Introduction to CNTK

Succinctly

By James McCaffrey

Succinctly E-Book Series

Syncfusion®

Introduction to CNTK Succinctly

By

James McCaffrey

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Technical Reviewer: Chris Lee
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: John Elderkin

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.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Face-book to help us spread the word about the *Succinctly* series!

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.

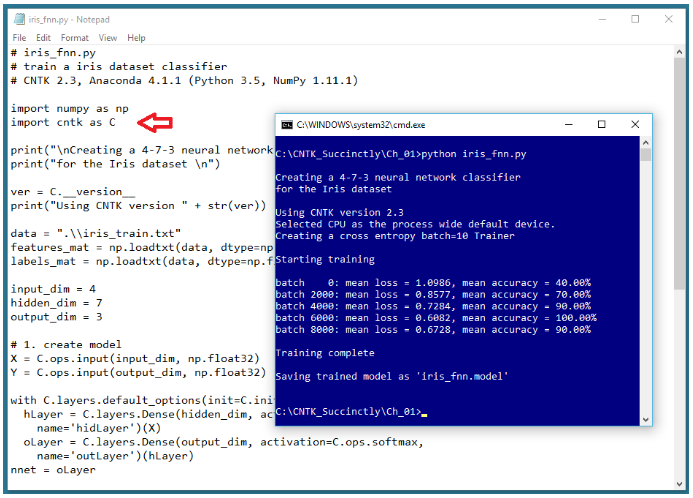
Chapter 1 Getting Started

Microsoft CNTK (Cognitive Toolkit, formerly Computational Network Toolkit) is an open source code framework that enables you to create deep learning systems, such as feed-forward neural network time series prediction systems and convolutional neural network image classifiers. Version 1.0 of CNTK was released in 2016. Version 2.0 was released in June 2017, and is a significant rewrite of the 1.0 version.

This e-book is based on CNTK version 2.3, released in late 2017. Because CNTK is still under active development, by the time you read this e-book, the latest version will likely be different. However, any changes will likely be relatively minor and consist mainly of additional functionality. In other words, the code presented here should work with any CNTK 2.x version.

The CNTK framework functions are written in C++ for optimal performance. Although you can call CNTK functions using a C++ program, the most common approach is to call CNTK functions using a Python program. CNTK version 1 used a proprietary scripting language, BrainScript, which is still supported by version 2.

CNTK v2 runs on both Windows (8.1, 10, Server 2012 R2 and later) and Linux systems, but not directly on Mac systems. The screenshot in Figure 1-1 shows a simplified CNTK session. Notice that the program is just an ordinary Python script, **iris_fnn.py**, that references CNTK as a Python package, and CNTK-Python programs run in an ordinary shell.



```
# iris_fnn.py
# train a iris dataset classifier
# CNTK 2.3, Anaconda 4.1.1 (Python 3.5, NumPy 1.11.1)

import numpy as np
import cntk as C

print("\nCreating a 4-7-3 neural network
print("for the Iris dataset \n")

ver = C.__version__
print("Using CNTK version " + str(ver))

data = ".\\iris_train.txt"
features_mat = np.loadtxt(data, dtype=np
labels_mat = np.loadtxt(data, dtype=np.f

input_dim = 4
hidden_dim = 7
output_dim = 3

# 1. create model
X = C.ops.input(input_dim, np.float32)
Y = C.ops.input(output_dim, np.float32)

with C.layers.default_options(init=C.ini
  hLayer = C.layers.Dense(hidden_dim, ac
    name='hidLayer')(X)
  oLayer = C.layers.Dense(output_dim, activation=C.ops.softmax,
    name='outLayer')(hLayer)
nnet = oLayer
```

```
C:\CNTK_Succinctly\Ch_01>python iris_fnn.py

Creating a 4-7-3 neural network classifier
for the Iris dataset

Using CNTK version 2.3
Selected CPU as the process wide default device.
Creating a cross entropy batch=10 Trainer

Starting training

batch    0: mean loss = 1.0986, mean accuracy = 40.00%
batch 2000: mean loss = 0.8577, mean accuracy = 70.00%
batch 4000: mean loss = 0.7284, mean accuracy = 90.00%
batch 6000: mean loss = 0.6082, mean accuracy = 100.00%
batch 8000: mean loss = 0.6728, mean accuracy = 90.00%

Training complete

Saving trained model as 'iris_fnn.model'


C:\CNTK_Succinctly\Ch_01>
```

Figure 1-1: Example CNTK Session

There are several other deep learning frameworks. Microsoft CNTK is most similar to Google TensorFlow. In my opinion, CNTK is easier to program with than TensorFlow; however, all deep-learning frameworks have a fairly steep learning curve. Both CNTK and TensorFlow can be accessed by the Keras wrapper framework, a topic that is outside the scope of this e-book.

This e-book assumes you have intermediate (or better) programming skills with a C-family language, but doesn't assume you know anything about CNTK. Enough chitchat already—let's get started.

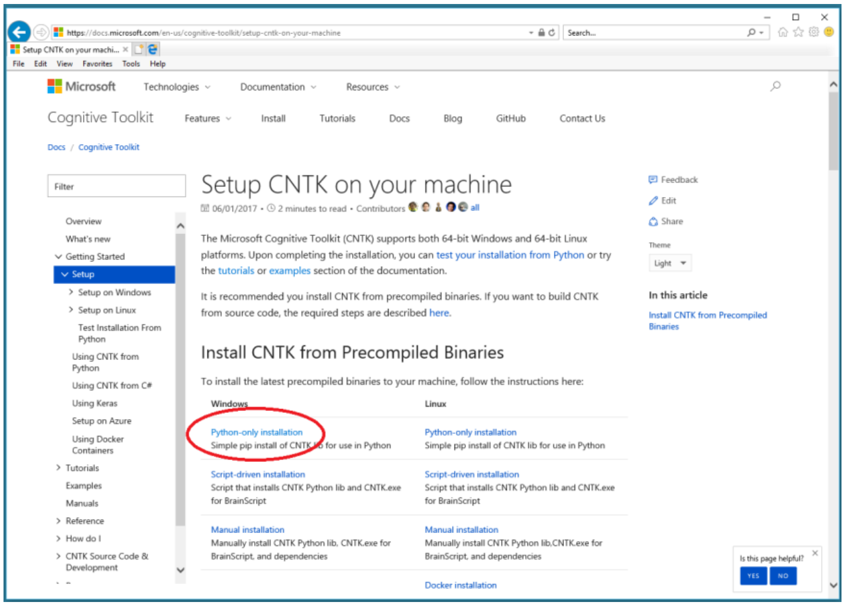
Installing CNTK

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 by making changes. So if you want to learn CNTK, the first step is to install it.

Installing CNTK is a bit different from many installs. You don't install CNTK directly. Instead, you install CNTK as an add-on package for Python. You first install a Python distribution, which contains the base Python language interpreter plus several additional packages that are required by CNTK, in particular the NumPy and SciPy packages. Then, you install CNTK as an extra package.

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

The first step is to determine which version of CNTK you want to use. In general, you'll want to install the most recent version of CNTK. However, this e-book is based on CNTK version 2.3, so you may want to install v2.3 instead of the latest version. There are several different kinds of CNTK installations to choose from. You'll find links to several installation types here, as shown in Figure 1-2.



Setup CNTK on your machine

06/01/2017 • 2 minutes to read • Contributors all

The Microsoft Cognitive Toolkit (CNTK) supports both 64-bit Windows and 64-bit Linux platforms. Upon completing the installation, you can test your installation from Python or try the tutorials or examples section of the documentation.

It is recommended you install CNTK from precompiled binaries. If you want to build CNTK from source code, the required steps are described here.

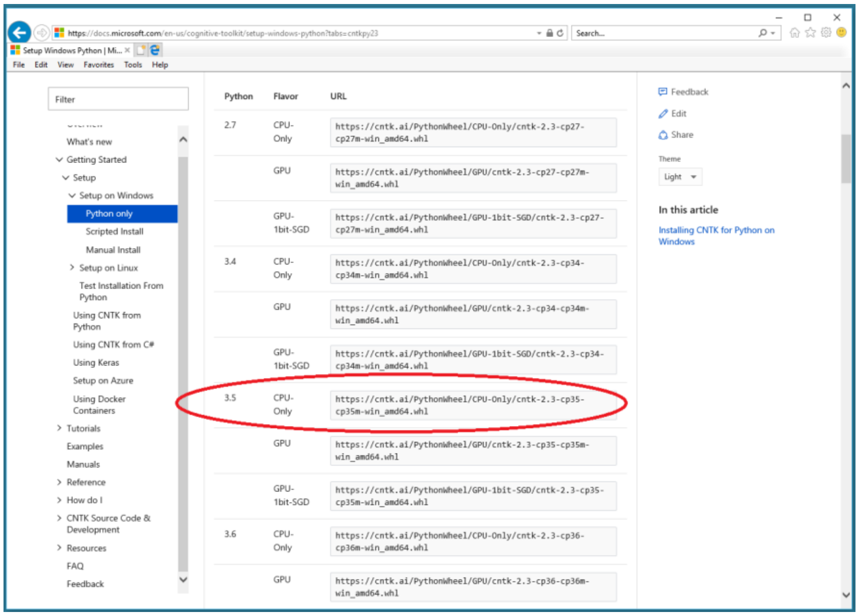
Install CNTK from Precompiled Binaries

To install the latest precompiled binaries to your machine, follow the instructions here:

Windows	Linux
Python-only installation Simple pip install of CNTK lib for use in Python	Python-only installation Simple pip install of CNTK lib for use in Python
Script-driven installation Script that installs CNTK Python lib and CNTK.exe for BrainScript	Script-driven installation Script that installs CNTK Python lib and CNTK.exe for BrainScript
Manual installation Manually install CNTK Python lib, CNTK.exe for BrainScript, and dependencies	Manual installation Manually install CNTK Python lib,CNTK.exe for BrainScript, and dependencies
	Docker installation

Figure 1-2: Select the Python-Only Link

I recommend that you select the **Python-only installation** option. After you click that link, you will be redirected to a page that lists the currently supported versions of Python, as shown in Figure 1-3.



Python	Flavor	URL
2.7	CPU-Only	https://cntk.ai/PythonWheel/CPU-Only/cntk-2.3-cp27-cp27m-win_amd64.whl
	GPU	https://cntk.ai/PythonWheel/GPU/cntk-2.3-cp27-cp27m-win_amd64.whl
	GPU-1bit-SGD	https://cntk.ai/PythonWheel/GPU-1bit-SGD/cntk-2.3-cp27-cp27m-win_amd64.whl
3.4	CPU-Only	https://cntk.ai/PythonWheel/CPU-Only/cntk-2.3-cp34-cp34m-win_amd64.whl
	GPU	https://cntk.ai/PythonWheel/GPU/cntk-2.3-cp34-cp34m-win_amd64.whl
	GPU-1bit-SGD	https://cntk.ai/PythonWheel/GPU-1bit-SGD/cntk-2.3-cp34-cp34m-win_amd64.whl
3.5	CPU-Only	https://cntk.ai/PythonWheel/CPU-Only/cntk-2.3-cp35-cp35m-win_amd64.whl
	GPU	https://cntk.ai/PythonWheel/GPU/cntk-2.3-cp35-cp35m-win_amd64.whl
	GPU-1bit-SGD	https://cntk.ai/PythonWheel/GPU-1bit-SGD/cntk-2.3-cp35-cp35m-win_amd64.whl
3.6	CPU-Only	https://cntk.ai/PythonWheel/CPU-Only/cntk-2.3-cp36-cp36m-win_amd64.whl
	GPU	https://cntk.ai/PythonWheel/GPU/cntk-2.3-cp36-cp36m-win_amd64.whl

Figure 1-3: Supported Versions of Python

Notice that each version of Python has three different flavors of CNTK: CPU-Only, GPU, and GPU-1bit-SGD. Each flavor has an associated .whl file (pronounced “wheel”). If you are new to Python, you can think of a Python .whl file as somewhat similar to a Windows .msi installer file. You might want to copy the .whl URI string, because you’ll need it after installing Python.

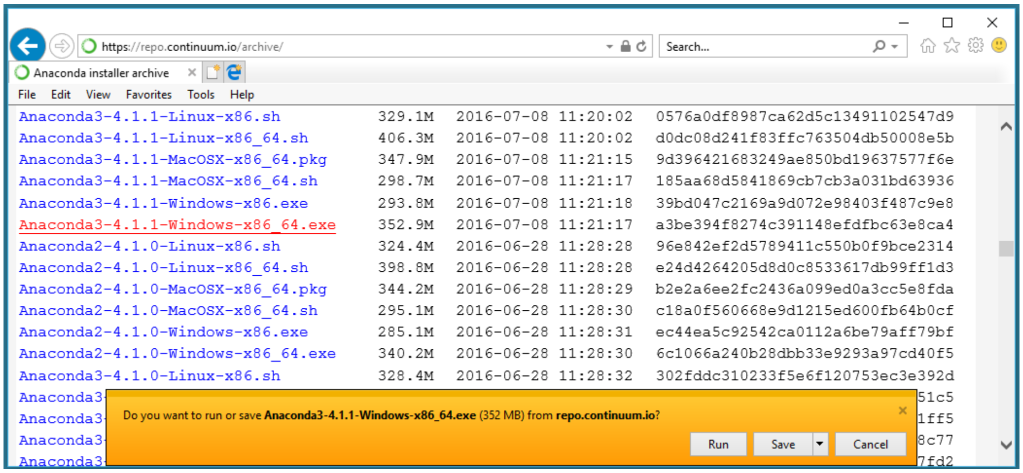
CNTK supports both CPU processing and GPU processing. If your machine does not have a GPU installed, then you should use the CPU-only version. At this point, your main decision is choosing the version of Python you’ll be using. In Figure 1-3 the choices are 2.7, 3.4, 3.5, and 3.6.

Before proceeding, you should check your machine to determine if you already have an existing Python installation. The simplest scenario is if your machine doesn’t have an existing Python installed, then 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 Python versioning issues at some point.

At the time of writing this e-book, Python version 3.5 was very stable, so I decided to use it rather than the newer version, 3.6. Once you decide on your Python version, you need to find the archived Anaconda distribution that has that version of Python.

Do *not* install the most recent version of Anaconda. If you scroll down the Setup information page, you’ll find which Anaconda version to use. For Python 3.5, you need Anacoinda3 4.1.1. Either do an internet search for “archive Anaconda install,” or go directly here.

On that page you’ll see many different Anaconda distributions. Be careful here; even though I’ve installed and uninstalled Anaconda for CNTK dozens of times, I’ve botched selecting the correct version of Anaconda several times.


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

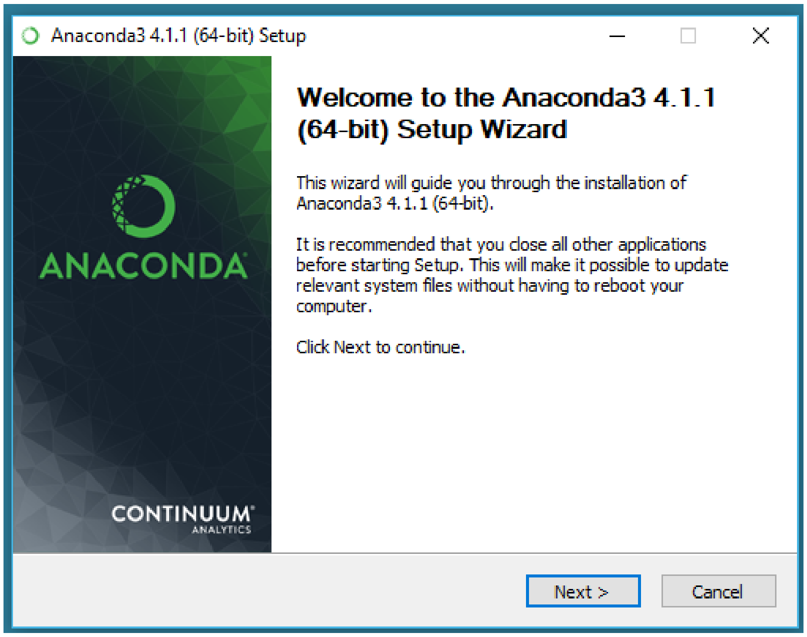
```
Anaconda3-4.1.1-Linux-x86.sh           329.1M  2016-07-08 11:20:02  0576a0df8987ca62d5c13491102547d9
Anaconda3-4.1.1-Linux-x86_64.sh        406.3M  2016-07-08 11:20:02  d0dc08d241f83ffc763504db50008e5b
Anaconda3-4.1.1-MacOSX-x86_64.pkg      347.9M  2016-07-08 11:21:15  9d396421683249ae850bd19637577f6e
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 1-4: 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 click the **Anaconda3-4.1.1-Windows-x86_64.exe** link. This will launch a self-extracting installation program. You can click **Run**.

To recap, at this point you’ve determined which version of CNTK you want to use (2.3 CPU-only in this example), and then determined which version of Anaconda to use (Anaconda3 4.1.1 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).

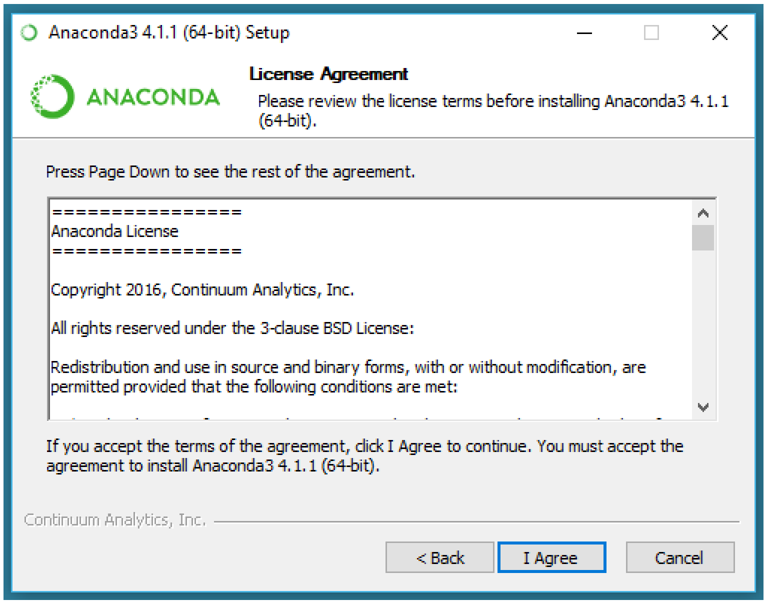
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Figure 1-5: Anaconda Installation Welcome

A few seconds after you click **Run**, the Anaconda installation Welcome window will appear, as shown in Figure 1-5. Click **Next**.

You will see the Anaconda License Agreement window, as shown in Figure 1-6. 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.

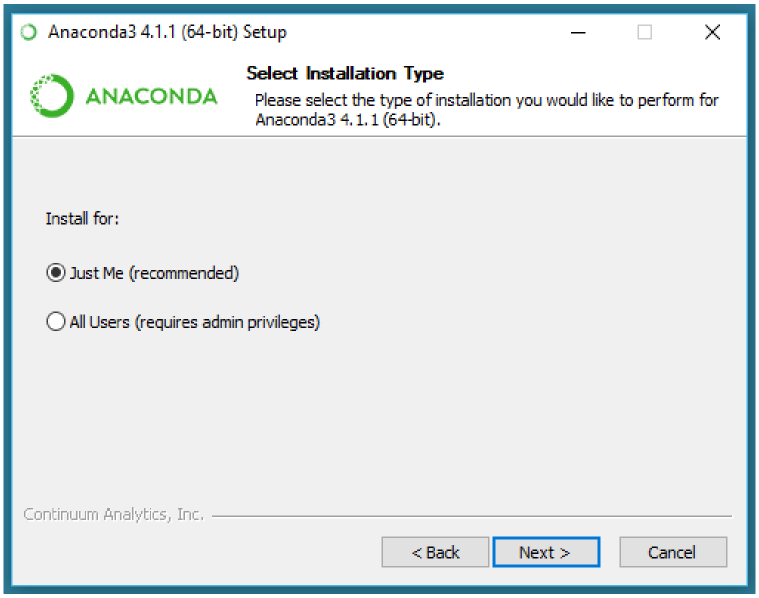
```
================
Anaconda License
================

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 1-6: Anaconda License Agreement

You will see the Select Installation Type window, as shown in Figure 1-7. 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:

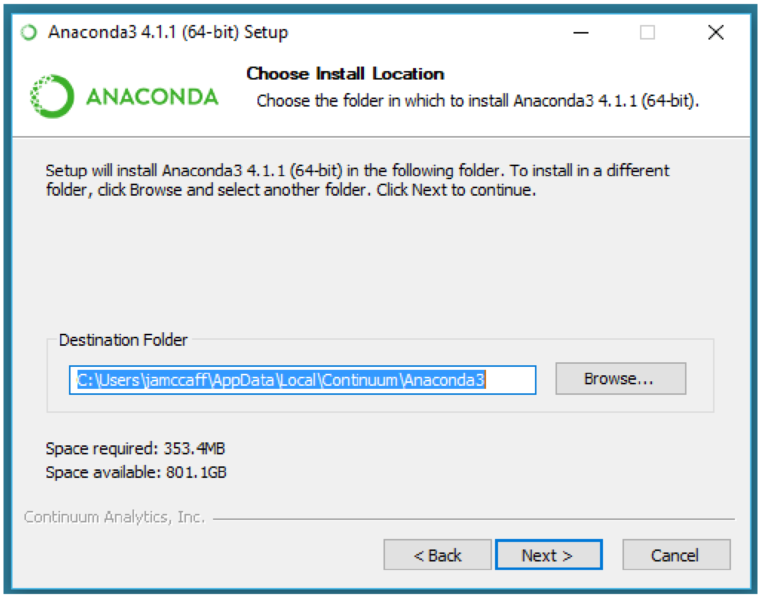
● Just Me (recommended)

○ All Users (requires admin privileges)

Continuum Analytics, Inc.

Figure 1-7: 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 non-CNTK packages may assume this location. Click **Next**.



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

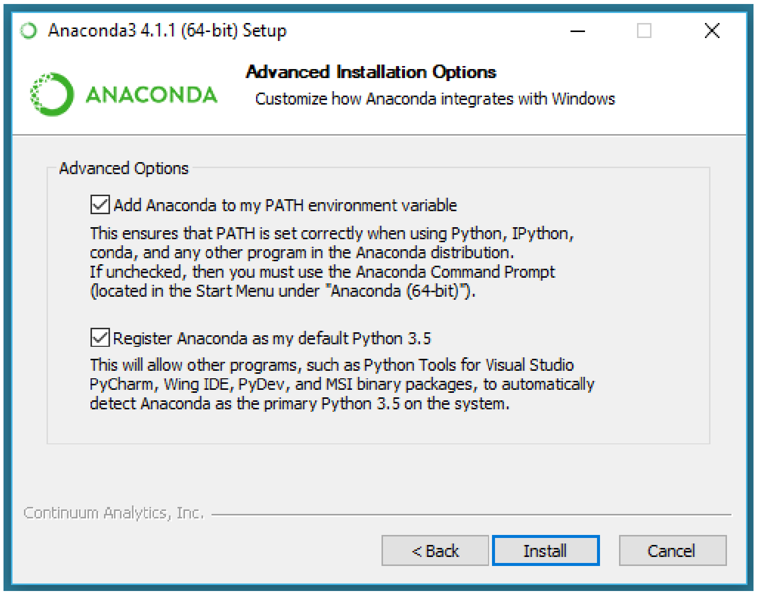
C:\Users\jamccaff\AppData\Local\Continuum\Anaconda3

Space required: 353.4MB
Space available: 801.1GB

Continuum Analytics, Inc.

Figure 1-8: Default Anaconda Python Installation Location

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**.



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)").

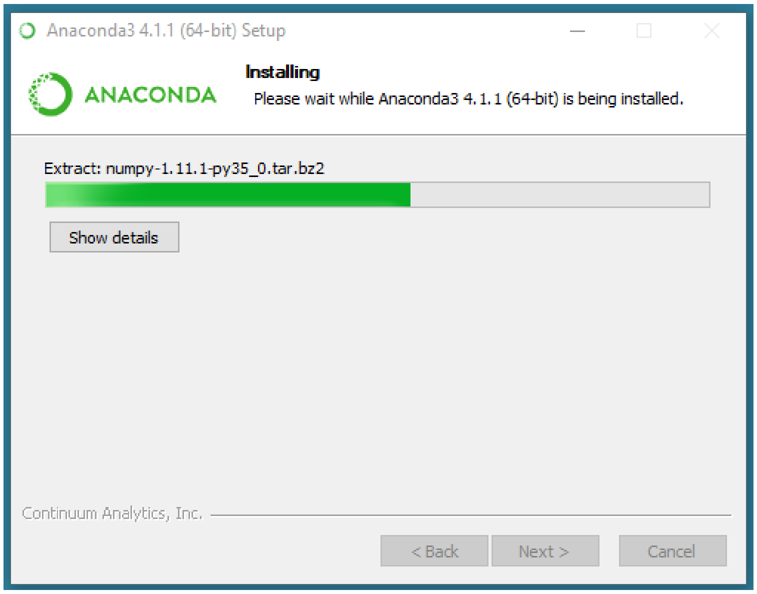
☑ Register Anaconda as my default Python 3.5

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 1-9: 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 the installation. However, you may want to observe the progress bar and see which packages are installed, such as NumPy, shown in Figure 1-10.



Anaconda3 4.1.1 (64-bit) Setup

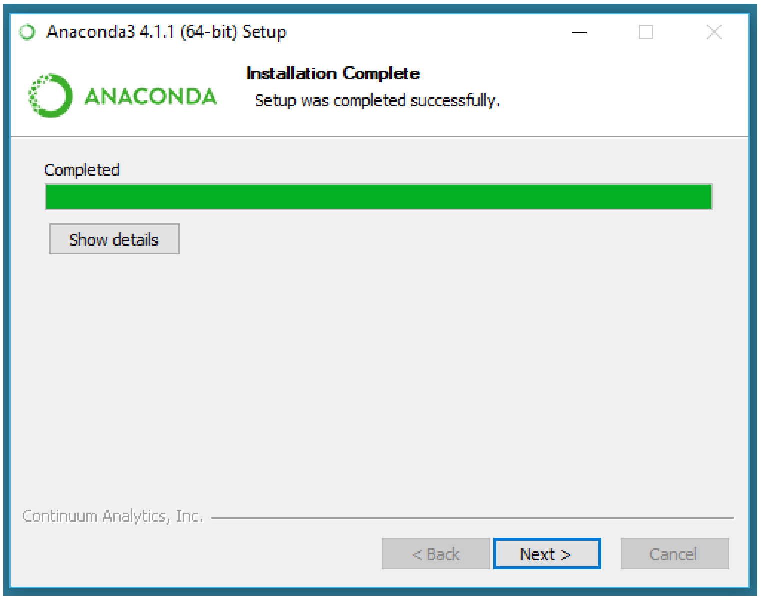
Installing

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

Extract: numpy-1.11.1-py35_0.tar.bz2

Figure 1-10: Anaconda Installation Progress

When installation completes successfully, you'll see an Installation Complete window. Click **Next**, as shown in Figure 1-11.



Anaconda3 4.1.1 (64-bit) Setup

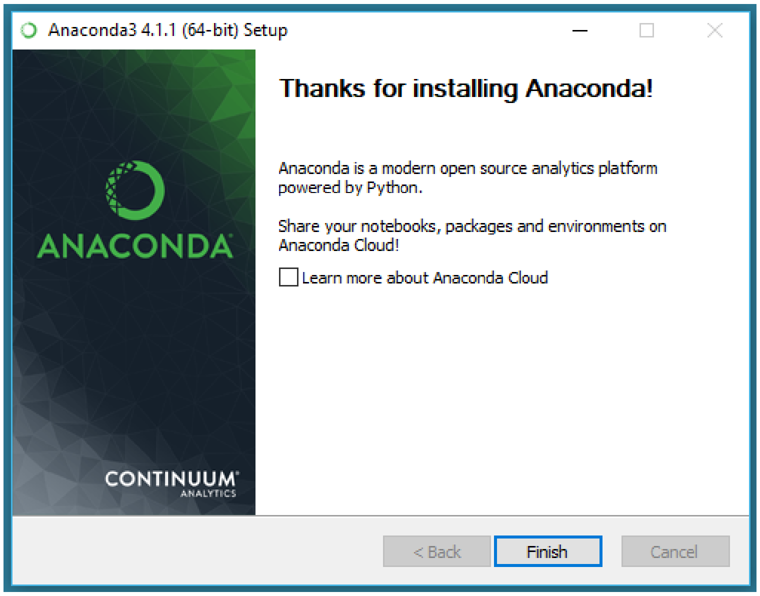
Installation Complete

Setup was completed successfully.

Completed

Figure 1-11: 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 1-12, 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!

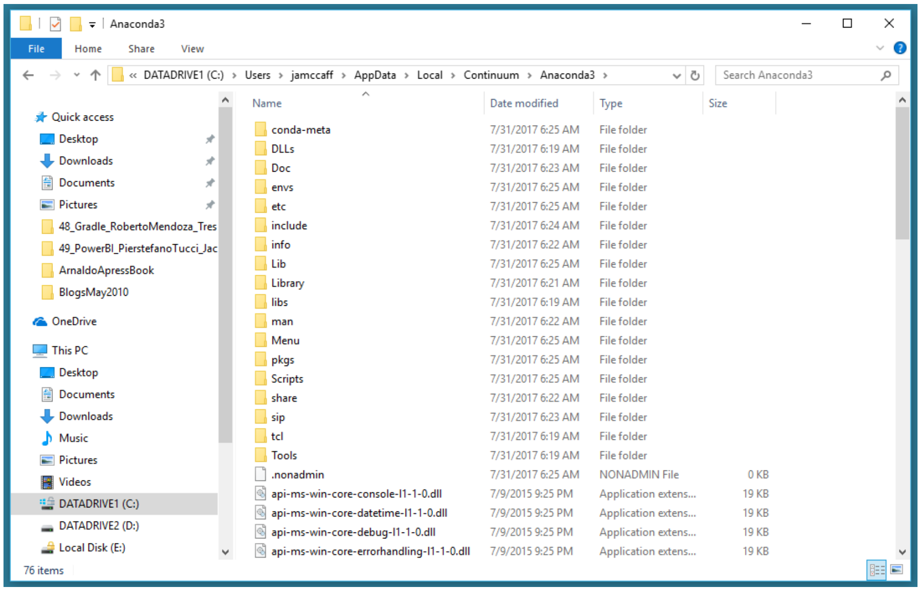
Anaconda is a modern open source analytics platform powered by Python.

Share your notebooks, packages and environments on Anaconda Cloud!

□ Learn more about Anaconda Cloud

Figure 1-12: Final Anaconda Installation Window

After the Anaconda installation is complete, you may want to look at the installation file and directory structure, as shown in Figure 1-13. Notice there are directories named **Lib**, **Library**, and **libs**. Just below the files shown in Figure 1-13 is the python.exe main execution engine.

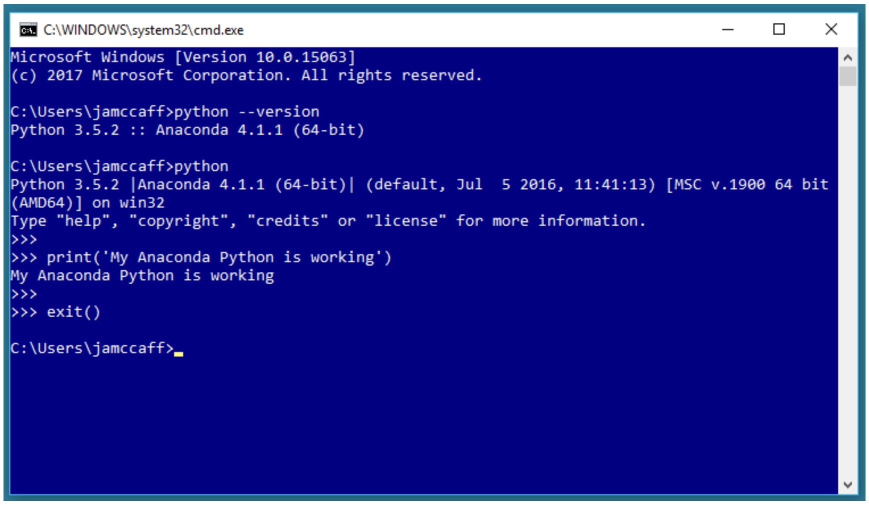


« DATADRIVE1 (C:) › Users › jamccaff › AppData › Local › Continuum › Anaconda3 ›

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 1-13: The Anaconda Installation Location

Before installing the CNTK add-on package, 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 1-14. You might want to test the Python interpreter by issuing the command **`python`** followed by a **`print('hello')`** statement.

