returns an object that holds complete logging information. This is sometimes useful for analysis of a model that refuses to train. The demo program does not capture the return history object.

After training, the model is evaluated:

```
  # 4. evaluate model
  loss_acc = model.evaluate(test_x, test_y, verbose=0)
  print("\nTest data loss = %0.4f  accuracy = %0.2f%%" % \
    (loss_acc[0], loss_acc[1]*100) )
```

The **evaluate()** function returns a Python list. The first value at index [0] is the always value of the required loss function specified in the **compile()** function, **categorical_crossentropy** in this case. Other values in the list are any optional **metrics** from the **compile()** function. In this example, the shortcut string **'acc'** was passed to **compile()**, so the value at index [1] holds the classification accuracy.

Saving and using the model

In most situations you'll want to save a trained model, especially if the training took hours or even longer. The demo program saves the trained model like so:

```
  # 5. save model
  print("Saving model to disk \n")
  mp = ".\\Models\\mnist_model.h5"
  model.save(mp)
```

Keras saves a trained model using the hierarchical data format (HDF) version 5. It's a binary format, so saved models can't be inspected with a text editor. In addition to saving an entire model, you can save just the model weights and biases, which is sometimes useful if you intend to transfer those values to another system. You can also save the model architecture without the weights.

A saved Keras model can be loaded from a different program like this:

```
print("Loading saved MNIST model")
mp = ".\\Models\\mnist_model.h5"
model = K.models.load_model(mp)
```

An alternative to saving the fully trained model is to save different versions of the model as they're trained. You could add the save code to the **on_epoch_end()** method of the program-defined **MyLogger** object, for example:

```
def on_epoch_end(self, epoch, logs={}):
  if epoch % self.n == 0:
    mdl_name = ".\\Models\\mnist_" + str(epoch) + "_model.h5"
    self.model.save(mdl_name)
```

The demo program concludes by making a prediction:

```
  # 6. use model
  print("Using model to predict dummy digit image: ")
  unknown = np.zeros(shape=(28,28), dtype=np.float32)
  for row in range(5,23): unknown[row][9] = 180  # vertical line
  for rc in range(9,19): unknown[rc][rc] = 250   # diagonal line
  plt.imshow(unknown, cmap=plt.get_cmap('gray_r'))
  plt.show()

  unknown = unknown.reshape(1, 28,28,1)
  predicted = model.predict(unknown)
  print("\nPredicted digit is: ")
  print(predicted)
```

The demo sets up a pseudo-digit image. The first **for** statement creates a vertical stroke that's 18 pixels tall with medium-high intensity (180). The second **for** statement creates a diagonal stroke connected to the first stroke, from upper left to lower right with high intensity (250).

Because the model was trained using non-normalized data, you must pass input value that are not normalized—values between 0 and 255.

The demo program displays a visual representation of the pseudo-digit using the PyPlot library's **imshow()** function ("image show"). Somewhat surprisingly, **imshow()** doesn't show anything when called—you must call the **show()** function.

To make a prediction, because the CNN model was trained using input with four dimensions, you must pass a multidimensional array that has four dimensions, (1 28, 28, 1). The first 1 value means one image, and the second 1 value means grayscale (a singe value between 0 and 255).

The output prediction is raw in the sense that it's just a value between 0.0 and 1.0, and therefore, it's up to you to interpret the meaning. You can do so programmatically along the lines of:

```
labels = ["zero", "one", "two", "three", "four", "five", "six" "seven",
"eight", "nine"]
idx = np.argmax(predicted[0])
result = labels[idx]
print("Predicted digit = " , result)
```

Summary and resources

To perform CNN image classification, you can encode and normalize data in a preprocessing step, or you can do so programmatically. The **Conv2D()** layer expects an input with shape (width, height, channels) where channels = 1 for a grayscale image. The number of filters, kernel size, and strides are hyperparameters, and their values must be determined by experimentation.

Using one or more **`MaxPooling2D()`** and **`Dropout()`** layers is optional, but common. You must use a **`Flatten()`** layer before a final **`Dense()`** output layer so you can use a cross entropy loss function. For training, Adagrad, Adadelta, RMSprop, and Adam are all reasonable choices. The batch size and the maximum number of training epochs to use are hyperparameters.

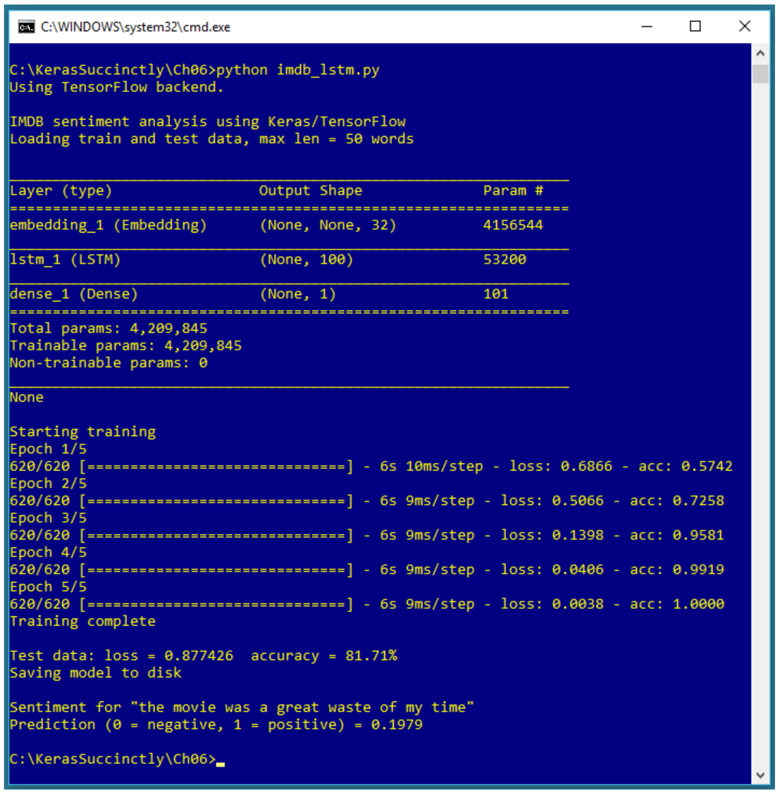
You can find the training and test data used by the demo program here.

The demo program uses only a few of the parameters to **`Conv2D()`**. For additional information, see the documentation here.

The demo program uses **`MaxPooling2D()`**. See additional details and information about other pooling methods here.

Chapter 6 Sentiment Analysis

The goal of sentiment analysis is to examine text data and predict if the mood is positive or negative. For example, you might want to programmatically process email messages from users of some product to predict if the sender is happy or not happy with the product.



```
C:\KerasSuccinctly\Ch06>python imdb_lstm.py
Using TensorFlow backend.

IMDB sentiment analysis using Keras/TensorFlow
Loading train and test data, max len = 50 words

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, None, 32)          4156544
_________________________________________________________________
lstm_1 (LSTM)                (None, 100)               53200
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 101
=================================================================
Total params: 4,209,845
Trainable params: 4,209,845
Non-trainable params: 0
_________________________________________________________________
None

Starting training
Epoch 1/5
620/620 [==============================] - 6s 10ms/step - loss: 0.6866 - acc: 0.5742
Epoch 2/5
620/620 [==============================] - 6s 9ms/step - loss: 0.5066 - acc: 0.7258
Epoch 3/5
620/620 [==============================] - 6s 9ms/step - loss: 0.1398 - acc: 0.9581
Epoch 4/5
620/620 [==============================] - 6s 9ms/step - loss: 0.0406 - acc: 0.9919
Epoch 5/5
620/620 [==============================] - 6s 9ms/step - loss: 0.0038 - acc: 1.0000
Training complete

Test data: loss = 0.877426  accuracy = 81.71%
Saving model to disk

Sentiment for "the movie was a great waste of my time"
Prediction (0 = negative, 1 = positive) = 0.1979

C:\KerasSuccinctly\Ch06>_
```

Figure 6-1: Sentiment Analysis using Keras

The screenshot in Figure 6-1 shows a demonstration of sentiment analysis. The demo program begins by loading 620 training data items and 667 test data items into memory. Each item is a movie review with up to 50 words, where the review can be positive (1) or negative (0).

Behind the scenes, the demo program creates an LSTM (long, short-term memory) neural network. The LSTM network has an embedding layer that converts each word in a review into a numeric vector with 32 values. The LSTM network has a memory cell size of 100. The network has a total of 4,209,845 weights and biases values that must be determined.

The LSTM model is trained using five epochs (a small number to keep the size of the screenshot in Figure 6-1 small). After training, the demo program computes the model's accuracy on the test data (81.71% or about 54 out of 67 correct). The demo concludes by making a prediction for a new, previously unseen review of "the movie was a great waste of my time," and correctly predicts the sentiment is negative.

Understanding the data

The IMDB (Internet Movie Database) movie review dataset consists of a total of 50,000 reviews. There are 25,000 training reviews and 25,000 test reviews. Each set has 12,500 positive review and 12,500 negative reviews. The raw data was collected as part of a research project and can be found here.

Getting the raw data into a usable format is a major challenge because the data is structured as one file per review. Here's an example of a positive review:

```
When I read other comment,i decided to watch this movie...<br /><br
/>First, cast specially Michael Madsen and Tamer Karadagli; good
enough...<br /><br />Film,very intelligence and interesting because
,cast have a lot of international specially European actor and actress
like from Turkey and Russsia...<br /><br />Second,Story is basic and
you can guess but if you interesting action good play you'll like in
my opinion...<br /><br />Third,Final chapter is not special or
interesting,it's regular like other action movies...<br /><br
/>Finally,i recommend to watch this movie...And i hope You'll love it
enjoy :D
```

Notice that there are misspelled words, incorrect grammar and punctuation, inconsistent capitalization, embedded HTML **`<br/>`** tags, and other factors to deal with. When working with natural language problems, the data preprocessing steps can often take 90 percent or more of the time and effort required to build a predictive model.

The Keras library has a built-in version of the IMDB dataset that can be loaded into memory like this:

```
from keras.datasets import imdb
(x_train, y_train), (x_test, y_test) = imdb.load_data()
```

However, using this approach is somewhat artificial, in my opinion, and hides many important details. For simplicity, I created a file of training data and a file of test data where each movie review is up to 50 words in length. The resulting data looks like this:

```
0 0 0 0 0 0 13 510 4 115 1331 363 . . . 1708 298 0
0 0 12 28 111 6 172 7 32188 9 4 88 31 . . . 1487 151 0
```

Each line is one review. The first few values on each line are zeroes for padding so that all reviews have exactly 50 values. The last value on each line is the sentiment: 0 for a negative review, and 1 for a positive review.

Each word is encoded using the same scheme as used by the built-in Keras IMDB dataset. Values 0 to 3 have special meaning. A value of 0 is used for padding. A value of 1 is used to indicate the start of a review in situations where the data is not delimited by newlines. A value of 2 is used for out-of-vocabulary (OOV)—words in the test data that were not seen in the training data. A value of 3 is reserved for custom usage.

Additionally, all words are converted to lower case, and all punctuation is removed, except for the single quote character, which is important for contractions like don't and wouldn't.

Each word ID is based upon the frequency of the word in the training data, where 4 is the most frequent ("the"), 5 is the second most frequent ("a"), and so on. The training data has a total of 129,888 distinct words, so the last word in the vocabulary has index 129,888 + 4 - 1 = 129,891.

As you'll see shortly, when using LSTM networks for natural language problems such as sentiment analysis, there's a tight coupling between data encoding and the LSTM network, and you need to know exactly how words are indexed.

The IMDB program

The complete program that generated the output shown in Figure 6-1 is shown in Code Listing 6-1. The program begins with comments for the program file name and versions of Python, TensorFlow, and Keras used, and then imports the NumPy, Keras, TensorFlow, and OS packages:

```
# imdb_lstm.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0
import numpy as np
import keras as K
import tensorflow as tf
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
```

In a non-demo scenario, you'd want to include additional details in the comments. Because Keras and TensorFlow are under rapid development, you should always document which versions are being used. Version incompatibilities can be a significant problem when working with Keras and other open-source software.

Code Listing 6-1: IMDB Movie Review Sentiment Analysis Program

```
# imdb_lstm.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0

# 
===========================================================================
=======

import numpy as np
import keras as K
import tensorflow as tf
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'

def main():
  # 0. get started
  print("\nIMDB sentiment analysis using Keras/TensorFlow ")
  np.random.seed(1)
  tf.set_random_seed(1)
  # 1. load data
  max_review_len = 50
  print("Loading train and test data, max len = %d words\n" %
max_review_len)

  train_x = np.loadtxt(".\\Data\\imdb_train_50w.txt", delimiter=" ",
    usecols=range(0,max_review_len), dtype=np.float32)
  train_y = np.loadtxt(".\\Data\\imdb_train_50w.txt", delimiter=" ",
    usecols=[max_review_len], dtype=np.float32)

  test_x = np.loadtxt(".\\Data\\imdb_test_50w.txt", delimiter=" ",
    usecols=range(0,max_review_len), dtype=np.float32)
  test_y = np.loadtxt(".\\Data\\imdb_test_50w.txt", delimiter=" ",
    usecols=max_review_len, dtype=np.float32)

  # 2. define model
  e_init = K.initializers.RandomUniform(-0.01, 0.01, seed=1)
  init = K.initializers.glorot_uniform(seed=1)
  simple_adam = K.optimizers.Adam()
  nw = 129892  # must be > vocabulary size (don't forget +4)
  embed_vec_len = 32  # values per word -- 100-500 is typical

  model = K.models.Sequential()
  model.add(K.layers.embeddings.Embedding(input_dim=nw,
output_dim=embed_vec_len,
    embeddings_initializer=e_init, mask_zero=True))
  model.add(K.layers.LSTM(units=100, kernel_initializer=init, dropout=0.2))
  model.add(K.layers.Dense(units=1, kernel_initializer=init,
activation='sigmoid'))
  model.compile(loss='binary_crossentropy', optimizer=simple_adam,
metrics=['acc'])
  print(model.summary())

  # 3. train model
  bat_size = 10
  max_epochs = 5
  print("\nStarting training ")
  model.fit(train_x, train_y, epochs=max_epochs, batch_size=bat_size,
    shuffle=True, verbose=1)
  print("Training complete \n")

  # 4. evaluate model
  loss_acc = model.evaluate(test_x, test_y, verbose=0)
  print("Test data: loss = %0.6f  accuracy = %0.2f%% " % \
    (loss_acc[0], loss_acc[1]*100))

  # 5. save model
  print("Saving model to disk \n")
  mp = ".\\Models\\imdb_model.h5"
  model.save(mp)

  # 6. use model
  print("Sentiment for \"the movie was a great waste of my time\"")
  rev = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0, 0, 0, 0, 0, 0, 0,
0, 0,
                   0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0, 0, 0, 0, 0, 0, 0,
0, 0,
                   0, 4, 20, 16, 6, 86, 425, 7, 58, 64]], dtype=np.float32)
  prediction = model.predict(rev)
  print("Prediction (0 = negative, 1 = positive) = ", end="")
  print("%0.4f" % prediction[0][0])

#
===========================================================================
=======

if __name__ == "__main__":
  main()
```

The program imports the entire Keras package and assigns an alias **K**. An alternative approach is to import only the modules you need, for example:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

Even though Keras uses TensorFlow as its backend engine, you don't need to explicitly import TensorFlow, except in order to set its random seed. The OS package is imported only so that an annoying TensorFlow startup warning message will be suppressed.

The program structure consists of a single main function, with no helper functions. The program begins with:

```
def main():
  # 0. get started
  print("\nIMDB sentiment analysis using Keras/TensorFlow ")
  np.random.seed(1)
  tf.set_random_seed(1)

  # 1. load data
  max_review_len = 50
  print("Loading train and test data, max len = %d words\n" % max_review_len)

  train_x = np.loadtxt(".\\Data\\imdb_train_50w.txt", delimiter=" ",
    usecols=range(0,max_review_len), dtype=np.float32)
  train_y = np.loadtxt(".\\Data\\imdb_train_50w.txt", delimiter=" ",
    usecols=[max_review_len], dtype=np.float32)
. . .
```

In most situations, you want to make your results reproducible. The Keras library makes extensive use of the NumPy global random-number generator, so it's good practice to set the seed value. The seed value used in the program, 1, is arbitrary. Similarly, because Keras uses TensorFlow, you'll usually want to set its seed, too. However, even if you set all random seeds, program results typically aren't completely reproducible due, in part, to order of numeric rounding of parallelized tasks.