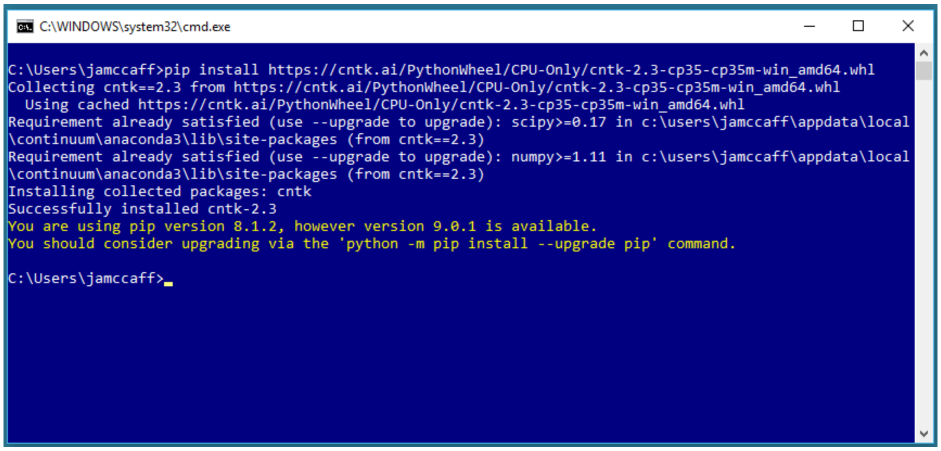


```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

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 1-14: Verifying the Anaconda Python Installation

After installing Anaconda, installing the CNTK package for Python is remarkably easy. From a command shell, enter **`pip install`** ***`uri-to-whl-file`***, where the URI is the .whl file associated with your Python install, as shown in Figure 1-3. For my installation, that URI is **https://cntk.ai/PythonWheel/CPU-Only/cntk-2.3-cp35-cp35m-win_amd64.whl**. Notice the reference to CNTK 2.3 in the URI. The "35" in the URI indicates Python 3.5.

Note that if your machine does not have a GPU, you can install the GPU version of CNTK anyway, because the GPU version has CPU code, too. However, I recommend CPU-only.

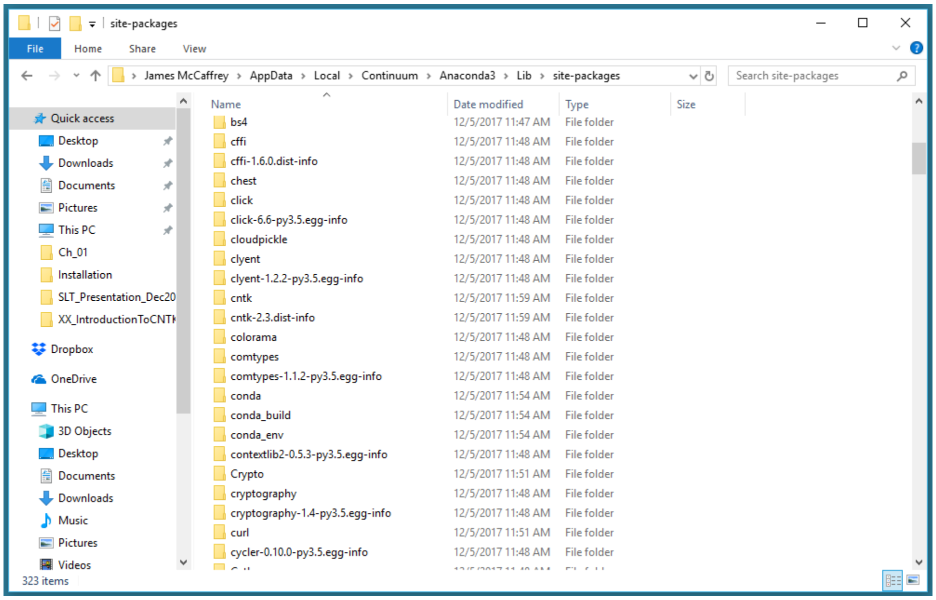


```
C:\Users\jamccaff>pip install https://cntk.ai/PythonWheel/CPU-Only/cntk-2.3-cp35-cp35m-win_amd64.whl
Collecting cntk==2.3 from https://cntk.ai/PythonWheel/CPU-Only/cntk-2.3-cp35-cp35m-win_amd64.whl
  Using cached https://cntk.ai/PythonWheel/CPU-Only/cntk-2.3-cp35-cp35m-win_amd64.whl
Requirement already satisfied (use --upgrade to upgrade): scipy>=0.17 in c:\users\jamccaff\appdata\local
\continuum\anaconda3\lib\site-packages (from cntk==2.3)
Requirement already satisfied (use --upgrade to upgrade): numpy>=1.11 in c:\users\jamccaff\appdata\local
\continuum\anaconda3\lib\site-packages (from cntk==2.3)
Installing collected packages: cntk
Successfully installed cntk-2.3
You are using pip version 8.1.2, however version 9.0.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

C:\Users\jamccaff>_
```

Figure 1-15: Using pip to Install the CNTK Package

CNTK will be installed very quickly, and you should see a success message, as shown in Figure 1-15.

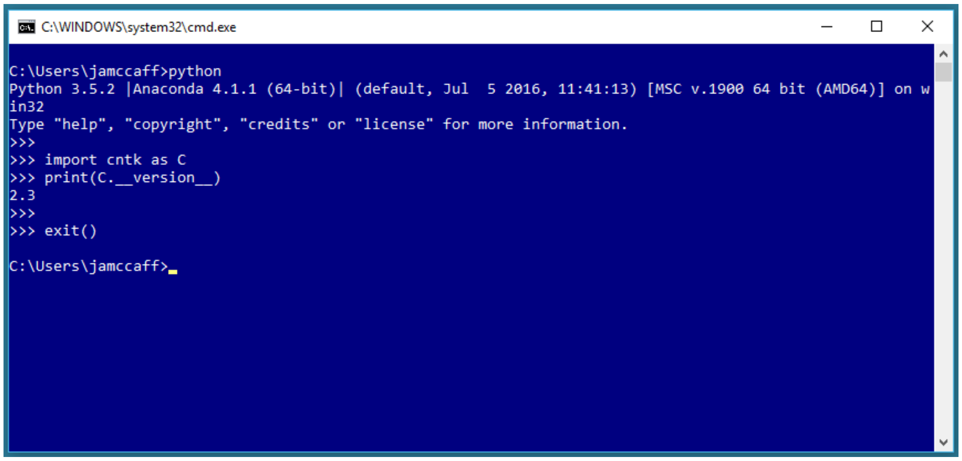
After CNTK is installed as a Python package, you might want to examine its file and directory structure, at **C:\Users\<user>\AppData\Local\Continuum\Anaconda3\Lib\site-packages\cntk**, as shown in Figure 1-16.



Name	Date modified	Type	Size
bs4	12/5/2017 11:47 AM	File folder	
cffi	12/5/2017 11:48 AM	File folder	
cffi-1.6.0.dist-info	12/5/2017 11:48 AM	File folder	
chest	12/5/2017 11:48 AM	File folder	
click	12/5/2017 11:48 AM	File folder	
click-6.6-py3.5.egg-info	12/5/2017 11:48 AM	File folder	
cloudpickle	12/5/2017 11:48 AM	File folder	
clyent	12/5/2017 11:48 AM	File folder	
clyent-1.2.2-py3.5.egg-info	12/5/2017 11:48 AM	File folder	
cntk	12/5/2017 11:59 AM	File folder	
cntk-2.3.dist-info	12/5/2017 11:59 AM	File folder	
colorama	12/5/2017 11:48 AM	File folder	
comtypes	12/5/2017 11:48 AM	File folder	
comtypes-1.1.2-py3.5.egg-info	12/5/2017 11:48 AM	File folder	
conda	12/5/2017 11:54 AM	File folder	
conda_build	12/5/2017 11:54 AM	File folder	
conda_env	12/5/2017 11:54 AM	File folder	
contextlib2-0.5.3-py3.5.egg-info	12/5/2017 11:48 AM	File folder	
Crypto	12/5/2017 11:51 AM	File folder	
cryptography	12/5/2017 11:48 AM	File folder	
cryptography-1.4-py3.5.egg-info	12/5/2017 11:48 AM	File folder	
curl	12/5/2017 11:51 AM	File folder	
cycler-0.10.0-py3.5.egg-info	12/5/2017 11:48 AM	File folder	

Figure 1-16: The CNTK Package Location

You should verify that CNTK has been installed correctly. From a command shell, start the Python interpreter by entering **`python`**. Then enter **`import cntk as C`** followed by **`print(C.__version__)`**, as shown in Figure 1-17. (Note the two underscores before and after **`version`** in the command).



```
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 w
in32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> import cntk as C
>>> print(C.__version__)
2.3
>>>
>>> exit()

C:\Users\jamccaff>_
```

Figure 1-17: Verifying the CNTK Installation

If the interpreter responds with your CNTK version, you can be certain that CNTK is installed properly on your machine, and you’re ready to start writing programs with the CNTK framework.

Editing and running CNTK programs

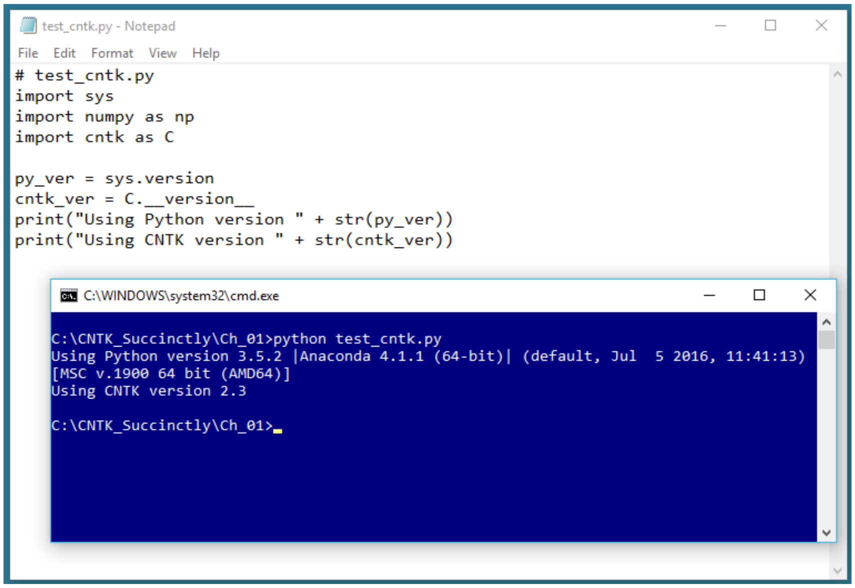
Because a CNTK program is a specialized Python program, you can use any Python editing environment. If you are relatively new to Python, selecting a Python editor or IDE (integrated development environment) can be a confusing task because there are dozens of editors and Python IDEs to choose from.

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 CNTK Version

```
# test_cntk.py
import sys
import numpy as np
import cntk as C

py_ver = sys.version
cntk_ver = C.__version__
print("Using Python version " + str(py_ver))
print("Using CNTK version " + str(cntk_ver))
```

For example, launch the Notepad text editor (or any other editor), and copy and paste the code from Code Listing 1-1. Save the file as **test_cntk.py** (being careful not to add an additional .txt extension) in a convenient directory. Open a command shell, navigate to the directory that holds your Python file, and execute by entering **`python test_cntk.py`**, as shown in Figure 1-18.

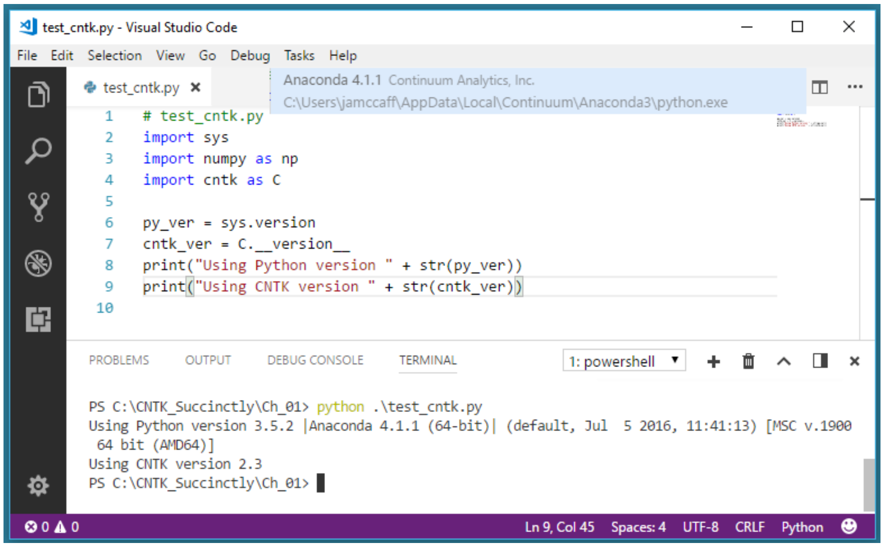


```
# test_cntk.py
import sys
import numpy as np
import cntk as C

py_ver = sys.version
cntk_ver = C.__version__
print("Using Python version " + str(py_ver))
print("Using CNTK version " + str(cntk_ver))
```

```
C:\CNTK_Succinctly\Ch_01>python test_cntk.py
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 CNTK version 2.3

C:\CNTK_Succinctly\Ch_01>
```

Figure 1-18: 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 then adding Python support is just a matter of a couple of clicks. You can download VS Code here.


test_cntk.py - Visual Studio Code
File Edit Selection View Go Debug Tasks Help
test_cntk.py
Anaconda 4.1.1 Continuum Analytics, Inc.
C:\Users\jamccaff\AppData\Local\Continuum\Anaconda3\python.exe
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: powershell
0 0
Ln 9, Col 45 Spaces: 4 UTF-8 CRLF Python

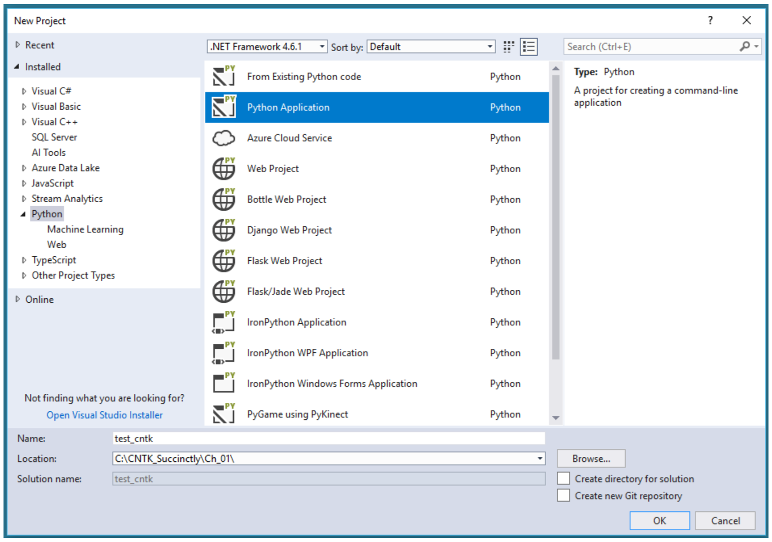
```
# test_cntk.py
import sys
import numpy as np
import cntk as C

py_ver = sys.version
cntk_ver = C.__version__
print("Using Python version " + str(py_ver))
print("Using CNTK version " + str(cntk_ver))
```

```
PS C:\CNTK_Succinctly\Ch_01> python .\test_cntk.py
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 CNTK version 2.3
PS C:\CNTK_Succinctly\Ch_01>
```

Figure 1-19: Using Visual Studio Code

Figure 1-19 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 the tutorial here to learn more.

Yet another option for editing and running CNTK 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 1-20 and 1-21.



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_cntk

Location: C:\CNTK_Succinctly\Ch_01\

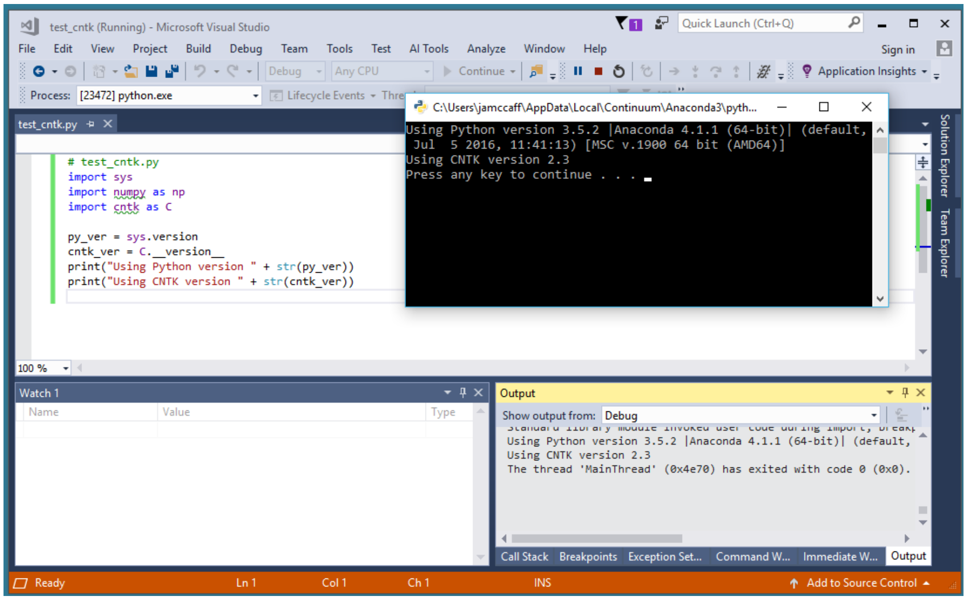
Solution name: test_cntk

☐ Create directory for solution

☐ Create new Git repository

Figure 1-20: 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 its very steep learning curve.



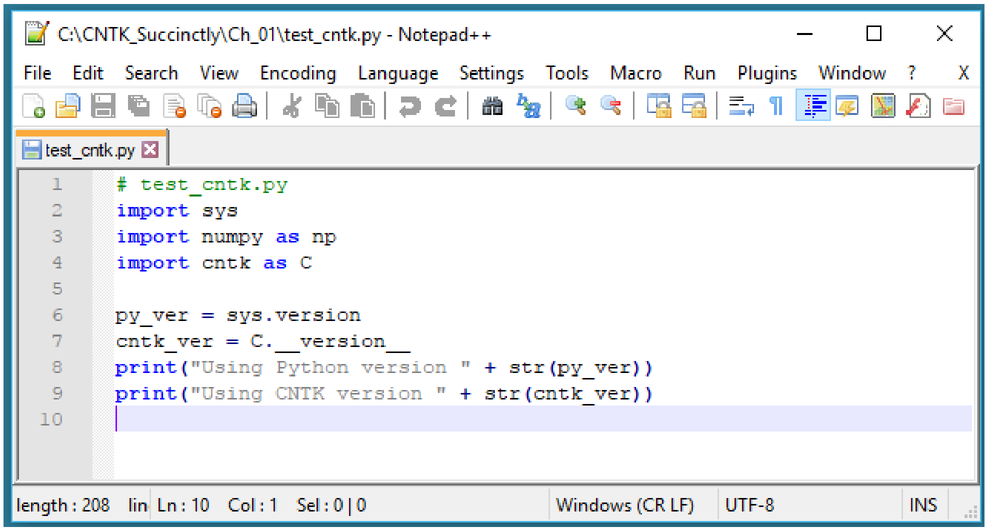
```python
# test_cntk.py
import sys
import numpy as np
import cntk as C

py_ver = sys.version
cntk_ver = C.__version__
print("Using Python version " + str(py_ver))
print("Using CNTK version " + str(cntk_ver))
```

```
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 CNTK version 2.3
Press any key to continue . . .
```

```
Standard library module invoked user code during import, breakp
Using Python version 3.5.2 |Anaconda 4.1.1 (64-bit)| (default,
Using CNTK version 2.3
The thread 'MainThread' (0x4e70) has exited with code 0 (0x0).
```

Figure 1-21: 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.

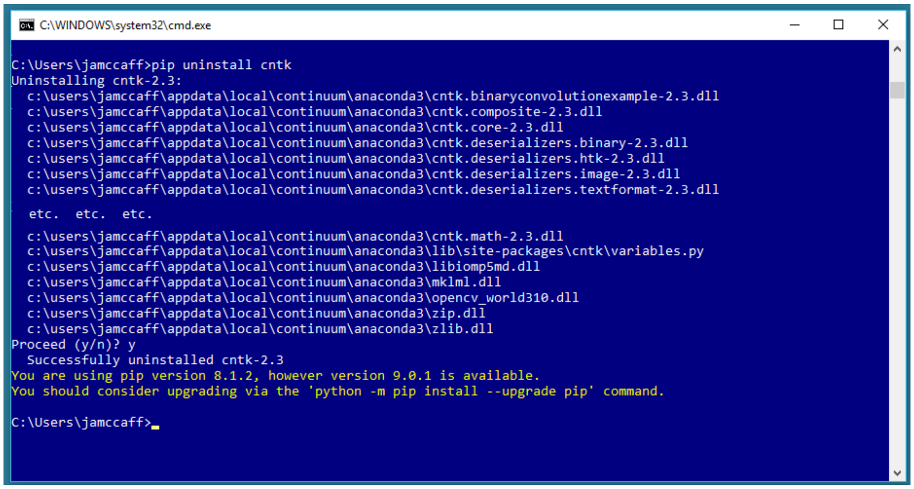

C:\CNTK_Succinctly\Ch_01\test_cntk.py - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
test_cntk.py
length : 208 lin Ln : 10 Col : 1 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS

```python
# test_cntk.py
import sys
import numpy as np
import cntk as C

py_ver = sys.version
cntk_ver = C.__version__
print("Using Python version " + str(py_ver))
print("Using CNTK version " + str(cntk_ver))

```

Figure 1-22: Using the Notepad++ Editor

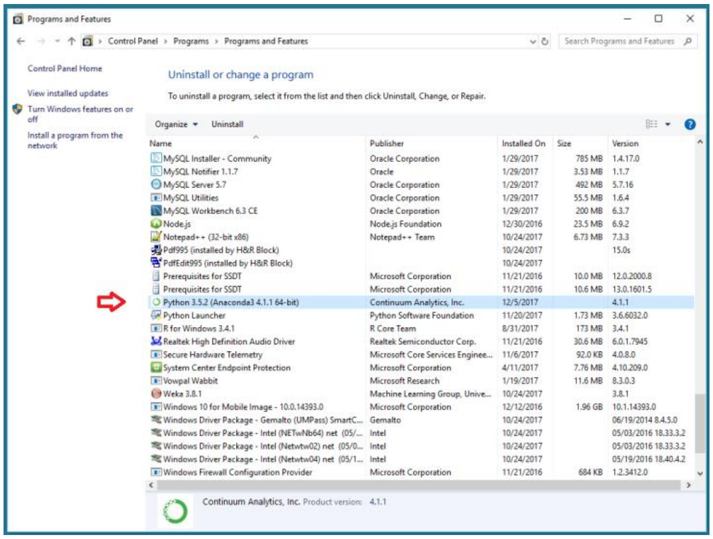
Uninstalling CNTK

CNTK and Anaconda Python have quick, easy, and reliable uninstall procedures. To uninstall CNTK, all you have to do is launch a command shell and issue the command **`pip uninstall cntk.`** It's that easy. The CNTK package will be removed from your Python system, as shown in Figure 1-23 (slightly edited for size). To uninstall Python, on Windows you can use the **Programs and Features** section of the Control Panel, as shown in Figure 1-24.



```
C:\Users\jamccaff>pip uninstall cntk
Uninstalling cntk-2.3:
  c:\users\jamccaff\appdata\local\continuum\anaconda3\cntk.binaryconvolutionexample-2.3.dll
  c:\users\jamccaff\appdata\local\continuum\anaconda3\cntk.composite-2.3.dll
  c:\users\jamccaff\appdata\local\continuum\anaconda3\cntk.core-2.3.dll
  c:\users\jamccaff\appdata\local\continuum\anaconda3\cntk.deserializers.binary-2.3.dll
  c:\users\jamccaff\appdata\local\continuum\anaconda3\cntk.deserializers.htk-2.3.dll
  c:\users\jamccaff\appdata\local\continuum\anaconda3\cntk.deserializers.image-2.3.dll
  c:\users\jamccaff\appdata\local\continuum\anaconda3\cntk.deserializers.textformat-2.3.dll

  etc.  etc.  etc.

  c:\users\jamccaff\appdata\local\continuum\anaconda3\cntk.math-2.3.dll
  c:\users\jamccaff\appdata\local\continuum\anaconda3\lib\site-packages\cntk\variables.py
  c:\users\jamccaff\appdata\local\continuum\anaconda3\libiomp5md.dll
  c:\users\jamccaff\appdata\local\continuum\anaconda3\mklml.dll
  c:\users\jamccaff\appdata\local\continuum\anaconda3\opencv_world310.dll
  c:\users\jamccaff\appdata\local\continuum\anaconda3\zip.dll
  c:\users\jamccaff\appdata\local\continuum\anaconda3\zlib.dll
Proceed (y/n)? y
  Successfully uninstalled cntk-2.3
You are using pip version 8.1.2, however version 9.0.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.

C:\Users\jamccaff>_
```

Figure 1-23: Uninstalling CNTK


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

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

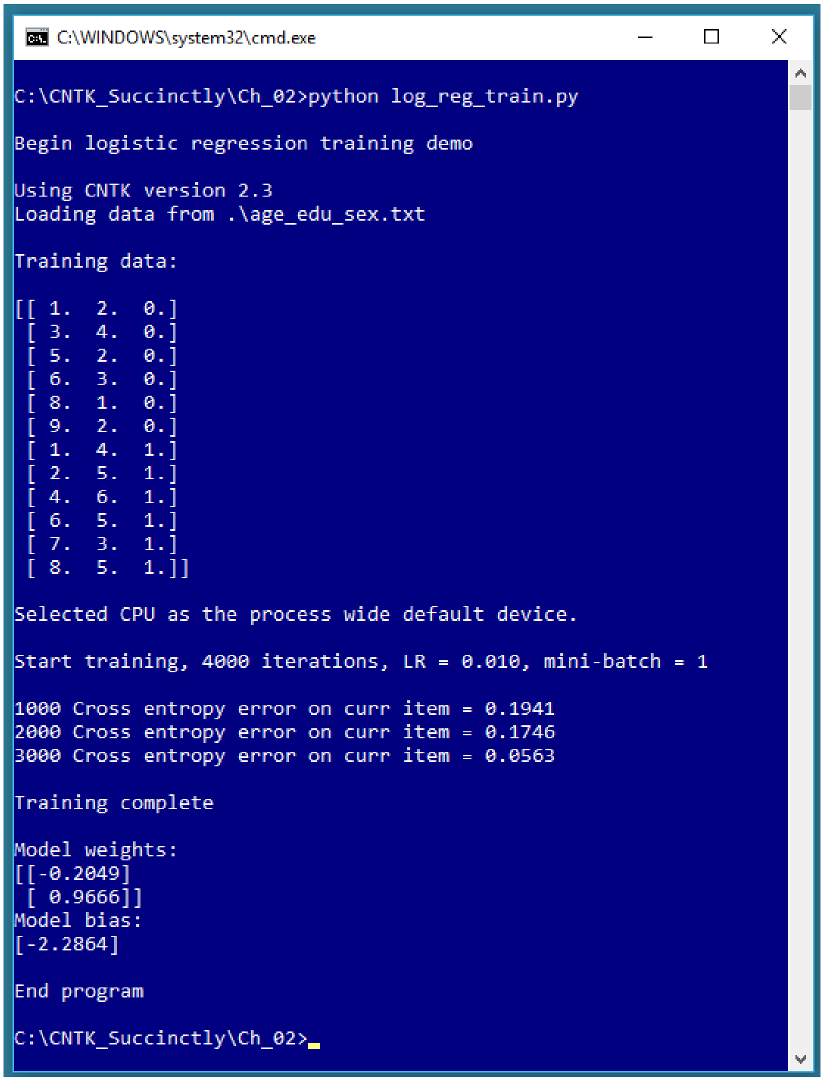
Continuum Analytics, Inc. Product version: 4.1.1

Figure 1-24: Uninstalling Anaconda Python

If you become a regular user of CNTK, you'll eventually 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 Logistic Regression

Logistic regression is one of the simplest machine-learning techniques. Logistic regression is a technique for a binary classification problem: to create a prediction model in situations where the value of the variable to predict (often called the *label* in machine learning terminology) can be one of just two categorical values. For example, you might want to predict the sex of a person (male or female) based on the person's age, years of education, and annual income.

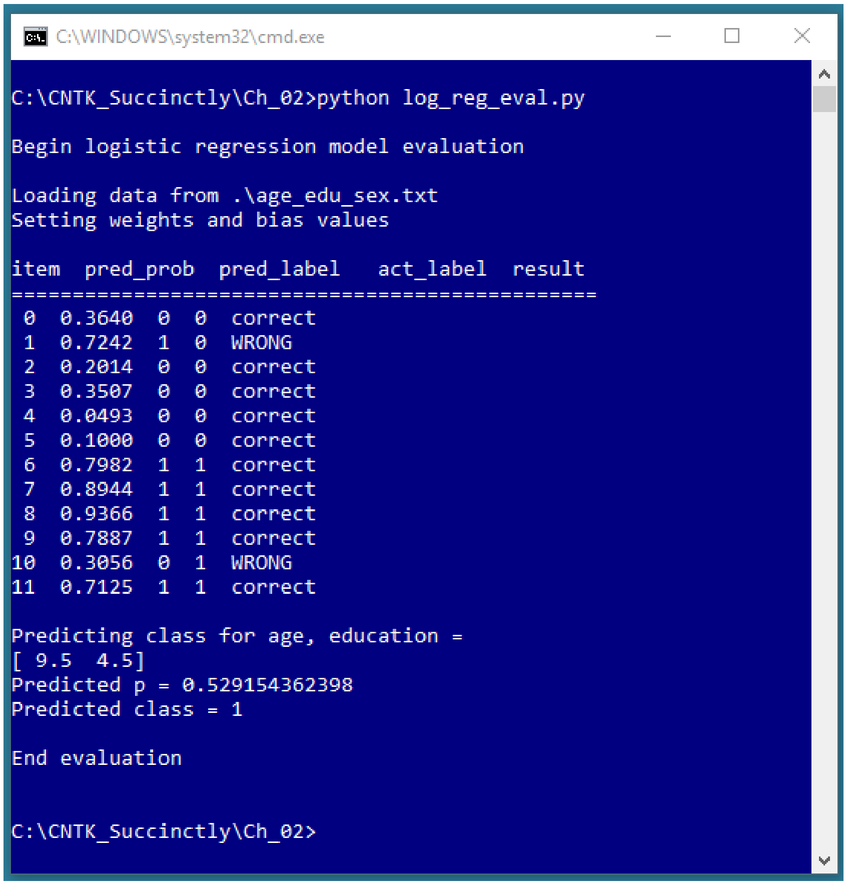


```
C:\CNTK_Succinctly\Ch_02>python log_reg_train.py

Begin logistic regression training demo

Using CNTK version 2.3
Loading data from .\age_edu_sex.txt

Training data:

[[ 1.  2.  0.]
 [ 3.  4.  0.]
 [ 5.  2.  0.]
 [ 6.  3.  0.]
 [ 8.  1.  0.]
 [ 9.  2.  0.]
 [ 1.  4.  1.]
 [ 2.  5.  1.]
 [ 4.  6.  1.]
 [ 6.  5.  1.]
 [ 7.  3.  1.]
 [ 8.  5.  1.]]

Selected CPU as the process wide default device.

Start training, 4000 iterations, LR = 0.010, mini-batch = 1

1000 Cross entropy error on curr item = 0.1941
2000 Cross entropy error on curr item = 0.1746
3000 Cross entropy error on curr item = 0.0563

Training complete

Model weights:
[[-0.2049]
 [ 0.9666]]
Model bias:
[-2.2864]

End program

C:\CNTK_Succinctly\Ch_02>
```

Figure 2-1: Logistic Regression Using CNTK

The screenshot in Figure 2-1 shows a simplified example of logistic regression. The demo program begins by loading 12 data items into memory. Each item represents a person's age, education, and sex. For example, the first data item is (1.0, 2.0, 0), which is a person with age = 1.0, education = 2.0, and sex = female.

Age and education are the predictor variables (often called features in CNTK and machine learning terminology). The values of the feature variables have been normalized in some way so that their magnitudes are all between 1.0 and 9.0. The variable-to-predict (often called the class or class label) is encoded with female = 0 and male = 1.



```
C:\CNTK_Succinctly\Ch_02>python log_reg_eval.py

Begin logistic regression model evaluation

Loading data from .\age_edu_sex.txt
Setting weights and bias values

item  pred_prob  pred_label   act_label  result
================================================
 0  0.3640  0  0  correct
 1  0.7242  1  0  WRONG
 2  0.2014  0  0  correct
 3  0.3507  0  0  correct
 4  0.0493  0  0  correct
 5  0.1000  0  0  correct
 6  0.7982  1  1  correct
 7  0.8944  1  1  correct
 8  0.9366  1  1  correct
 9  0.7887  1  1  correct
10  0.3056  0  1  WRONG
11  0.7125  1  1  correct

Predicting class for age, education =
[ 9.5  4.5]
Predicted p = 0.529154362398
Predicted class = 1

End evaluation


C:\CNTK_Succinctly\Ch_02>
```

Figure 2-2: Evaluating the Logistic Regression Model

The CNTK demo training program creates a logistic regression prediction model using 4,000 iterations of the stochastic gradient descent algorithm with binary cross-entropy error. After training completed, the demo displayed the values of the weights (-0.2049, 0.9666) and the value of the bias (-2.2864) that define the logistic regression model.

The screenshot in Figure 2-2 shows an evaluation of the trained logistic regression model. The program walks through each of the 12 training items, and computes and displays a prediction probability, the associated predicted class, the actual class, and a tag indicating if the prediction is correct or incorrect. The logistic regression prediction model correctly predicted 10 of the training items, for a classification accuracy of 10 / 12 = 83%.

After evaluating the model, the program shown in Figure 2-2 makes a prediction for a new, previously unseen person. The unknown item has normalized age = 9.5 and normalized education = 4.5. The prediction probability is 0.5291 and so the predicted class is 1 (male).

Understanding logistic regression

In order to understand the CNTK code that generated the output shown in Figures 2-1 and 2-2, you need a basic understanding of logistic regression. The key ideas are best explained by example. Suppose you want to predict the credit worthiness of a loan application (0 = reject, 1 = approve) based on the application's x1 = debt, x2 = income, x3 = credit rating. In logistic regression, you determine a weight value, w, for each feature, and a single bias value b.

Suppose x1 = 3.0, x2 = -2.0, and x3 = 1.0. And suppose you determine that w1 = 0.65, w2 = 1.75, w3 = 2.05, and b = 0.33. To compute the predicted class, you first compute z = (x1 * w1) + (x2 * w2) + (x3 * w3) + b:

```
z = (3.0)(0.65) + (-2.0)(1.75) + (1.0)(2.05) + 0.33
   = 1.95 - 3.50 + 2.05 + 0.33
   = 0.83
```

Next you compute p = 1.0 / (1.0 + exp(-z)) where the **exp()** function is Euler's number, e (~2.71828) raised to a power:

```
p = 1.0 / (1.0 + exp(-0.83))
   = 1.0 / (1.0 + 0.4360)
   = 1.0 / 1.43630
   = 0.6963
```

The p-value can be interpreted as the probability that the class is 1. Put another way, if p < 0.5 the prediction is class = 0, otherwise (if p >= 0.5) the prediction is class = 1.

OK, but where do the weights and bias values come from? To determine the values of the weights and bias, you must obtain a set of training data that has known input predictor values, and known, correct class label values. Then you use an algorithm (typically gradient descent, or a variation called stochastic gradient descent) to find values for the weights and bias so that computed output values closely match the known correct class labels.

An important aspect of logistic regression training is measuring the error between computed output values (using a set of weights and a bias) and the correct class label. Suppose, for a given set of input values and a given set of weights and bias values, the computed p value is 0.7000 and the known correct class label is 1. You could compute a squared error:

```
se = (1 - 0.7000) * (1 - 0.7000)
     = 0.09
```

And in fact, this approach can be used with logistic regression. However, for rather complex technical reasons, it's considered preferable to use what's called binary cross-entropy error (also called log-loss).

For the numeric example above (p = 0.7000, c = 1), binary cross-entropy error is calculated as:

```
cee = - [c * ln(p)] + [(1-c) * ln(1-p)]
        = - [1 * ln(0.7000)] + [0 * ln(0.3000)]
        = -ln(0.7000)
        = 0.3567
```

Cross-entropy error is a bit difficult to interpret, but smaller values mean smaller error (which means a more accurate prediction). As you'll see shortly, CNTK has functions that support both squared error and cross-entropy error. The key thing to remember is that when performing logistic regression training is that unless you have an existing system committed to using squared error, it's recommended that you use binary cross-entropy error, and smaller values are better.

Setting up the training data

The training data used by the programs shown in Figures 2-1 and 2-2 is graphed in Figure 2-3 and is:

```
1.0, 2.0, 0
3.0, 4.0, 0
5.0, 2.0, 0
6.0, 3.0, 0
8.0, 1.0, 0
9.0, 2.0, 0
1.0, 4.0, 1
2.0, 5.0, 1
4.0, 6.0, 1
6.0, 5.0, 1
7.0, 3.0, 1
8.0, 5.0, 1
```

When using CNTK for logistic regression, it's up to you to prepare your training data, including normalizing feature values, and encoding class labels as `0` or `1`. For example, the raw data might have resembled:

```
25.0, 12, female
37.0, 16, male
. . .
```

The demo program uses only two predictor variables, just to keep things simple, and so that the data can be displayed in a two-dimensional graph. Logistic regression can be used with any number of predictor variables. The predictor variables must be numeric, but it is possible to convert non-numeric predictor values into numeric values so that you can use logistic regression.

Compared to other binary classification techniques, the main advantage of logistic regression is simplicity. The primary disadvantage of logistic regression is that it only works well with data that is linearly separable. Conceptually, logistic regression finds a straight line that separates the two classes. For the demo data, no straight line will do better than 10 of 12 correct predictions.

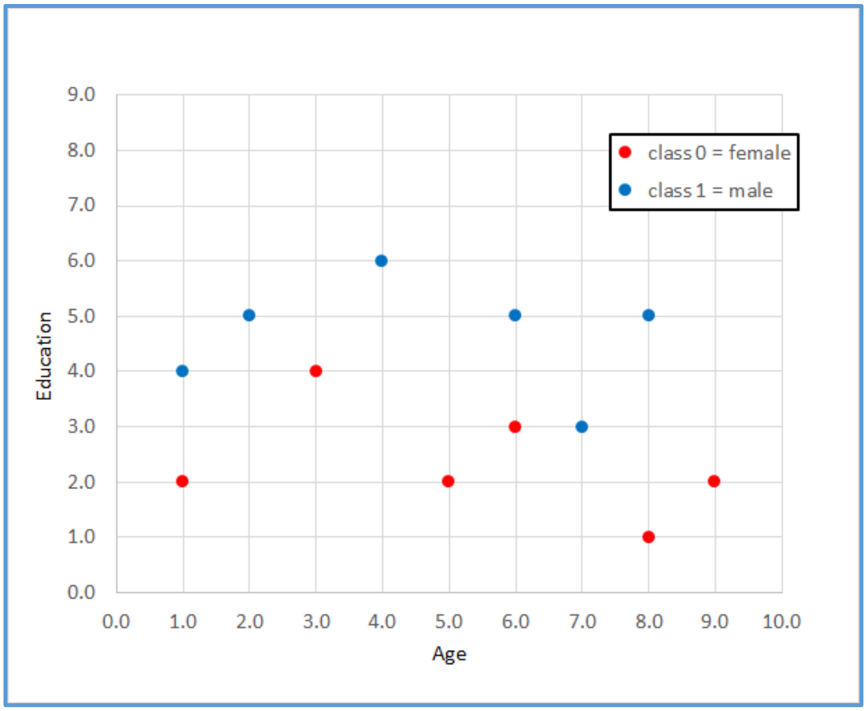


Figure 2-3: Data for Logistic Regression

Feature normalization and class encoding are explained in detail in Chapter 3. The data was copied into Notepad and saved as **age_edu_sex.txt** on a local machine. In a non-demo scenario, you might have hundreds, or thousands of training items.

Creating a logistic regression model

The program code that generated the screenshot shown in Figure 2-1 is presented in Code Listing 2-1. The program begins by commenting the name of the file and version of CNTK used, and importing the NumPy and CNTK packages:

```
# log_reg_train.py
# logistic regression age-education-sex synthetic data
# CNTK 2.3
import numpy as np
import cntk as C
```

In a non-demo scenario, you'd want to include additional details. The program structure consists of a single **`main`** function, with no helper functions:

```
def main():
  print("\nBegin logistic regression training demo \n")
  ver = C.__version
  print("Using CNTK version " + str(ver))
  . . .
  print("End program ")
if __name__ == "__main__":
  main()
```

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 clear. Because CNTK is young and under active development, it's good practice to indicate which version is being used in a code comment, and to programmatically verify the version, as shown.

Code Listing 2-1: Logistic Regression Training

```
# log_reg_train.py
# logistic regression age-education-sex synthetic data
# CNTK 2.3

import numpy as np
import cntk as C

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

def main():
  print("\nBegin logistic regression training demo \n")
  ver = C.__version__
  print("Using CNTK version " + str(ver))

  # training data format:
  # 4.0, 3.0, 0
  # 9.0, 5.0, 1
  # . . .

  data_file = ".\\age_edu_sex.txt"
  print("Loading data from " + data_file + "\n")
  features_mat = np.loadtxt(data_file, dtype=np.float32, delimiter=",",
    skiprows=0, usecols=[0,1])
  labels_mat = np.loadtxt(data_file, dtype=np.float32, delimiter=",",
    skiprows=0, usecols=[2], ndmin=2)

  print("Training data: \n")
  combined_mat = np.concatenate((features_mat, labels_mat), axis=1)
  print(combined_mat); print("")

  # create model
  features_dim = 2  # x1, x2
  labels_dim = 1    # always 1 for log regression

  X = C.ops.input_variable(features_dim, np.float32)    # cntk.Variable
  y = C.input_variable(labels_dim, np.float32)    # correct class value
  W = C.parameter(shape=(features_dim, 1))   # trainable cntk.Parameter
  b = C.parameter(shape=(labels_dim))
  z = C.times(X, W) + b        # or z = C.plus(C.times(X, W), b)
  p = 1.0 / (1.0 + C.exp(-z))  # or p = C.ops.sigmoid(z)
  model = p  # create an alias

  # create Learner and Trainer
  ce_error = C.binary_cross_entropy(model, y)  # CE a bit more principled
for LR
  fixed_lr = 0.010
  learner = C.sgd(model.parameters, fixed_lr)
  trainer = C.Trainer(model, (ce_error), [learner])
  max_iterations = 4000

  # train
  print("\nStart training, 4000 iterations, LR = 0.010, mini-batch = 1 \n")
  np.random.seed(4)
  N = len(features_mat)
  for i in range(0, max_iterations):
    row = np.random.choice(N,1)  # pick a random row from training items
    trainer.train_minibatch({ X: features_mat[row], y: labels_mat[row] })

    if i % 1000 == 0 and i > 0:
      mcee = trainer.previous_minibatch_loss_average
      print(str(i) + " Cross-entropy error on curr item = %0.4f " % mcee)

  print("\nTraining complete \n")

  np.set_printoptions(precision=4, suppress=True)
  print("Model weights: ")
  print(W.value)
  print("Model bias:")
  print(b.value)
  print("")
  # np.set_printoptions(edgeitems=3,infstr=‘inf’,
  #   linewidth=75, nanstr=‘nan’, precision=8,
  #   suppress=False, threshold=1000, formatter=None)  # reset all

  print("End program ")

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

Next, the program loads the training data into memory:

```
  # training data format:
  # 4.0, 3.0, 0
  # 9.0, 5.0, 1
  # . . .

  data_file = ".\\age_edu_sex.txt"
  print("Loading data from " + data_file + "\n")
  features_mat = np.loadtxt(data_file, dtype=np.float32, delimiter=",",
    skiprows=0, usecols=[0,1])
  labels_mat = np.loadtxt(data_file, dtype=np.float32, delimiter=",",
    skiprows=0, usecols=[2], ndmin=2)
```

The training data is located in the same directory as the training program. The CNTK training functions expect training data to be supplied in two matrices: one for the features, and one for the class labels. The demo uses the **numpy.loadtxt()** function. Note that this technique only works if your training data is small enough to be completely stored in memory. For large datasets, CNTK supports a **Reader** object, which is described in Chapter 4.