I indent with two spaces rather than the normal four spaces because of page-width limitations. All normal error-checking has been removed to keep the main ideas as clear as possible.

The test data is read into memory using the same technique:

```
  test_x = np.loadtxt(".\\Data\\imdb_test_50w.txt", delimiter=" ",
    usecols=range(0,max_review_len), dtype=np.float32)
  test_y = np.loadtxt(".\\Data\\imdb_test_50w.txt", delimiter=" ",
    usecols=max_review_len, dtype=np.float32)
```

The program assumes that the training and test data files are located in a subdirectory named **`Data`**. The program doesn't have any information about the structure of the data files. I strongly recommend that you include program comments describing your data format. Data format information is easy to remember when you're writing your program, but difficult to remember a couple of weeks later.

The training data is read into memory using the NumPy **`loadtxt()`** function. There are many ways to read data into memory, but the **`loadtxt()`** function is versatile enough to meet most problem scenarios. The NumPy **`genfromtxt()`** function is very similar but gives you a few additional options, such as dealing with missing data. The **`loadtxt()`** function has a large number of parameters, but in most cases you only need **`usecols`**, **`delimiter`**, and **`dtype`**.

Notice that **`usecols`** can accept a list such as **`[max_review_len]`** or a Python range such as **`range(0,max_review_len)`**. If you use the **`range()`** function, be careful to remember that the first parameter is inclusive, but the second parameter is exclusive.

The default **`dtype`** parameter value is **`numpy.float`**, which is an alias for Python **`float`**, and is the exact same as **`numpy.float64`**. But the default data type for almost all Keras functions is **`numpy.float32`**, so the program specifies this type. The idea is that for the majority of machine learning problems, the advantage in precision gained by using 64-bit values is not worth the memory and performance penalty.

Instead of using a NumPy function such as **`loadtxt()`** to read data into memory, a different approach is to use the Pandas ("panel data" or "Python Data Analysis Library") library, which has many advanced data manipulation features. However, Pandas has a significant learning curve.

Defining the LSTM neural network model

The program defines an LSTM neural network using this code:

```
  # 2. define model
  e_init = K.initializers.RandomUniform(-0.01, 0.01, seed=1)
  init = K.initializers.glorot_uniform(seed=1)
  simple_adam = K.optimizers.Adam()
  nw = 129892
  embed_vec_len = 32

  model = K.models.Sequential()
  model.add(K.layers.embeddings.Embedding(input_dim=nw,
output_dim=embed_vec_len,
    embeddings_initializer=e_init, mask_zero=True))
  model.add(K.layers.LSTM(units=100, kernel_initializer=init, dropout=0.2))
  model.add(K.layers.Dense(units=1, kernel_initializer=init,
activation='sigmoid'))
  model.compile(loss='binary_crossentropy', optimizer=simple_adam,
metrics=['acc'])

  print(model.summary())
```

There's a lot going on here, so bear with me. The LSTM has two major components and several minor components. The first major component is the **`Embedding()`** layer. When working with natural language, you can feed word indexes such as 4 for "the" and 20 for "movie" directly to an LSTM network. However, this approach doesn't give very good results. A better approach is to convert each word index into a vector of real values such as (0.4508, 1.3233, . . 0.9305).

The vectors must be constructed in a way so that words that are close semantically, such as "excellent" and "wonderful," have vectors that are close numerically. There are three major ways to construct a set of word embeddings. First, you can create a custom set of embeddings based on your training data, using a separate tool, such as the Word2Vec library. Second, you can use a set of pre-built word embeddings based on a standard corpus, such as a large news feed of several hundred thousand stories from Google, or the text of all Wikipedia articles. The demo program uses a third approach, which is to compute the word embeddings on the fly, using the training data. This is a difficult problem, and is responsible for 4,156,444 of the 4,209,845 weights and biases of the model.

Notice that the **`Embedding()`** constructor requires the largest word index value. The demo uses 129,892 rather than 129,891 to indicate that you can have extra indexes if you wish. The demo program specifies an embedding vector length of 32. This value is a free parameter. For larger problems, a typical vector length is 100 to 500. Table 6-1 summarizes the seven parameters for an **`Embedding()`** layer.

Table 6-1: Embedding Layer Parameters

Name	Description
`input_dim`	Size of the vocabulary, i.e. maximum integer index + 1
`output_dim`	Dimension of the dense embedding
`embeddings_initializer`	Initializer for the embeddings matrix

Name	Description
`embeddings_regularizer`	Regularizer function applied to the embeddings matrix
`embeddings_constraint`	Constraint function applied to the embeddings matrix
`mask_zero`	Whether or not the input value 0 is a padding value
`input_length`	Length of input sequences, when it is constant

The second major component of the LSTM network is the **`LSTM()`** layer. LSTMs are fantastically complex software modules, but the key idea is that they have a memory, or equivalently, they have state. Suppose you knew that one word of a sentence was "few" and you wanted to predict the next word. You'd certainly have to take a wild guess. But if you knew the previous words were "You can't make an omelet without breaking a few…", then you'd almost surely predict the next word to be "eggs." In short, LSTM networks have state and can work well for sequences of input words.

You can get a rough idea of what an LSTM cell is by examining the diagram in Figure 6-2.

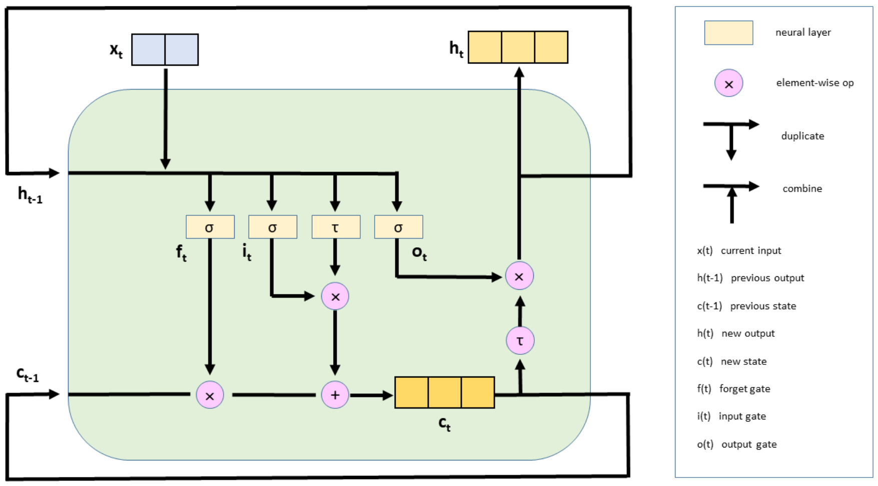


Figure 6-2: A Simplified LSTM Cell

In Figure 6-2, x(t) is the input at time t and h(t) is the corresponding output. The vector c(t) is the cell state, or memory. The output, h(t), depends on the current input and the cell state. The internal plumbing of an LSTM cell is very complex, but fortunately, when using Keras you only need a few of the 23 **`LSTM()`** parameters.

The demo program specifies the memory via the **`units=100`** argument. Memory size is a free parameter. Because of the complexity of an **`LSTM()`** layer, you can't apply dropout by using a standard **`Dropout()`** layer, so there's an internal dropout mechanism that the demo applies as a **`dropout=0.2`** argument.

After the **LSTM()** layer, the model has a single **Dense()** layer with sigmoid activation. The idea here is to compress the output of the **LSTM()** layer down to a single value between 0.0 and 1.0, which can be interpreted as the probability that the predicted class = 1. Put another way, this means that if the output is less than 0.5, the model predicts 0 = negative sentiment; otherwise, the model predicts 1 = positive sentiment.

The LSTM model is compiled using binary cross entropy as the loss function because the class labels are 0 or 1. In sentiment analysis scenarios where you have three or more class labels, such as negative = (1, 0, 0), neutral = (0, 1, 0) and positive = (0, 0, 1), you would change the activation function on the last network layer from **sigmoid** to **softmax**, and use categorical cross entropy for the **loss** function.

You can loosely think of the compilation process as translating Keras code into TensorFlow code (or CNTK code or Theano code). You must pass values to the **optimizer** and **loss** parameters so that the **fit()** method will know how to train the model. The **metrics** parameter is optional. The program passes a Python list containing just **'acc'** to indicate that classification accuracy (percentage correct predictions) should be computed during training.

The demo program displays a summary of the LSTM model using the **summary()** function. The primary purpose of using **summary()** is to check how many weights and biases your model has, which gives you an idea of how long training will take. It's possible to construct deep networks that just aren't trainable because they have too many weights and biases.

Training and evaluating the model

After training data has been read into memory and the LSTM network has been created, the demo program trains the model using these statements:

```
# 3. train model
bat_size = 10
max_epochs = 5
print("\nStarting training ")
model.fit(train_x, train_y, epochs=max_epochs, batch_size=bat_size,
  shuffle=True, verbose=1)
print("Training complete \n")
```

The batch size is set to 10, which is called online training. The batch size is a free parameter that must be determined by trial and error. Some of my colleagues like to use powers of two for their batch size: 4, 8, 16, etc., but there is no research evidence that I'm aware of that indicates this practice is better or worse. As a general rule of thumb, LSTM neural networks are very sensitive to the batch size.

The **max_epochs** variable controls how many iterations will be used for training. The **shuffle** parameter in the **fit()** function indicates that the training items should be processed in random order. The default value is **True**, so the parameter could have been omitted. The **verbose** parameter controls how much information to display during training: 0 means display no information, 1 means display full information, and 2 means display a medium amount of information.

The **fit()** function returns a dictionary object that has the recorded training history. The demo program does not capture this information.

After training, the demo program evaluates the model on the test data:

```
  # 4. evaluate model
  loss_acc = model.evaluate(test_x, test_y, verbose=0)
  print("Test data: loss = %0.6f  accuracy = %0.2f%% " % \
    (loss_acc[0], loss_acc[1]*100))
```

The **evaluate()** function returns a list of values. The first value at index [0] is the always value of the required **loss** function specified in the **compile()** function, binary cross entropy in this case. Other values in the list are any optional **metrics** from the **compile()** function. In this example, **'acc'** was passed, so the value at index [1] holds the classification accuracy. The program multiples by 100 to convert accuracy from a proportion (like 0.8123) to a percentage (like 81.23 percent).

Saving and using the model

In most situations you'll want to save a trained model, especially if the training took hours or even longer. The demo program saves the trained model like so:

```
  # 5. save model
  print("Saving model to disk \n")
  mp = ".\\Models\\imdb_model.h5"
  model.save(mp)
```

The Keras **save()** function saves a trained model using the hierarchical data format (HDF) version 5. It is a binary format, so saved models can't be inspected with a text editor. In addition to saving an entire model, you can save just the model weights and biases, which is sometimes useful. You can also save the just model architecture without the weights.

You can load a saved Keras model from a different program like this:

```
print("Loading saved IMDB sentiment model")
mp = ".\\Models\\imdb_model.h5"
model = K.models.load_model(mp)
```

The whole point of creating and training a model is so that it can be used to make predictions for new, previously unseen data:

```
  # 6. use model
  print("Sentiment for \"the movie was a great waste of my time\"")
  rev = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
                   0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
                   0, 4, 20, 16, 6, 86, 425, 7, 58, 64]], dtype=np.float32)
  prediction = model.predict(rev)
  print("Prediction (0 = negative, 1 = positive) = ", end="")
  print("%0.4f" % prediction[0][0])
```

Because the LSTM model was trained using reviews that have length padded to 50 encoded words, when making a prediction you must pass a new review to the **`predict()`** method using the same format. The encoded values for "the movie was a great waste of my time" were hard-coded. However, in a non-demo scenario, when you create the training and test data files, you would save the encodings to a text file, typically named something like **vocab.txt**, along the lines of:

```
the   4
waste 425
time  64
. . .
```

Then you could write a script that opens the vocabulary file and reads the file into a dictionary object, where a word is the dictionary key ,and the encoded index is the dictionary value.

Summary and resources

To create a classification prediction model where the input is a sequence of text such as sentences, you can use an LSTM network that consists of one or more LSTM cells plus some additional plumbing such as a dense layer.

When working with text input, words should be encoded as numeric vectors, a process called embedding. You can either create embeddings in a preprocessing phase, or you can create an embedding on the fly using an **`Embedding()`** layer.

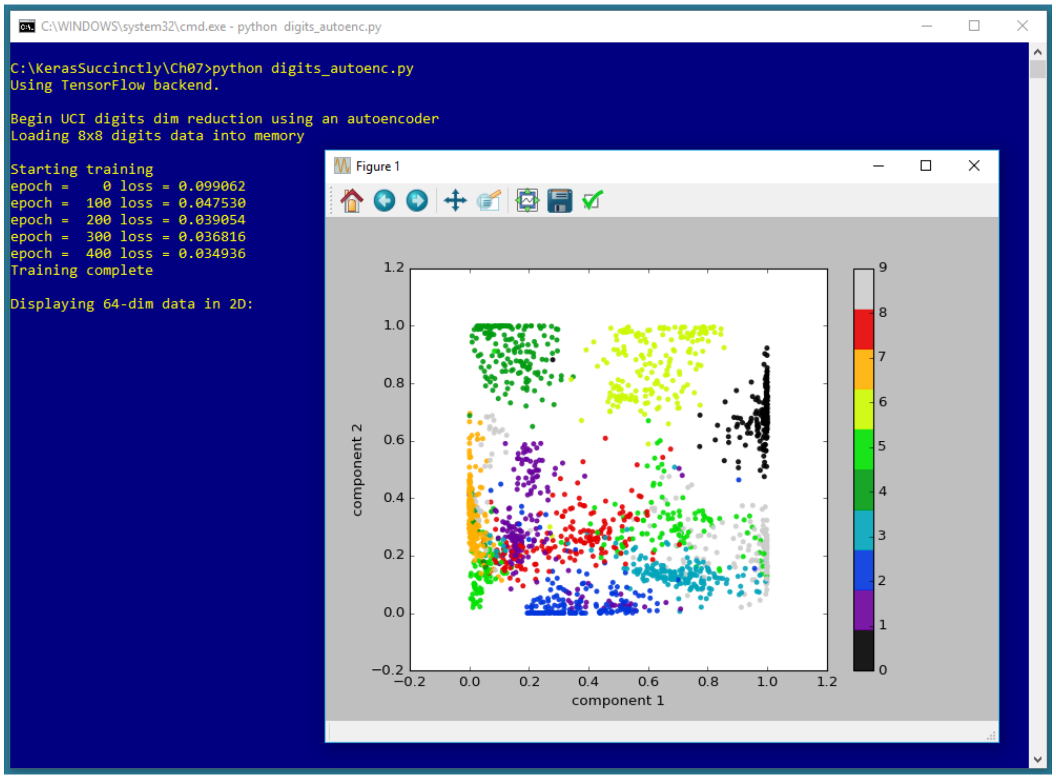
Free parameters for LSTM models include weight-initialization algorithms, optimization algorithm and its parameters, dropout rate, batch size, and number of training iterations.

You can find the training and test data used by the demo program here.

The demo program uses just three of the 23 parameters for the **`LSTM()`** constructor. You can find additional information here.

Chapter 7 Autoencoders

An autoencoder is a type of neural network that can perform dimensionality reduction for visualization. For example, suppose you have data that represents the age and height of men and women. If you want to graph your data, you can do so easily by plotting age on the x-axis and height on the y-axis, with blue dots for men and pink dots for women. But if your data has five dimensions, such as (age, height, weight, income, years-education), then there's no easy way to graph the data.



```
C:\KerasSuccinctly\Ch07>python digits_autoenc.py
Using TensorFlow backend.

Begin UCI digits dim reduction using an autoencoder
Loading 8x8 digits data into memory

Starting training
epoch =    0 loss = 0.099062
epoch =  100 loss = 0.047530
epoch =  200 loss = 0.039054
epoch =  300 loss = 0.036816
epoch =  400 loss = 0.034936
Training complete

Displaying 64-dim data in 2D:
```

Figure 7-1: Autoencoder Dimensionality Reduction for Visualization using Keras

The screenshot in Figure 7-1 shows a demonstration of autoencoder dimensionality reduction for visualization. The demo program begins by loading 1,797 data items into memory. Each data item has 64 dimensions. The demo program creates an autoencoder that encodes/compresses each 64-dimensional data item down to two dimensions, and then graphs the result. Each of the data items belong to one of ten classes, and this information is used to color each point on the graph.