Most machine learning models use the **float32** data type, rather than the more common **float64** type, because the extra precision gained by using 64 bits is rarely worth the performance penalty incurred. The **skiprows** parameter is useful when a dataset has one or more header lines. Notice that columns are zero-base indexed.

The shape of the **features_mat** matrix is determined by the structure of the data file, and so in this case is 12×2 (12 rows, 2 columns). Notice the **ndmin=2** ("minimum dimensions") for the **labels_mat** matrix. Without that parameter, **loadtxt()** would infer a one-dimensional vector return type rather than the required 12×1 two-dimensional matrix.

Incompatible and incorrect matrix shapes are a very common source of CNTK errors. One way to track down matrix shape errors during development is to print shape information using the **shape** property, for example, **print(labels_mat.shape)**, which will display (12, 1).

After loading the training data into memory, the program displays the data like so:

```
  print("Training data: \n")
  combined_mat = np.concatenate((features_mat, labels_mat), axis=1)
  print(combined_mat); print("")
```

The **concatenate()** function combines two or more arrays or matrices. This is done, instead of displaying the features matrix and the labels matrix separately, only to make the display easier to read. Instead of printing the entire combined matrix, you could display just the first three rows using special Python syntax: **print(combined_mat[(0,1,2),:])** or **print(combined_mat(range(0,3),:])**. You can interpret either as, "Rows 0, 1, and 2, and all columns."

Next, the training program creates an (untrained) logistic regression model that is compatible with the training data:

```
 # create model
  features_dim = 2  # x0, x1
  labels_dim = 1    # always 1 for log regression
  X = C.ops.input_variable(features_dim, np.float32)    # cntk.Variable
  y = C.input_variable(labels_dim, np.float32)    # correct class value
  W = C.parameter(shape=(features_dim, 1))   # trainable cntk.Parameter
  b = C.parameter(shape=(labels_dim))
  z = C.times(X, W) + b        # or z = C.plus(C.times(X, W), b)
  p = 1.0 / (1.0 + C.exp(-z))  # or p = C.ops.sigmoid(z)
  model = p  # create an alias
```

The **features_dim** variable holds the number of predictor variables (two, for age and education in this case). The **labels_dim** holds the number of class labels. For logistic regression, this will always be 1 (even though the class label can take one of two possible values, 0 or 1, corresponding to female and male—one variable, two values). Neither dim variable name is a CNTK keyword so you can use names like **x_dim** and **y_dim** if you wish.

The CNTK **input_variable()** function creates storage to hold feature and label values. Notice that the assignment to object **X** uses the fully qualified name **C.ops.input_variable()** but the assignment to object **y** uses the shortcut syntax **C.input_variable()**. An alternative approach is to import specific functions and omit the qualification. For example:

```
import numpy as np
from cntk.ops import input_variable
. . .
X = input_variable(features_dim, np.float32)
```

The **input_variable()** function, like most CNTK functions, has a large number of parameters that can be omitted in a function call, because most parameters have default values, which makes the an explicit inclusion of those parameters optional. For example, **input_variable()** has optional parameters **needs_gradient**, **is_sparse, dynamix_axes**, and **name**. In fact, the **dtype** (data type) parameter has default value **np.float32**, so that argument could have been omitted, too.

The **W** matrix holds the logistic regression weights, and the **b** scalar holds the single bias value. Note that both are created with a call to the **parameter()** function, which means that when a CNTK training method is called, those object values will be updated automatically.

The logistic regression model is specified by computing a **z** value as the sum of products of weight and feature values, plus the bias **b**. The probability value, **p**, is computed as described previously. Note that I use the CNTK **exp()** function rather than the NumPy **np.exp()** function or the base Python **math.exp()** function; I could have used any of these three.

The function **f(x) = 1.0 / (1.0 + exp(-x))** occurs often in machine learning, and is called the logistic sigmoid function (sometimes not entirely correctly abbreviated to just sigmoid). CNTK has a built-in **sigmoid()** function, so the code that computes **p** can be written as:

```
p = C.ops.sigmoid(z)
```

At this point, at a low level, **p** represents the probability that the class has value 1, and at a higher level of abstraction, **p** represents the overall logistic regression model. The demo program creates a name-friendly alias by assigning object **p** to object **model**. This is completely optional and is done just for readability. CNTK also has an **ops.alias()** function you can use.

Training a logistic regression model

After reading the training data into memory and creating a logistic regression model, the next step is to train the model. The demo program continues by creating a **Learner** and a **Trainer** object:

```
  # create Learner and Trainer
  ce_error = C.binary_cross_entropy(model, y)
  fixed_lr = 0.010
  learner = C.sgd(model.parameters, fixed_lr)
  trainer = C.Trainer(model, (ce_error), [learner])
  max_iterations = 4000
```

You can think of a **Learner** as an algorithm to minimize error, and a **Trainer** as an object that uses the **Learner** to compute the weights and bias values. The demo program uses the binary cross-entropy function from the CNTK **losses** package to measure the error between computed output probabilities and known correct label values (0 or 1). If you wanted to use squared error, the code could be **s_error = C.squared_error(model, y)**.

The demo sets up a stochastic gradient descent learning algorithm using the **sgd()** function. CNTK supports many advanced learning algorithms such as **adadelta()**, **adam()**, and **nesterov()**. Advanced algorithms are most often used with deep neural networks. Stochastic gradient descent is the simplest learning algorithm, and generally works well for logistic regression problems.

The **sgd()** function requires a learning rate, which is a value that influences how much the weights and bias value change on each training iteration. The value of the learning rate must be determined by trial and error. If the learning rate is too small, training will be very slow. If the learning rate is too large, training may skip over a good result.

Instead of using a fixed learning rate, you can specify a learning rate schedule where the learning rate varies. Typically, this is a relatively large learning rate during the early iterations, and then smaller rate(s) during later iterations.

After a **Learner** object has been created, it is passed as an argument to the **trainer()** function to create a **Trainer** object. The second parameter to **trainer()** is a tuple, as indicated by the parentheses, because you can pass more than one error function. For linear regression, you should pass the binary cross-entropy function (or squared error). You can optionally create a second error function that measures classification error—the percentage of incorrect predictions made. For example:

```
clas_err = C.classification_error(model, y)
trainer = C.Trainer(model, (ce_error, clas_err), [learner])
```

However, as you'll see shortly, because the demo program adjusts weights and bias values after processing a single training item, computing classification accuracy is better done in a different way.

The third parameter to **`trainer()`** is a list, as indicated by the square brackets, because you can pass multiple **`Learner`** objects. This technique is usually used with deep neural networks—different parts of the network can use different learner algorithm objects.

Stochastic gradient descent is an iterative technique, so you must have some stopping condition. The simplest approach is to specify a fixed number of training iterations. The number of iterations must be determined by trial and error.

After the learner and trainer objects are created, the demo program trains the logistic regression model like so:

```
  # train
  print("\nStart training, 4000 iterations, LR = 0.010, mini-batch = 1 \n")
  np.random.seed(4)
  N = len(features_mat)

  for i in range(0, max_iterations):
    row = np.random.choice(N,1)  # pick a random row from training items
    trainer.train_minibatch({ X: features_mat[row], y: labels_mat[row] })

    if i % 1000 == 0 and i > 0:
      mcee = trainer.previous_minibatch_loss_average
      print(str(i) + " Cross-entropy error on curr item = %0.4f " % mcee)

  print("\nTraining complete \n")
```

When training a model, you must decide how many training items to use for each weight and bias update operation. For logistic regression, the most common approach (and the one used by the demo) is to update weights and bias values after processing a single training item. This is often called *online training*. An alternative is to process all training items and then perform an update operation. This is often called *batch training*.

A third approach is to process a certain number of training items—for example, four or five—before updating. This is often called mini-batch training. Notice that mini-batch training with a batch size equal to 1 is equivalent to online training, and that mini-batch training with a batch size equal to the size of the training dataset is equivalent to batch training.

When using the online or mini-batch approach, it's important that the training items be visited in random order. The demo sets **`np.random.seed`** to 4 so that results are reproducible. The seed value of 4 was used only because it gave a representative demo output. Inside the training loop, function **`np.random.choice()`** is used to select a single row index.

The **train_minibatch()** function accepts the predictor values, creates an anonymous dictionary object with **X** as the key, and the known correct class label (an anonymous dictionary object with **y** as the key). Behind the scenes, **train_minibatch()** computes a predicted output, then uses binary cross entropy to update each weight and bias value slightly, governed by the learning rate.

When training a model, it's important to display the current error so you can catch situations where training isn't working. Training failure is very common. The demo uses the **previous_minibatch_loss_average** property to display the binary cross-entropy error for the previous training item, once every 1,000 iterations. The previous item is used rather than the current item because of the internal architecture of CNTK. In a non-demo scenario, you'd likely display error more frequently, for example, once every 100 iterations.

Note that when using online training, because you're only processing one training item at a time, the error value will be quite volatile. What you hope to see are error values that generally decrease. What you don't want to see are error values that don't really change, or worse, values that steadily increase.

When using online training, it's not too helpful to display the classification error, or its complement, classification accuracy, because the accuracy applies to a single training item and will be either 0% (if the item is incorrectly predicted) or 100% (for a correct prediction).

The online training algorithm used in the demo selects a random row from the training data, 4,000 times. This doesn't guarantee that each training item will be used the same number of times. A more sophisticated approach is to walk through each training item, in random order, and then repeat that process several times. For example:

```
# train
np.random.seed(0)
N = len(features_mat)
for sweep in range(0, 100):
  indices = np.random.choice(N, N, replace=False)
  for idx in indices:
    trainer.train_minibatch({ X: features_mat[idx], y: labels_mat[idx] })
  mcee = trainer.previous_minibatch_loss_average
  print(" Cross-entropy error on curr item = %0.4f \n" % mcee)
```

Because there are N = 12 training items, the **np.random.choice()** function generates an array of indices, from 0 to 11, in random order. The training item at each index is processed, and then another set of indices in random order is generated. This means there would be 100 * 12 = 1200 total update operations and each training item is used the same number of times.

The demo training program concludes by displaying the values of the two weights and the value of the bias:

```
np.set_printoptions(precision=4, suppress=True)
print("Model weights: ")
print(W.value)
print("Model bias:")
print(b.value)
print("")
```

Notice that you can't simply print `W` and `b` directly because they are CNTK objects of type **`Parameter`**, so you must use their **`value`** property. The values of `W` and `b` effectively define the logistic regression model so instead of just displaying them; you might want to write them to a text file so they can be retrieved by another program.

Using a trained logistic regression model

After a logistic regression model has been trained, you'll usually want to evaluate it for classification accuracy, and eventually use it to make a prediction for new, previously unseen data. The program **log_reg_eval.py** that generated the output shown in Figure 2-2 is presented in Code Listing 2-2. Note that the **** character is used by Python for line continuation.

Code Listing 2-2: Evaluating a Trained Model

```
# log_reg_eval.py
# logistic regression age-education-sex data

import numpy as np

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

def main():
  print("\nBegin logistic regression model evaluation \n")

  data_file = ".\\age_edu_sex.txt"
  print("Loading data from " + data_file)
  features_mat = np.loadtxt(data_file, dtype=np.float32, delimiter=",",
    skiprows=0, usecols=(0,1))
  labels_mat = np.loadtxt(data_file, dtype=np.float32, delimiter=",",
    skiprows=0, usecols=[2], ndmin=2)

  print("Setting weights and bias values \n")
  weights = np.array([-0.2049, 0.9666], dtype=np.float32)
  bias = np.array([-2.2864], dtype=np.float32)
  N = len(features_mat)
  features_dim = 2

  print("item  pred_prob  pred_label   act_label  result")
  print("================================================")
  for i in range(0, N):  # each item
    x = features_mat[i]
    z = 0.0
    for j in range(0, features_dim):  # each feature
      z += x[j] * weights[j]
    z += bias[0]
    pred_prob = 1.0 / (1.0 + np.exp(-z))
    pred_label = 0 if pred_prob < 0.5 else 1
    act_label = labels_mat[i]
    pred_str = 'correct' if np.absolute(pred_label - act_label) < 1.0e-5 \
      else 'WRONG'
    print("%2d  %0.4f  %0.0f  %0.0f  %s" % \
      (i, pred_prob, pred_label, act_label, pred_str))

  x = np.array([9.5, 4.5], dtype=np.float32)
  print("\nPredicting class for age, education = ")
  print(x)
  z = 0.0
  for j in range(0, features_dim):
    z += x[j] * weights[j]
  z += bias[0]
  p = 1.0 / (1.0 + np.exp(-z))
  print("Predicted p = " + str(p))
  if p < 0.5: print("Predicted class = 0")
  else: print("Predicted class = 1")

  print("\nEnd evaluation \n")

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

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

The evaluation program imports the **numpy** package but does not need the **cntk** package, because the classification accuracy of the trained model will be computed from scratch rather than by using the **previous_minibatch_evaluation_average** property of the CNTK **Trainer** object.

The evaluation program loads the training data into a features matrix and a class label matrix in exactly the same way as the training program. In many scenarios you will want to combine training and evaluation into the same program; two different programs are used here to emphasize the distinction between the two operations.

Next, the evaluation program sets the values of the weights and the bias that were determined by the training program:

```
print("Setting weights and bias values \n")
weights = np.array([0.0925, 1.1722], dtype=np.float32)
bias = np.array([-4.5400], dtype=np.float32)
N = len(features_mat)
features_dim = 2
```

Notice that the evaluation program uses ordinary NumPy arrays rather than CNTK **Parameter** objects. Next, the evaluation program walks through each training item and computes the logistic regression probability:

```
  print("item  pred_prob  pred_label   act_label  result")
  print("================================================")
  for i in range(0, N):  # each item
    x = features_mat[i]
    z = 0.0
    for j in range(0, features_dim):  # each feature
      z += x[j] * weights[j]
    z += bias[0]
    pred_prob = 1.0 / (1.0 + np.exp(-z))
. . .
```

An alternative approach is to write a program-defined function to perform this calculation. For example:

```
def compute_p(x, w, b):
  z = 0.0
  for i in range(len(x)):
    z += x[i] * w[i]
  z += b
  p = 1.0 / (1.0 + np.exp(-z))
  return p
```

After computing the prediction probability for the current training item, the program determines the associated predicted 0 or 1 label, and compares that with the actual class label:

```
  . . .
  pred_label = 0 if pred_prob < 0.5 else 1
  act_label = labels_mat[i]
  pred_str = ‘correct’ if np.absolute(pred_label - act_label) < 1.0e-5 \
    else ‘WRONG’
  print("%2d  %0.4f  %0.0f  %0.0f  %s" % \
    (i, pred_prob, pred_label, act_label, pred_str))
```

Comparing the computed class with the actual class is a bit tricky because the actual class is type **float32**, so the comparison checks to see if the two values are within 0.00001 of each other rather than check for exact equality. In practice you can usually compare directly, for example: **pred_str = ‘correct’ if pred_label == act_label else ‘WRONG**.

If you’re new to Python, the conditional if-statements may look a bit odd. You could in fact use standard C-family language syntax, for example:

```
if np.absolute(pred_label - act_label) < 1.0e-5:
  pred_str = 'correct'
else:
  pred_str = 'WRONG'
```

Because there are only 12 training items, the evaluation program just displays **correct** or **WRONG**. In a non-demo scenario, you'd likely want to use counter variables and then compute classification accuracy code along the lines of **class_acc = num_corr * 1.0 / (num_corr + num_wrong)**.

In some situations, you may want to compute and display the final average binary cross-entropy error for all training data. If you are using batch training, you can use the **Trainer** object's **previous_minibatch_loss_average** property, and then use the last value. A more flexible approach is to use the trained model and walk through each training item, and compute error yourself. For example, you could modify the code in the evaluation program to include:

```
sum_bcee = 0.0  # sum of binary cross-entropy error terms
for i in range(0, N):  # each item
  x = features_mat[i]
  z = 0.0
  for j in range(0, features_dim):  # each feature
    z += x[j] * weights[j]
  z += bias[0]                      # add the bias
  p = 1.0 / (1.0 + np.exp(-z))      # prediction prob
  y = labels_mat[i]                 # actual label/prob
  if y == 1:
    sum_bcee += -np.log(p)
  else:
    sum_bcee += -np.log(1-p)
mean_bcee = sum_bcee / N
print("mean binary cross-entropy error = " + str(mean_bcee))
```

The demo programs in this chapter use only one set of training data. In most situations, you'd have an additional test dataset that is held out from training. You'd then evaluate the classification accuracy, and possibly the cross-entropy error, on the test data. If the classification accuracy of the test data is significantly worse than the classification accuracy of the training data, your model may be overfitted.

The entire point of logistic regression classification is to make a prediction using new, previously unseen data. The evaluation demo concludes by demonstrating one way to make a prediction:

```
x = np.array([9.5, 4.5], dtype=np.float32)
print("\nPredicting class for age, education = ")
print(x)
z = 0.0
for j in range(0, features_dim):
  z += x[j] * weights[j]
z += bias[0]
p = 1.0 / (1.0 + np.exp(-z))
print("Predicted p = " + str(p))
if p < 0.5: print("Predicted class = 0")
else: print("Predicted class = 1")
```

Notice that because the logistic regression model was trained using normalized age and education values, you make predictions using normalized feature values, (9.5, 4.5) in this example.

The prediction code doesn't use any CNTK functionality; therefore, you can use this approach in any system. For example, you could write equivalent code in some C# program. The trained weights and bias values define the model, so they can be used anywhere.

Exercise

There are several repositories for standard benchmark datasets for machine learning. One well-known dataset is the Wisconsin Cancer data. Using the programs in Code Listing 2-1 and 2-2 as guides, create and evaluate a logistic regression prediction model for the Wisconsin Cancer dataset.

The raw data is located at the University of California at Irvine repository.

There are several different versions of the dataset, and the documentation is quite confusing. Use the version of the dataset that has 699 items. Each item has an ID number, followed by nine normalized predictor values, followed by a value of 2 or 4, where 2 = benign and 4 = malignant.

The data resembles the following:

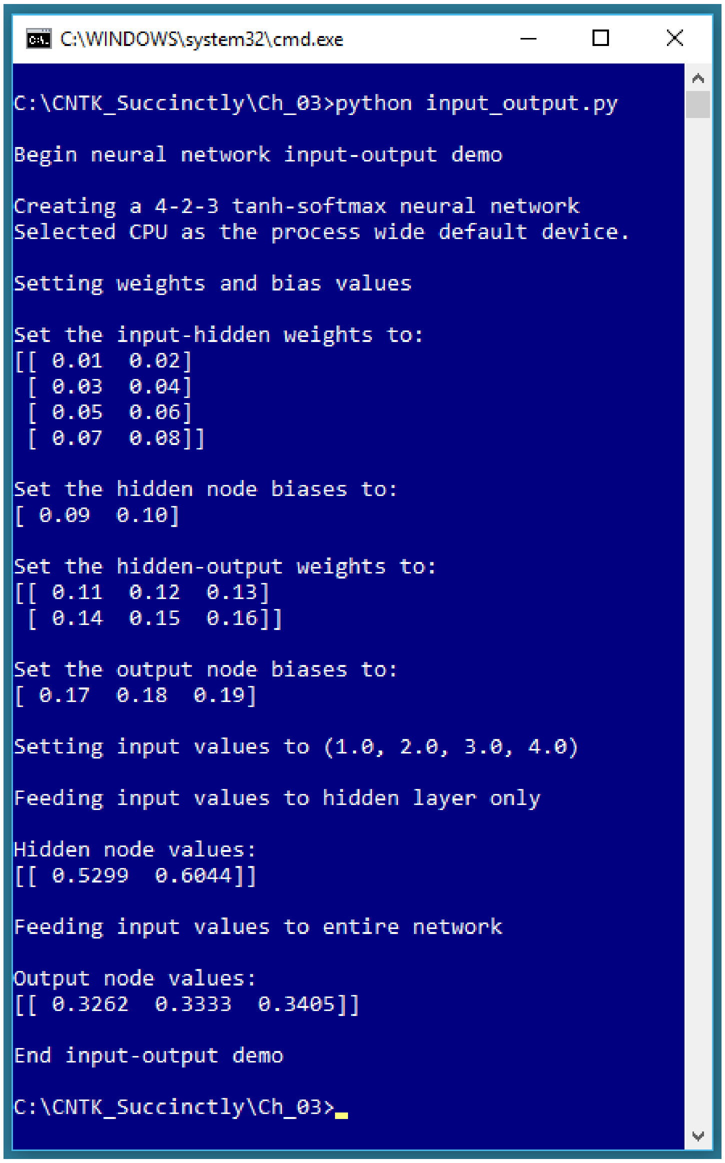
```
1000025,5,1,1,1,2,1,3,1,1,2
1002945,5,4,4,5,7,10,3,2,1,2
. . .
1017122,8,10,10,8,7,10,9,7,1,4
1018099,1,1,1,1,2,10,3,1,1,2
. . .
897471,4,8,8,5,4,5,10,4,1,4
```

The demo program in Code Listing 2-1 just displays the weights and bias values to the shell. So when you evaluate the model as in Code Listing 2-2, you need to copy those values into the program. As an alternative, you might want to write the weights and bias values to a text file using the Python **`open()`** and **`write()`** functions, and then load them using the **`read()`** function.

CNTK has **`save()`** and **`load()`** functions that make this process easier. The process for saving and loading a trained model using CNTK functions is explained in Chapter 4.

Chapter 3 Fundamental Concepts

In order to effectively use the CNTK deep-learning code library functions, you must have a basic grasp of a handful of key machine learning concepts. This chapter explains neural network architecture and input-output, error and accuracy, data encoding and normalization, batch and online training, model overfitting, and train-test validation. The screenshot in Figure 3-1 gives you an idea of what this chapter covers.



```
C:\CNTK_Succinctly\Ch_03>python input_output.py

Begin neural network input-output demo

Creating a 4-2-3 tanh-softmax neural network
Selected CPU as the process wide default device.

Setting weights and bias values

Set the input-hidden weights to:
[[ 0.01  0.02]
 [ 0.03  0.04]
 [ 0.05  0.06]
 [ 0.07  0.08]]

Set the hidden node biases to:
[ 0.09  0.10]

Set the hidden-output weights to:
[[ 0.11  0.12  0.13]
 [ 0.14  0.15  0.16]]

Set the output node biases to:
[ 0.17  0.18  0.19]

Setting input values to (1.0, 2.0, 3.0, 4.0)

Feeding input values to hidden layer only

Hidden node values:
[[ 0.5299  0.6044]]

Feeding input values to entire network

Output node values:
[[ 0.3262  0.3333  0.3405]]

End input-output demo

C:\CNTK_Succinctly\Ch_03>_
```

Figure 3-1: Neural Network Input-Output

Neural network architecture

The program that generate the output shown in Figure 3-1 is presented in Code Listing 3-1. A diagram that represents the demo program neural network is shown in Figure 3-2.

Code Listing 3-1: Neural Network Input-Output Mechanism

```
# input_output.py
# demo the NN input-output mechanism
# CNTK 2.3

import numpy as np
import cntk as C

print("\nBegin neural network input-output demo \n")
np.set_printoptions(precision=4, suppress=True,
  formatter={'float': '{: 0.2f}'.format})

i_node_dim = 4
h_node_dim = 2
o_node_dim = 3

X = C.ops.input_variable(i_node_dim, np.float32)
Y = C.ops.input_variable(o_node_dim, np.float32)

print("Creating a 4-2-3 tanh-softmax neural network")
h = C.layers.Dense(h_node_dim, activation=C.ops.tanh,
  name='hidLayer')(X)
o = C.layers.Dense(o_node_dim, activation=C.ops.softmax,
  name='outLayer')(h)
nnet = o

print("\nSetting weights and bias values")
ih_wts = np.array([[0.01, 0.02],
                   [0.03, 0.04],
                   [0.05, 0.06],
                   [0.07, 0.08]], dtype=np.float32)

h_biases = np.array([0.09, 0.10])

ho_wts = np.array([[0.11, 0.12, 0.13],
                   [0.14, 0.15, 0.16]], dtype=np.float32)

o_biases = np.array([0.17, 0.18, 0.19], dtype=np.float32)

h.hidLayer.W.value = ih_wts
h.hidLayer.b.value = h_biases
o.outLayer.W.value = ho_wts
o.outLayer.b.value = o_biases
```

```
print("\nSet the input-hidden weights to: ")
print(h.hidLayer.W.value)
print("\nSet the hidden node biases to: ")
print(h.hidLayer.b.value)
print("\nSet the hidden-output weights to: ")
print(o.outLayer.W.value)
print("\nSet the output node biases to: ")
print(o.outLayer.b.value)

print("\nSetting input values to (1.0, 2.0, 3.0, 4.0)")
x_vals = np.array([1.0, 2.0, 3.0, 4.0], dtype=np.float32)

np.set_printoptions(formatter={'float': '{: 0.4f}'.format})
print("\nFeeding input values to hidden layer only ")
h_vals = h.eval({X: x_vals})
print("\nHidden node values:")
print(h_vals)

print("\nFeeding input values to entire network ")
y_vals = nnet.eval({X: x_vals})
print("\nOutput node values:")
print(y_vals)

print("\nEnd input-output demo ")
```

The demo program begins by importing the **`numpy`** and **`cntk`** packages. The **`set_printoptions()`** function suppresses printing very small values with scientific notation (such as 1.23e-6) and instructs **`print()`** to display floating point values to exactly two decimals.

Next, the demo prepares to create a neural network:

```
i_node_dim = 4
h_node_dim = 2
o_node_dim = 3

X = C.ops.input_variable(i_node_dim, np.float32)
Y = C.ops.input_variable(o_node_dim, np.float32)
```

The first three statements specify the number of input, hidden, and output nodes. The next two statements set up a CNTK type **`Variable`** object (**`X`**) to hold input data from a data source, usually a text file of training data, and a second **`Variable`** object (**`Y`**) to hold known, correct output values from the training data. The function name **`input_variable()`** may be a bit confusing—the “input” refers to input into the program from a data source, not necessarily input to a neural network.

Now the demo can create a neural network:

```
print("Creating a 4-2-3 tanh-softmax neural network")
h = C.layers.Dense(hnode_dim, activation=C.ops.tanh,
  name='hidLayer')(X)
o = C.layers.Dense(onode_dim, activation=C.ops.softmax,
  name='outLayer')(h)
nnet = o
```

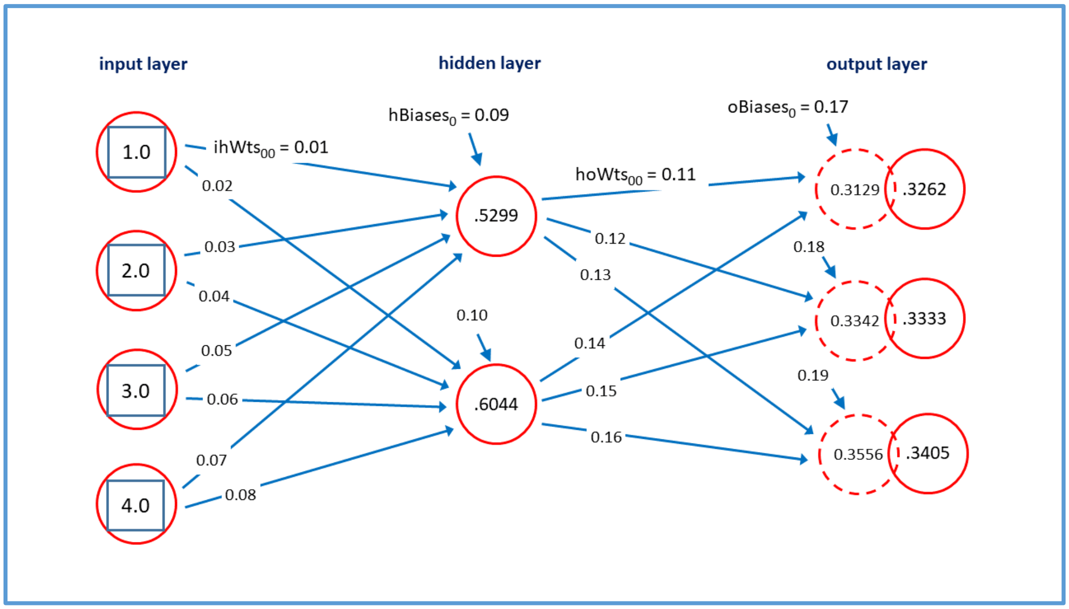


Figure 3-2: Neural Network Architecture

The **Dense()** function is used to create a fully connected layer of hidden nodes or output nodes, but not input nodes. The syntax is a bit strange. The **X Variable** object acts as input to the hidden layer **h**, and **h** acts as the input to the output layer **o**. The demo creates an alias of **nnet** to the output layer just for readability—the output of the neural network is conceptually the same as the network.

Notice that **X** is not really a layer of the network. If you wanted to create an explicit input layer, you could write:

```
i = X
h = C.layers.Dense(h_node_dim, activation=C.ops.tanh,
  name='hidLayer')(i)
o = C.layers.Dense(o_node_dim, activation=C.ops.softmax,
  name='outLayer')(h)
nnet = o  # or nnet = C.ops.alias(o)
```

The hidden layer is assigned a **name** property. This is optional, but useful in some situations. The hidden layer is configured to use **tanh** (hyperbolic tangent) activation. Activation functions are explained in the next section of this chapter. The output layer is also assigned a name, and is configured to use **softmax** activation. The **nnet** object is an alias for the output layer and can be omitted. You can also use the **alias()** function, as shown in the code comment.

Next, the demo program sets up the weights and biases. In the diagram in Figure 3-2, each blue arrow connecting a node to another node represents a numeric constant called a weight. There are 4 * 2 = 8 input-to-hidden weights and 2 * 3 = 6 hidden-to-output weights. Each short blue arrow pointing into a hidden or output node is a special weight called a bias. There are 2 + 3 = 5 biases. In general, for a fully connected neural network with **ni** input nodes, **nh** hidden nodes and **no** output nodes, there will be a total of (**ni** * **nh**) + (**nh** * **no**) + (**nh** + **no**) weights and biases.

The demo sets up the input-to-hidden weights and the hidden node biases like this:

```
ih_wts = np.array([[0.01, 0.02],
                   [0.03, 0.04],
                   [0.05, 0.06],
                   [0.07, 0.08]], dtype=np.float32)

h_biases = np.array([0.09, 0.10])
```

The weights are placed into a NumPy array-of-arrays style matrix. The row index indicates the index of the input node, and the column index is the index of the hidden node. For example, the value at [2, 1] is 0.06 and is the weight connecting input node [2] to hidden node [1].

The demo sets up the hidden-to-output weights and the output node biases in the same way:

```
ho_wts = np.array([[0.11, 0.12, 0.13],
                   [0.14, 0.15, 0.16]], dtype=np.float32)

o_biases = np.array([0.17, 0.18, 0.19], dtype=np.float32)
```

Next, the weights and bias values are copied into the neural network:

```
h.hidLayer.W.value = ih_wts
h.hidLayer.b.value = h_biases
o.outLayer.W.value = ho_wts
o.outLayer.b.value = o_biases
```

To access the weights in a CNTK neural network, the demo program uses the pattern:

```
layer_object.layer_name.W.value
```

Using the layer **name** property was required in previous versions of CNTK, but now you can just use a **Layer** object directly, for example:

```
h.W.value = ih_wts
h.b.value = h_biases
o.W.value = ho_wts
o.b.value = o_biases
```

Instead of placing the values of the weights and biases into NumPy arrays, and then transferring the values into the neural network, the demo could have set the weights and bias values directly. For example:

```
h.W.value = np.array([[0.01, 0.02],
                      [0.03, 0.04],
                      [0.05, 0.06],
                      [0.07, 0.08]], dtype=np.float32)
```

You have a lot of flexibility when using CNTK. This is good in general, but because there are many ways to perform most tasks, understanding the CNTK documentation examples is a bit more difficult than it would be if there were only one way to perform a task.

Neural network input-output mechanism

After the neural network has been created, and its weights and bias values have been set, the demo program in Code Listing 3-1 creates some input node values and feeds those values to the network:

```
print("\nSetting input values to (1.0, 2.0, 3.0, 4.0)")
x_vals = np.array([1.0, 2.0, 3.0, 4.0], dtype=np.float32)
np.set_printoptions(formatter={‘float’: ‘{: 0.4f}’.format})

print("\nFeeding input values to hidden layer only ")
h_vals = h.eval({X: x_vals})  # or just h.eval(x_vals)
print("\nHidden node values:")
print(h_vals)

print("\nFeeding input values to entire network ")
y_vals = nnet.eval({X: x_vals})
print("\nOutput node values:")
print(y_vals)

print("\nEnd input-output demo ")
```

This code is a bit subtle. The call to **h.eval()** uses the values in array **x_vals** to create an anonymous dictionary object with **X** as the key, and then the hidden node values are computed and returned into array **h_vals**, using the input-to-hidden weights and the **tanh** activation function. Using a dictionary is optional, and you can just pass the array directly. The call to **nnet.eval()** accepts the values in **x_vals**, transfers them into an anonymous dictionary object, recomputes the values of the hidden nodes, and then computes the values of the output nodes, which are then returned into array **y_vals**. The **Y** object is not used by the demo. Typically, **Y** holds known, correct output node values from a set of training data.

To recap, when you create a neural network, you can easily access the values of the weights and biases using the **W** and **b** properties of a **Layer** object. There is no easy way to access the values of the hidden and output layer nodes—you must use the return value from a call to the **eval()** function applied to a **Layer** object.

The neural network input-output mechanism isn’t as complicated as it might seem, and is best explained by example. Suppose the input values are (1.0, 2.0, 3.0, 4.0) and the weights and bias values are as shown in Figure 3-2. The topmost hidden node value is 0.5299 and is computed as:

```
h[0] = tanh( (1.0)(0.01) + (2.0)(0.03) + (3.0)(0.05) + (4.0)(0.07) + 0.09 )
       = tanh( 0.01 + 0.06 + 0.15 + 0.28 + 0.09 )
       = tanh(0.59)
       = 0.5299
```

In words, you compute the sum of products of each input value and its associated weight, add the bias, and take the hyperbolic tangent of the sum.

After the hidden node values are computed, they are used to compute the output nodes. The preliminary values of the output nodes are (0.3129, 0.3342, 0.3556) and are computed as:

```
pre_o[0] = (0.5299)(0.11) + (0.6044)(0.14) + 0.17
              = 0.0583 + 0.0846 + 0.17
              = 0.3129

pre_o[1] = (0.5299)(0.12) + (0.6044)(0.15) + 0.18
              = 0.0636 + 0.0907 + 0.18
              = 0.3342

pre_o[2] = (0.5299)(0.13) + (0.6044)(0.16) + 0.19
              = 0.0689 + 0.0967 + 0.19
              = 0.3556
```

In words, you just compute the sum of products of hidden nodes and associated weights, and then add the associated bias value.

After the preliminary node values are computed, they are coerced so that the sum to 1.0 using the softmax function. (Note: in this example, the preliminary node values almost sum to 1.0, but this is just a coincidence, and in most cases the sum of the preliminary node values will not be close to 1.0).

Softmax is best explained by example. The softmax function is applied to the preliminary output node values as:

```
 o[0] = exp(0.3129) / [ exp(0.3129) + exp(0.3342) + exp(0.3556) ]
        = 1.3674 / (1.3674 + 1.3969 + 1.4270)
        = 1.3674 / 4.1913
        = 0.3262

o[1] = exp(0.3342) / [ exp(0.3129) + exp(0.3342) + exp(0.3556) ]
        = 1.3969 / (1.3674 + 1.3969 + 1.4270)
        = 1.3969 / 4.1913
        = 0.3333

o[2] = exp(0.3556) / [ exp(0.3129) + exp(0.3342) + exp(0.3556) ]
        = 1.4270 / (1.3674 + 1.3969 + 1.4270)
        = 1.4270 / 4.1913
        = 0.3405
```

These are the final output values. Because they sum to 1.0, they can loosely be interpreted as probabilities. In words, the softmax of one of a set of values is the **`exp()`** of the value divided by the sum of the **`exp()`** of all the values. The **`exp(x)`** function is Euler's number (approximately 2.71828) raised to **`x`**.

The two most common activation functions used by single hidden layer neural networks are hyperbolic tangent (**`tanh`**) and logistic sigmoid (logsig or just **`sigmoid`**). The graphs of those two functions are shown in Figure 3-3. The functions are closely related. The mathematical **`tanh`** function accepts any real value from negative infinity to positive infinity, and returns a value between -1.0 and +1.0. The **`logsig(x) = 1.0 / (1.0 + exp(-x))`** function accepts any real value and returns a value between 0.0 and 1.0.

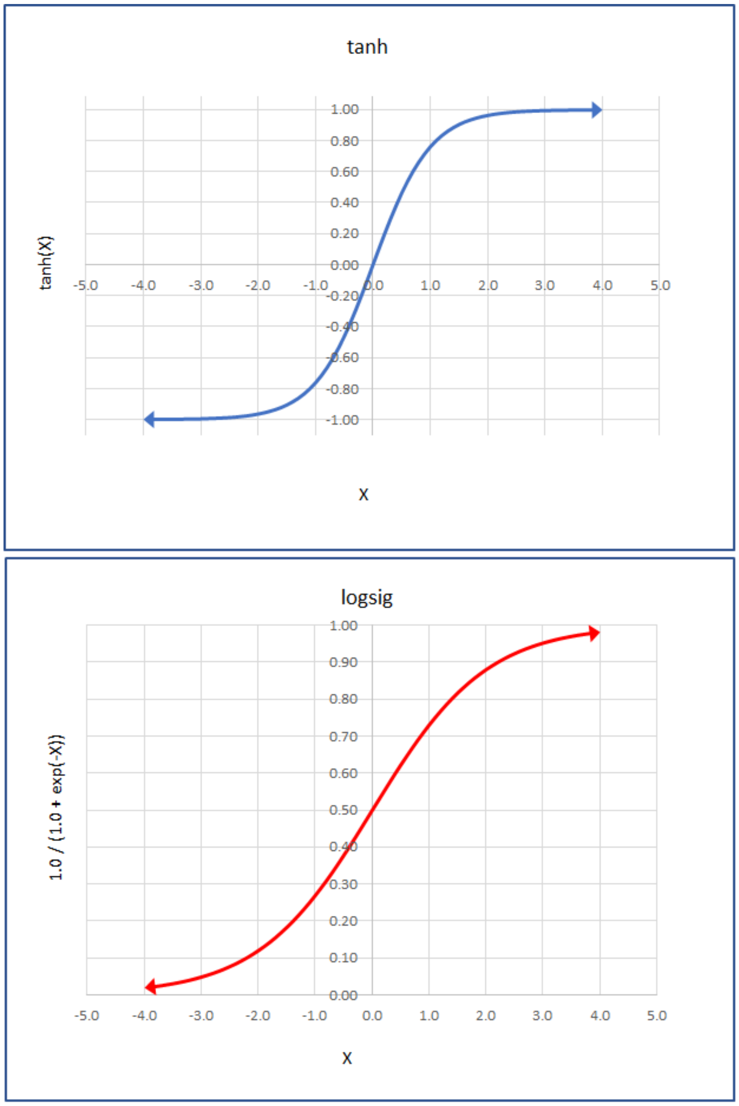


Figure 3-3: The Tanh and LogSig Activation Functions

In practice, for relatively simple neural networks, both activation functions usually give similar results, but **`tanh`** is used a bit more often than **`sigmoid`** for hidden layer activation. Other, less-common activation functions supported by CNTK include **`softplus`**, **`relu`** (rectified linear unit), **`leaky_relu`**, **`param_relu`** (parametric **`relu`**), and **`selu`** (scaled exponential linear unit).

Encoding and normalization

Neural networks only work directly with numeric data. Non-numeric data must be encoded. Numeric data should often be normalized (also called scaled) so that the magnitudes of the values are roughly the same. Suppose you have some training data that looks like:

```
27  42,000.00  liberal
56  68,000.00  conservative
37  94,000.00  moderate
```

The goal is to predict political leaning from a person's age and income. You should normalize the age and income predictor values so that the relatively large incomes don't swamp the ages. There are three common normalization techniques. The first technique, which doesn't have a standard name, is to just multiply or divide all values by an appropriate power of 10. For example, you could divide all age values by 10, and all income values by 10,000:

```
2.7  4.20  liberal
5.6  6.80  conservative
3.7  9.40  moderate
```

Another technique is called min-max normalization. For each predictor variable, you replace a value with (max - value) / (max - min). For example, for the three raw age values, the max = 56 and the min = 27. The normalized age value for 37 is (37 - 27) / (56 - 27) = 0.3448. The min-max normalized data would be:

```
0.0000  1.0000  liberal
1.0000  0.5000  conservative
0.3448  0.0000  moderate
```

Using min-max normalization, all normalized values will be between 0.0 and 1.0 where a 0.0 is the normalized value for the smallest raw value, and a 1.0 is the largest value.

A third technique is called z-score (or Gaussian or normal) normalization. For each predictor variable, you replace the value with (value - mean) / (std dev). For example, for the raw age values, the mean is 40.0 and the standard deviation is 12.03. The normalized age value for 37 is (37 - 40.0) / 12.03 = -0.2494. The z-score normalized data would be:

```
-1.0808  -1.2247  liberal
 1.3303   0.0000  conservative
-0.2494   1.2247  moderate
```

Using z-score normalization, normalized values with typically be between -10.0 and +10.0 where a positive value is one that greater than the mean, a negative value is smaller than the mean, and a 0.0 is the mean value.

As a rule of thumb, I use divide-by-power-of-10 normalization when all values of all predictor variables have the same order of magnitude, because the technique is simple and preserves the sign of the raw data. I use z-score normalization when the underlying data is likely to be Gaussian distributed. I use min-max normalization when I can't use power-of-10 or z-score normalization.

There are several ways to encode non-numeric predictor/feature values, but just one standard way to encode non-numeric class label values. Suppose you have some training data like:

```
female  2.5  democrat     white
male    4.0  republican   red
female  3.1  democrat     silver
male    5.3  independent  black
male    2.9  democrat     silver
```

The goal is to predict a person's car color (black, red, silver, or white) from their sex (male or female), age, and political affiliation (democrat, republican, or independent). The most common way to encode non-numeric predictor values is to use what's called 1-of-(N-1) encoding. For a predictor variable that can be just one of two possible values, you encode as -1 or +1.

The 1-of-(N-1) technique for a predictor variable that can be three or more values is best explained by example. For the political affiliation data shown previously, you'd encode democrat as (1, 0), encode republican as (0, 1), and encode independent as (-1, -1).

Suppose a predictor variable can be one of five categorical values from a survey: terrible, weak, average, good, and excellent. You could encode terrible = (1, 0, 0, 0), weak = (0, 1, 0, 0), average = (0, 0, 1, 0), good = (0, 0, 0, 1), excellent = (-1, -1, -1, -1). In general, to encode a categorical variable that can take one of N values, you use N-1 values, where one value is a 1 and the other values are 0, except for the last categorical value which you encode as all -1 values.

Encoding a non-numeric class label (value to predict) is almost always done using what is called 1-of-N (also known as "one-hot") encoding. If the class label can be one of three or more categorical values, you use N values where one value is a 1 and the other values are 0. For the data above where car color can be (black, red, silver, white), you could encode black = (1, 0, 0, 0), red = (0, 1, 0, 0), silver = (0, 0, 1, 0), and white = (0, 0, 0, 1).

The idea behind 1-of-N encoding is that you will design a neural network so that it outputs N values that sum to 1.0 and which can be interpreted as probabilities. Suppose an output is (0.20, 0.55, 0.15, 0.10) then because the largest probability is 0.55, the prediction maps to (0, 1 0, 0) which is the encoding for red.

To recap, if the original, raw car data is:

```
female  2.5  democrat     white
male    4.0  republican   red
female  3.1  democrat     silver
male    5.3  independent  black
male    2.9  democrat     silver
```

Then the encoded data would be:

```
 1  2.5   1  0   0 0 0 1
-1  4.0   0  1   0 1 0 0
 1  3.1   1  0   0 0 1 0
-1  5.3  -1 -1   1 0 0 0
-1  2.9   1  0   0 0 1 0
```

A special case occurs when the value to predict is binary. You can encode normally using (1, 0) and (0, 1), or you can encode as a single value that's either 0 or 1. Binary classification and the two types of encoding the class label values are discussed in Chapter 5.

Normalizing and encoding training data is often time consuming and annoying. The CNTK library does not have any built-in normalization and encoding functions. Libraries such as pandas ("panel data" library) and scikit-learn ("science kit machine learning") do have such functions, but those libraries have non-trivial learning curves. Typically, you must use a variety of techniques for data preprocessing.

Squared error and cross-entropy error

You can think of a neural network as a complex math function. The behavior of a neural network is determined by its input values, its weights and bias values, and its activation functions. Determining the values of the weights and biases is called training the network. The idea is to use a set of training data that has known input values, and known, correct output values, and then use some algorithm to find values of the weights and biases so that the error between computed output values, and the known correct output values, is minimized.

When using CNTK, you must specify which error metric to use, which learning algorithm to use, and how many training items to use at a time. The two most common forms of error metric for training are squared error and cross-entropy error (also called log loss). The demo program in Code Listing 3-2 shows how to compute squared error and cross-entropy error.

Code Listing 3-2: Squared Error and Cross-Entropy Error

```python
# error_demo.py
# CNTK 2.3, Anaconda 4.1.1

import numpy as np
import cntk as C

targets = np.array([[1, 0, 0],
                    [0, 1, 0],
                    [0, 0, 1],
                    [1, 0, 0]], dtype=np.float32)

computed = np.array([[7.0, 2.0, 1.0],
                     [1.0, 9.0, 6.0],
                     [2.0, 1.0, 5.0],
                     [4.0, 7.0, 3.0]], dtype=np.float32)

np.set_printoptions(precision=4, suppress=True)

print("\nTargets = ")
print(targets, "\n")

sm = C.ops.softmax(computed).eval()  # apply softmax to computed values
print("\nSoftmax applied to computed = ")
print(sm)

N = len(targets)  # 4
n = len(targets[0])  # 3

sum_se = 0.0
for i in range(N):  # each item
  for j in range(n):
    err = (targets[i,j] - sm[i,j]) * (targets[i,j] - sm[i,j])
    sum_se += err  # accumulate
mean_se = sum_se / N
print("\nMean squared error from scratch = %0.4f" % mean_se)

mean_se = C.losses.squared_error(sm, targets).eval() / 4.0
print("\nMean squared error from CNTK = %0.4f" % mean_se)

sum_cee = 0.0
for i in range(N):  # each item
  for j in range(n):
    err = -np.log(sm[i,j]) * targets[i,j]
    sum_cee += err  # accumulate
mean_cee = sum_cee / N
print("\nMean cross-entropy error w/ softmax from scratch = %0.4f" %
mean_cee)

sum_cee = 0.0
for i in range(N):
  err = C.losses.cross_entropy_with_softmax(computed[i].reshape(1,3), \
    targets[i].reshape(1,3)).eval()
  sum_cee += err
mean_cee = sum_cee / N
print("\nMean cross-entropy error w/ softmax from CNTK = %0.4f" %
mean_cee)
```

The demo program first sets up four target items from a hypothetical training dataset:

```
targets = np.array([[1, 0, 0],
                    [0, 1, 0],
                    [0, 0, 1],
                    [1, 0, 0]], dtype=np.float32)
```

You can imagine this corresponds to four 1-of-N encoded class label items. Next, the demo sets up hypothetical associated raw (before **softmax**) computed output values:

```
computed = np.array([[7.0, 2.0, 1.0],
                     [1.0, 9.0, 6.0],
                     [2.0, 1.0, 5.0],
                     [4.0, 7.0, 3.0]], dtype=np.float32)
```

Notice that the row values do not sum to 1.0. Next, the demo applies **softmax** to the raw computed output values:

```
sm = C.ops.softmax(computed).eval()
print("\nSoftmax applied to computed = ")
print(sm)
```

The results are:

```
[[ 0.9909  0.0067  0.0025]
 [ 0.0003  0.9523  0.0474]
 [ 0.0466  0.0171  0.9362]
 [ 0.0466  0.9362  0.0171]]
```

Notice that each row now sums to 1.0, and that the smallest-to-largest ordering is maintained. The demo program calculates both mean squared error and mean cross-entropy error from scratch, using just Python, and then using CNTK functions. The point is that in most situations you'll want to use the built-in CNTK functions, but if you need a custom error metric, you can implement it without too much difficulty.

In order to understand the code, it's useful to see how mean squared error and mean cross-entropy error are calculated by hand. The mean squared error could be calculated by hand as:

```
[ (0.9909 - 1)^2 + (0.0067 - 0)^2 + (0.0025 - 0)^2 +
  (0.0003 - 0)^2 + (0.9523 - 1)^2 + (0.0474 - 0)^2 +
  (0.0466 - 0)^2 + (0.0171 - 0)^2 + (0.9362 - 1)^2 +
  (0.0466 - 1)^2 + (0.9362 - 0)^2 + (0.0171 - 0)^2 ] / 4

= (0.0001 + 0.0045 + 0.0065 + 1.7857) / 4
= 1.7969 / 4
= 0.4492
```

In words, you compute the squared error for each item, add the squared error terms, and divide by the number of items. The demo prepares calculating mean squared error from scratch:

```
N = len(targets)  # 4
n = len(targets[0])  # 3
```

Then the demo walks through each row, using **i** as an index, and computes the squared error for the row as the sum of squared differences between the target values and the softmax values using **j** as an index. Then it accumulates the sum, and divides the accumulated sum by 4:

```
sum_se = 0.0
for i in range(N):  # each item
  for j in range(n):
    err = (sm[i,j] - targets[i,j]) * (sm[i,j] - targets[i,j])
    sum_se += err  # accumulate
mean_se = sum_se / N
print("\nMean squared error from scratch = %0.4f" % mean_se)
```

The output of this part of the demo program matches the result calculated by hand:

```
Mean squared error from scratch = 0.4492
```

Notice that if all softmax output values exactly equal the target values, mean squared error will be zero. The largest possible squared error for an item is 1.0, and therefore, the largest possible mean squared error for a set of items is 1.0. Next, the demo program calculates mean squared error using the built-in CNTK **`squared_error()`** function, like this:

```
mean_se = C.losses.squared_error(sm, targets).eval() / 4.0
print("\nMean squared error from CNTK = %0.4f" % mean_se)
```

As you'd expect, the output matches the previous computation and output:

```
Mean squared error from CNTK = 0.4492
```

Notice that **`squared_error()`** can operate on the full matrices rather than a row at a time, and that because **`squared_error()`** is type CNTK **`Function`**, you have to call the **`eval()`** function.

The squared error can be calculated for any two vectors of the same length. Cross entropy is a specialized form of error that applies only to two sets of probabilities, in other words, two vectors that have the same length, and that have values that sum to 1.0. If you have a set of predicted probabilities and a set of actual probabilities, the cross-entropy error is calculated as the negative of the sum of the products of the log of the predicted times the actual.

For example, suppose a set of predicted probabilities is (0.10, 0.60, 0.30), and the corresponding set of actual probabilities is (0.15, 0.50, 0.35). The cross-entropy error is:

```
CEE = - [ log(0.10) * 0.15 + log(0.60) * 0.50 + log(0.30) * 0.35 ]
        = - ( -0.3453 + -0.2554 + -0.4214 )
        = 1.0222
```

In classification problems, the softmax output values will sum to 1.0 and can be interpreted as probabilities, and a target vector will consist of one 1 value, and the rest 0 values, so it too can be interpreted as a probability set. For example, the first softmax item in the demo program is (0.9909, 0.0067, 0.0025), and the corresponding target vector is (1, 0, 0). Cross-entropy error is:

```
CEE = - [ log(0.9909) * 1 + log(0.0067) * 0 + log(0.0025) * 0 ]
        = - ( -0.0091 + 0 + 0 )
        = 0.0091
```

Notice that for classification problems, all but one term of the cross-entropy error will drop out. The demo program computes the cross-entropy error from scratch for the hypothetical classification data with these statements:

```
sum_cee = 0.0
for i in range(N):  # each item
  for j in range(n):
    err = -np.log(sm[i,j]) * targets[i,j]
    sum_cee += err  # accumulate
mean_cee = sum_cee / N
print("\nMean cross-entropy error w/ softmax from scratch = %0.4f" %
mean_cee)
```

The demo code follows the hand calculation closely. The output is:

```
Mean cross-entropy error w/ softmax from scratch = 0.7975
```

The demo program computes mean cross-entropy error in a slightly different way:

```
sum_cee = 0.0
for i in range(N):  # each item
  err = C.losses.cross_entropy_with_softmax(computed[i].reshape(1,3), \
    targets[i].reshape(1,3)).eval()
  sum_cee += err
mean_cee = sum_cee / N
print("\nMean cross-entropy error w/ softmax from CNTK = %0.4f" % mean_cee)
```

The output is:

```
Mean cross-entropy error w/ softmax from CNTK = 0.7975
```

Unlike **squared_error()**, which can easily compute over multiple items (in a matrix), **cross_entropy_with_softmax()** works more easily with one item at a time. Additionally, and somewhat surprisingly, CNTK v2.3 does not have a plain, cross-entropy error function—there's only **cross_entropy_with_softmax()**. This means the function expects raw, pre-softmax values. Notice the function accepts **computed** rather than **sm**.

The **reshape()** function is a required detail. Because **computed** is a 4×3 matrix, **computed[i]** is the **i**th row which is an array. The **cross_entropy_with_softmax()** function requires a matrix, so, the array is cast to a 1×3 matrix.

To summarize, when you train a CNTK model you must specify which error function to use. The two main options are **squared_error()** and **cross_entropy_with_softmax()**. For classification problems with three or more class labels, either error function can be used, but **cross_entropy_with_softmax()** is recommended. For binary classification problems, if you set up a model with a single output node, you should use the special **binary_cross_entropy()** function, but if you set up a model with two output nodes, **cross_entropy_with_softmax()** is recommended. If you do use **crosss_entropy_with_softmax()**, you must remember not to explicitly apply **softmax** so that it isn't applied twice. For all regression problems, **squared_error()** is recommended.

Stochastic gradient descent

Suppose you are training a model, and one of the weights currently has a value of 2.58. For a given training data item, you feed the input values to the model and compute the output values. Then you use an error function to determine something called the gradient, which is a number that indicates how far off, and in what direction, the computed output value is compared to the known correct output value. Suppose the gradient is +1.84. If you adjust (usually by subtraction) the weight by a fraction, called the learning rate (often indicated by Greek letter eta which looks like a script lower case "n") then the computed output values will become slightly closer to the target output values. Suppose the learning rate is 0.01, then the new weight value is 2.58 - (1.84)(0.01) = 2.58 - 0.0184 = 2.5616.

Stochastic gradient descent is a complex topic, but CNTK handles most of the details for you. Typical CNTK code looks like:

```
batch_size = 10
learn_rate = 0.01
learner = C.sgd(nnet.parameters, learn_rate)
trainer = C.Trainer(nnet, (tr_loss, tr_clas), [learner])
. . .
curr_batch = rdr.next_minibatch(batch_size, input_map=my_input_map)
trainer.train_minibatch(curr_batch)
```

When stochastic gradient descent is used to train a neural network, the technique is often called *back-propagation*. In addition to the basic **sgd()** function, CNTK has several advanced variations, including **adam()** ("adaptive moment estimation"), and **rmsprop()** ("root mean squared propagation").

When using any form of stochastic gradient descent, you must choose between online training (update weights after processing one training data item), batch training (update weights after processing all training items), or mini-batch training (update weights after processing a subset of all training items). In high-level pseudo-code, online training is:

```
loop max_iterations times
  for-each training item
    compute output values
    compute gradients
    use gradients to update each weight and bias value
  end-for
end-loop
```

In pseudo-code, batch training is:

```
loop max_iterations times
  set all accumulated_gradients = 0
  for-each training item
    compute output values
    compute gradients
    accumulate gradients
  end-for
  use accumulated gradients to update each weight and bias value
end-loop
```

In the early days of neural networks, batch training was most often used. Then online training became more common. Currently, mini-batch training is most often used. The CNTK library functions make it easy for you to try all three approaches.

Setting a value for the learning rate is mostly a matter of trial and error. A very small learning rate can make training very, very slow. A too-large learning rate can make training fail entirely. One technique for dealing with the learning rate magnitude problem is called *momentum*. Using momentum adds a boost to each weight update. What this means is you can use a relatively small learning rate, and the rate will effectively increase automatically for you. CNTK has a **`momentum_sgd()`** function for basic momentum, and a **`nesterov()`** function, which is an advanced version of momentum.

In the previous pseudo-code example, notice that training is an iterative process. You might guess that the more training iterations you can perform, the better. However, this is not always the case. If you over-train a model, you might get model overfitting. This is a situation where you have very low error (and high accuracy for a classification model) on the training data, but when presented with new, previously unseen data, your model predicts poorly.

There are several ways to reduce the likelihood of overfitting. As it turns out, overfitting is often associated with weights and bias values that have large magnitudes. Regularization is a technique that limits the magnitudes of weights and biases. There are two main forms of regularization, named L1 and L2. L1 regularization tends to lead to some weight values that are close to zero (effectively dropping their associated input nodes), and L2 regularization tends to lead to all weights and bias values being small, but few very close to zero. You can supply L1 or L2 regularization information as a parameter, for example:

```
learner = C.sgd(nnet.parameters, learn_rate, l2_regularization_weight=0.10)
```

Another technique used to limit overfitting is called *train-validate-test*. Using normal train-test, you divide all available training data into a training set (typically about 80% of available data items) and a test set (the remaining 20%). You use the training data to train your model, then feed the test dataset to the trained model. The accuracy of the trained model on the test data is a very rough estimate of the accuracy you can expect on new, previously unseen data.

With the train-validate-test technique, you divide all available training data into three sets: a training set (usually about 80% of the items), a validation set (10% of the items), and a test set (10% of the items). You train your model using the training dataset, but every few training iterations, you feed the validation data to the model. At some point, the error on the validation data may start to increase rather than decrease, indicating that model overfitting may be occurring. You can stop training, then evaluate the model on the test data. The train-validate-test technique is a meta-heuristic, not a well-defined algorithm. CNTK does not have any built-in functions that directly support train-validate-test.

The CNTK library organization

Technically, CNTK is a Python package. CNTK is organized into 13 high-level sub-packages and eight sub-modules (smaller sub-packages). The 10 most frequently-used of these are shown in Table 3-1.

Table 3-1: Most Important CNTK Packages

Package	Description
cntk.io	Functions for reading data, ex: **next_minibatch()**
cntk.layers	High-level functions for creating neural layers, ex: **Dense()**
cntk.learners	Functions for training, ex: **sgd()**
cntk.losses	Functions to measure training error, ex: **squared_error()**
cntk.metrics	Functions to measure model error, ex: **classification_error()**
cntk.ops	Low-level functions, ex: **input_variable()** and **tanh()**
cntk.random	Functions to generate random numbers, ex: **normal()** and **uniform()**
cntk.train	Training functions, ex: **train_minibatch()**
cntk.initializer	Model parameter initializers, ex: **normal()** and **uniform()**
cntk.variables	Low-level constructs, ex: **Parameter()** and **Variable()**

Notice that some function names appear in more than one package; for example, both the cntk.random and cntk.initializer packages have **normal()** and **uniform()** functions. These duplications are relatively rare, and in practice you shouldn't have any trouble distinguishing between duplicate function names.

When writing CNTK with Python code, your indispensable resource is the API documentation. Because CNTK v2 is so new, there are relatively few good code examples available online. However, this will change as the use of CNTK increases.

Writing CNTK code is not easy, and you'll run into problems. When I get an error, as a general rule of thumb, my first action is to determine the type of the offending object(s), for example:

```
t = type(foo)
print(t)
```

Then I'll look up that type in the API reference. One of the most common sources of errors in CNTK programs is an incorrect shape of a matrix or array. Most CNTK objects are NumPy multidimensional arrays. You can use the NumPy **`shape`** property to determine an object's shape, for example:

```
shp = foo.shape
print(shp)
```

In some rare situations, you may need to change the shape of an object. To do so, you can use the NumPy **`reshape()`** function. Note: throughout this e-book, I use the terms *function* and *method*, and the terms *variable* and *object*, more or less interchangeably.

Exercise

Using the program in Code Listing 3-1 as a guide, write a CNTK program that creates a 3-2-4 neural network with logistic sigmoid hidden layer activation and softmax output layer activation. Set the input-to-hidden weights to:

```
[[0.01, 0.02],
 [0.03, 0.04],
 [0.05, 0.06]]
```

Set the hidden-to-output weights to:

```
[[0.07, 0.08, 0.09, 0.10],
 [0.11, 0.12, 0.13, 0.14]]
```

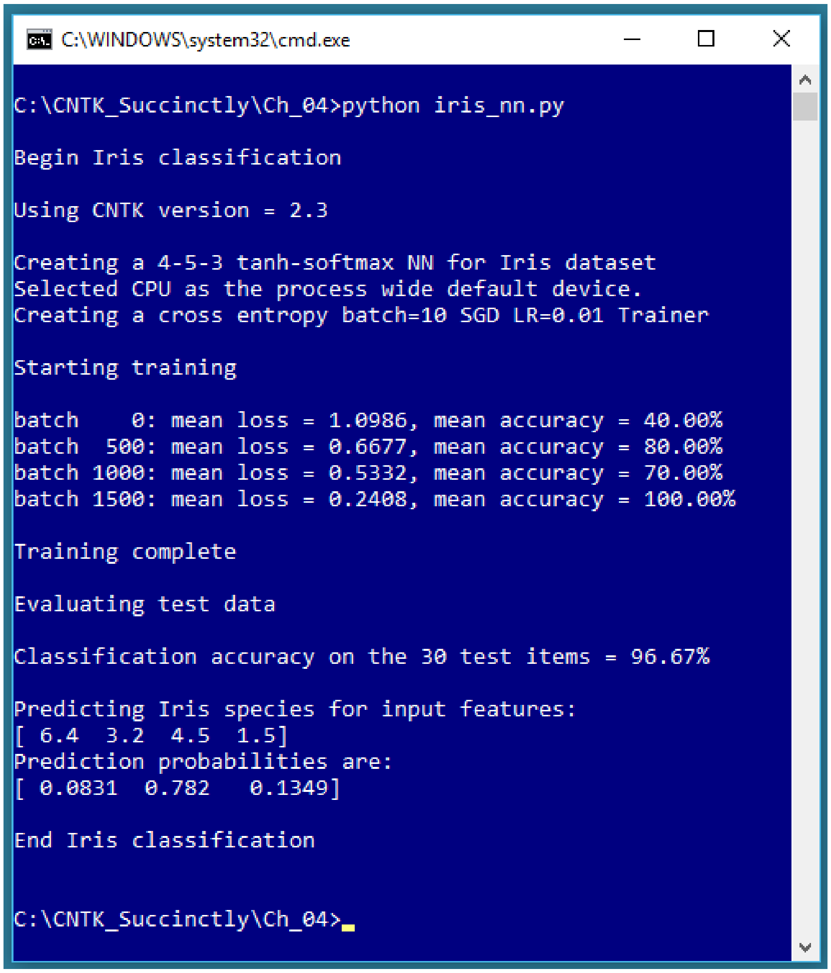
Set the hidden node bias values to `[0.15, 0.16]` and the output node bias values to `[0.17, 0.18, 0.19, 0.20]`.

Compute the output values for input values `= [2.0, 4.0, 6.0]`.

Answer: `[0.2414, 0.2470, 0.2528, 0.2588]`

Chapter 4 Neural Network Classification

This chapter explains how to perform classification using CNTK. Take a look at Figure 4-1 to see where the chapter is headed. The problem is to predict the species of an iris flower (*setosa*, *versicolor*, or *virginica*) based on four predictor variables (sepal length, sepal width, petal length, and petal width). A sepal is a leaf-like structure.