The 1,797 data items represent crude 8x8 bitmaps of handwritten digits from 0 to 9. In other words, the autoencoder visualization is applied to data that is itself a representation of a visualization. However, autoencoder dimensionality reduction for visualization can be applied to any type of data. For example, the Fisher's Iris dataset has four dimensions (sepal length, sepal width, petal length, and petal width), and the data could be reduced to two dimensions for visualization using an autoencoder.

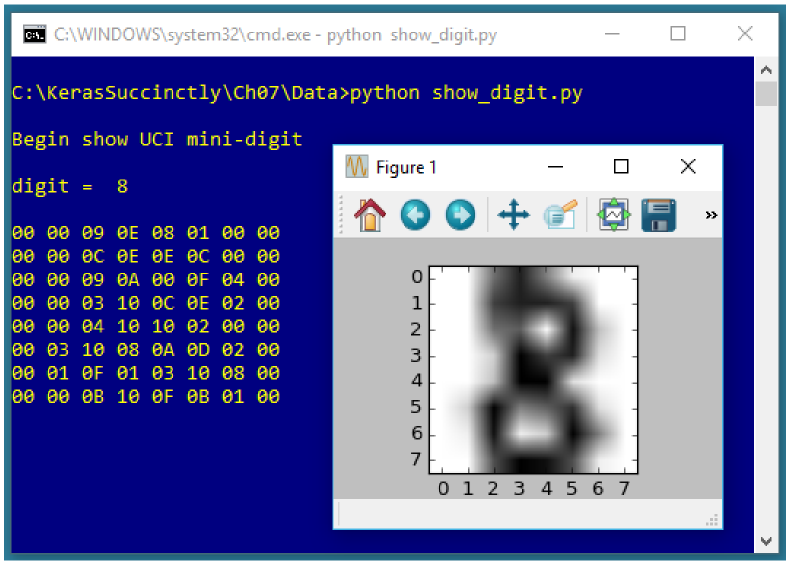
Understanding the data

The demo data looks like this:

```
0,0,0,1,11,0,0,0,0,0,0,7,8, . . . 16,4,0,0,4
0,0,9,14,8,1,0,0,0,0,12,14, . . . 5,11,1,0,8
. . .
```

Each line of data represents a handwritten digit. The first 64 values on a line are grayscale pixel values between 0 and 16. The last value on a line is the digit value.

The screenshot in Figure 7-2 shows one of the data items. The data item is first displayed in the shell, using the raw pixel values expressed in hexadecimal. Then the item is displayed graphically.



```
C:\KerasSuccinctly\Ch07\Data>python show_digit.py

Begin show UCI mini-digit

digit =  8

00 00 09 0E 08 01 00 00
00 00 0C 0E 0E 0C 00 00
00 00 09 0A 00 0F 04 00
00 00 03 10 0C 0E 02 00
00 00 04 10 10 02 00 00
00 03 10 08 0A 0D 02 00
00 01 0F 01 03 10 08 00
00 00 0B 10 0F 0B 01 00
```

Figure 7-2: One of the UCI Digits

The goal of an autoencoder is to reduce the 64 dimensions of an item down to just two values so the item can be graphed as a point on an x-y graph.

The Autoencoder program

The complete program that generated the output shown in Figure 7-1 is shown in Code Listing 7-1. The program begins with comments for the program file name and versions of Python, TensorFlow, and Keras used, and then imports the NumPy, Keras, TensorFlow, PyPlot and OS packages:

```
# digits_autoenc.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0
import numpy as np
import keras as K
import tensorflow as tf
import matplotlib.pyplot as plt
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
```

In a non-demo scenario, you'd want to include additional details in the comments. Because Keras and TensorFlow are under rapid development, you should always document which versions are being used. Version incompatibilities can be a significant problem when working with Keras and other open-source software.

Code Listing 7-1: Autoencoder Program

```
# digits_autoenc.py
# Python 3.5.2, TensorFlow 2.1.5, Keras 1.7.0

#
===========================================================================
=======

import numpy as np
import keras as K
import tensorflow as tf
import matplotlib.pyplot as plt
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='2'  # suppress CPU msg

class MyLogger(K.callbacks.Callback):
  def __init__(self, n):
    self.n = n   # print loss every n epochs

  def on_epoch_end(self, epoch, logs={}):
    if epoch % self.n == 0:
      curr_loss =logs.get('loss')
      print("epoch = %4d loss = %0.6f" % (epoch, curr_loss))

def main():
  # 0. get started
  print("\nBegin UCI digits dim reduction using an autoencoder")
  np.random.seed(1)
  tf.set_random_seed(1)

  # 1. load data into memory
  print("Loading 8x8 digits data into memory \n")
  data_file = ".\\Data\\digits_uci_test_1797.txt"
  data_x = np.loadtxt(data_file, delimiter=",", usecols=range(0,64),
    dtype=np.float32)
  labels = np.loadtxt(data_file, delimiter=",", usecols=[64],
    dtype=np.float32)
  data_x = data_x / 16

  # 2. define autoencoder
  my_init = K.initializers.glorot_uniform(seed=1)
  X = K.layers.Input(shape=[64])
  layer1 = K.layers.Dense(units=32, activation='sigmoid',
    kernel_initializer=my_init)(X)
  layer2 = K.layers.Dense(units=2, activation='sigmoid',
    kernel_initializer=my_init)(layer1)
  layer3 = K.layers.Dense(units=32, activation='sigmoid',
    kernel_initializer=my_init)(layer2)
  layer4 = K.layers.Dense(units=64, activation='sigmoid',
    kernel_initializer=my_init)(layer3)

  enc_dec = K.models.Model(X, layer4)
  encoder = K.models.Model(X, layer2)

  # 3. compile model
  simple_adam = K.optimizers.Adam()
  enc_dec.compile(loss='mean_squared_error',
    optimizer=simple_adam)

  # 4. train model
  print("Starting training")
  max_epochs = 500
  my_logger = MyLogger(n=100)
  h = enc_dec.fit(x=data_x, y=data_x, batch_size=8, epochs=max_epochs,
    verbose=0, callbacks=[my_logger])
  print("Training complete \n")

  # 5. generate (x,y) pairs for each digit
  reduced = encoder.predict(data_x)

  # 6. graph the digits in 2D
  print("Displaying 64-dim data in 2D: \n")
  plt.scatter(x=reduced[:, 0], y=reduced[:, 1],
    c=labels, edgecolors='none', alpha=0.9,
    cmap=plt.cm.get_cmap('nipy_spectral', 10), s=20)

  plt.xlabel('component 1')
  plt.ylabel('component 2')
  plt.colorbar()
  plt.show()

#
===========================================================================
=======

if __name__ == "__main__":
  main()
```

The program imports the entire Keras package and assigns an alias **`K`**. An alternative approach is to import only the modules you need, for example:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

Even though Keras uses TensorFlow as its backend engine, you don't need to explicitly import TensorFlow, except in order to set its random seed. The OS package is imported only so that an annoying TensorFlow startup warning message will be suppressed.

The program structure consists of a single **`main`** function, plus a helper class for displaying messages during training. The helper class definition is:

```
class MyLogger(K.callbacks.Callback):
  def __init__(self, n):
    self.n = n   # print loss every n epochs

  def on_epoch_end(self, epoch, logs={}):
    if epoch % self.n == 0:
      curr_loss =logs.get('loss')
      print("epoch = %4d loss = %0.6f" % (epoch, curr_loss))
```

The **`MyLogger`** class is used to print the value of the built-in loss function every 100 epochs. The idea is that the **`fit()`** method can display progress messages every epoch, or not at all, but if you want messages every few epochs, you must define a custom callback class.

The **`main()`** code begins with:

```
def main():
  # 0. get started
  print("\nBegin UCI digits dim reduction using an autoencoder")
  np.random.seed(1)
  tf.set_random_seed(1)

  # 1. load data into memory
  print("Loading 8x8 digits data into memory \n")
  data_file = ".\\Data\\digits_uci_test_1797.txt"
  data_x = np.loadtxt(data_file, delimiter=",", usecols=range(0,64),
    dtype=np.float32)
  labels = np.loadtxt(data_file, delimiter=",", usecols=[64],
    dtype=np.float32)
  data_x = data_x / 16
. . .
```

In most situations, you want to make your results reproducible. The Keras library makes extensive use of the NumPy global random-number generator, so it's good practice to set the seed value. The seed value used in the program, **`1`**, is arbitrary. Similarly, because Keras uses TensorFlow, you'll often want to set its seed, too. Unfortunately, program results typically aren't completely reproducible due to order of numeric rounding of parallelized tasks.

My preferred style is to indent with two spaces rather than the normal four spaces. All normal error-checking has been removed to keep the main ideas as clear as possible.

The program assumes that the training and test data files are located in a subdirectory named **`Data`**. The program itself doesn't have any information about the structure of the data files. I strongly recommend that you include in your program comments such as:

```
# data has 1797 items, is comma-delimited and looks like:
# 0,0,12,10, . . 13,11,5
# 0,0,0,2,8, . . 12,10,8
# first 64 values are grayscale pixel values between 0 and 16
# last value is the class label, '0' through '9'
```

This kind of information is easy to remember when you're writing your program, but can be very difficult to remember a couple of weeks later.

The single data file is read into memory using the **`np.loadtxt()`** function. There are many ways to read data into memory, but the **`loadtxt()`** function is versatile enough to meet most problem scenarios. The NumPy **`genfromtxt()`** function is very similar but gives you a few additional options, such as dealing with missing data. The **`loadtxt()`** function has a large number of parameters, but in most cases, you only need **`usecols`**, **`delimiter`**, and **`dtype`**.

Notice that **`usecols`** can accept a list such as **`[64]`** or a Python range such as **`range(0,64)`**. If you use the **`range()`** function, be careful to remember that the first parameter is inclusive, but the second parameter is exclusive.

The default **`dtype`** parameter value is **`numpy.float`**, which is an alias for Python **`float`**, and is the exact same as **`numpy.float64`**. But the default data type for almost all Keras functions is **`numpy.float32`**, so the program specifies this type. The idea is that for the majority of machine learning problems, the advantage in precision gained by using 64-bit values is not worth the memory and performance penalty.

Instead of using a NumPy function such as **`loadtxt()`** to read data into memory, a different approach is to use the Pandas (originally "panel data," now "Python Data Analysis") library, which has many advanced data manipulation features. However, Pandas has a non-trivial learning curve and requires significant investment of your time.

After the digits pixel x-data has been loaded into memory, the data is normalized by dividing all values by 16. This makes all pixel values between 0.0 and 1.0, which makes the autoencoder a bit easier to train.

Defining the autoencoder model

The program defines a 64-32-2-32-64 autoencoder using this code:

```
# 2. define autoencoder
my_init = K.initializers.glorot_uniform(seed=1)
X = K.layers.Input(shape=[64])
layer1 = K.layers.Dense(units=32, activation='sigmoid',
  kernel_initializer=my_init)(X)
layer2 = K.layers.Dense(units=2, activation='sigmoid',
  kernel_initializer=my_init)(layer1)
layer3 = K.layers.Dense(units=32, activation='sigmoid',
  kernel_initializer=my_init)(layer2)
layer4 = K.layers.Dense(units=64, activation='sigmoid',
  kernel_initializer=my_init)(layer3)

enc_dec = K.models.Model(X, layer4)
encoder = K.models.Model(X, layer2)
```

The architecture of an autoencoder for dimensionality reduction is best explained by a diagram—see Figure 7-3. The diagram shows a small 6-3-2-3-6 autoencoder rather than the large 64-32-2-32-64 architecture of the demo program.

There are two key ideas. First, an autoencoder's input and output are the same. Second, the inner-most layer has two nodes, which correspond to the x-axis and y-axis values for graphing.

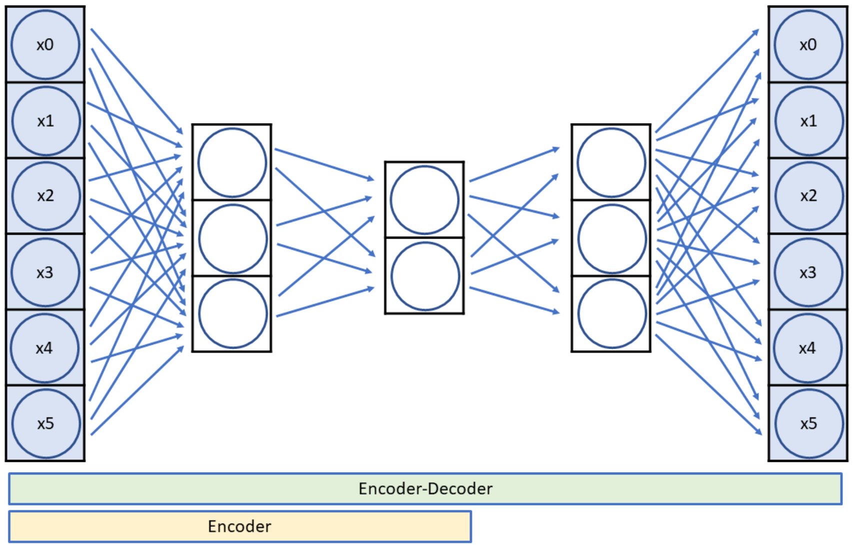


Figure 7-3: A 6-3-2-3-6 Autoencoder

An autoencoder is a special neural network that learns to predict its own input. After training, the inner-most two nodes are a reduced dimensionality representation of the input. An autoencoder is a specific type of neural network called an encoder-decoder. The encoder part of the network extracts the compressed representation of the network's input.

The number of autoencoder input nodes and output nodes is determined by the dimensionality of your data. The inner-most layer will usually have two or three nodes if the goal is dimensionality reduction for visualization. The number of other hidden layers and the number of nodes in each layer are free parameters.

Compiling and training the autoencoder

After training data has been read into memory and the autoencoder has been defined, the model is compiled and trained:

```
  # 3. compile model
  simple_adam = K.optimizers.Adam()
  enc_dec.compile(loss='mean_squared_error', optimizer=simple_adam)

  # 4. train model
  print("Starting training")
  max_epochs = 500
  my_logger = MyLogger(n=100)
  h = enc_dec.fit(x=data_x, y=data_x, batch_size=8, epochs=max_epochs,
    verbose=0, callbacks=[my_logger])
  print("Training complete \n")
```

The **Adam** (adaptive moment estimation) optimizer is a good general-purpose learner for deep neural networks, but **Adagrad**, **Adadelta**, and **RMSprop** are reasonable alternatives. Because the target values are type **np.float32**, the autoencoder is compiled using mean-squared error rather than cross-entropy error.

The number of training epochs and the batch size are free parameters. Notice that the **fit()** method is passed **data_x** for both the **x** and **y** parameters. The demo program captures the return object holding the training history from the **fit()** method, but doesn't make use of it. If you want to see the loss values, you can do so like this:

```
print(h.history['loss'])
```

The **verbose=0** parameter suppresses all built-in logging messages so that only the ones generated by the **my_logger** callback object are displayed.

Saving and using the autoencoder

In most situations, you'll want to save a trained model, especially if the training took hours or even longer. The demo program does not save the trained autoencoder, but you can do so like this:

```
  # save autoencoder model
  print("Saving model to disk \n")
  mp = ".\\Models\\autoenc_model.h5"
  encoder.save(mp)
```

Keras saves trained models using the hierarchical data format (HDF) version 5. It is a binary format, so saved models can't be inspected with a text editor. In addition to saving an entire model, you can save only the model weights and biases, which is sometimes useful. You can also save the model architecture without the weights.