```
C:\CNTK_Succinctly\Ch_04>python iris_nn.py

Begin Iris classification

Using CNTK version = 2.3

Creating a 4-5-3 tanh-softmax NN for Iris dataset
Selected CPU as the process wide default device.
Creating a cross entropy batch=10 SGD LR=0.01 Trainer

Starting training

batch    0: mean loss = 1.0986, mean accuracy = 40.00%
batch  500: mean loss = 0.6677, mean accuracy = 80.00%
batch 1000: mean loss = 0.5332, mean accuracy = 70.00%
batch 1500: mean loss = 0.2408, mean accuracy = 100.00%

Training complete

Evaluating test data

Classification accuracy on the 30 test items = 96.67%

Predicting Iris species for input features:
[ 6.4  3.2  4.5  1.5]
Prediction probabilities are:
[ 0.0831  0.782   0.1349]

End Iris classification


C:\CNTK_Succinctly\Ch_04>
```

Figure 4-1: Iris Dataset Classification Using CNTK

Predicting a categorical value is often called *classification*, as opposed to predicting a numeric value, which is often called a *regression* problem. Note that logistic regression, explained in Chapter 2, is both a regression technique (the value to predict is a numeric probability between 0.0 and 1.0) and a classification technique (the prediction probability is mapped to one of two categorical values, such as “male” or “female”).

The example program creates a 4-5-3 neural network, that is, one with four input nodes (one for each predictor value), five hidden processing nodes (the number of hidden nodes must be determined by trial and error), and three output nodes (because there are three possible species). The neural network uses tanh activation in the hidden nodes and softmax activation in the output nodes.

The underlying error function used for training is cross-entropy, instead of the main alternative, squared error. The neural network is trained on 120 training items using the mini-batch approach, 10 training items at a time, and 2,000 iterations. During training, the average error/loss gradually decreases and the classification accuracy mostly increases, suggesting that training is working.

After training completed, the trained model was applied to 30 previously unseen test data items. The model scored 96.67% accuracy (29 of 30 correct predictions). The program concluded by using the trained model to predict the species for predictor values (6.4, 3.2, 4.5, 1.5). The predicted probabilities were (0.0831, 0.7820, 0.1349), which maps to (0, 1, 0), which is the encoding for the *Iris versicolor* species.

Preparing the Iris Data training and test files

Fisher's Iris Data, created in the 1930s, is arguably the most well-known dataset in machine learning. There are 150 data items. The raw data looks like the following:

```
5.1  3.5  1.4  0.2  setosa
4.9  3.0  1.4  0.2  setosa
. . .
7.0  3.2  4.7  1.4  versicolor
6.4  3.2  4.5  1.5  versicolor
. . .
6.3  3.3  6.0  2.5  virginica
5.8  2.7  5.1  1.9  virginica
```

The first four values on each line are sepal length, sepal width, petal length, and petal width. The last value on each line is the species. There are 50 of items each of the three species. The example program uses data in a format that can be used efficiently by CNTK. That data looks like the following:

```
|attribs 5.1 3.5 1.4 0.2 |species 1 0 0
|attribs 4.9 3.0 1.4 0.2 |species 1 0 0
. . .
|attribs 7.0 3.2 4.7 1.4 |species 0 1 0
|attribs 6.4 3.2 4.5 1.5 |species 0 1 0
. . .
|attribs 6.3 3.3 6.0 2.5 |species 0 0 1
|attribs 5.8 2.7 5.1 1.9 |species 0 0 1
```

The **|attribs** and **|species** tags mark the start of the feature values and the class label values, respectively. Each line is space-delimited. The feature values don't need to be normalized because they're all roughly in the same magnitude. The species label is 1-of-N (also called one-hot) encoded.

You can use whatever tag names you wish. And you can add item ID numbers or comments using the tag mechanism, for example:

```
|ID 001 |attribs 5.1 3.5 1.4 0.2 |species 1 0 0 |# setosa
|ID 002 |attribs 4.9 3.0 1.4 0.2 |species 1 0 0 |#
. . .
|ID 041 |attribs 7.0 3.2 4.7 1.4 |species 0 1 0 |# versicolor
. . .
```

The 150-item dataset was split into a 120-item training set (the first 40 of each species) and a 30-item hold-out test set (the remaining 10 of each species). The CNTK-format training and test data can be found in the Appendix to this e-book.

Because there are four features, it's not feasible to graph the data. However, you can get a rough idea of the data's structure by examining the two-dimensional graph, based on just sepal length and petal length, shown in Figure 4-2.

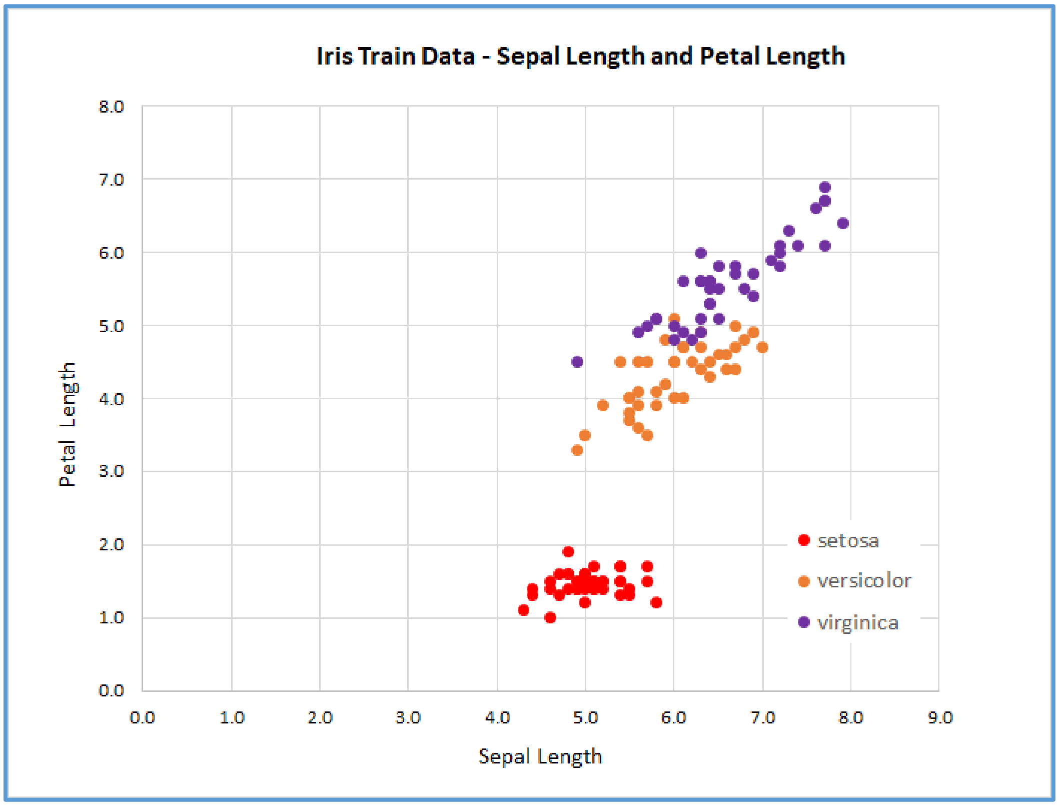


Figure 4-2: Iris Data

The classification program

The code for the program that generated the output shown in Figure 4-1 is presented in Code Listing 4-1. The program begins by defining a **`create_reader()`** helper function that can process data files in CNTK format:

```
def create_reader(path, input_dim, output_dim, rnd_order, sweeps):
  # rnd_order -> usually True for training
  # sweeps -> usually C.io.INFINITELY_REPEAT for training OR 1 for eval
  x_strm = C.io.StreamDef(field='attribs', shape=input_dim, is_sparse=False)
  y_strm = C.io.StreamDef(field='species', shape=output_dim, is_sparse=False)
  streams = C.io.StreamDefs(x_src=x_strm, y_src=y_strm)
  deserial = C.io.CTFDeserializer(path, streams)
  mb_src = C.io.MinibatchSource(deserial, randomize=rnd_order,
max_sweeps=sweeps)
  return mb_src
```

There is a lot going on in the helper function, but for most classification problems you can think of the code as a boilerplate, and the only thing you'll need to change is the references to the tag names in your CNTK file, **`attribs`** and **`species`** in this example.

Code Listing 4-1: Iris Flower Classification

```
# iris_nn.py
# CNTK 2.3 with Anaconda 4.1.1 (Python 3.5, NumPy 1.11.1)

# Use a one-hidden layer simple NN with 5 hidden nodes
# iris_train_cntk.txt - 120 items (40 each class)
# iris_test_cntk.txt - remaining 30 items

import numpy as np
import cntk as C

def create_reader(path, input_dim, output_dim, rnd_order, sweeps):
  # rnd_order -> usually True for training
  # sweeps -> usually C.io.INFINITELY_REPEAT for training OR 1 for eval
  x_strm = C.io.StreamDef(field='attribs', shape=input_dim,
is_sparse=False)
  y_strm = C.io.StreamDef(field='species', shape=output_dim,
is_sparse=False)
  streams = C.io.StreamDefs(x_src=x_strm, y_src=y_strm)
  deserial = C.io.CTFDeserializer(path, streams)
  mb_src = C.io.MinibatchSource(deserial, randomize=rnd_order,
max_sweeps=sweeps)
  return mb_src
```

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

def main():
  print("\nBegin Iris classification \n")
  print("Using CNTK version = " + str(C.__version__) + "\n")

  input_dim = 4
  hidden_dim = 5
  output_dim = 3

  train_file = ".\\Data\\iris_train_cntk.txt"
  test_file = ".\\Data\\iris_test_cntk.txt"

  # 1. create network
  X = C.ops.input_variable(input_dim, np.float32)
  Y = C.ops.input_variable(output_dim, np.float32)

  print("Creating a 4-5-3 tanh-softmax NN for Iris dataset ")
  with C.layers.default_options(init=C.initializer.uniform(scale=0.01,
seed=1)):
    hLayer = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer')(X)
    oLayer = C.layers.Dense(output_dim, activation=None,
     name='outLayer')(hLayer)
  nnet = oLayer
  model = C.ops.softmax(nnet)

  # 2. create learner and trainer
  print("Creating a cross entropy batch=10 SGD LR=0.01 Trainer \n")
  tr_loss = C.cross_entropy_with_softmax(nnet, Y)  # not model!
  tr_clas = C.classification_error(nnet, Y)

  max_iter = 2000
  batch_size = 10
  learn_rate = 0.01
  learner = C.sgd(nnet.parameters, learn_rate)
  trainer = C.Trainer(nnet, (tr_loss, tr_clas), [learner])

  # 3. create reader for train data
  rdr = create_reader(train_file, input_dim, output_dim,
    rnd_order=True, sweeps=C.io.INFINITELY_REPEAT)
  iris_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }
  # 4. train
  print("Starting training \n")
  for i in range(0, max_iter):
    curr_batch = rdr.next_minibatch(batch_size, input_map=iris_input_map)
    trainer.train_minibatch(curr_batch)
    if i % 500 == 0:
      mcee = trainer.previous_minibatch_loss_average
      macc = (1.0 - trainer.previous_minibatch_evaluation_average) * 100
      print("batch %4d: mean loss = %0.4f, mean accuracy = %0.2f%% " \
        % (i, mcee, macc))
  print("\nTraining complete")

  # 5. evaluate model using test data
  print("\nEvaluating test data \n")

  rdr = create_reader(test_file, input_dim, output_dim,
    rnd_order=False, sweeps=1)
  iris_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }
  num_test = 30
  all_test = rdr.next_minibatch(num_test, input_map=iris_input_map)
  acc = (1.0 - trainer.test_minibatch(all_test)) * 100
  print("Classification accuracy on the 30 test items = %0.2f%%" % acc)

  # (could save model here - see text)

  # 6. use trained model to make prediction
  np.set_printoptions(precision = 1)
  unknown = np.array([[6.4, 3.2, 4.5, 1.5]], dtype=np.float32)  # (0 1 0)
  print("\nPredicting Iris species for input features: ")
  print(unknown[0])

  # pred_prob = model.eval({X: unknown})
  pred_prob = model.eval(unknown)  # simple form works too
  np.set_printoptions(precision = 4, suppress=True)
  print("Prediction probabilities are: ")
  print(pred_prob[0])

  print("\nEnd Iris classification \n ")

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

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

Program execution begins by setting up the architecture arguments for the neural network, and the location of the data files:

```
def main():
  print("\nBegin Iris classification \n")
  print("Using CNTK version = " + str(C.__version__) + "\n")
  input_dim = 4
  hidden_dim = 5
  output_dim = 3
  train_file = ".\\Data\\iris_train_cntk.txt"
  test_file = ".\\Data\\iris_test_cntk.txt"
. . .
```

The dimensions for the neural network input nodes and output nodes are determined by the structure of your data, but the number of hidden nodes is a free parameter (often called a *hyperparameter*), and must be determined by trial and error. The data files are placed in a Data subdirectory. When working with CNTK, a standard approach is to have a root project directory for code, with Data and Models subdirectories.

Next, the classification program creates the untrained neural network:

```
  # 1. create network
  X = C.ops.input_variable(input_dim, np.float32)
  Y = C.ops.input_variable(output_dim, np.float32)

  print("Creating a 4-5-3 tanh-softmax NN for Iris dataset ")
  with C.layers.default_options(init=C.initializer.uniform(scale=0.01,
seed=1)):
    hLayer = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer')(X)
    oLayer = C.layers.Dense(output_dim, activation=None,
     name='outLayer')(hLayer)
  nnet = oLayer  # nnet = C.ops.alias(oLayer)
  model = C.ops.softmax(nnet)
```

The hidden layer nodes and the output layer nodes are created using the **Dense()** function, which fully connects all nodes in one layer to the other layer. The syntax is rather unusual. The **X** object, which is type CNTK **Variable**, acts as input to the **hLayer** object (the hidden layer), and the **hLayer** object acts as input to the **oLayer** object (the output layer).

The hidden layer uses **tanh** activation, which often (but not always) performs a bit better than the logistic sigmoid, **C.ops.sigmoid()** function. If you have experience with other deep learning libraries, you'll likely be surprised that the output layer uses no activation function, or stated equivalently, uses the identity function. Many other deep learning libraries have a plain vanilla cross-entropy error function for training. Somewhat unusually, as of version 2.3, CNTK has a **cross_entropy_with_softmax()** function, but does not have a plain cross-entropy function. So, during training you must use **cross_entropy_with_softmax()** which expects raw, un-softmaxed values. If you apply softmax to the output layer, then during training softmax will be applied twice. This doesn't break training, but for interesting technical reasons, double application of softmax often slows training significantly.

This explains why you don’t want to apply **softmax** to the output layer nodes. However, when it comes time to use the neural network, for evaluation or to make predictions, you typically want to apply **softmax**. So, the example program creates an **nnet** object, which is used during training, and a **model** object, which is used for evaluation or prediction.

The Python **with** statement is a shortcut that can be used to apply specified values to multiple layers of the network. In this case, the **init** parameter value means that all weights are initialized to a uniform random value between -0.01 and +0.01 inclusive. The example does not use the related **init_bias** parameter, so all biases are initialized to 0.0. Neural networks are often highly sensitive to initialization values, so if training fails, one of the first things to try is a different initialization algorithm. Some common alternatives to **uniform()** initialization are **normal()**, **glorot_normal()**, and **glorot_uniform()**.

After creating the dual untrained models, the example program sets up a **Learner** algorithm object, and then uses it to create a **Trainer** training object:

```
  # 2. create learner and trainer
  print("Creating a cross-entropy batch=10 SGD LR=0.01 Trainer \n")
  tr_loss = C.cross_entropy_with_softmax(nnet, Y)  # not model!
  tr_clas = C.classification_error(nnet, Y)
  max_iter = 2000

  batch_size = 10
  learn_rate = 0.01
  learner = C.sgd(nnet.parameters, learn_rate)
  trainer = C.Trainer(nnet, (tr_loss, tr_clas), [learner])
```

As explained earlier, the **nnet** object, not the **model** object, is trained. The **tr_loss** object specifies cross-entropy error rather than the main alternative, squared error, for training. The **tr_clas** object is created so that classification error (percentage of incorrect predictions) can be automatically computed during training.

The **learner** object is a stochastic gradient descent object created using the **sgd()** function with a constant, fixed learning rate, which is the simplest possible algorithm. The CNTK library has a large number of sophisticated learning algorithms, many of which are very complex. For example, the using the CNTK **fsadagrad()** (“specialized adaptive gradient”) function, the learning algorithm could be coded as:

```
  max_iter = 2000
  batch_size = 10
  lr_schedule = C.learning_parameter_schedule_per_sample([(1000, 0.05), (1,
0.01)])
  mom_sch = C.momentum_schedule([(100, 0.99), (0, 0.95)], batch_size)
  learner = C.fsadagrad(nnet.parameters, lr=lr_schedule, momentum=mom_sch)
  trainer = C.Trainer(nnet, (tr_loss, tr_clas), [learner])
```

When advanced learner algorithms work, they can work very well. But you have many additional free parameters to deal with. As a general rule of thumb, I recommend that you try **sgd()** first, and resort to exotic learning algorithms only when you have a good reason to do so (such as repeated training failures, or known research results).

After creating the **Trainer** object, the example program calls the program-defined **create_reader()** function to create a **Reader** for the training data:

```
# 3. create reader for train data
rdr = create_reader(train_file, input_dim, output_dim,
  rnd_order=True, sweeps=C.io.INFINITELY_REPEAT)
iris_input_map = {
  X : rdr.streams.x_src,
  Y : rdr.streams.y_src
}
```

When training a neural network, you want to visit the training data in a random order so that the training doesn't fall into some sort of oscillating pattern and stall. The sweeps parameter is set to the CNTK integer constant **INFINITELY_REPEAT** (18,446,744,073,709,551,615) so that training items can be visited multiple times.

The **iris_input_map** object is a dictionary collection, which is easy to botch. Notice that there are CNTK **Variable** objects (**X** and **Y**) on the left side of the colon, and assignment receivers from the call to the **StreamDefs()** function (**x_src** and **y_src**) on the right-hand side of the colon. As before, however, you can consider this code boilerplate, and just replace the map name (**iris_input_map**) with something pertinent to your classification problem.

Next, the example classification program trains the network like so:

```
# 4. train
print("Starting training \n")
for i in range(0, max_iter):
  curr_batch = rdr.next_minibatch(batch_size, input_map=iris_input_map)
  trainer.train_minibatch(curr_batch)
  if i % 500 == 0:
    mcee = trainer.previous_minibatch_loss_average
    macc = (1.0 - trainer.previous_minibatch_evaluation_average) * 100
    print("batch %4d: mean loss = %0.4f, accuracy = %0.2f%% " \
      % (i, mcee, macc))
print("\nTraining complete")
```

The **next_minibatch()** function fetches a batch of training items from the training data file (10 in this case), and knows where the feature values are, and where the class labels values are from the information supplied by the **iris_input_map** object. The call to **train_minibatch()** is almost too simple—all the real work was done in the preparation to the call.

It's important to monitor the error/loss during training because training failure is very common, and so you want to catch it as early as possible. The example program displays the average cross-entropy error on the just-used mini-batch of 10 training items, and the classification accuracy of those 10 items, every 500 iterations/mini-batches. The CNTK library has a built-in **ProgressPrinter** class in the logging package. Many of my colleagues like and use a **ProgressPrinter** object, but I prefer to create logging messages using custom code.

After training using the 120-item training data has finished, the example program evaluates the classification accuracy of the trained model on the held-out 30-item test data:

```
  # 5. evaluate model using test data
  print("\nEvaluating test data \n")
  rdr = create_reader(test_file, input_dim, output_dim,
    rnd_order=False, sweeps=1)
  iris_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }
  num_test = 30
  all_test = rdr.next_minibatch(num_test, input_map=iris_input_map)
  acc = (1.0 - trainer.test_minibatch(all_test)) * 100
  print("Classification accuracy on the 30 test items = %0.2f%%" % acc)
```

The **Reader** for the test data specifies the location of the test data file and passes **False** to the **rnd_order** parameter because there's no need to process the test data in a random order to determine classification accuracy. The **sweeps** parameter is set to 1 because you only want to visit each test item once.

The call to **next_minibatch()** will return all test items because the **num_test** variable is set to 30, the number of test items. The **test_minibatch()** function returns the classification error, which in my opinion isn't quite as natural a metric as classification accuracy, so the example program computes accuracy from error, and prints the accuracy using a 0% to 100% format, in this case 96.67% (29 of 30 correct).

It's sometimes useful to see exactly which data items were predicted correctly and incorrectly. You can walk through the test data and get detailed information like this:

```
  for i in range(0, num_test):
    curr_item = rdr.next_minibatch(1, input_map=iris_input_map)
    x = curr_item[X].asarray()[0]  # (1,4)
    y = curr_item[Y].asarray()[0]  # (1,3)
    p_prob = model.eval(x)  # prediction probabilities
    p_class = np.argmax(p_prob[0])  # predicted class 0, 1, or 2
    a_class = np.argmax(y)  # actual class 0, 1, or 2
    print(x, end="")
    if p_class == a_class:  # if predicted class == actual class
      print("  correct")
    else:
      print("  WRONG")
```

The code is a bit trickier than it might appear at first glance. The first argument to **next_minibatch()** is 1, so the reader will fetch one test item at a time. The **curr_item** object is type **dict**, and you must pass the **X** or **Y** object as the key, then cast the result to a (three-dimensional) array, and then peel away the first dimension to get a 1×4 or 1×3 matrix.

After evaluating the accuracy of the trained model, the example program shows how to make a prediction for new, previously unseen data:

```
# 6. use trained model to make prediction
np.set_printoptions(precision = 1, suppress=True)
unknown = np.array([[6.4, 3.2, 4.5, 1.5]], dtype=np.float32)  # (0 1 0)
print("\nPredicting Iris species for input features: ")
print(unknown[0])
pred_prob = model.eval(unknown)
np.set_printoptions(precision = 4, suppress=True)
print("Prediction probabilities are: ")
print(pred_prob[0])
```

Notice that **unknown** is a 1×4 matrix, as indicated by the double square brackets, rather than an array, because the **eval()** function expects a matrix as input. The return prediction probabilities result is a 1×3 matrix, so the example program displays row [0], the only row.

Saving and loading a trained model

Because the Iris dataset has only 150 items, training a neural network classifier model takes only a few seconds. However, in non-demo scenarios, training on a large dataset can take hours or even days. You'll usually want to save your trained model so you won't have to retrain from scratch.

You can save a trained model like this:

```
mdl = ".\\Models\\iris_nn.model"
model.save(mdl, format=C.ModelFormat.CNTKv2)
```

The first argument passed to **save()** is just a file name, possibly including a path. There is no required file extension, but **.model** is common and makes sense. The **format** parameter has default value **ModelFormat.CNTKv2**, and so could have been omitted. An alternative is to use the ONNX (Open Neural Network Exchange) format: **model.save(mdl, format=ONNX)**.

Recall that the training program created both a **nnet** object (where output is not softmaxed) and a **model** object (with softmaxed output). You'll normally want to save the softmaxed version of a trained model, but you can save the non-softmaxed network object if you wish.

Loading a saved CNTK model is very easy, as shown by the short program in Code Listing 4-2. All you have to do is use the **load()** function.

Code Listing 4-2: Loading a Saved Model

```
# iris_load.py
# CNTK 2.3

import numpy as np
import cntk as C

print("Loading saved Iris model")
model = C.ops.functions.Function.load(".\\Models\\iris_nn.model")
```

```
np.set_printoptions(precision = 1, suppress=True)
unknown = np.array([[6.4, 3.2, 4.5, 1.5]], dtype=np.float32)  # (0 1 0)
print("\nPredicting Iris species for input features: ")
print(unknown[0])

pred_prob = model.eval(unknown)
np.set_printoptions(precision = 4, suppress=True)
print("Prediction probabilities are: ")
print(pred_prob[0])

print("\nDone \n")
```

After loading a saved model, the model can be used exactly as if the model had just been trained. Notice that there's a bit of asymmetry in the calls to **`save()`** and **`load()`**; **`save()`** is a method on a **`Function`** object, and **`load()`** is a static method from the **`Function`** class.

Deep neural networks

There is no standard definition of a deep neural network, but in general the term means any neural network architecture that has two or more hidden layers. For example:

```
  with C.layers.default_options(init=C.initializer.uniform(scale=0.01,
seed=1)):
    hLayer1 = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer1')(X)
    hLayer2 = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer2')(hLayer1)
    oLayer = C.layers.Dense(output_dim, activation=None,
     name='outLayer')(hLayer2)
  nnet = C.ops.alias(oLayer)
  model = C.ops.softmax(nnet)
```

In theory, and loosely speaking, a neural network with just one hidden layer can compute any model that a network with two or more hidden layers can (subject to certain conditions). This is called the universal approximation theorem, or sometimes, the Cybenko theorem. But in practice, for complex datasets, using multiple hidden layers can sometimes produce a better predictive model. Note, however, that for relatively simple data, a deep neural network can sometimes actually generate a worse model than a network with a single hidden layer.

One of the pitfalls of deep neural networks is that they tend to be more prone to overfitting than networks with a single hidden layer. One common technique for reducing the likelihood of overfitting in a deep neural network is to insert one or more dropout layers. When dropout is applied, a specified proportion of nodes in a layer is randomly dropped, meaning that during training, on each iteration, the learning algorithm essentially pretends the dropped nodes don't exist. Exactly why dropout often prevents overfitting is a complex topic and not fully understood, but it's a standard technique with deep neural networks.

For example, the following code inserts a dropout layer between two hidden layers:

```
  with C.layers.default_options(init=C.initializer.uniform(scale=0.01,
seed=1)):
    hLayer1 = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer1')(X)
    dLayer = C.layers.Dropout(0.20, name='dropLayer')(hLayer1)
    hLayer2 = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer2')(dLayer)
    oLayer = C.layers.Dense(output_dim, activation=None,
     name='outLayer')(hLayer2)
  nnet = C.ops.alias(oLayer)
  model = C.ops.softmax(nnet)
```

In this example, the dropout is applied to the nodes in the first hidden layer.

Instead of manually chaining layers together, the CNTK library has a **Sequential()** function that can be used as a syntactic substitute. For example:

```
  model = C.layers.Sequential([
    C.layers.Dense(40, activation=C.ops.tanh),
    C.layers.Dense(20, activation=C.ops.sigmoid),
    C.layers.Dropout(0.25),
    C.layers.Dense(3, activation=C.ops.softmax)])
```

The CNTK library also has a **For()** method that can be used to programmatically generate multiple layers. Let me emphasize that **Sequential()** and **For()** are merely syntactic-sugar mechanisms, and so they don’t provide additional functionality.

Exercise

Using the program in Code Listing 4-1 as a guide, create, train, and evaluate a neural network classification prediction model for the Wheat Seeds dataset.

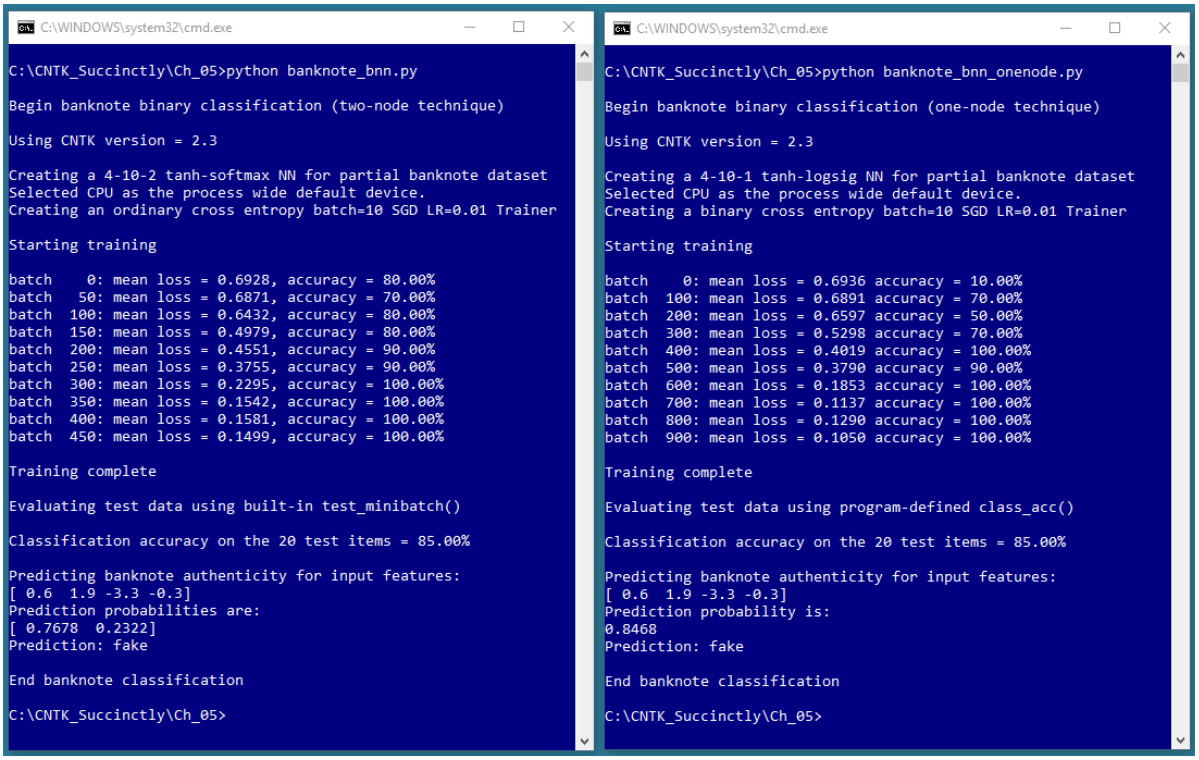
The Wheat Seeds dataset is a well-known benchmark. It can be found here. There are 210 items. Each item has seven predictor values followed by a variety of wheat, encoded as 1 = Kama, 2 = Rosa, 3 = Canadian. There are 70 of each variety. The tab-delimited raw data looks like:

```
15.26  14.84  0.871  5.763  3.312  2.221  5.22   1
14.88  14.57  0.8811 5.554  3.333  1.018  4.956  1
. . .
12.3   13.34  0.8684 5.243  2.974  5.637  5.063  3
```

When you create the training and test datasets, I suggest using 150 items for training and 60 items for testing. You will get better results if you normalize the values of the predictor variables. Don’t forget to encode the seed variety value.

Chapter 5 Neural Binary Classification

This chapter explains how to perform binary classification (the variable to predict can take one of just two possible values) using a neural network. Neural network binary classification is significantly more powerful than logistic regression binary classification, at the expense of a moderate increase in complexity.



```
C:\CNTK_Succinctly\Ch_05>python banknote_bnn.py

Begin banknote binary classification (two-node technique)

Using CNTK version = 2.3

Creating a 4-10-2 tanh-softmax NN for partial banknote dataset
Selected CPU as the process wide default device.
Creating an ordinary cross entropy batch=10 SGD LR=0.01 Trainer

Starting training

batch    0: mean loss = 0.6928, accuracy = 80.00%
batch   50: mean loss = 0.6871, accuracy = 70.00%
batch  100: mean loss = 0.6432, accuracy = 80.00%
batch  150: mean loss = 0.4979, accuracy = 80.00%
batch  200: mean loss = 0.4551, accuracy = 90.00%
batch  250: mean loss = 0.3755, accuracy = 90.00%
batch  300: mean loss = 0.2295, accuracy = 100.00%
batch  350: mean loss = 0.1542, accuracy = 100.00%
batch  400: mean loss = 0.1581, accuracy = 100.00%
batch  450: mean loss = 0.1499, accuracy = 100.00%

Training complete

Evaluating test data using built-in test_minibatch()

Classification accuracy on the 20 test items = 85.00%

Predicting banknote authenticity for input features:
[ 0.6  1.9 -3.3 -0.3]
Prediction probabilities are:
[ 0.7678  0.2322]
Prediction: fake

End banknote classification

C:\CNTK_Succinctly\Ch_05>
```

```
C:\CNTK_Succinctly\Ch_05>python banknote_bnn_onenode.py

Begin banknote binary classification (one-node technique)

Using CNTK version = 2.3

Creating a 4-10-1 tanh-logsig NN for partial banknote dataset
Selected CPU as the process wide default device.
Creating a binary cross entropy batch=10 SGD LR=0.01 Trainer

Starting training

batch    0: mean loss = 0.6936 accuracy = 10.00%
batch  100: mean loss = 0.6891 accuracy = 70.00%
batch  200: mean loss = 0.6597 accuracy = 50.00%
batch  300: mean loss = 0.5298 accuracy = 70.00%
batch  400: mean loss = 0.4019 accuracy = 100.00%
batch  500: mean loss = 0.3790 accuracy = 90.00%
batch  600: mean loss = 0.1853 accuracy = 100.00%
batch  700: mean loss = 0.1137 accuracy = 100.00%
batch  800: mean loss = 0.1290 accuracy = 100.00%
batch  900: mean loss = 0.1050 accuracy = 100.00%

Training complete

Evaluating test data using program-defined class_acc()

Classification accuracy on the 20 test items = 85.00%

Predicting banknote authenticity for input features:
[ 0.6  1.9 -3.3 -0.3]
Prediction probability is:
0.8468
Prediction: fake

End banknote classification

C:\CNTK_Succinctly\Ch_05>
```

Figure 5-1: Banknote Dataset Classification

If you’re new to machine learning, your first thought might be something like, “Binary classification using a neural network is no different than multi-class classification—there’s just two output nodes instead of three or more.” And you’d be mostly correct. However, the two side-by-side screenshots in Figure 5-1 indicate that there are two different techniques for neural network binary classification.

The two programs shown in Figure 5-1 work on the same raw data. The goal is to create a classification model that predicts whether a banknote (think a dollar bill or a euro) is a forgery or is authentic, based on four predictor variables: variance, skewness, kurtosis, and entropy. The program shown on the left uses essentially the same technique that we used in Chapter 4 to classify an iris flower as *setosa*, *versicolor*, or *virginica*. For neural network binary classification, this is called the two-node technique. The program shown on the right uses a significantly different approach, called the one-node technique.

Let me cut to the chase and state that in my opinion, the two-node technique is preferable to the one-node technique. But, for historical reasons, the neural network one-node technique is more common. You'll almost certainly encounter the one-node technique, and should understand how it works.

Recall that the iris data was encoded as *setosa* = (1, 0, 0), *versicolor* = (0, 1, 0), *virginica* = (0, 0, 1). For the two-node technique, a banknote is encoded as forgery = (1, 0), authentic = (0, 1). After training, the two-node model is fed inputs (0.6, 1.9, -3.3, -0.3), and the prediction probabilities are (0.7678, 0.2322). This maps to (1, 0), and so the prediction is forgery/fake. In short, the neural network two-node binary classification technique is essentially the same as multi-class neural network classification.

The program shown on the right of Figure 5-1 uses a neural network with just a single output node. As you'll see, this requires a change in the output layer activation function, a change in training error function, and the need for a program-defined classification accuracy function. For the one-node technique, authentic is encoded as 0, and forgery is encoded as 1. After training, the one-node model is fed the same inputs, (0.6, 1.9, -3.3, -0.3). The single output probability is 0.8468, which maps to 1, so the prediction is forgery/fake.

Both techniques give a similar quality prediction model—85% classification accuracy on the test data. Notice that during training, both techniques have roughly the same cross-entropy error, but that the one-node technique requires twice as many training iterations, 1000 versus 500, as the two-node technique. However, the one-node technique only updates half as many hidden-to-output node weights per iteration, so the increase in number of iterations is offset by faster training per iteration. The bottom line is that there's no significant technical advantage to either of the techniques. I prefer the two-node technique because it's slightly simpler, in my opinion.

Preparing the banknote training and test data

The raw banknote data looks like:

```
3.6216,8.6661,-2.8073,-0.44699,0
4.5459,8.1674,-2.4586,-1.4621,0
. . .
-1.3971,3.3191,-1.3927,-1.9948,1
0.39012,-0.14279,-0.031994,0.35084,1
```

You can find this dataset here. The full dataset has 1,372 items. For simplicity, I selected just the first 50 authentic items (class forgery = 0) and the first 50 fake items (class forgery = 1). I wrote a short helper program to convert the raw data into two-node CNTK format that looks like:

```
|stats  3.62160000  8.66610000 -2.80730000 -0.44699000 |forgery 0 1 |# authentic
|stats  4.54590000  8.16740000 -2.45860000 -1.46210000 |forgery 0 1 |# authentic
. . .
|stats -1.39710000  3.31910000 -1.39270000 -1.99480000 |forgery 1 0 |# fake
|stats  0.39012000 -0.14279000 -0.03199400  0.35084000 |forgery 1 0 |# fake
```

The code for the helper program is shown in Code Listing 5-1. Because there are only 100 lines of data, the helper program uses **`print()`** to emit output to the shell in which it's running. For larger datasets, you'd want to open a text file for writing and use the **`write()`** function.

I scraped the shell output and copied 80 items (40 authentic, 40 forgery) into a training file, and 20 items (10 authentic, 10 forgery) into a test file. The CNTK-formatted data can be found in the Appendix to this e-book.

Code Listing 5-1: Helper to Create CNTK-Format Data from Raw Data

```
# make_banknote_data.py
# input: raw banknote_100.txt
# output: banknote data in CNTK two-node format to screen
# for scraping (manually divide into train/test)

fin = open(".\\banknote_100_raw.txt", "r")

for line in fin:
  line = line.strip()
  tokens = line.split(",")
  if tokens[4] == "0":
    print("|stats %12.8f %12.8f %12.8f %12.8f |forgery 0 1 |# authentic"
% \
      (float(tokens[0]), float(tokens[1]), float(tokens[2]),
float(tokens[3])) )
  else:
    print("|stats %12.8f %12.8f %12.8f %12.8f |forgery 1 0 |# fake" % \
      (float(tokens[0]), float(tokens[1]), float(tokens[2]),
float(tokens[3])) )

fin.close()
```

Because there are four predictors/features, it's not feasible to graph the data. However, you can get a rough idea of the data's structure by examining the two-dimensional graph, based on just variance and skewness, of the 80-item training data as shown in Figure 5-2. The data is not linearly separable, so logistic regression would not work well.

Neural network two-node binary classification

The program that produced the output shown in the left side of Figure 5-1 is presented in Code Listing 5-2. If you scan through the code listing, you'll see there's very little difference between two-node binary classification and three-node (or more) multi-class classification. In the program-defined **`create_reader()`** function, the **`field`** properties are changed to correspond to the **`stats`** and **`forgery`** tags in the training and test data files:

```
  x_strm = C.io.StreamDef(field='stats', shape=input_dim, is_sparse=False)
  y_strm = C.io.StreamDef(field='forgery', shape=output_dim, is_sparse=False)
```

The neural network output dimension is changed to 2 for the two-node technique:

```
  input_dim = 4
  hidden_dim = 10
  output_dim = 2
```

As always, determining the number of hidden nodes to use is a matter of trial and error. As in multi-class classification, you use the **`cross_entropy_with_softmax()`** error metric because CNTK doesn't have a non-softmax version, so you do not use activation on the output layer:

```
   with C.layers.default_options(init=C.initializer.uniform(scale=0.01,
seed=1)):
    hLayer = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer')(X)
    oLayer = C.layers.Dense(output_dim, activation=None,
     name='outLayer')(hLayer)
  nnet = oLayer                # train this
  model = C.ops.softmax(nnet)  # predict with this
```

Note that because Python assigns by reference rather than by value, when you train the **`nnet`** object, the **`model`** object will be updated, too.

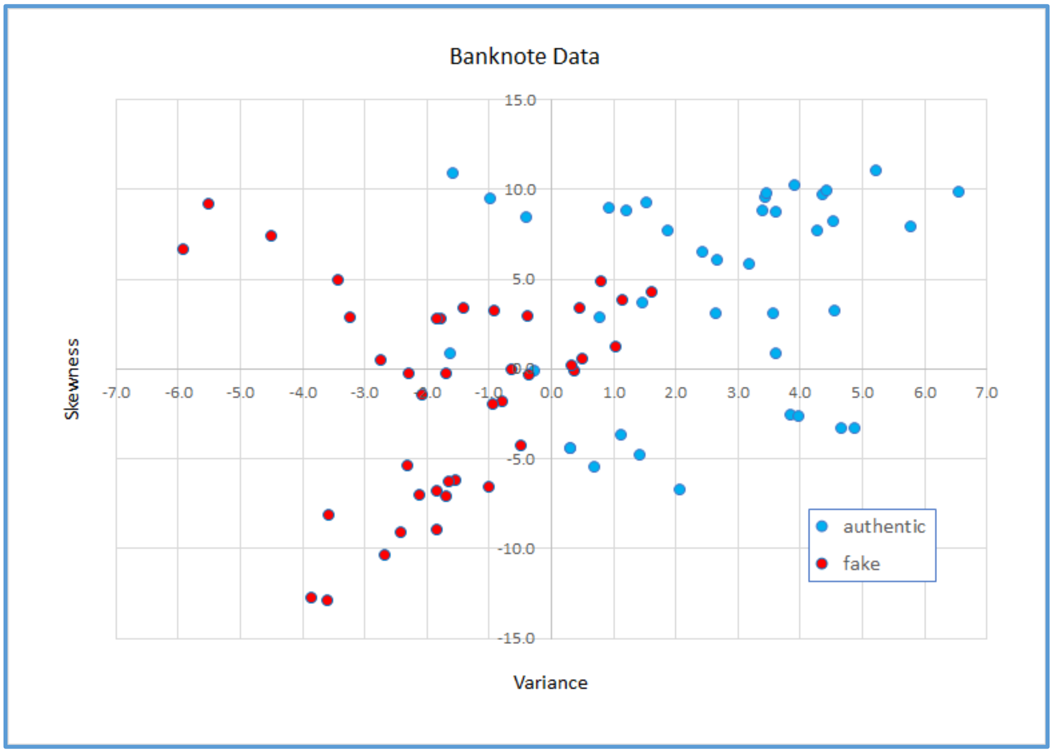


Figure 5-2 Partial Banknote Data

As in multi-class classification, you set up objects to monitor cross-entropy error and classification error/accuracy:

```
  tr_loss = C.cross_entropy_with_softmax(nnet, Y)
  tr_clas = C.classification_error(nnet, Y)
```

This is significant, because as you'll see shortly, you can't use the built-in classification error function when using the one-node technique.

Code Listing 5-2: Two-Node Technique Binary Classification

```
# banknote_bnn.py
# CNTK 2.3 with Anaconda 4.1.1 (Python 3.5, NumPy 1.11.1)

# Use a one-hidden layer simple NN with 10 hidden nodes
# banknote_train_cntk.txt - 80 items (40 authentic, 40 fake)
# banknote_test_cntk.txt - 20 items (10 authentic, 10 fake)

import numpy as np
import cntk as C

def create_reader(path, input_dim, output_dim, rnd_order, sweeps):
  # rnd_order -> usually True for training
  # sweeps -> usually C.io.INFINITELY_REPEAT for training OR 1 for eval
  x_strm = C.io.StreamDef(field='stats', shape=input_dim, is_sparse=False)
  y_strm = C.io.StreamDef(field='forgery', shape=output_dim,
is_sparse=False)
  streams = C.io.StreamDefs(x_src=x_strm, y_src=y_strm)
  deserial = C.io.CTFDeserializer(path, streams)
  mb_src = C.io.MinibatchSource(deserial, randomize=rnd_order,
max_sweeps=sweeps)
  return mb_src

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

def main():
  print("\nBegin banknote binary classification (two-node technique) \n")
  print("Using CNTK version = " + str(C.__version__) + "\n")

  input_dim = 4
  hidden_dim = 10
  output_dim = 2

  train_file = ".\\Data\\banknote_train_cntk.txt"
  test_file = ".\\Data\\banknote_test_cntk.txt"

  # two-node data files:
  # |stats   4.17110   8.72200  -3.02240  -0.59699 |forgery 0 1 |#
authentic
```

```
  # |stats  -0.20620   9.22070  -3.70440  -6.81030 |forgery 0 1 |#
authentic
  # . . .
  # |stats   0.60050   1.93270  -3.28880  -0.32415 |forgery 1 0 |# fake
  # |stats   0.91315   3.33770  -4.05570  -1.67410 |forgery 1 0 |# fake

  # 1. create network
  X = C.ops.input_variable(input_dim, np.float32)
  Y = C.ops.input_variable(output_dim, np.float32)

  print("Creating a 4-10-2 tanh-softmax NN for partial banknote dataset ")
  with C.layers.default_options(init=C.initializer.uniform(scale=0.01,
seed=1)):
    hLayer = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer')(X)
    oLayer = C.layers.Dense(output_dim, activation=None,
     name='outLayer')(hLayer)
  nnet = oLayer
  model = C.ops.softmax(nnet)

  # 2. create learner and trainer
  print("Creating an ordinary cross entropy batch=10 SGD LR=0.01 Trainer ")
  tr_loss = C.cross_entropy_with_softmax(nnet, Y)  # not model!
  tr_clas = C.classification_error(nnet, Y)

  max_iter = 500
  batch_size = 10
  learn_rate = 0.01
  learner = C.sgd(nnet.parameters, learn_rate)
  trainer = C.Trainer(nnet, (tr_loss, tr_clas), [learner])

  # 3. create reader for train data
  rdr = create_reader(train_file, input_dim, output_dim,
    rnd_order=True, sweeps=C.io.INFINITELY_REPEAT)
  banknote_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }

  # 4. train
  print("\nStarting training \n")
  for i in range(0, max_iter):
    curr_batch = rdr.next_minibatch(batch_size,
input_map=banknote_input_map)
    trainer.train_minibatch(curr_batch)
    if i % 50 == 0:
      mcee = trainer.previous_minibatch_loss_average
      macc = (1.0 - trainer.previous_minibatch_evaluation_average) * 100
      print("batch %4d: mean loss = %0.4f, accuracy = %0.2f%% " \
        % (i, mcee, macc))
  print("\nTraining complete")

  # 5. evaluate model using test data
  print("\nEvaluating test data using built-in test_minibatch() \n")
  rdr = create_reader(test_file, input_dim, output_dim,
    rnd_order=False, sweeps=1)
  banknote_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }
  num_test = 20
  all_test = rdr.next_minibatch(num_test, input_map=banknote_input_map)
  acc = (1.0 - trainer.test_minibatch(all_test)) * 100
  print("Classification accuracy on the 20 test items = %0.2f%%" % acc)

  # (could save model here)

  # 6. use trained model to make prediction
  np.set_printoptions(precision = 1, suppress=True)
  unknown = np.array([[0.6, 1.9, -3.3, -0.3]], dtype=np.float32) # likely 1
0 = fake
  print("\nPredicting banknote authenticity for input features: ")
  print(unknown[0])

  pred_prob = model.eval({X: unknown})
  np.set_printoptions(precision = 4, suppress=True)
  print("Prediction probabilities are: ")
  print(pred_prob[0])
  if pred_prob[0,0] < pred_prob[0,1]: # maps to (0,1)
    print("Prediction: authentic")
  else:                               # maps to (1,0)
    print("Prediction: fake")

  print("\nEnd banknote classification ")

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

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

Training a two-node technique binary classifier neural network is exactly the same as training a multi-class network. When using a trained model to make a prediction, you have to map the two prediction probabilities to a predicted class:

```
  unknown = np.array([[0.6, 1.9, -3.3, -0.3]], dtype=np.float32)
  pred_prob = model.eval({X: unknown})
  print("Prediction probabilities are: ")
  print(pred_prob[0])
  if pred_prob[0,0] < pred_prob[0,1]: # maps to (0,1)
    print("Prediction: authentic")
  else:                               # maps to (1,0)
    print("Prediction: fake")
```

The return result from a call to the **eval()** function is an array-of-arrays, like [[0.7678, 0.2322]]. By selecting index [0] you get a single array with the two prediction probabilities, like [0.7678, 0.2322]. If the first of the two probabilities is less than the second, the prediction maps to the class that is encoded (0, 1); otherwise, the prediction maps to the class encoded as (1, 0).

Because the outputs are probabilities and there are only two values, you could also map a predicted class like this:

```
  if pred_prob[0,0] < 0.5:            # maps to (0,1)
    print("Prediction: authentic")
  else:                               # maps to (1,0)
    print("Prediction: fake")
```

When using two-node neural network binary classification, you can encode either class as (0, 1), but it's up to you to maintain the encoding meaning—it's surprisingly easy to botch this.

Neural network one-node binary classification

I prepared the one-node version of the banknote classification technique by modifying the training and test data files, replacing (0, 1) with 0, and (1, 0) with 1:

```
|stats  3.62160000  8.66610000 -2.80730000 -0.44699000 |forgery 0 |#
authentic
|stats  4.54590000  8.16740000 -2.45860000 -1.46210000 |forgery 0 |#
authentic
. . .
|stats -1.39710000  3.31910000 -1.39270000 -1.99480000 |forgery 1 |# fake
|stats  0.39012000 -0.14279000 -0.03199400  0.35084000 |forgery 1 |# fake
```

Because there's no change to the tag names **stats** and **forgery**, there's no need to modify the program-defined **create_reader()** function. The complete program that generated the output shown on the right side of Figure 5-1 is presented in Code Listing 5-3.

Code Listing 5-3: One-Node Binary Classification Technique

```
# banknote_bnn_onenode.py
# CNTK 2.3 with Anaconda 4.1.1 (Python 3.5, NumPy 1.11.1)

# Use a one-hidden layer simple NN with 10 hidden nodes
# banknote_train_cntk.txt - 80 items (40 authentic, 40 fake)
# banknote_test_cntk.txt - 20 items(10 authentic, 10 fake)
```

```
import numpy as np
import cntk as C

def create_reader(path, input_dim, output_dim, rnd_order, sweeps):
  # rnd_order -> usually True for training
  # sweeps -> usually C.io.INFINITELY_REPEAT for training OR 1 for eval
  x_strm = C.io.StreamDef(field='stats', shape=input_dim, is_sparse=False)
  y_strm = C.io.StreamDef(field='forgery', shape=output_dim,
is_sparse=False)
  streams = C.io.StreamDefs(x_src=x_strm, y_src=y_strm)
  deserial = C.io.CTFDeserializer(path, streams)
  mb_src = C.io.MinibatchSource(deserial, randomize=rnd_order,
max_sweeps=sweeps)
  return mb_src

def class_acc(mb, x_var, y_var, model):
  num_correct = 0; num_wrong = 0
  x_mat = mb[x_var].asarray()  # batch_size x 1 x features_dim
  y_mat = mb[y_var].asarray()  # batch_size x 1 x 1

  for i in range(mb[x_var].shape[0]):  # each item in the batch
    p = model.eval(x_mat[i])           # 1 x 1
    y = y_mat[i]                       # 1 x 1
    if p[0,0] < 0.5 and y[0,0] == 0.0 or p[0,0] >= 0.5 and y[0,0] == 1.0:
      num_correct += 1
    else:
      num_wrong += 1
  return (num_correct * 100.0) / (num_correct + num_wrong)

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

def main():
  print("\nBegin banknote binary classification (one-node technique) \n")
  print("Using CNTK version = " + str(C.__version__) + "\n")

  input_dim = 4
  hidden_dim = 10
  output_dim = 1  # NOTE

  train_file = ".\\Data\\banknote_train_cntk_onenode.txt"  # NOTE:
different file
  test_file = ".\\Data\\banknote_test_cntk_onenode.txt"    # NOTE

  # one-node data files:
  # |stats    4.17110   8.72200  -3.02240  -0.59699 |forgery 0 |# authentic
  # |stats   -0.20620   9.22070  -3.70440  -6.81030 |forgery 0 |# authentic
  # . . .
  # |stats   0.60050   1.93270  -3.28880  -0.32415 |forgery 1 |# fake
  # |stats   0.91315   3.33770  -4.05570  -1.67410 |forgery 1 |# fake

  # 1. create network
  X = C.ops.input_variable(input_dim, np.float32)
  Y = C.ops.input_variable(output_dim, np.float32)

  print("Creating a 4-10-1 tanh-logsig NN for partial banknote dataset ")
  with C.layers.default_options(init=C.initializer.uniform(scale=0.01,
seed=1)):
    hLayer = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer')(X)
    oLayer = C.layers.Dense(output_dim, activation=C.ops.sigmoid,
     name='outLayer')(hLayer)  # NOTE: sigmoid activation
  model = oLayer  # alias

  # 2. create learner and trainer
  print("Creating a binary cross entropy batch=10 SGD LR=0.01 Trainer \n")
  tr_loss = C.binary_cross_entropy(model, Y)    # NOTE: use model
  # tr_clas = C.classification_error(model, Y)  # NOTE: not available for
one-node

  max_iter = 1000
  batch_size = 10
  learn_rate = 0.01
  learner = C.sgd(model.parameters, learn_rate)  # NOTE: use model
  trainer = C.Trainer(model, (tr_loss), [learner])  # NOTE: no
classification error

  # 3. create reader for train data
  rdr = create_reader(train_file, input_dim, output_dim,
    rnd_order=True, sweeps=C.io.INFINITELY_REPEAT)
  banknote_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }

  # 4. train
  print("Starting training \n")
  for i in range(0, max_iter):
    curr_batch = rdr.next_minibatch(batch_size,
input_map=banknote_input_map)
    trainer.train_minibatch(curr_batch)
    if i % 100 == 0:
      mcee = trainer.previous_minibatch_loss_average  # built-in
      ca = class_acc(curr_batch, X, Y, model)         # program-defined
      print("batch %4d: mean loss = %0.4f accuracy = %0.2f%%" % (i, mcee,
ca))
```

```
  print("\nTraining complete")

  # 5. evaluate test data (cannot use trainer.test_minibatch)
  print("\nEvaluating test data using program-defined class_acc() \n")
  rdr = create_reader(test_file, input_dim, output_dim,
    rnd_order=False, sweeps=1)

  banknote_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }

  num_test = 20
  all_test = rdr.next_minibatch(num_test, input_map=banknote_input_map)
  acc = class_acc(all_test, X, Y, model)
  print("Classification accuracy on the 20 test items = %0.2f%%" % acc)

  # (could save model here)

  # 6. use trained model to make prediction
  np.set_printoptions(precision = 1, suppress=True)
  unknown = np.array([[0.6, 1.9, -3.3, -0.3]], dtype=np.float32)  # likely
fake
  print("\nPredicting banknote authenticity for input features: ")
  print(unknown[0])

  pred_prob = model.eval({X: unknown})
  print("Prediction probability is: ")
  print("%0.4f" % pred_prob[0,0])

  if pred_prob[0,0] < 0.5:             # prob(forgery) < 0.5
    print("Prediction: authentic")
  else:
    print("Prediction: fake")

  print("\nEnd banknote classification ")

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

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

The first change to the program is the addition of a program-defined **`class_acc()`** function to compute the classification accuracy of a mini-batch. When using the two-node classification technique, you can use the CNTK built-in **`classification_error()`** function, but the one-node technique doesn't support **`classification_error()`**, so you must implement a program-defined function yourself.

In the **main()** function, you change the number of output nodes to 1 because you're using the one-node technique:

```
  input_dim = 4
  hidden_dim = 10
  output_dim = 1  # NOTE: instead of 2
```

When creating the neural network, you change the output activation from **None** to **sigmoid()**, and you need just one neural network object that is used for both training and prediction:

```
  with C.layers.default_options(init=C.initializer.uniform(scale=0.01,
seed=1)):
    hLayer = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer')(X)
    oLayer = C.layers.Dense(output_dim, activation=C.ops.sigmoid,
     name='outLayer')(hLayer)  # change from None

  model = oLayer  # NOTE: use for both training and prediction
```

The logistic **sigmoid** activation function scales the single output node value to the range [0.0, 1.0), which can be interpreted as the probability of getting class 1. This means if the output value is less than 0.5, your prediction is class 0; otherwise, your prediction is class 1.

For the one-node technique binary classification technique, when you set up the training error functions, you use **binary_cross_entropy()** instead of **cross_entropy_with_softmax()**, and you drop the **classification_error()** function:

```
  print("Creating a binary cross-entropy batch=10 SGD LR=0.01 Trainer \n")
  tr_loss = C.binary_cross_entropy(model, Y)    # NOTE: use model
  # tr_clas = C.classification_error(model, Y)  # NOTE: not available for
one-node
```

The **binary_cross_entropy()** function is used when you have a single output node with a value between 0.0 and 1.0. You could use **squared_error()**, but **binary_cross_entropy()** is more principled. If you include and then use **classification_error()** with the one-node technique, your program will run, but the function will give you meaningless results.

When you set up the trainer, you use the training error/loss function, but not the classification error:

```
 trainer = C.Trainer(model, (tr_loss), [learner])  # NOTE: no classification
error
```

The one-node training code is essentially the same as the two-node technique, except you don't have the **previous_minibatch_evaluation_average()** function available because there's no **classification_error()** function defined. So if you want to monitor classification accuracy (optional but recommended), you must call a program-defined function:

```
  print("Starting training \n")
  for i in range(0, max_iter):
    curr_batch = rdr.next_minibatch(batch_size, input_map=banknote_input_map)
    trainer.train_minibatch(curr_batch)
    if i % 100 == 0:
      mcee = trainer.previous_minibatch_loss_average  # built-in
      ca = class_acc(curr_batch, X, Y, model)         # program-defined
      print("batch %4d: mean loss = %0.4f accuracy = %0.2f%%" % (i, mcee,
ca))
  print("\nTraining complete")
```

The helper function **class_acc()** is at the top of Code Listing 5-3. The function accepts a mini-batch object for which you want to compute classification accuracy, a **Variable** object that holds the structure of the input values, a **Variable** object that holds the structure of the known correct output values, and a **model** object that is the neural network being trained:

```
def class_acc(mb, x_var, y_var, model):
  num_correct = 0; num_wrong = 0:
  . . .
```

The **x** and **y** values are pulled out of the mini-batch object like this:

```
  x_mat = mb[x_var].asarray()  # batch_size x 1 x features_dim
  y_mat = mb[y_var].asarray()  # batch_size x 1 x 1
```

This code is not at all obvious, but can be considered boilerplate. A CNTK mini-batch object is implemented as a Python dictionary. The **X** and **Y** variable objects act as keys for the dictionary, but the dictionary values must be explicitly cast as NumPy array types. The resulting arrays have three dimensions, where the first is the number of training items in the mini-batch.

Next, the function walks through each training item in the mini-batch:

```
  for i in range(mb[x_var].shape[0]):  # each item in the batch
    p = model.eval(x_mat[i])           # 1 x 1
    y = y_mat[i]                       # 1 x 1
    if p[0,0] < 0.5 and y[0,0] == 0.0 or p[0,0] >= 0.5 and y[0,0] == 1.0:
      num_correct += 1
    else:
      num_wrong += 1
  return (num_correct * 100.0) / (num_correct + num_wrong)
```

The input values are fed to the model using the **eval()** function, and the computed **y** value is returned into a matrix **p**, which has dimensions 1×1, so the probability value is in **p[0,0]**. The known correct output value is also in a 1×1 matrix.

The one-node technique program concludes by making a prediction for an unknown banknote:

```
  pred_prob = model.eval({X: unknown})
  print("Prediction probability is: ")
  print("%0.4f" % pred_prob[0,0])
  if pred_prob[0,0] < 0.5:             # prob(forgery) < 0.5
    print("Prediction: authentic")
  else:
    print("Prediction: fake")
```

The return value from the call to **`eval()`** is a 1×1 matrix with a value between 0.0 and 1.0. A value less than 0.5 maps to the class encoded as 0 (authentic banknote in this case), and a value greater than or equal to 0.5 maps to the class encoded as 1 (forgery/fake banknote).

Exercise

Using either the two-node technique (Code Listing 5-2) or the one-node technique (Code Listing 5-3), create, train, and evaluate a neural network binary classifier for the (processed) Cleveland Heart Disease dataset. You can find the raw data here.

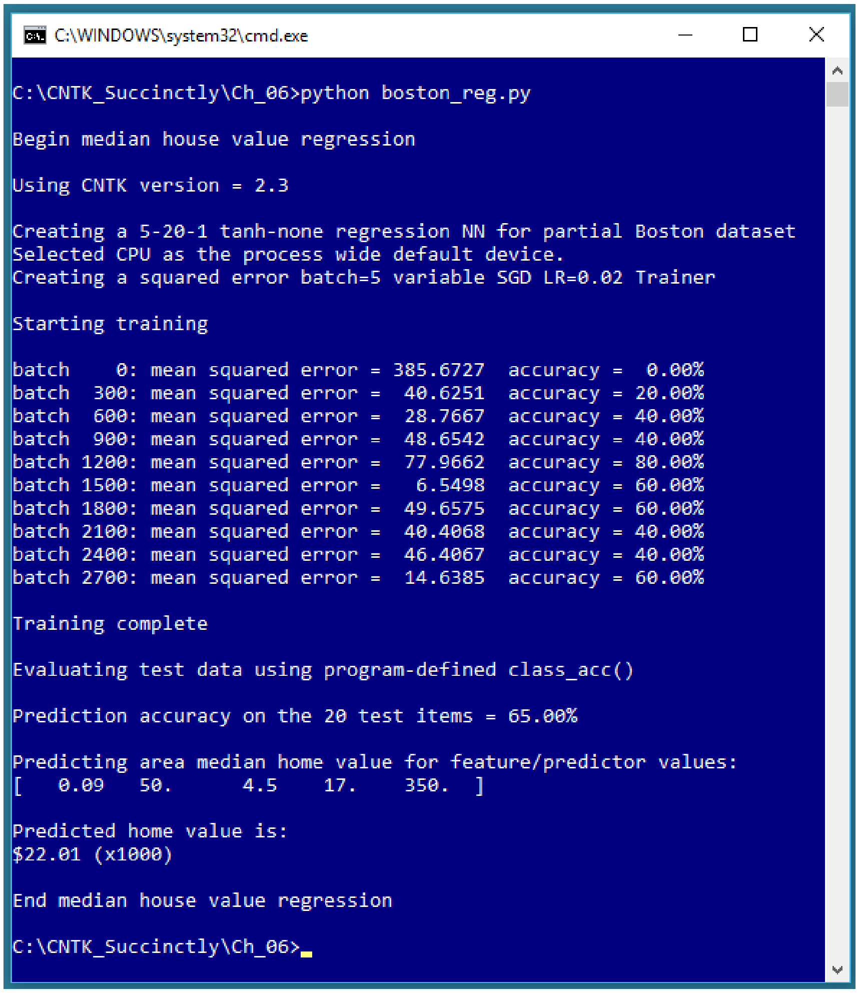
There are 303 data items, six of which have a missing value. Each item has 13 features followed by a value from 0 to 4, where 0 means no heart disease, and values 1 through 4 mean presence of heart disease.

```
63.0,1.0,1.0,145.0,233.0,1.0,2.0,150.0,0.0,2.3,3.0,0.0,6.0,0
67.0,1.0,4.0,160.0,286.0,0.0,2.0,108.0,1.0,1.5,2.0,3.0,3.0,2
67.0,1.0,4.0,120.0,229.0,0.0,2.0,129.0,1.0,2.6,2.0,2.0,7.0,1
. . .
```

I suggest removing the six items that have missing values. Because of the different scales of features, I recommend using some form of normalization—dividing by a power of 10 works reasonably well, but min-max works better. When dividing the normalized and encoded data into a training and a test set, I suggest using 80% of the data for training (about 238 items) and 20% for testing.

Chapter 6 Neural Network Regression

The goal of a regression problem is to predict a numeric value from one or more predictor variables. For example, suppose you want to predict the median value of a house in one of 100 towns near Boston. You have data that includes a crime statistic for each town, the age of the houses in each town, a measure of the distance from each town to Boston, the pupil-to-teacher ratio in each town, a racial demographic statistic for each town, and the median house value in each town. Using the first five predictor variables, you want to create a model to predict median house value.



```
C:\CNTK_Succinctly\Ch_06>python boston_reg.py

Begin median house value regression

Using CNTK version = 2.3

Creating a 5-20-1 tanh-none regression NN for partial Boston dataset
Selected CPU as the process wide default device.
Creating a squared error batch=5 variable SGD LR=0.02 Trainer

Starting training

batch    0: mean squared error = 385.6727  accuracy =  0.00%
batch  300: mean squared error =  40.6251  accuracy = 20.00%
batch  600: mean squared error =  28.7667  accuracy = 40.00%
batch  900: mean squared error =  48.6542  accuracy = 40.00%
batch 1200: mean squared error =  77.9662  accuracy = 80.00%
batch 1500: mean squared error =   6.5498  accuracy = 60.00%
batch 1800: mean squared error =  49.6575  accuracy = 60.00%
batch 2100: mean squared error =  40.4068  accuracy = 40.00%
batch 2400: mean squared error =  46.4067  accuracy = 40.00%
batch 2700: mean squared error =  14.6385  accuracy = 60.00%

Training complete

Evaluating test data using program-defined class_acc()

Prediction accuracy on the 20 test items = 65.00%

Predicting area median home value for feature/predictor values:
[   0.09   50.      4.5    17.    350.  ]

Predicted home value is:
$22.01 (x1000)

End median house value regression

C:\CNTK_Succinctly\Ch_06>
```

Figure 6-1: Median Home Value Regression

You could create a linear regression model along the lines of **`Y = a0 + (a1)(crime) + (a2)(age) + (a3)(distance) + (a4)(ratio) + (a5)(racial)`** where **`Y`** is the predicted median value, **`a0`** is a constant, and **`a1`** through **`a5`** are constants associated with the five predictor variables. An alternative approach, which can often create a more accurate prediction model, is to use a neural network.

Preparing the Boston area house values data

The Boston area house values dataset is a well-known benchmark collection dating from a 1978 research paper. The full dataset has 14 attributes/variables, and 506 instances. You can find the full dataset here. For simplicity, the demo program shown in Figure 6-1 uses just six of the 14 attributes (five as predictors, one as a value-to-predict), and 100 instances (80 training and 20 test).

The value to predict is the median house price in a town or census tract. The first predictor is crime per capita in the town or United States census tract, so you'd expect smaller values to be associated with higher house values. The second is the proportion of owner-occupied units built before 1940, so larger values mean older, but it's not obvious if older houses would be associated with higher house values or lower house values. The third predictor is a weighted distance of the town to five Boston employment centers. The fourth predictor is the area school pupil-to-teacher ratio. The fifth predictor is an indirect metric of the proportion of black residents in the town (`= 1000 * (proportion_Black - 0.63)^2`), so you'd expect higher values to be associated with lower house values. The values to predict are median house values that have been divided by 1,000, for example 25.50 means $25,500.00—homes were much less expensive in 1978.

Using the first 80 items of the full dataset, I created an 80-item tab-delimited training data file that looks like this:

```
|predictors  1.612820  96.90  3.76  21.00  248.31  |medval  13.50
|predictors  0.064170  68.20  3.36  19.20  396.90  |medval  18.90
|predictors  0.097440  61.40  3.38  19.20  377.56  |medval  20.00
. . .
```

I also created a 20-item test set with the same format. You can find both datasets in the Appendix of this e-book. Because there are five predictor variables, it's not possible to graph the dataset, but you can get an idea of the structure of the data by the graph in Figure 6-2. The graph plots the 80-item training set with the age attribute (proportion of houses built before 1940) on the x-axis, and median house value on the y-axis. Linear regression would fit a nearly horizontal line through the middle of the data, meaning the prediction equation would predict a median house value of about 22.50 regardless of age—and result in a poor prediction model.

For simplicity, I did not normalize the data, but this is an example of data that should definitely be normalized. Notice that the racial predictor variable, with values like 396.55, is much larger than the crime variable, with values like 0.0131. One simple approach would be to scale all predictor variables so that most are between 1.0 and 10.0—you could multiply crime by 10, divide age by 10, leave distance alone, divide ratio by 10, and divide racial by 100.

Another approach would be to use min-max normalization, or z-score normalization. In fact, some experiments showed that all three forms of normalization gave a significantly better predictive model.

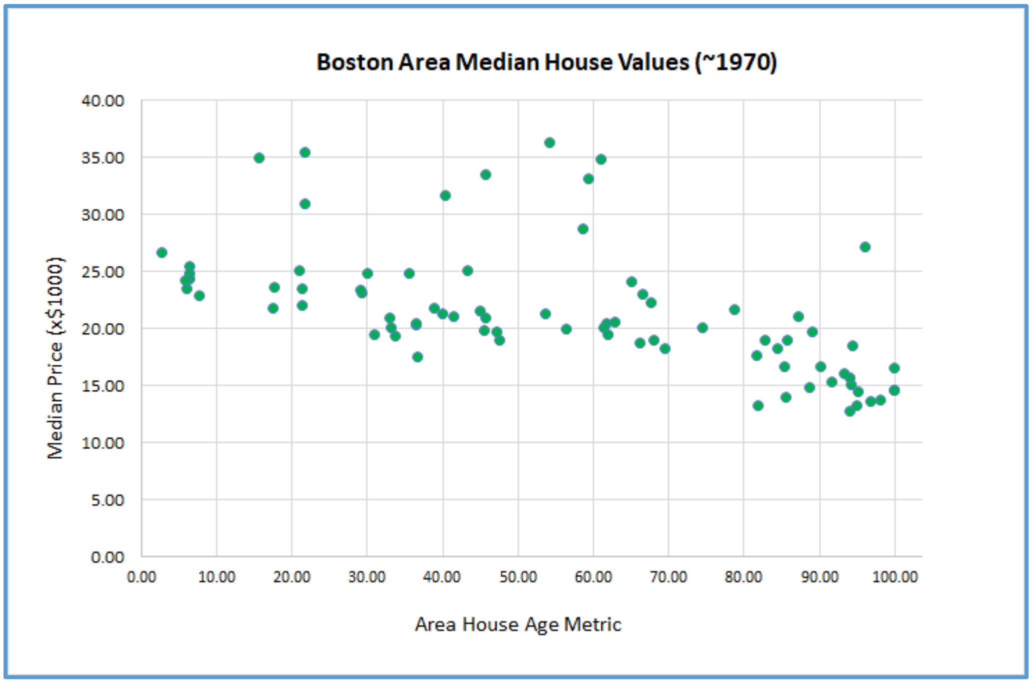


Figure 6-2: Partial Boston Area Median House Values Data

In situations in which you have a dataset with a large number of variables, some of those variables may not be useful for a prediction model, and including some variables may actually create a worse model than you'd get by leaving those variables out. Determining which predictor variables to use, and which not to use, is called feature selection.

The neural network regression program

The program code that generated the output shown in the screenshot in Figure 6-1 is presented in Code Listing 6-1. After importing the required NumPy and CNTK packages, the demo program defines a helper function to read data from a CNTK format file into a mini-batch object:

```
def create_reader(path, input_dim, output_dim, rnd_order, sweeps):
  # rnd_order -> usually True for training
  # sweeps -> usually C.io.INFINITELY_REPEAT for training OR 1 for eval
  x_strm = C.io.StreamDef(field='predictors', shape=input_dim,
is_sparse=False)
  y_strm = C.io.StreamDef(field='medval', shape=output_dim, is_sparse=False)
  streams = C.io.StreamDefs(x_src=x_strm, y_src=y_strm)
  # streams = C.variables.Record(x_src=x_strm, y_src=y_strm)
  deserial = C.io.CTFDeserializer(path, streams)
  mb_src = C.io.MinibatchSource(deserial, randomize=rnd_order,
max_sweeps=sweeps)
  return mb_src
```

You can consider the code in this helper function as boilerplate for neural network regression problems. Because CNTK is so new and evolving so quickly, sometimes the documentation gets out of sync with the code For example, at the time I'm writing this e-book, the documentation makes no mention of the **`StreamDefs()`** function. It's not clear if **`StreamDefs()`** has been deprecated or if it's just missing from the documentation.

If you apply the **`type()`** function to the return result **`streams`**, you'll see it is type **`cntk.variables.Record`**, and so you could code as:

```
streams = C.variables.Record(x_src=x_strm, y_src=y_strm)
```

Using the Python **`type()`** function to examine CNTK objects is an indispensable debugging technique.

Code Listing 6-1: Neural Network Regression

```
# boston_reg.py
# CNTK 2.3 with Anaconda 4.1.1 (Python 3.5, NumPy 1.11.1)

# Predict median value of a house in an area near Boston based on area's
# crime rate, age of houses, distance to Boston, pupil-teacher
# ratio, percentage black residents
# boston_train_cntk.txt - 100 items
# boston_test_cntk.txt - 20 items

import numpy as np
import cntk as C

def create_reader(path, input_dim, output_dim, rnd_order, sweeps):
  # rnd_order -> usually True for training
  # sweeps -> usually C.io.INFINITELY_REPEAT for training OR 1 for eval
  x_strm = C.io.StreamDef(field='predictors', shape=input_dim,
is_sparse=False)
  y_strm = C.io.StreamDef(field='medval', shape=output_dim,
is_sparse=False)
  streams = C.io.StreamDefs(x_src=x_strm, y_src=y_strm)
  # streams = C.variables.Record(x_src=x_strm, y_src=y_strm)
  deserial = C.io.CTFDeserializer(path, streams)
  mb_src = C.io.MinibatchSource(deserial, randomize=rnd_order,
max_sweeps=sweeps)
  return mb_src

def mb_accuracy(mb, x_var, y_var, model, delta):
  num_correct = 0
  num_wrong = 0

  x_mat = mb[x_var].asarray()  # batch_size x 1 x features_dim
  y_mat = mb[y_var].asarray()  # batch_size x 1 x 1

  # for i in range(mb[x_var].shape[0]):  # each item in the batch
  for i in range(len(mb[x_var])):
    v = model.eval(x_mat[i])           # 1 x 1 predicted value
    y = y_mat[i]                       # 1 x 1 actual value
    if np.abs(v[0,0] - y[0,0]) <  delta:  # close enough?
      num_correct += 1
    else:
      num_wrong += 1
  return (num_correct * 100.0) / (num_correct + num_wrong)

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

def main():
  print("\nBegin median house value regression \n")
  print("Using CNTK version = " + str(C.__version__) + "\n")

  input_dim = 5  # crime, age, distance, pupil-teach, black
  hidden_dim = 20
  output_dim = 1  # median value (x$1000)

  train_file = ".\\Data\\boston_train_cntk.txt"
  test_file = ".\\Data\\boston_test_cntk.txt"

  # data resembles:
  # |predictors  0.041130  33.50  5.40  19.00  396.90  |medval  28.00
  # |predictors  0.068600  62.50  3.50  18.00  393.53  |medval   33.20

  # 1. create network
  X = C.ops.input_variable(input_dim, np.float32)
  Y = C.ops.input_variable(output_dim, np.float32)

  print("Creating a 5-20-1 tanh-none regression NN for partial Boston
dataset ")
  with C.layers.default_options(init=C.initializer.uniform(scale=0.01,
seed=1)):
    hLayer = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer')(X)
    oLayer = C.layers.Dense(output_dim, activation=None,
     name='outLayer')(hLayer)
  model = C.ops.alias(oLayer)  # alias

  # 2. create learner and trainer
  print("Creating a squared error batch=5 variable SGD LR=0.02 Trainer \n")
  tr_loss = C.squared_error(model, Y)

  max_iter = 3000
  batch_size = 5
  base_learn_rate = 0.02
  sch = C.learning_parameter_schedule([base_learn_rate, base_learn_rate/2],
    minibatch_size=batch_size,
    epoch_size=int((max_iter*batch_size)/2))
  learner = C.sgd(model.parameters, sch)
  trainer = C.Trainer(model, (tr_loss), [learner])

  # 3. create reader for train data
  rdr = create_reader(train_file, input_dim, output_dim,
    rnd_order=True, sweeps=C.io.INFINITELY_REPEAT)
  boston_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }

  # 4. train
  print("Starting training \n")
  for i in range(0, max_iter):
    curr_batch = rdr.next_minibatch(batch_size, input_map=boston_input_map)
    trainer.train_minibatch(curr_batch)
    if i % int(max_iter/10) == 0:
      mcee = trainer.previous_minibatch_loss_average
      acc = mb_accuracy(curr_batch, X, Y, model, delta=3.00)  # program-
defined
      print("batch %4d: mean squared error = %8.4f  accuracy = %5.2f%%" \
% (i, mcee, acc))
  print("\nTraining complete")

  # 5. evaluate test data (cannot use trainer.test_minibatch)
  print("\nEvaluating test data using program-defined class_acc() \n")
  rdr = create_reader(test_file, input_dim, output_dim,
    rnd_order=False, sweeps=1)

  boston_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }

  num_test = 20
  all_test = rdr.next_minibatch(num_test, input_map=boston_input_map)
  acc = mb_accuracy(all_test, X, Y, model, delta=3.00)
  print("Prediction accuracy on the 20 test items = %0.2f%%" % acc)

  # (could save model here)

  # 6. use trained model to make prediction
  np.set_printoptions(precision = 2, suppress=True)
  unknown = np.array([[0.09, 50.00, 4.5, 17.00, 350.00]], dtype=np.float32)
  print("\nPredicting area median home value for feature/predictor values:
")
```

```
  print(unknown[0])

  pred_value = model.eval({X: unknown})
  print("\nPredicted home value is: ")
  print("$%0.2f (x1000)" % pred_value[0,0])

  print("\nEnd median house value regression ")

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

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

When working with neural network regression, you'll need to define a custom accuracy function. With classification problems, a prediction is either correct or wrong. But with regression problems, you must define what it means for an output value to be correct—how close is close enough?