A saved Keras autoencoder can be loaded from a different program like this:

```
print("Loading a saved model")
mp = ".\\Models\\autoenc_model.h5"
encoder = K.models.load_model(mp)
```

The demo program generates a two-dimensional visualization of the 1797 64-dimensional data items using these statements:

```
  # 5. generate (x,y) pairs for each digit
  reduced = encoder.predict(data_x)

  # 6. graph the digits in 2D
  print("Displaying 64-dim data in 2D: \n")
  plt.scatter(x=reduced[:, 0], y=reduced[:, 1],
    c=labels, edgecolors='none', alpha=0.9,
    cmap=plt.cm.get_cmap('nipy_spectral', 10), s=20)

  plt.xlabel('component 1')
  plt.ylabel('component 2')
  plt.colorbar()
  plt.show()
```

The call to the **`predict()`** method returns a NumPy array-of-arrays style matrix named **`reduced`** with 1797 rows and two columns, where column [0] is the x-axis value and column [1] is the y-axis value. The PyPlot **`scatter()`** function is used to generate a scatter plot.

The **`scatter()`** parameter names are a bit cryptic. Parameter **`c`** is a sequence of n numbers to be mapped to colors using the **`cmap`** parameter. Recall that array **`label`** holds the class labels 0 through 9 (as type **`np.float32`**) for each data item.

The **`cmap`** parameter ("colormap") has value **`'nipy_spectral'`**, which uses a continuous set of colors from dark purple, to green, to dark red. There are many other PyPlot colormaps, including **`'rainbow'`**, **`'jet'`**, **`'Dark1'`**, and **`'cool'`**.

The **`s`** parameter controls the size (measured in points) of the marker dots on the scatter plot. The **`alpha`** parameter controls the transparency of the marker dots.

Summary and resources

An autoencoder is a special type of neural network that learns to predict its own input values. Because autoencoders don't use labeled data during training, autoencoders are an example of an unsupervised technique.

One common use of autoencoders is for dimensionality reduction, so that high-dimensionality data can be visualized on a two-dimensional or three-dimensional graph.

You can find the 1797-item data file used by the demo program here.

Other resources:

- Complete UCI digits dataset
- Reference for the PyPlot scatter function
- Colormap examples
- Reference for the Keras Model class API

Appendix

The program presented in Code Listing A-1 was used in Chapter 4 to split the Cleveland Heart Disease dataset file into training, validation, and test files.

Code Listing A-1: Program split_file.py

```
# split_file.py
# does not read source into memory
# useful when no processing/normalization needed

import numpy as np

def file_len(fname):
 f = open(fname)
 for (i, line) in enumerate(f): pass
 f.close()
 return i+1

def main():
  source_file = ".\\cleveland_norm.txt"
  train_file = ".\\cleveland_train.txt"
  validate_file = ".\\cleveland_validate.txt"
  test_file = ".\\cleveland_test.txt"

  N = file_len(source_file)
  num_train = int(0.60 * N)
  num_validate = int(0.20 * N)
  num_test = N - (num_train + num_validate) # ~20%

  np.random.seed(1)
  indices = np.arange(N)  # array [0, 1, . . N-1]
  np.random.shuffle(indices)

  train_dict = {}
  test_dict = {}
  validate_dict = {}
  for i in range(0,num_train):
    k = indices[i]; v = i  # i is not used
    train_dict[k] = v

  for i in range(num_train,(num_train+num_validate)):
    k = indices[i]; v = i
    validate_dict[k] = v

  for i in range((num_train+num_validate),N):
    k = indices[i]; v = i
    test_dict[k] = v

  f_source = open(source_file, "r")
  f_train = open(train_file, "w")
  f_validate = open(validate_file, "w")
  f_test = open(test_file, "w")

  line_num = 0
  for line in f_source:
    if line_num in train_dict: # checks for key
      f_train.write(line)
    elif line_num in validate_dict:
      f_validate.write(line)
    else:
      f_test.write(line)
    line_num += 1

  f_source.close()
  f_train.close()
  f_validate.close()
  f_test.close()

if __name__ == "__main__":
  main()
```

The program presented in Code Listing A-2 was used in Chapter 5 to create the MNIST training and test files.

Code Listing A-2: Program make_data.py

```
# make_data.py
# raw binary MNIST to Keras text file
#
# go to http://yann.lecun.com/exdb/mnist/ and
# download the four g-zipped files:
# train-images-idx3-ubyte.gz (60,000 train images)
# train-labels-idx1-ubyte.gz (60,000 train labels)
# t10k-images-idx3-ubyte.gz  (10,000 test images)
# t10k-labels-idx1-ubyte.gz  (10,000 test labels)
#
# use the 7-Zip program to unzip the four files.
# I recommend adding a .bin extension to remind
# you they're in a proprietary binary format
#
# run the script twice, once for train data, once for
# test data, changing the file names as appropriate.
```

```python
# uses pure Python only

# target format:
# 5 ** 0 0 152 27 .. 0
# 7 ** 0 0 38 122 .. 0
# label digit at [0]    784 vals at [2-786]
# dummy ** seperator at [1]

def generate(img_bin_file, lbl_bin_file,
            result_file, n_images):

  img_bf = open(img_bin_file, "rb")    # binary image pixels
  lbl_bf = open(lbl_bin_file, "rb")    # binary labels
  res_tf = open(result_file, "w")      # result file

  img_bf.read(16)   # discard image header info
  lbl_bf.read(8)    # discard label header info

  for i in range(n_images):   # number images requested
    # digit label first
    lbl = ord(lbl_bf.read(1))  # get label like '3' (one byte)
    res_tf.write(str(lbl))

    # encoded = [0] * 10         # make one-hot vector
    # encoded[lbl] = 1
    # for i in range(10):
    #  res_tf.write(str(encoded[i]))
    #  res_tf.write(" ")  # like 0 0 0 1 0 0 0 0 0 0

    res_tf.write(" ** ")  # arbitrary seperator char for readibility

    # now do the image pixels
    for j in range(784):  # get 784 vals for each image file
      val = ord(img_bf.read(1))
      res_tf.write(str(val))
      if j != 783: res_tf.write(" ")  # avoid trailing space
    res_tf.write("\n")  # next image

  img_bf.close(); lbl_bf.close();  # close the binary files
  res_tf.close()                   # close the result text file

# ================================================================

def main():
  # generate(".\\UnzippedBinary\\train-images.idx3-ubyte.bin",
  #          ".\\UnzippedBinary\\train-labels.idx1-ubyte.bin",
  #          ".\\mnist_train_keras_1000.txt",
  #          n_images = 1000)  # first n images
  generate(".\\UnzippedBinary\\t10k-images.idx3-ubyte.bin",
          ".\\UnzippedBinary\\t10k-labels.idx1-ubyte.bin",
          ".\\mnist_test_keras_foo.txt",
          n_images = 100)  # first n images

if __name__ == "__main__":
  main()
```

The program presented in Code Listing A-3 was used in Chapter 5 to display an MNIST digit.

Code Listing A-3: Program show_image.py

```
# show_image.py

import numpy as np
import matplotlib.pyplot as plt

# data file looks like:
# 5 ** 0 .. 23 157 .. 0
# 4 ** 0 .. 255 16 .. 0
# note dummy separator at [1]

def display(txt_file, idx):
  # values between 0-255
  # data file has 1 + 1 + 784 = 786 vals per line, [0] to [785]

  y_data = np.loadtxt(txt_file, delimiter = " ",
    usecols=[0], dtype=np.float32)
  x_data = np.loadtxt(txt_file, delimiter = " ",
    usecols=range(2,786), dtype=np.float32)

  label = int(y_data[idx])  # like '5'
  print("digit = ", str(label), "\n")

  pixels = np.array(x_data[idx,], dtype=np.int)  # to int
  pixels = pixels.reshape((28,28))
  for i in range(28):
    for j in range(28):
      print("%.2X" % pixels[i,j], end="")
      print(" ", end="")
    print("")

  img = np.array(x_data[idx,])   # as float32
  img = img.reshape((28,28))
  plt.imshow(img, cmap=plt.get_cmap('gray_r'))
  plt.show()
```

```
def main():
  print("\nBegin show MNIST image demo \n")

  img_file = ".\\mnist_train_keras_1000.txt"
  display(img_file, idx=0)  # first image

  print("\nEnd \n")

if __name__ == "__main__":
  main()
```

The program presented in Code Listing A-4 was used in Chapter 6 to create the IMDB Movie Review training and test files.

Code Listing A-4: Program make_data_files.py

```
# make_data_files.py
#
# input: source Stanford 50,000 data files reviews
# output: one combined train file, one combined test file
# output files are in index version, using the Keras dataset
# format where 0 = padding, 1 = 'start', 2 = OOV, 3 = unused
# 4 = most frequent word ('the'), 5 = next most frequent, etc.
# i'm skipping the start=1 because it makes no sense.
# these data files will be loaded into memory then feed
# a built-in Embedding layer (rather than custom embeddings)

import os

# allow the Windws cmd shell to deal with wacky characters
import sys
import codecs
sys.stdout = codecs.getwriter('utf8')(sys.stdout.buffer)

# ---------------------------------------------------------------

def get_reviews(dir_path, num_reviews, punc_str):
  punc_table = {ord(char): None for char in punc_str}  # dictionary
  reviews = []  # list-of-lists of words
  ctr = 1
  for file in os.listdir(dir_path):
    if ctr > num_reviews: break
    curr_file = os.path.join(dir_path, file)
    f = open(curr_file, "r", encoding="utf8")   # each review has only
one line . . .
    for line in f:
      line = line.strip()
      if len(line) > 0:  # number characters
        # print(line)  # to show non-ASCII == errors
        line = line.translate(punc_table)  # remove punc
        line = line.lower()  # lower case
        line = " ".join(line.split())  # remove consecutive WS
        word_list = line.split(" ")  # one review is a list of words
        reviews.append(word_list)    #
    f.close()  # close curr file
    ctr += 1
  return reviews

# ---------------------------------------------------------------

def make_vocab(all_reviews):
  word_freq_dict = {}   # key = word, value = frequency

  for i in range(len(all_reviews)):
    reviews = all_reviews[i]
    for review in reviews:
      for word in review:
        if word in word_freq_dict:
          word_freq_dict[word] += 1
        else:
          word_freq_dict[word] = 1

  kv_list = []  # list of word-freq tuples so can sort
  for (k,v) in word_freq_dict.items():
    kv_list.append((k,v))

  # list of tuples where index is 0-based rank, val is (word,freq)
  sorted_kv_list = sorted(kv_list, key=lambda x: x[1], reverse=True)  #
sort by freq

  f = open(".\\vocab_file.txt", "w", encoding="utf8")
  vocab_dict = {}  # key = word, value = 1-based rank ('the' = 1, 'a' =
2, etc.)
  for i in range(len(sorted_kv_list)):
    w = sorted_kv_list[i][0]  # word is at [0]
    vocab_dict[w] = i+1       # 1-based as in Keras dataset

    f.write(w + " " + str(i+1) + "\n")  # save word-space-index
  f.close()

  return vocab_dict

# ---------------------------------------------------------------

def generate_file(reviews_lists, outpt_file, w_or_a, vocab_dict,
max_review_len, label_char):
  fout = open(outpt_file, w_or_a, encoding="utf8")  # write first time,
append later
  offset = 3  # Keras offset: 'the' = 1 (most frequent) -> 1+3 = 4

  for i in range(len(reviews_lists)):  # walk thru each review-list
    curr_review = reviews_lists[i]
    n_words = len(curr_review)
    if n_words > max_review_len:
      continue  # next i, continue without writing anything
    n_pad = max_review_len - n_words   # number of 0s to prepend
    for j in range(n_pad):
      fout.write("0 ")
    for word in curr_review:
      if word not in vocab_dict:  # a word in test set not in training
set
        fout.write("2 ")   # 2 is the special out-of-vocab index
      else:
        idx = vocab_dict[word] + offset
        fout.write("%d " % idx)
    fout.write(label_char + "\n")  # like '0' or '1', or 'N' or 'P'

  fout.close()

# ----------------------------------------------------------------

def main():
  remove_chars = "!\"#$%&()*+,-./:;<=>?@[\\]^_`{|}~"   # leave ' for
words like it's

  print("\nLoading all reviews into memory - be patient ")
  pos_train_reviews = get_reviews(".\\SourceFiles\\train\\pos", 12500,
remove_chars)
  neg_train_reviews = get_reviews(".\\SourceFiles\\train\\neg", 12500,
remove_chars)
  pos_test_reviews = get_reviews(".\\SourceFiles\\test\\pos", 12500,
remove_chars)
  neg_test_reviews = get_reviews(".\\SourceFiles\\test\\neg", 12500,
remove_chars)

  mp = max(len(l) for l in pos_train_reviews)  # 2469
  mn = max(len(l) for l in neg_train_reviews)  # 1520
  mm = max(mp, mn)  # longest (in words) review in training set  # 2469
  # print(mp, mn)

  print("\nAnalyzing reviews and making vocabulary ")
  vocab_dict = make_vocab([pos_train_reviews, neg_train_reviews])  # key
= word, value = word rank
  v_len = len(vocab_dict)  # need this value, plus 4, for Embedding
Layer: 129888+4 = 129892
```

```
  print("\nVocab size = %d -- use this +4 for Embedding nw " % v_len)

  max_review_len = 50   # use None for all reviews (any len)
  if max_review_len == None or max_review_len > mm:
    max_review_len = mm

  print("\nGenerating training file with len %d words or less " %
max_review_len)

  generate_file(pos_train_reviews, ".\\imdb_train_50w.txt", "w",
vocab_dict, max_review_len, "1")
  generate_file(neg_train_reviews, ".\\imdb_train_50w.txt", "a",
vocab_dict, max_review_len, "0")

  print("\nGenerating test file with len %d words or less " %
max_review_len)

  generate_file(pos_test_reviews, ".\\imdb_test_50w.txt", "w",
vocab_dict, max_review_len, "1")
  generate_file(neg_test_reviews, ".\\imdb_test_50w.txt", "a",
vocab_dict, max_review_len, "0")

  # inspect a generated file
  # vocab_dict was used indirectly (offset)

  # print("Displaying encoded training file: \n")
  # f = open(".\\imdb_train_50w.txt", "r", encoding="utf8")
  # for line in f:
  #   print(line, end="")
  # f.close()

  # print("\nDisplaying decoded training file: \n")

  # index_to_word = {}
  # index_to_word[0] = "<PAD>"
  # index_to_word[1] = "<ST>"
  # index_to_word[2] = "<OOV>"
  # for (k,v) in vocab_dict.items():
  #   index_to_word[v+3] = k

  # f = open(".\\imdb_train_50w.txt", "r", encoding="utf8")
  # for line in f:
  #   line = line.strip()
  #   indexes = line.split(" ")
  #   for i in range(len(indexes)-1):  # last is '0' or '1'
  #     idx = (int)(indexes[i])
  #     w = index_to_word[idx]
  #     print("%s " % w, end="")
  #   print("%s \n" % indexes[len(indexes)-1])
  # f.close()

if __name__ == "__main__":
  main()
```

The program presented in Code Listing A-5 was used in Chapter 7 to display a UCI Digit Dataset image.

Code Listing A-5: Program show_digit.py

```
# show_digit.py

import numpy as np
import matplotlib.pyplot as plt

# data file looks like:
# 0,0,5,16 . . 12,0,0,7
# first 64 values are grayscale pixel (0-16), last is digit (0-9)

def display(data_file, idx):
  x_data = np.loadtxt(data_file, delimiter = ",",
    usecols=range(0,64), dtype=np.int)
  y_data = np.loadtxt(data_file, delimiter = ",",
    usecols=[64], dtype=np.int)

  label = y_data[idx]  # like '5'
  print("digit = ", str(label), "\n")

  pixels = np.array(x_data[idx])  # target row of pixels
  pixels = pixels.reshape((8,8))
  for i in range(8):
    for j in range(8):
      print("%.2X" % pixels[i,j], end="")
      print(" ", end="")
    print("")

  plt.imshow(pixels, cmap=plt.get_cmap('gray_r'))
  plt.show()

def main():
  print("\nBegin show UCI mini-digit \n")

  data_file = ".\\digits_uci_test_1797.txt"
  display(data_file, idx=8)

  print("\nEnd \n")
```

```
if __name__ == "__main__":
  main()
```