The demo program defines a helper function that accepts a CNTK mini-batch object and computes a custom accuracy metric. The definition begins:

```
def mb_accuracy(mb, x_var, y_var, model, delta):
  num_correct = 0
  num_wrong = 0
```

The function accepts a CNTK mini-batch object of training data, a model to evaluate, and a **delta** value that defines how close a predicted output value must be to the known correct value in order to be considered correct. The **x_var** and **y_var** parameters aren't necessary from a conceptual point of view, but they're required to access data in the mini-batch object:

```
  x_mat = mb[x_var].asarray()  # batch_size x 1 x features_dim
  y_mat = mb[y_var].asarray()  # batch_size x 1 x 1
```

A CNTK mini-batch object is essentially a Python dictionary that has CNTK **Variable** object as keys. The values must be explicitly coerced into three-dimensional NumPy array objects using the **asarray()** function.

The first dimension of the array shape is the number of items in the mini-batch collection, so that value can be used to iterate through the collection. The input values from each item are fed to the regression model, and the output values are computed (using the current weights and bias values):

```
  for i in range(mb[x_var].shape[0]):  # each item in the batch
    v = model.eval(x_mat[i])           # 1 x 1 predicted value
    y = y_mat[i]                       # 1 x 1 actual value
  . . .
```

Because the **len()** function applied to an **n**-dim NumPy array returns the first dimension, you can also iterate using **for i in range(len(mb[x_var]))** instead of using **shape** directly. The return value from the call to **eval()** is a 1×1 matrix. Information like this is not obvious and can best be determined during program debugging by inserting statements that display objects' **shape** properties.

The **mb_accuracy()** function computes and returns the percentage of correct predictions:

```
. . .
    if np.abs(v[0,0] - y[0,0]) <  delta:  # close enough?
      num_correct += 1
    else:
      num_wrong += 1
  return (num_correct * 100.0) / (num_correct + num_wrong)
```

An alternative is to return the accuracy as a proportion such as 0.6500, rather than a percentage like 65.00%. Instead of counting a predicted output value as correct if it is within a fixed value of the correct output value, you can check if the computed value is within a specified percentage/proportion of the correct output value. For example, you might want to count a predicted area median house value correct if it's within 10% of the true value.

The demo regression program prepares to create a neural network like so:

```
  input_dim = 5  # crime, age, distance, pupil-teach, racial
  hidden_dim = 20
  output_dim = 1  # median value (x$1000)

  X = C.ops.input_variable(input_dim, np.float32)
  Y = C.ops.input_variable(output_dim, np.float32)
```

There are no good rules of thumb for determining the number of hidden nodes to use. It is possible to create a regression model with two or more output nodes, but such problems are quite rare.

The model is created with these statements:

```
  with C.layers.default_options(init=C.initializer.uniform(scale=0.01,
seed=1)):
    hLayer = C.layers.Dense(hidden_dim, activation=C.ops.tanh,
      name='hidLayer')(X)
    oLayer = C.layers.Dense(output_dim, activation=None,
     name='outLayer')(hLayer)
  model = C.ops.alias(oLayer)  # alias
```

Note that there is no activation function applied to the single output node. This is sometimes called applying the **Identify** function, which is just **f(x) = x**.

The learner algorithm object and trainer object are created with a variable learning rate:

```
  tr_loss = C.squared_error(model, Y)
  max_iter = 3000
  batch_size = 5
  base_learn_rate = 0.02
  sch = C.learning_parameter_schedule([base_learn_rate, base_learn_rate/2],
    minibatch_size=batch_size,
    epoch_size=int((max_iter*batch_size)/2))
  learner = C.sgd(model.parameters, sch)
  trainer = C.Trainer(model, (tr_loss), [learner])
```

The demo program uses **squared_error()** because cross-entropy is not applicable for regression problems. It is possible to extend CNTK to use a custom error function, but that's a topic outside of the scope of this e-book.

Instead of using a fixed learning rate, the demo creates a learning rate schedule that uses a learning rate of 0.02 for the first half of the 3,000 training iterations, and then a 0.01 rate for the second half of iterations. In general, learning rate schedules aren't necessary for simple neural network, but they're often useful for deep neural networks with many hidden layers.

A reader for the training data is created in the usual way:

```
  rdr = create_reader(train_file, input_dim, output_dim,
    rnd_order=True, sweeps=C.io.INFINITELY_REPEAT)
  boston_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }
```

Instead of reusing a single **rdr** object for training and testing, some of my colleagues prefer creating two objects, such as **rdr_train** and **rdr_test**.

Training is performed with these statements:

```
  for i in range(0, max_iter):
    curr_batch = rdr.next_minibatch(batch_size, input_map=boston_input_map)
    trainer.train_minibatch(curr_batch)
    if i % int(max_iter/10) == 0:
      mcee = trainer.previous_minibatch_loss_average
      acc = mb_accuracy(curr_batch, X, Y, model, delta=3.00)  # program-
defined
      print("batch %4d: mean squared error = %8.4f  accuracy = %5.2f%%" \
        % (i, mcee, acc))
```

Just as with classification, it's important to monitor error/loss during regression training, because training can often fail in spectacular fashion. Here average squared error is displayed every 1/10 of the specified iterations. Mean squared error is a bit easier to interpret than cross-entropy error. Suppose the mean squared error is 25.00; then the mean absolute error is 5.00, which means an actual median house price of 30.00 ($30,000) is being predicted as either 25.00 or 35.00.

The demo program concludes by making a prediction for new, previously unseen predictor values. The prediction is prepared:

```
  unknown = np.array([[0.09, 50.00, 4.5, 17.00, 350.00]], dtype=np.float32)
# 1x5
  print("\nPredicting area median home value for feature/predictor values: ")
  print(unknown[0])
```

The prediction is made like this:

```
  pred_value = model.eval({X: unknown})
  print("\nPredicted home value is: ")
  print("$%0.2f (x1000)" % pred_value[0,0])
  print("\nEnd median house value regression ")
```

Because of Python's permissiveness, you pass the unknown matrix directly to **eval()**:

```
 pred_value = model.eval(unknown)
```

The Python language's lax handling of argument types isn't necessarily always a benefit—it allows many different ways to write code, which can be reflected in somewhat apparently inconsistent code examples you find online.

Exercise

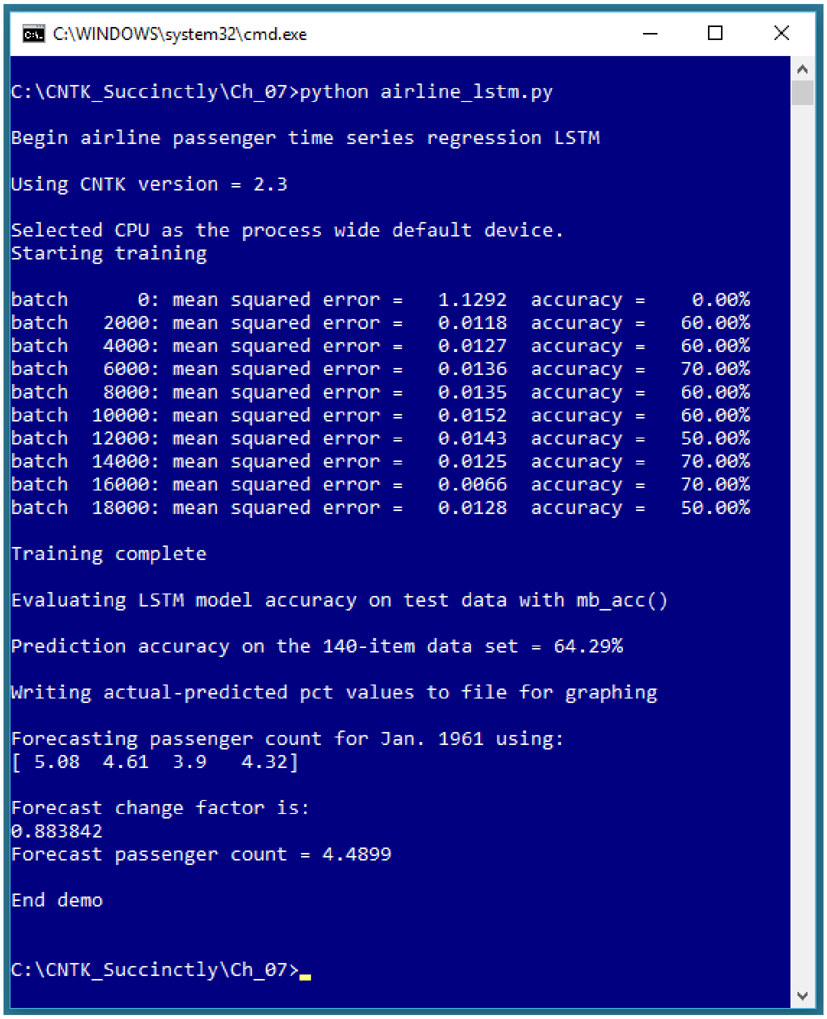
Using the program in Code Listing 6-1 as a guide, create, train, and evaluate a neural network regression model for the Yacht Hydrodynamics dataset. You can find the raw data here. There are 308 data items. There are six predictor values that describe the shape of a yacht hull, followed by the value to predict which is residuary resistance. The data looks like this:

```
-2.3 0.568 4.78 3.99 3.17 0.125 0.11
-2.3 0.568 4.78 3.99 3.17 0.150 0.27
-2.3 0.568 4.78 3.99 3.17 0.175 0.47
. . .
```

I recommend normalizing the second and sixth predictors by multiplying by 10, or by normalizing all predictor values using min-max normalization. When you split the 308 items into a training set and a test set, I suggest using 80% for training (about 246 items), and the remaining 20% (62 items) for testing.

Chapter 7 LSTM Time Series Regression

The goal of a time series regression problem is to make predictions based on historical time data. For example, if you have monthly sales data over the course of a year or two, you might want to predict sales for the upcoming month. Time series regression problems are almost always extremely difficult, and there are many different techniques you can use.



```
C:\CNTK_Succinctly\Ch_07>python airline_lstm.py

Begin airline passenger time series regression LSTM

Using CNTK version = 2.3

Selected CPU as the process wide default device.
Starting training

batch      0: mean squared error =   1.1292  accuracy =   0.00%
batch   2000: mean squared error =   0.0118  accuracy =  60.00%
batch   4000: mean squared error =   0.0127  accuracy =  60.00%
batch   6000: mean squared error =   0.0136  accuracy =  70.00%
batch   8000: mean squared error =   0.0135  accuracy =  60.00%
batch  10000: mean squared error =   0.0152  accuracy =  60.00%
batch  12000: mean squared error =   0.0143  accuracy =  50.00%
batch  14000: mean squared error =   0.0125  accuracy =  70.00%
batch  16000: mean squared error =   0.0066  accuracy =  70.00%
batch  18000: mean squared error =   0.0128  accuracy =  50.00%

Training complete

Evaluating LSTM model accuracy on test data with mb_acc()

Prediction accuracy on the 140-item data set = 64.29%

Writing actual-predicted pct values to file for graphing

Forecasting passenger count for Jan. 1961 using:
[ 5.08  4.61  3.9   4.32]

Forecast change factor is:
0.883842
Forecast passenger count = 4.4899

End demo


C:\CNTK_Succinctly\Ch_07>_
```

Figure 7-1: Time Series Regression Using a Neural Network

The screenshot in Figure 7-1 shows an example of time series regression using an LSTM (long short-term memory) deep neural network. The data is a well-known dataset of the monthly number of international airline passengers, from January 1949 through December 1960. The data and resulting prediction model are graphed in Figure 7-2.

Unlike the other examples in this e-book, in time series regression there isn't a clear distinction between feature/predictor values and class/values-to-be predicted. Each predicted value is based on one or more previous values.

The distinction is a bit subtle. In the graph in Figure 7-2, the first passenger count is 1.12 (112,000 passengers) for month 1, and the second passenger count is 1.18 for month 2. It's not correct to think that f(1) = 1.12 and f(2) = 1.18, and so on for some function f(t). Each passenger count is part of the series of values, and not an explicit function of a time variable of some sort.

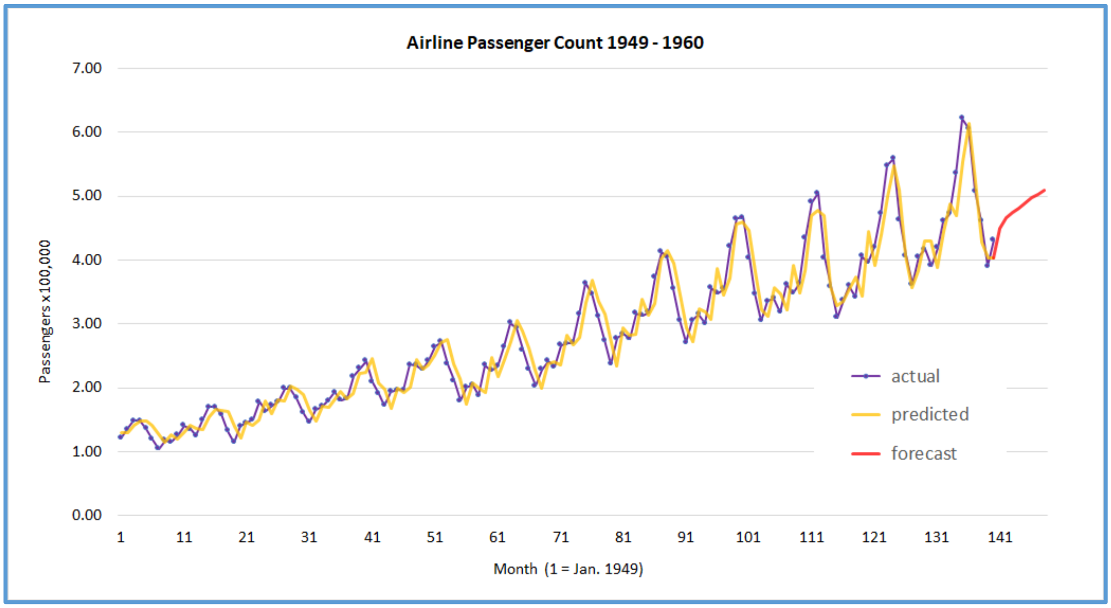


Figure 7-2: Time Series Regression Actual vs. Predicted

After a time series regression model has been created, it is typically used in one of two ways. The model can be used to identify anomalous data points in the historical data, or the model can be used to forecast one or more time steps ahead (sometimes called extrapolation). In the demo, there aren't any clearly anomalous data points. The forecast for January 1961 is 4.49 (449,000) passengers.

Preparing the airline passenger data

The airline passenger dataset appears to have first been published in the 1976 book *Time Series Analysis: Forecasting and Control* by Box and Jenkins; however, it's not clear exactly where the authors obtained the raw data. The 144-item dataset can be found in several formats in many places on the Internet.

One form of the raw data is:

```
"1949-01";112
"1949-02";118
"1949-03";132
. . .
```

The LSTM time series regression demo program uses a rolling window technique, which is best explained by example. The data file used by the demo looks like:

```
|prev 1.12 1.18 1.32 1.29 |next 1.21 |change 1.08035714
|prev 1.18 1.32 1.29 1.21 |next 1.35 |change 1.14406780
|prev 1.32 1.29 1.21 1.35 |next 1.48 |change 1.12121212
```

The **`prev`**, **`next`**, and **`change`** tags were inserted so that the data could be easily processed by a CNTK **`Reader`** object. Each sequence of four consecutive data points is used as the predictor set for the next data point. Here the size of the input sequence is set to 4. The sequence size for LSTM is a free parameter, and must be determined by trial and error.

Each raw passenger count has been normalized by dividing by 100,000, so 1.12 means 112,000 passengers. Almost everything about time series regression problems is tricky; standard min-max and z-score normalization are often ineffective. You can find the complete dataset used by the demo program in the Appendix to this e-book.

The LSTM does not predict the next passenger count directly. Instead, the model predicts a change factor relative to the first item in the input sequence. For example, the first set of four input values are (1.12, 1.18, 1.32, 1.29) and the predicted change value is 1.08035714, which means that the next predicted passenger count is 1.12 * 1.08035714 = 1.2100, as shown in the data. The data item identified by the **`next`** tag is 1.21, and is the raw actual next count. The actual passenger count values are included only for convenience when computing an accuracy metric—the model predicts a change factor, not a count.

After formatting the 144 raw data points, there are 140 data sequences (the first four items have no predictors). Notice that setting up rolling window data results in some duplication of data because most data points appear four times. An alternative approach is to just maintain a single sequence (on file or in memory) and programmatically read from that sequence. But in most time series regression problem scenarios, there isn’t a huge amount of data, so the convenience of using duplicate data often outweighs the minor inefficiency.

LSTM networks

Unlike regular neural networks, LSTM networks maintain state. Informally, LSTM networks accept a sequence as input and remember a bit of their previous input and output values. LSTM networks are one of several types of recurrent neural networks.

LSTM networks were designed to work with natural language data. For example, suppose you were asked to predict the next word in the sentence, “I’m taking lessons in _____.” Without any context, you’d have a difficult time making an accurate prediction. But suppose knew a previous sentence in the paragraph was, “I’m looking forward to my trip to Spain.” You’d probably predict the unknown word in the other sentence is “Spanish.”

Let me emphasize that the use of LSTM networks for time series regression is speculative, and at the time this e-book is being written, there are no definitive research results that indicate whether LSTM networks work better than, or worse than, traditional techniques. However, using LSTMs for time series regression appears promising. Additionally, using an LSTM network with numeric data is easier than working with natural language data, and gives you a good introduction to LSTMs.

The CNTK library supports LSTM networks. An LSTM cell is shown in Figure 7-3. Even a quick glance at the figure should convince you that LSTMs are complicated, and that implementing an LSTM network from scratch would be extremely challenging. In the figure, the X_t vector represents an input sequence of four consecutive airline passenger counts. The Y_t vector represents four output values. To compute output values, the LSTM cell uses the input sequence, plus the previous output values (the “short-term” memory), plus a state that holds information about all previous input and output values (the “long” memory).

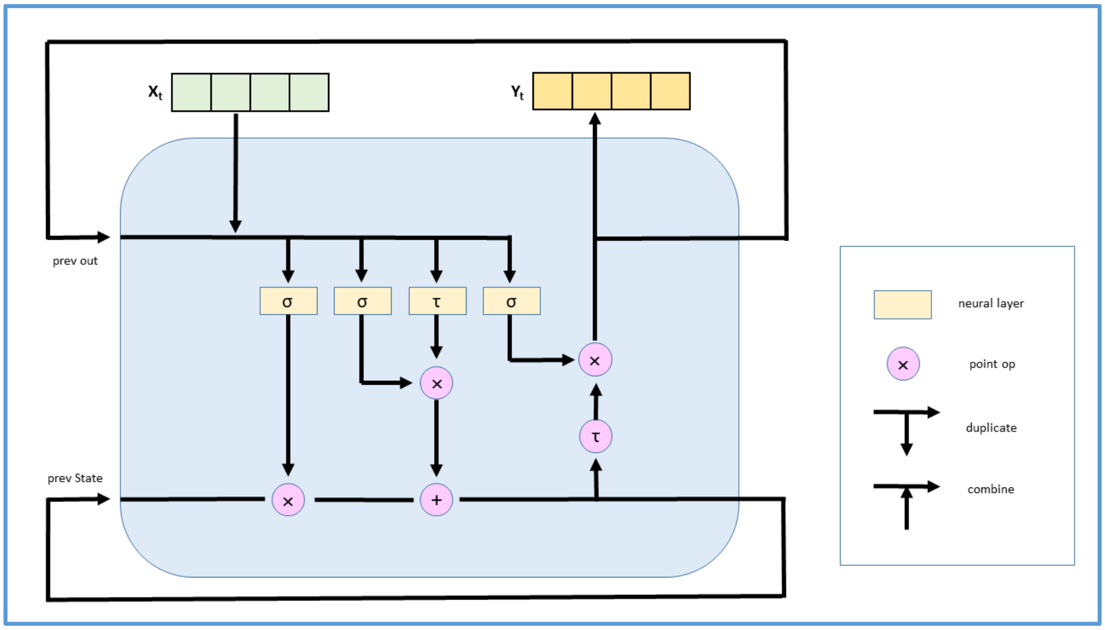


Figure 7-3: An LSTM Network Cell

The internal structure of an LSTM cell uses sigmoid neural layers, called gates, to control how much information is remembered and how much information is forgotten. A tanh neural layer holds state information. An LSTM cell also uses vector multiplication and addition.

The LSTM time series regression program

The LSTM time series regression program code is presented in Code Listing 7-1. After importing the required NumPy and CNTK packages, the demo program defines a helper function to read data from a CNTK format file into a mini-batch object:

```
def create_reader(path, input_dim, output_dim, rnd_order, sweeps):
  x_strm = C.io.StreamDef(field='prev', shape=input_dim, is_sparse=False)
  y_strm = C.io.StreamDef(field='change', shape=output_dim, is_sparse=False)
  z_strm = C.io.StreamDef(field='next', shape=output_dim, is_sparse=False)
  streams = C.io.StreamDefs(x_src=x_strm, y_src=y_strm, z_src=z_strm)
  deserial = C.io.CTFDeserializer(path, streams)
  mb_source = C.io.MinibatchSource(deserial, randomize=rnd_order,
max_sweeps=sweeps)
  return mb_source
```

Notice that **`create_reader()`** defines three streams instead of the usual two streams. It's not necessary to define a stream that corresponds to each tag in a CNTK-format data file. If you don't intend to use a field in a CNTK-format data file, you can ignore the field by not creating a stream for the field.

Code Listing 7-1: Time Series Regression

```
# airline_lstm.py
# time series regression with a CNTK LSTM

import numpy as np
import cntk as C

# data resembles:
# |prev 1.12 1.18 1.32 1.29 |next 1.21 |change 1.08035714
# |prev 1.18 1.32 1.29 1.21 |next 1.35 |change 1.14406780

def create_reader(path, input_dim, output_dim, rnd_order, sweeps):
  # rnd_order -> usually True for training
  # sweeps -> usually C.io.INFINITELY_REPEAT Or 1
  x_strm = C.io.StreamDef(field='prev', shape=input_dim, is_sparse=False)
  y_strm = C.io.StreamDef(field='change', shape=output_dim,
is_sparse=False)
  z_strm = C.io.StreamDef(field='next', shape=output_dim, is_sparse=False)
  streams = C.io.StreamDefs(x_src=x_strm, y_src=y_strm, z_src=z_strm)
  deserial = C.io.CTFDeserializer(path, streams)
  mb_source = C.io.MinibatchSource(deserial, randomize=rnd_order,
max_sweeps=sweeps)
  return mb_source

def mb_accuracy(mb, x_var, y_var, model, delta):
  num_correct = 0
  num_wrong = 0

  x_mat = mb[x_var].asarray()  # batch_size x 1 x features_dim
  y_mat = mb[y_var].asarray()  # batch_size x 1 x 1

  for i in range(len(mb[x_var])):
    v = model.eval(x_mat[i])           # 1 x 1 predicted change factor
    y = y_mat[i]                       # 1 x 1 actual change factor
    if np.abs(v[0,0] - y[0,0]) <  delta:  # close enough?
      num_correct += 1
    else:
      num_wrong += 1
  return (num_correct * 100.0) / (num_correct + num_wrong)

#
===========================================================================
=======
def main():
  print("\nBegin airline passenger time series regression LSTM \n")
  print("Using CNTK version = " + str(C.__version__) + "\n")

  train_file = ".\\Data\\airline_train_cntk.txt"

  # 1. create model
  input_dim = 4   # context pattern window
  output_dim = 1  # passenger count increase/decrease factor

  X = C.ops.sequence.input_variable(input_dim)  # sequence of 4 items
  Y = C.ops.input_variable(output_dim)  # change from X[0]
  Z = C.ops.input_variable(output_dim)  # actual next passenger count

  model = None
  with C.layers.default_options():
    model = C.layers.Recurrence(C.layers.LSTM(shape=256))(X)
    model = C.sequence.last(model)
    model = C.layers.Dense(output_dim)(model)

  # 2. create the learner and trainer
  learn_rate = 0.01
  tr_loss = C.squared_error(model, Y)
  learner = C.adam(model.parameters, learn_rate, 0.99)
  trainer = C.Trainer(model, (tr_loss), [learner])

  # 3. create the training reader; note rnd_order
  rdr = create_reader(train_file, input_dim, output_dim,
    rnd_order=True, sweeps=C.io.INFINITELY_REPEAT)
  airline_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src
  }

  # 4. train
  max_iter = 20000
  batch_size = 10

  print("Starting training \n")
  for i in range(0, max_iter):
    curr_mb = rdr.next_minibatch(batch_size, input_map=airline_input_map)
    trainer.train_minibatch(curr_mb)
    if i % int(max_iter/10) == 0:
      mcee = trainer.previous_minibatch_loss_average
      acc = mb_accuracy(curr_mb, X, Y, model, delta=0.10)  # program-
defined
      print("batch %6d: mean squared error = %8.4f  accuracy = %7.2f%%" \
        % (i, mcee, acc))
```

```
print("\nTraining complete")

mdl_name = ".\\Models\\airline_lstm.model"
model.save(mdl_name)  # CNTK v2 format is default

# 5. compute model accuracy on data
print("\nEvaluating LSTM model accuracy on test data with mb_acc() \n")
rdr = create_reader(train_file, input_dim, output_dim,
  rnd_order=False, sweeps=1)

airline_input_map = {
  X : rdr.streams.x_src,
  Y : rdr.streams.y_src
  # no need for Z at this point
}

num_items = 140
all_items = rdr.next_minibatch(num_items, input_map=airline_input_map)
acc = mb_accuracy(all_items, X, Y, model, delta=0.10)
print("Prediction accuracy on the 140-item data set = %0.2f%%" % acc)

# 6. save actual-predicted values to make a graph later
print("\nWriting actual-predicted pct values to file for graphing")
fout = open(".\\Data\\actual_predicted_lstm.txt", "w")
rdr = create_reader(train_file, input_dim, output_dim,
  rnd_order=False, sweeps=1)
airline_input_map = {
  X : rdr.streams.x_src,
  Y : rdr.streams.y_src,
  Z : rdr.streams.z_src   # actual next passenger count
}
num_items = 140
all_items = rdr.next_minibatch(num_items, input_map=airline_input_map)

x_mat = all_items[X].asarray()
y_mat = all_items[Y].asarray()
z_mat = all_items[Z].asarray()

for i in range(all_items[X].shape[0]):  # each item in the batch
  v = model.eval(x_mat[i])          # 1 x 1 predicted change
  y = y_mat[i]                      # 1 x 1 actual change
  z = z_mat[i]                      # actual next count
  x = x_mat[i]                      # first item in sequence
  p = x[0,0] * v[0,0]               # predicted next count
  fout.write("%0.2f, %0.2f\n" % (z[0,0], p))
fout.close()

# 7. predict passenger count for Jan. 1961
np.set_printoptions(precision = 4, suppress=True)
```

```
  in_seq = np.array([[5.08, 4.61, 3.90, 4.32]], dtype=np.float32)  # last 4
actual
  print("\nForecasting passenger count for Jan. 1961 using: ")
  print(in_seq[0])

  pred_change = model.eval({X: in_seq})
  print("\nForecast change factor is: ")
  print("%0.6f " % pred_change[0,0])

  pred_count = in_seq[0,0] * pred_change[0,0]
  print("Forecast passenger count = %0.4f" % pred_count)

  print("\nEnd demo \n")

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

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

Because the output of the demo LSTM network is a numeric value, it's necessary to implement a custom accuracy function if you want to measure accuracy. The demo program defines a function **mb_accuracy()** ("mini-batch accuracy") that computes the percentage of correct output values, given a mini-batch of input and output values, and a model to compute predicted output values:

```
def mb_accuracy(mb, x_var, y_var, model, delta):
  num_correct = 0; num_wrong = 0
  x_mat = mb[x_var].asarray()  # batch_size x 1 x features_dim
  y_mat = mb[y_var].asarray()  # batch_size x 1 x 1
  for i in range(len(mb[x_var])):
    v = model.eval(x_mat[i])           # 1 x 1 predicted change factor
    y = y_mat[i]                       # 1 x 1 actual change factor
    if np.abs(v[0,0] - y[0,0]) <  delta:  # close enough?
      num_correct += 1
    else:
      num_wrong += 1
  return (num_correct * 100.0) / (num_correct + num_wrong)
```

Recall that the LSTM network predicts a change factor, not a passenger count. So the accuracy function computes accuracy based on the predicted change factor, not the predicted passenger count. The actual next-month passenger count values in the data file are not used at all.

In the **main()** function, the demo program prepares to create an LSTM model with these statements:

```
train_file = ".\\Data\\airline_train_cntk.txt"

input_dim = 4   # context pattern window
output_dim = 1  # passenger count increase/decrease factor
X = C.ops.sequence.input_variable(input_dim)  # sequence of 4 items
Y = C.ops.input_variable(output_dim)  # pct change from X[0]
Z = C.ops.input_variable(output_dim)  # actual next passenger count
```

The number of values in each input sequence is set to 4. LSTMs can handle variable-sized input sequences. Such sequences often occur when processing natural language (sentence length for example). But for numeric data, a fixed sequence/window size is much more common.

When using an LSTM, instead of creating an input **Variable** object using the **cntk.ops.input_variable()** function, you create a special **Variable** object using the **cntk.ops.sequence.input_variable()** function.

The LSTM network cell is created like so:

```
model = None
with C.layers.default_options():
  model = C.layers.Recurrence(C.layers.LSTM(shape=256))(X)
  model = C.sequence.last(model)
  model = C.layers.Dense(output_dim)(model)
```

An LSTM cell is created inside a **Recurrence** layer so that its output values and state are fed back as inputs as shown in Figure 7-3. The **shape=256** argument to **LSTM()** is the size of the state-output and internal tanh layer that remembers previous inputs and output values. Instead of just passing an integer directly, that argument is sometimes passed indirectly as a variable, named something like **hidden_dim**.

Roughly speaking, larger values of the **shape** parameter mean the LSTM remembers more, but this isn’t necessarily always a good thing. For time series regression, the network probably won’t benefit from remembering input patterns far in the past.

Using the **sequence.last()** function is syntactically required. The **last()** function returns the last state-output in the LSTM cell (256 values), because only the last state generated by the set of input sequences is wanted.

LSTM networks often use a dropout layer that randomly drops nodes during training in order to reduce the likelihood of model overfitting. The demo doesn’t use dropout.

The last layer in the network is a standard **Dense** layer that adds a final node and condenses the 256 final-state values to a single value. Technically this isn’t entirely necessary, but adding an extra layer adds additional weights and an additional bias value, which in theory, gives the model more predictive power.

You could add multiple **Dense** layers, for example:

```
model = C.layers.Recurrence(C.layers.LSTM(shape=256))(X)
model = C.sequence.last(model)
model = C.layers.Dense(10)(model)
model = C.layers.Dense(output_dim)(model)
```

And you could add additional **Recurrence** layers, for example:

```
model = C.layers.Recurrence(C.layers.LSTM(shape=256))(X)
model = C.layers.Recurrence(C.layers.LSTM(shape=256))(model)
model = C.sequence.last(model)
model = C.layers.Dense(output_dim)(model)
```

Little is known about exotic LSTM network architectures, and investigating these architectures is an active area of deep neural network research.

Training an LSTM network

After the LSTM network is created, **Learner** and **Trainer** objects are created in the usual way:

```
learn_rate = 0.01
tr_loss = C.squared_error(model, Y)
learner = C.adam(model.parameters, learn_rate, 0.99)
trainer = C.Trainer(model, (tr_loss), [learner])
```

As a rule of thumb, as network architectures increase in size and complexity, the more important advanced **Learner** algorithms become. The demo uses the Adam (adaptive moment estimation) learner, which is a good first try when working with LSTM networks. The 0.99 argument is assigned to the **adam()** function's **momentum** parameter, but somewhat confusingly is described by the CNTK documentation as the **beta1** parameter. Choosing reasonable values for advanced learning functions when there are no default values requires research, trial and error, and patience.

The demo program creates a reader for the time series training data like so:

```
rdr = create_reader(train_file, input_dim, output_dim,
  rnd_order=True, sweeps=C.io.INFINITELY_REPEAT)

airline_input_map = {
  X : rdr.streams.x_src,
  Y : rdr.streams.y_src
}
```

The **create_reader()** function is called using **rnd_order=True**, which means that each batch of training items will be processed in a random order. This seems counterintuitive because the training data is one big sequence of values. In discussions with my colleagues, there is roughly a 50-50 split in opinion regarding whether training data should be processed in a random order or in a sequential order. As was the case with simple neural networks when they were new (in the 1980s), many fundamental concepts about LSTM networks are not currently well understood.

The actual training of the LSTM network for time series regression is performed by these statements:

```
max_iter = 20000
batch_size = 10
print("Starting training \n")
for i in range(0, max_iter):
  curr_mb = rdr.next_minibatch(batch_size, input_map=airline_input_map)
  trainer.train_minibatch(curr_mb)
  if i % int(max_iter/10) == 0:
    mcee = trainer.previous_minibatch_loss_average
    acc = mb_accuracy(curr_mb, X, Y, model, delta=0.10)  # program-defined
    print("batch %6d: mean squared error = %8.4f  accuracy = %7.2f%%" \
      % (i, mcee, acc))
print("\nTraining complete")

mdl_name = ".\\Models\\airline_lstm.model"
model.save(mdl_name)  # CNTK v2 format is default
```

Typically, training an LSTM network requires many more training iterations than training a single hidden layer neural network. Monitoring error/loss during training is critically important because when working with LSTM networks, based on my experience, training failure is the rule rather than the exception. The demo prints error 10 times during training: **`int(max_iter/10)`**.

The demo program saves the trained model after training completes (using the default CNTK v2 format). An alternative is to also save intermediate models during training, every so often. For example, inside the **`if`** condition in the code above, you could write:

```
mdl_name = ".\\Models\\airline_lstm_" + str(i) + ".model"
model.save(mdl_name)
```

And instead of embedding just the training iteration into the intermediate model name, you could also embed information such as the loss/error associated with the model. The point is that for non-demo datasets, training can take hours or days, and saving intermediate models is necessary in practice. CNTK supports a checkpoint mechanism that is outside the scope of this e-book.

The average model accuracy on the current batch of 10 training items is printed every so often. After training, the demo program also computes and prints the model accuracy on the entire 140-item dataset, using the program-defined **`mb_accuracy()`** function. In a non-demo scenario, you'd likely also want to compute and print the average loss/error for the entire dataset. For example, you could code a program-defined function:

```
def mb_error(mb, x_var, y_var, model):
  sum_sq_err = 0.0
  x_mat = mb[x_var].asarray()  # batch_size x 1 x features_dim
  y_mat = mb[y_var].asarray()  # batch_size x 1 x 1
  for i in range(len(mb[x_var])):
    v = model.eval(x_mat[i])          # 1 x 1 predicted change factor
    y = y_mat[i]                      # 1 x 1 actual change factor
    sum_sq_err += (v[0,0] - y[0,0]) * (v[0,0] - y[0,0])
  return sum_sq_err / len(mb[x_var])
```

It would likely be a mistake to try and use the built-in **previous_minibatch_loss_average** property of the **Trainer** object, because you'd have to call the **train()** method, which would modify the trained model.

Prediction and forecasting

When performing time series regression, it's almost always useful to graph results. After training, the demo LSTM program walks through the 140-item training data, computes a predicted passenger count, and then writes actual count and predicted count pairs to a text file.

First, the program prepares the analysis:

```
  print("\nWriting actual-predicted passenger counts to file for graphing")
  fout = open(".\\Data\\actual_predicted_lstm.txt", "w")
  rdr = create_reader(train_file, input_dim, output_dim,
    rnd_order=False, sweeps=1)
  airline_input_map = {
    X : rdr.streams.x_src,
    Y : rdr.streams.y_src,
    Z : rdr.streams.z_src   # actual next passenger count
  }
  num_items = 140
  all_items = rdr.next_minibatch(num_items, input_map=airline_input_map)
```

Even though the LSTM network model predicts a change factor rather than a direct passenger count, it makes more sense to graph actual-predicted passenger counts rather than actual-predicted change values (although you could graph both).

Instead of computing an actual passenger count from the actual change factor and the first value of the input sequence, the demo program reads the actual counts, which were stored in the training data file for convenience. Notice the input map for the **Reader** object sets up all three data streams. Next, the demo program extracts the input sequences, the actual changes, and the actual next count data into three matrices:

```
  x_mat = all_items[X].asarray()
  y_mat = all_items[Y].asarray()
  z_mat = all_items[Z].asarray()
```

Recall that **all_items** is a CNTK mini-batch object which is implemented as a dictionary object, and the **X**, **Y**, and **Z Variable** objects are used as keys. Next, each item is processed:

```
  for i in range(all_items[X].shape[0]):  # each item in the batch
    v = model.eval(x_mat[i])           # 1 x 1 predicted change
    y = y_mat[i]                       # 1 x 1 actual change
    z = z_mat[i]                       # actual next count
    x = x_mat[i]                       # first item in sequence
    p = x[0,0] * v[0,0]                # predicted next count
    fout.write("%0.2f, %0.2f\n" % (z[0,0], p))

  fout.close()
```

The **eval()** function computes the predicted change values into **v**, so the predicted passenger count is the first item in the current sequence, **x[0,0]**, times **v[0,0]**, because both values are returned into 1×1 matrices. In practice, you will spend a lot of time determining the shape of different CNTK objects using the **shape** property and a **print()** function.

Each actual-predicted comma-separated pair is written to the specified text file. An alternative is to store actual-predicted pairs into a Python list using the **append()** or **insert()** function.

After writing actual-predicted data to the text file, I opened the file with Excel and created the graph shown in Figure 7-2. An alternative is to programmatically create a graph from within your script using the Python **matplotlib** package. Many of the CNTK documentation examples use this approach. However, **matplotlib** code can obscure the main ideas of CNTK model creation, training, and evaluation, so I've used an external technique to create the graph.

In time series regression, making a prediction for input values beyond the range of the input values in the training data is usually called forecasting or extrapolation. The demo program predicts the change for the first time period past the training data (January 1961, month 141) and then uses the change to compute the predicted next passenger count:

```
  in_seq = np.array([[5.08, 4.61, 3.90, 4.32]], dtype=np.float32)  # last 4
actual
  print("\nForecasting passenger count for Jan. 1961 using: ")
  print(in_seq[0])
  pred_change = model.eval({X: in_seq})
  print("\nForecast change factor is: ")
  print("%0.6f " % pred_change[0,0])
  pred_count = in_seq[0,0] * pred_change[0,0]
  print("Forecast passenger count = %0.4f" % pred_count)
```

The **in_seq** 1×4 array holds the last four normalized actual passenger counts. The array is passed to the **eval()** function via the **X** key to create an anonymous dictionary object. The array could have been passed directly. The return value is a 1×1 matrix that holds the predicted change. That change value is multiplied by the first item in the input sequence to determine the predicted passenger count.

In many situations, you only want to predict one time-unit ahead. However, you can extrapolate as many time units as you wish. The short program in Code Listing 7-2 extrapolates eight time units past the training data.

Code Listing 7-2: Extrapolating Time Series

```
# airline_forecast_lstm.py
# CNTK v2.3 Ananconda 4.1.1

import numpy as np
import cntk as C

model = C.ops.functions.Function.load(".\\Models\\airline_lstm.model")

np.set_printoptions(precision = 2, suppress=True)
curr = np.array([[5.08, 4.61, 3.90, 4.32]], dtype=np.float32)

for i in range(8):
  pred_change = model.eval(curr)
  print("\nCurrent counts : ", end=""); print(curr)
  print("Forecast change factor = %0.4f" % pred_change[0,0])
  pred_count = curr[0,0] * pred_change[0,0]
  print("Forecast passenger count = %0.2f" % pred_count)

  for j in range(3):
    curr[0,j] = curr[0,j+1]
  curr[0,3] = pred_count
```

As you can see in the graph in Figure 7-2, you must use great caution when extrapolating time series data, because the accuracy of the predictions will often degrade quickly.

Exercise

The Appendix of this e-book has raw data for monthly initial unemployment claims for King County Washington (Seattle area) from January 2005 through December 2014. Using the program in Code Listing 7-1 as a guide, create an LSTM time series regression model.

The raw data came from here.

The raw comma-delimited data has 120 items and looks like:

```
8/1/2007,6110
9/1/2007,5298
. . .
11/1/2014,6993
12/1/2014,7978
```

You will have to decide how to normalize the data, and how to set up a CNTK-format data file. I suggest using Excel or a similar spreadsheet to preprocess the raw data.

Appendix A Datasets

Chapter 2: age_edu_sex.txt (comma delimited)

```
1.0, 2.0, 0
3.0, 4.0, 0
5.0, 2.0, 0
6.0, 3.0, 0
8.0, 1.0, 0
9.0, 2.0, 0
1.0, 4.0, 1
2.0, 5.0, 1
4.0, 6.0, 1
6.0, 5.0, 1
7.0, 3.0, 1
8.0, 5.0, 1
```

Chapter 4: iris_train_cntk.txt (space delimited)

```
|attribs 5.1 3.5 1.4 0.2 |species 1 0 0
|attribs 4.9 3.0 1.4 0.2 |species 1 0 0
|attribs 4.7 3.2 1.3 0.2 |species 1 0 0
|attribs 4.6 3.1 1.5 0.2 |species 1 0 0
|attribs 5.0 3.6 1.4 0.2 |species 1 0 0
|attribs 5.4 3.9 1.7 0.4 |species 1 0 0
|attribs 4.6 3.4 1.4 0.3 |species 1 0 0
|attribs 5.0 3.4 1.5 0.2 |species 1 0 0
|attribs 4.4 2.9 1.4 0.2 |species 1 0 0
|attribs 4.9 3.1 1.5 0.1 |species 1 0 0
|attribs 5.4 3.7 1.5 0.2 |species 1 0 0
|attribs 4.8 3.4 1.6 0.2 |species 1 0 0
|attribs 4.8 3.0 1.4 0.1 |species 1 0 0
|attribs 4.3 3.0 1.1 0.1 |species 1 0 0
|attribs 5.8 4.0 1.2 0.2 |species 1 0 0
|attribs 5.7 4.4 1.5 0.4 |species 1 0 0
|attribs 5.4 3.9 1.3 0.4 |species 1 0 0
|attribs 5.1 3.5 1.4 0.3 |species 1 0 0
|attribs 5.7 3.8 1.7 0.3 |species 1 0 0
|attribs 5.1 3.8 1.5 0.3 |species 1 0 0
|attribs 5.4 3.4 1.7 0.2 |species 1 0 0
|attribs 5.1 3.7 1.5 0.4 |species 1 0 0
|attribs 4.6 3.6 1.0 0.2 |species 1 0 0
|attribs 5.1 3.3 1.7 0.5 |species 1 0 0
|attribs 4.8 3.4 1.9 0.2 |species 1 0 0
|attribs 5.0 3.0 1.6 0.2 |species 1 0 0
|attribs 5.0 3.4 1.6 0.4 |species 1 0 0
|attribs 5.2 3.5 1.5 0.2 |species 1 0 0
|attribs 5.2 3.4 1.4 0.2 |species 1 0 0
|attribs 4.7 3.2 1.6 0.2 |species 1 0 0
|attribs 4.8 3.1 1.6 0.2 |species 1 0 0
|attribs 5.4 3.4 1.5 0.4 |species 1 0 0
|attribs 5.2 4.1 1.5 0.1 |species 1 0 0
|attribs 5.5 4.2 1.4 0.2 |species 1 0 0
```

```
|attribs 4.9 3.1 1.5 0.2 |species 1 0 0
|attribs 5.0 3.2 1.2 0.2 |species 1 0 0
|attribs 5.5 3.5 1.3 0.2 |species 1 0 0
|attribs 4.9 3.6 1.4 0.1 |species 1 0 0
|attribs 4.4 3.0 1.3 0.2 |species 1 0 0
|attribs 5.1 3.4 1.5 0.2 |species 1 0 0
|attribs 7.0 3.2 4.7 1.4 |species 0 1 0
|attribs 6.4 3.2 4.5 1.5 |species 0 1 0
|attribs 6.9 3.1 4.9 1.5 |species 0 1 0
|attribs 5.5 2.3 4.0 1.3 |species 0 1 0
|attribs 6.5 2.8 4.6 1.5 |species 0 1 0
|attribs 5.7 2.8 4.5 1.3 |species 0 1 0
|attribs 6.3 3.3 4.7 1.6 |species 0 1 0
|attribs 4.9 2.4 3.3 1.0 |species 0 1 0
|attribs 6.6 2.9 4.6 1.3 |species 0 1 0
|attribs 5.2 2.7 3.9 1.4 |species 0 1 0
|attribs 5.0 2.0 3.5 1.0 |species 0 1 0
|attribs 5.9 3.0 4.2 1.5 |species 0 1 0
|attribs 6.0 2.2 4.0 1.0 |species 0 1 0
|attribs 6.1 2.9 4.7 1.4 |species 0 1 0
|attribs 5.6 2.9 3.6 1.3 |species 0 1 0
|attribs 6.7 3.1 4.4 1.4 |species 0 1 0
|attribs 5.6 3.0 4.5 1.5 |species 0 1 0
|attribs 5.8 2.7 4.1 1.0 |species 0 1 0
|attribs 6.2 2.2 4.5 1.5 |species 0 1 0
|attribs 5.6 2.5 3.9 1.1 |species 0 1 0
|attribs 5.9 3.2 4.8 1.8 |species 0 1 0
|attribs 6.1 2.8 4.0 1.3 |species 0 1 0
|attribs 6.3 2.5 4.9 1.5 |species 0 1 0
|attribs 6.1 2.8 4.7 1.2 |species 0 1 0
|attribs 6.4 2.9 4.3 1.3 |species 0 1 0
|attribs 6.6 3.0 4.4 1.4 |species 0 1 0
|attribs 6.8 2.8 4.8 1.4 |species 0 1 0
|attribs 6.7 3.0 5.0 1.7 |species 0 1 0
|attribs 6.0 2.9 4.5 1.5 |species 0 1 0
|attribs 5.7 2.6 3.5 1.0 |species 0 1 0
|attribs 5.5 2.4 3.8 1.1 |species 0 1 0
|attribs 5.5 2.4 3.7 1.0 |species 0 1 0
|attribs 5.8 2.7 3.9 1.2 |species 0 1 0
|attribs 6.0 2.7 5.1 1.6 |species 0 1 0
|attribs 5.4 3.0 4.5 1.5 |species 0 1 0
|attribs 6.0 3.4 4.5 1.6 |species 0 1 0
|attribs 6.7 3.1 4.7 1.5 |species 0 1 0
|attribs 6.3 2.3 4.4 1.3 |species 0 1 0
|attribs 5.6 3.0 4.1 1.3 |species 0 1 0
|attribs 5.5 2.5 4.0 1.3 |species 0 1 0
|attribs 6.3 3.3 6.0 2.5 |species 0 0 1
|attribs 5.8 2.7 5.1 1.9 |species 0 0 1
|attribs 7.1 3.0 5.9 2.1 |species 0 0 1
|attribs 6.3 2.9 5.6 1.8 |species 0 0 1
|attribs 6.5 3.0 5.8 2.2 |species 0 0 1
|attribs 7.6 3.0 6.6 2.1 |species 0 0 1
|attribs 4.9 2.5 4.5 1.7 |species 0 0 1
|attribs 7.3 2.9 6.3 1.8 |species 0 0 1
|attribs 6.7 2.5 5.8 1.8 |species 0 0 1
|attribs 7.2 3.6 6.1 2.5 |species 0 0 1
|attribs 6.5 3.2 5.1 2.0 |species 0 0 1
|attribs 6.4 2.7 5.3 1.9 |species 0 0 1
```

```
|attribs 6.8 3.0 5.5 2.1 |species 0 0 1
|attribs 5.7 2.5 5.0 2.0 |species 0 0 1
|attribs 5.8 2.8 5.1 2.4 |species 0 0 1
|attribs 6.4 3.2 5.3 2.3 |species 0 0 1
|attribs 6.5 3.0 5.5 1.8 |species 0 0 1
|attribs 7.7 3.8 6.7 2.2 |species 0 0 1
|attribs 7.7 2.6 6.9 2.3 |species 0 0 1
|attribs 6.0 2.2 5.0 1.5 |species 0 0 1
|attribs 6.9 3.2 5.7 2.3 |species 0 0 1
|attribs 5.6 2.8 4.9 2.0 |species 0 0 1
|attribs 7.7 2.8 6.7 2.0 |species 0 0 1
|attribs 6.3 2.7 4.9 1.8 |species 0 0 1
|attribs 6.7 3.3 5.7 2.1 |species 0 0 1
|attribs 7.2 3.2 6.0 1.8 |species 0 0 1
|attribs 6.2 2.8 4.8 1.8 |species 0 0 1
|attribs 6.1 3.0 4.9 1.8 |species 0 0 1
|attribs 6.4 2.8 5.6 2.1 |species 0 0 1
|attribs 7.2 3.0 5.8 1.6 |species 0 0 1
|attribs 7.4 2.8 6.1 1.9 |species 0 0 1
|attribs 7.9 3.8 6.4 2.0 |species 0 0 1
|attribs 6.4 2.8 5.6 2.2 |species 0 0 1
|attribs 6.3 2.8 5.1 1.5 |species 0 0 1
|attribs 6.1 2.6 5.6 1.4 |species 0 0 1
|attribs 7.7 3.0 6.1 2.3 |species 0 0 1
|attribs 6.3 3.4 5.6 2.4 |species 0 0 1
|attribs 6.4 3.1 5.5 1.8 |species 0 0 1
|attribs 6.0 3.0 4.8 1.8 |species 0 0 1
|attribs 6.9 3.1 5.4 2.1 |species 0 0 1
```

Chapter 4: iris_test_cntk.txt (space delimited)

```
|attribs 5.0 3.5 1.3 0.3 |species 1 0 0
|attribs 4.5 2.3 1.3 0.3 |species 1 0 0
|attribs 4.4 3.2 1.3 0.2 |species 1 0 0
|attribs 5.0 3.5 1.6 0.6 |species 1 0 0
|attribs 5.1 3.8 1.9 0.4 |species 1 0 0
|attribs 4.8 3.0 1.4 0.3 |species 1 0 0
|attribs 5.1 3.8 1.6 0.2 |species 1 0 0
|attribs 4.6 3.2 1.4 0.2 |species 1 0 0
|attribs 5.3 3.7 1.5 0.2 |species 1 0 0
|attribs 5.0 3.3 1.4 0.2 |species 1 0 0
|attribs 5.5 2.6 4.4 1.2 |species 0 1 0
|attribs 6.1 3.0 4.6 1.4 |species 0 1 0
|attribs 5.8 2.6 4.0 1.2 |species 0 1 0
|attribs 5.0 2.3 3.3 1.0 |species 0 1 0
|attribs 5.6 2.7 4.2 1.3 |species 0 1 0
|attribs 5.7 3.0 4.2 1.2 |species 0 1 0
|attribs 5.7 2.9 4.2 1.3 |species 0 1 0
|attribs 6.2 2.9 4.3 1.3 |species 0 1 0
|attribs 5.1 2.5 3.0 1.1 |species 0 1 0
|attribs 5.7 2.8 4.1 1.3 |species 0 1 0
|attribs 6.7 3.1 5.6 2.4 |species 0 0 1
|attribs 6.9 3.1 5.1 2.3 |species 0 0 1
|attribs 5.8 2.7 5.1 1.9 |species 0 0 1
|attribs 6.8 3.2 5.9 2.3 |species 0 0 1
|attribs 6.7 3.3 5.7 2.5 |species 0 0 1
|attribs 6.7 3.0 5.2 2.3 |species 0 0 1
```

```
|attribs 6.3 2.5 5.0 1.9 |species 0 0 1
|attribs 6.5 3.0 5.2 2.0 |species 0 0 1
|attribs 6.2 3.4 5.4 2.3 |species 0 0 1
|attribs 5.9 3.0 5.1 1.8 |species 0 0 1
```

Chapter 5: banknote_train_cntk.txt (tab delimited)

```
|stats   3.62160000   8.66610000  -2.80730000  -0.44699000 |forgery 0 1 |# authentic
|stats   4.54590000   8.16740000  -2.45860000  -1.46210000 |forgery 0 1 |# authentic
|stats   3.86600000  -2.63830000   1.92420000   0.10645000 |forgery 0 1 |# authentic
|stats   3.45660000   9.52280000  -4.01120000  -3.59440000 |forgery 0 1 |# authentic
|stats   0.32924000  -4.45520000   4.57180000  -0.98880000 |forgery 0 1 |# authentic
|stats   4.36840000   9.67180000  -3.96060000  -3.16250000 |forgery 0 1 |# authentic
|stats   3.59120000   3.01290000   0.72888000   0.56421000 |forgery 0 1 |# authentic
|stats   2.09220000  -6.81000000   8.46360000  -0.60216000 |forgery 0 1 |# authentic
|stats   3.20320000   5.75880000  -0.75345000  -0.61251000 |forgery 0 1 |# authentic
|stats   1.53560000   9.17720000  -2.27180000  -0.73535000 |forgery 0 1 |# authentic
|stats   1.22470000   8.77790000  -2.21350000  -0.80647000 |forgery 0 1 |# authentic
|stats   3.98990000  -2.70660000   2.39460000   0.86291000 |forgery 0 1 |# authentic
|stats   1.89930000   7.66250000   0.15394000  -3.11080000 |forgery 0 1 |# authentic
|stats  -1.57680000  10.84300000   2.54620000  -2.93620000 |forgery 0 1 |# authentic
|stats   3.40400000   8.72610000  -2.99150000  -0.57242000 |forgery 0 1 |# authentic
|stats   4.67650000  -3.38950000   3.48960000   1.47710000 |forgery 0 1 |# authentic
|stats   2.67190000   3.06460000   0.37158000   0.58619000 |forgery 0 1 |# authentic
|stats   0.80355000   2.84730000   4.34390000   0.60170000 |forgery 0 1 |# authentic
|stats   1.44790000  -4.87940000   8.34280000  -2.10860000 |forgery 0 1 |# authentic
|stats   5.24230000  11.02720000  -4.35300000  -4.10130000 |forgery 0 1 |# authentic
|stats   5.78670000   7.89020000  -2.61960000  -0.48708000 |forgery 0 1 |# authentic
|stats   0.32920000  -4.45520000   4.57180000  -0.98880000 |forgery 0 1 |# authentic
|stats   3.93620000  10.16220000  -3.82350000  -4.01720000 |forgery 0 1 |# authentic
|stats   0.93584000   8.88550000  -1.68310000  -1.65990000 |forgery 0 1 |# authentic
|stats   4.43380000   9.88700000  -4.67950000  -3.74830000 |forgery 0 1 |# authentic
|stats   0.70570000  -5.49810000   8.33680000  -2.87150000 |forgery 0 1 |# authentic
|stats   1.14320000  -3.74130000   5.57770000  -0.63578000 |forgery 0 1 |# authentic
|stats  -0.38214000   8.39090000   2.16240000  -3.74050000 |forgery 0 1 |# authentic
|stats   6.56330000   9.81870000  -4.41130000  -3.22580000 |forgery 0 1 |# authentic
|stats   4.89060000  -3.35840000   3.42020000   1.09050000 |forgery 0 1 |# authentic
|stats  -0.24811000  -0.17797000   4.90680000   0.15429000 |forgery 0 1 |# authentic
|stats   1.48840000   3.62740000   3.30800000   0.48921000 |forgery 0 1 |# authentic
|stats   4.29690000   7.61700000  -2.38740000  -0.96164000 |forgery 0 1 |# authentic
|stats  -0.96511000   9.41110000   1.73050000  -4.86290000 |forgery 0 1 |# authentic
|stats  -1.61620000   0.80908000   8.16280000   0.60817000 |forgery 0 1 |# authentic
|stats   2.43910000   6.44170000  -0.80743000  -0.69139000 |forgery 0 1 |# authentic
|stats   2.68810000   6.01950000  -0.46641000  -0.69268000 |forgery 0 1 |# authentic
|stats   3.62890000   0.81322000   1.62770000   0.77627000 |forgery 0 1 |# authentic
|stats   4.56790000   3.19290000  -2.10550000   0.29653000 |forgery 0 1 |# authentic
|stats   3.48050000   9.70080000  -3.75410000  -3.43790000 |forgery 0 1 |# authentic
|stats  -1.39710000   3.31910000  -1.39270000  -1.99480000 |forgery 1 0 |# fake
|stats   0.39012000  -0.14279000  -0.03199400   0.35084000 |forgery 1 0 |# fake
|stats  -1.66770000  -7.15350000   7.89290000   0.96765000 |forgery 1 0 |# fake
|stats  -3.84830000 -12.80470000  15.68240000  -1.28100000 |forgery 1 0 |# fake
|stats  -3.56810000  -8.21300000  10.08300000   0.96765000 |forgery 1 0 |# fake
|stats  -2.28040000  -0.30626000   1.33470000   1.37630000 |forgery 1 0 |# fake
|stats  -1.75820000   2.73970000  -2.53230000  -2.23400000 |forgery 1 0 |# fake
|stats  -0.89409000   3.19910000  -1.82190000  -2.94520000 |forgery 1 0 |# fake
|stats   0.34340000   0.12415000  -0.28733000   0.14654000 |forgery 1 0 |# fake
|stats  -0.98540000  -6.66100000   5.82450000   0.54610000 |forgery 1 0 |# fake
```

```
|stats  -2.41150000  -9.13590000   9.34440000  -0.65259000 |forgery 1 0 |# fake
|stats  -1.52520000  -6.25340000   5.35240000   0.59912000 |forgery 1 0 |# fake
|stats  -0.61442000  -0.09105800  -0.31818000   0.50214000 |forgery 1 0 |# fake
|stats  -0.36506000   2.89280000  -3.64610000  -3.06030000 |forgery 1 0 |# fake
|stats  -5.90340000   6.56790000   0.67661000  -6.67970000 |forgery 1 0 |# fake
|stats  -1.82150000   2.75210000  -0.72261000  -2.35300000 |forgery 1 0 |# fake
|stats  -0.77461000  -1.87680000   2.40230000   1.13190000 |forgery 1 0 |# fake
|stats  -1.81870000  -9.03660000   9.01620000  -0.12243000 |forgery 1 0 |# fake
|stats  -3.58010000 -12.93090000  13.17790000  -2.56770000 |forgery 1 0 |# fake
|stats  -1.82190000  -6.88240000   5.46810000   0.05731300 |forgery 1 0 |# fake
|stats  -0.34810000  -0.38696000  -0.47841000   0.62627000 |forgery 1 0 |# fake
|stats   0.47368000   3.36050000  -4.50640000  -4.04310000 |forgery 1 0 |# fake
|stats  -3.40830000   4.85870000  -0.76888000  -4.86680000 |forgery 1 0 |# fake
|stats  -1.66620000  -0.30005000   1.42380000   0.02498600 |forgery 1 0 |# fake
|stats  -2.09620000  -7.10590000   6.61880000  -0.33708000 |forgery 1 0 |# fake
|stats  -2.66850000 -10.45190000   9.11390000  -1.73230000 |forgery 1 0 |# fake
|stats  -0.47465000  -4.34960000   1.99010000   0.75170000 |forgery 1 0 |# fake
|stats   1.05520000   1.18570000  -2.64110000   0.11033000 |forgery 1 0 |# fake
|stats   1.16440000   3.80950000  -4.94080000  -4.09090000 |forgery 1 0 |# fake
|stats  -4.47790000   7.37080000  -0.31218000  -6.77540000 |forgery 1 0 |# fake
|stats  -2.73380000   0.45523000   2.43910000   0.21766000 |forgery 1 0 |# fake
|stats  -2.28600000  -5.44840000   5.80390000   0.88231000 |forgery 1 0 |# fake
|stats  -1.62440000  -6.34440000   4.65750000   0.16981000 |forgery 1 0 |# fake
|stats   0.50813000   0.47799000  -1.98040000   0.57714000 |forgery 1 0 |# fake
|stats   1.64080000   4.25030000  -4.90230000  -2.66210000 |forgery 1 0 |# fake
|stats   0.81583000   4.84000000  -5.26130000  -6.08230000 |forgery 1 0 |# fake
|stats  -5.49010000   9.10480000  -0.38758000  -5.97630000 |forgery 1 0 |# fake
|stats  -3.22380000   2.79350000   0.32274000  -0.86078000 |forgery 1 0 |# fake
|stats  -2.06310000  -1.51470000   1.21900000   0.44524000 |forgery 1 0 |# fake
|stats  -0.91318000  -2.01130000  -0.19565000   0.06636500 |forgery 1 0 |# fake
```

Chapter 5: banknote_test_cntk.txt (tab delimited)

```
|stats   4.17110000   8.72200000  -3.02240000  -0.59699000 |forgery 0 1 |# authentic
|stats  -0.20620000   9.22070000  -3.70440000  -6.81030000 |forgery 0 1 |# authentic
|stats  -0.00689190   9.29310000  -0.41243000  -1.96380000 |forgery 0 1 |# authentic
|stats   0.96441000   5.83950000   2.32350000   0.06636500 |forgery 0 1 |# authentic
|stats   2.85610000   6.91760000  -0.79372000   0.48403000 |forgery 0 1 |# authentic
|stats  -0.78690000   9.56630000  -3.78670000  -7.50340000 |forgery 0 1 |# authentic
|stats   2.08430000   6.62580000   0.48382000  -2.21340000 |forgery 0 1 |# authentic
|stats  -0.78690000   9.56630000  -3.78670000  -7.50340000 |forgery 0 1 |# authentic
|stats   3.91020000   6.06500000  -2.45340000  -0.68234000 |forgery 0 1 |# authentic
|stats   1.63490000   3.28600000   2.87530000   0.08705400 |forgery 0 1 |# authentic
|stats   0.60050000   1.93270000  -3.28880000  -0.32415000 |forgery 1 0 |# fake
|stats   0.91315000   3.33770000  -4.05570000  -1.67410000 |forgery 1 0 |# fake
|stats  -0.28015000   3.07290000  -3.38570000  -2.91550000 |forgery 1 0 |# fake
|stats  -3.60850000   3.32530000  -0.51954000  -3.57370000 |forgery 1 0 |# fake
|stats  -6.20030000   8.68060000   0.00913440  -3.70300000 |forgery 1 0 |# fake
|stats  -4.29320000   3.34190000   0.77258000  -0.99785000 |forgery 1 0 |# fake
|stats  -3.02650000  -0.06208800   0.68604000  -0.05518600 |forgery 1 0 |# fake
|stats  -1.70150000  -0.01035600  -0.99337000  -0.53104000 |forgery 1 0 |# fake
|stats  -0.64326000   2.47480000  -2.94520000  -1.02760000 |forgery 1 0 |# fake
|stats  -0.86339000   1.93480000  -2.37290000  -1.08970000 |forgery 1 0 |# fake
```

Chapter 5: banknote_train_cntk_onenode.txt (tab delimited)

```
|stats   3.62160000   8.66610000  -2.80730000  -0.44699000 |forgery 0 |# authentic
|stats   4.54590000   8.16740000  -2.45860000  -1.46210000 |forgery 0 |# authentic
|stats   3.86600000  -2.63830000   1.92420000   0.10645000 |forgery 0 |# authentic
|stats   3.45660000   9.52280000  -4.01120000  -3.59440000 |forgery 0 |# authentic
|stats   0.32924000  -4.45520000   4.57180000  -0.98880000 |forgery 0 |# authentic
|stats   4.36840000   9.67180000  -3.96060000  -3.16250000 |forgery 0 |# authentic
|stats   3.59120000   3.01290000   0.72888000   0.56421000 |forgery 0 |# authentic
|stats   2.09220000  -6.81000000   8.46360000  -0.60216000 |forgery 0 |# authentic
|stats   3.20320000   5.75880000  -0.75345000  -0.61251000 |forgery 0 |# authentic
|stats   1.53560000   9.17720000  -2.27180000  -0.73535000 |forgery 0 |# authentic
|stats   1.22470000   8.77790000  -2.21350000  -0.80647000 |forgery 0 |# authentic
|stats   3.98990000  -2.70660000   2.39460000   0.86291000 |forgery 0 |# authentic
|stats   1.89930000   7.66250000   0.15394000  -3.11080000 |forgery 0 |# authentic
|stats  -1.57680000  10.84300000   2.54620000  -2.93620000 |forgery 0 |# authentic
|stats   3.40400000   8.72610000  -2.99150000  -0.57242000 |forgery 0 |# authentic
|stats   4.67650000  -3.38950000   3.48960000   1.47710000 |forgery 0 |# authentic
|stats   2.67190000   3.06460000   0.37158000   0.58619000 |forgery 0 |# authentic
|stats   0.80355000   2.84730000   4.34390000   0.60170000 |forgery 0 |# authentic
|stats   1.44790000  -4.87940000   8.34280000  -2.10860000 |forgery 0 |# authentic
|stats   5.24230000  11.02720000  -4.35300000  -4.10130000 |forgery 0 |# authentic
|stats   5.78670000   7.89020000  -2.61960000  -0.48708000 |forgery 0 |# authentic
|stats   0.32920000  -4.45520000   4.57180000  -0.98880000 |forgery 0 |# authentic
|stats   3.93620000  10.16220000  -3.82350000  -4.01720000 |forgery 0 |# authentic
|stats   0.93584000   8.88550000  -1.68310000  -1.65990000 |forgery 0 |# authentic
|stats   4.43380000   9.88700000  -4.67950000  -3.74830000 |forgery 0 |# authentic
|stats   0.70570000  -5.49810000   8.33680000  -2.87150000 |forgery 0 |# authentic
|stats   1.14320000  -3.74130000   5.57770000  -0.63578000 |forgery 0 |# authentic
|stats  -0.38214000   8.39090000   2.16240000  -3.74050000 |forgery 0 |# authentic
|stats   6.56330000   9.81870000  -4.41130000  -3.22580000 |forgery 0 |# authentic
|stats   4.89060000  -3.35840000   3.42020000   1.09050000 |forgery 0 |# authentic
|stats  -0.24811000  -0.17797000   4.90680000   0.15429000 |forgery 0 |# authentic
|stats   1.48840000   3.62740000   3.30800000   0.48921000 |forgery 0 |# authentic
|stats   4.29690000   7.61700000  -2.38740000  -0.96164000 |forgery 0 |# authentic
|stats  -0.96511000   9.41110000   1.73050000  -4.86290000 |forgery 0 |# authentic
|stats  -1.61620000   0.80908000   8.16280000   0.60817000 |forgery 0 |# authentic
|stats   2.43910000   6.44170000  -0.80743000  -0.69139000 |forgery 0 |# authentic
|stats   2.68810000   6.01950000  -0.46641000  -0.69268000 |forgery 0 |# authentic
|stats   3.62890000   0.81322000   1.62770000   0.77627000 |forgery 0 |# authentic
|stats   4.56790000   3.19290000  -2.10550000   0.29653000 |forgery 0 |# authentic
|stats   3.48050000   9.70080000  -3.75410000  -3.43790000 |forgery 0 |# authentic
|stats  -1.39710000   3.31910000  -1.39270000  -1.99480000 |forgery 1 |# fake
|stats   0.39012000  -0.14279000  -0.03199400   0.35084000 |forgery 1 |# fake
|stats  -1.66770000  -7.15350000   7.89290000   0.96765000 |forgery 1 |# fake
|stats  -3.84830000 -12.80470000  15.68240000  -1.28100000 |forgery 1 |# fake
|stats  -3.56810000  -8.21300000  10.08300000   0.96765000 |forgery 1 |# fake
|stats  -2.28040000  -0.30626000   1.33470000   1.37630000 |forgery 1 |# fake
|stats  -1.75820000   2.73970000  -2.53230000  -2.23400000 |forgery 1 |# fake
|stats  -0.89409000   3.19910000  -1.82190000  -2.94520000 |forgery 1 |# fake
|stats   0.34340000   0.12415000  -0.28733000   0.14654000 |forgery 1 |# fake
|stats  -0.98540000  -6.66100000   5.82450000   0.54610000 |forgery 1 |# fake
|stats  -2.41150000  -9.13590000   9.34440000  -0.65259000 |forgery 1 |# fake
|stats  -1.52520000  -6.25340000   5.35240000   0.59912000 |forgery 1 |# fake
|stats  -0.61442000  -0.09105800  -0.31818000   0.50214000 |forgery 1 |# fake
|stats  -0.36506000   2.89280000  -3.64610000  -3.06030000 |forgery 1 |# fake
|stats  -5.90340000   6.56790000   0.67661000  -6.67970000 |forgery 1 |# fake
|stats  -1.82150000   2.75210000  -0.72261000  -2.35300000 |forgery 1 |# fake
```

```
|stats  -0.77461000  -1.87680000   2.40230000   1.13190000 |forgery 1 |# fake
|stats  -1.81870000  -9.03660000   9.01620000  -0.12243000 |forgery 1 |# fake
|stats  -3.58010000 -12.93090000  13.17790000  -2.56770000 |forgery 1 |# fake
|stats  -1.82190000  -6.88240000   5.46810000   0.05731300 |forgery 1 |# fake
|stats  -0.34810000  -0.38696000  -0.47841000   0.62627000 |forgery 1 |# fake
|stats   0.47368000   3.36050000  -4.50640000  -4.04310000 |forgery 1 |# fake
|stats  -3.40830000   4.85870000  -0.76888000  -4.86680000 |forgery 1 |# fake
|stats  -1.66620000  -0.30005000   1.42380000   0.02498600 |forgery 1 |# fake
|stats  -2.09620000  -7.10590000   6.61880000  -0.33708000 |forgery 1 |# fake
|stats  -2.66850000 -10.45190000   9.11390000  -1.73230000 |forgery 1 |# fake
|stats  -0.47465000  -4.34960000   1.99010000   0.75170000 |forgery 1 |# fake
|stats   1.05520000   1.18570000  -2.64110000   0.11033000 |forgery 1 |# fake
|stats   1.16440000   3.80950000  -4.94080000  -4.09090000 |forgery 1 |# fake
|stats  -4.47790000   7.37080000  -0.31218000  -6.77540000 |forgery 1 |# fake
|stats  -2.73380000   0.45523000   2.43910000   0.21766000 |forgery 1 |# fake
|stats  -2.28600000  -5.44840000   5.80390000   0.88231000 |forgery 1 |# fake
|stats  -1.62440000  -6.34440000   4.65750000   0.16981000 |forgery 1 |# fake
|stats   0.50813000   0.47799000  -1.98040000   0.57714000 |forgery 1 |# fake
|stats   1.64080000   4.25030000  -4.90230000  -2.66210000 |forgery 1 |# fake
|stats   0.81583000   4.84000000  -5.26130000  -6.08230000 |forgery 1 |# fake
|stats  -5.49010000   9.10480000  -0.38758000  -5.97630000 |forgery 1 |# fake
|stats  -3.22380000   2.79350000   0.32274000  -0.86078000 |forgery 1 |# fake
|stats  -2.06310000  -1.51470000   1.21900000   0.44524000 |forgery 1 |# fake
|stats  -0.91318000  -2.01130000  -0.19565000   0.06636500 |forgery 1 |# fake
```

Chapter 5: banknote_test_cntk_onenode.txt (tab delimited)

```
|stats   4.17110000   8.72200000  -3.02240000  -0.59699000 |forgery 0 |# authentic
|stats  -0.20620000   9.22070000  -3.70440000  -6.81030000 |forgery 0 |# authentic
|stats  -0.00689190   9.29310000  -0.41243000  -1.96380000 |forgery 0 |# authentic
|stats   0.96441000   5.83950000   2.32350000   0.06636500 |forgery 0 |# authentic
|stats   2.85610000   6.91760000  -0.79372000   0.48403000 |forgery 0 |# authentic
|stats  -0.78690000   9.56630000  -3.78670000  -7.50340000 |forgery 0 |# authentic
|stats   2.08430000   6.62580000   0.48382000  -2.21340000 |forgery 0 |# authentic
|stats  -0.78690000   9.56630000  -3.78670000  -7.50340000 |forgery 0 |# authentic
|stats   3.91020000   6.06500000  -2.45340000  -0.68234000 |forgery 0 |# authentic
|stats   1.63490000   3.28600000   2.87530000   0.08705400 |forgery 0 |# authentic
|stats   0.60050000   1.93270000  -3.28880000  -0.32415000 |forgery 1 |# fake
|stats   0.91315000   3.33770000  -4.05570000  -1.67410000 |forgery 1 |# fake
|stats  -0.28015000   3.07290000  -3.38570000  -2.91550000 |forgery 1 |# fake
|stats  -3.60850000   3.32530000  -0.51954000  -3.57370000 |forgery 1 |# fake
|stats  -6.20030000   8.68060000   0.00913440  -3.70300000 |forgery 1 |# fake
|stats  -4.29320000   3.34190000   0.77258000  -0.99785000 |forgery 1 |# fake
|stats  -3.02650000  -0.06208800   0.68604000  -0.05518600 |forgery 1 |# fake
|stats  -1.70150000  -0.01035600  -0.99337000  -0.53104000 |forgery 1 |# fake
|stats  -0.64326000   2.47480000  -2.94520000  -1.02760000 |forgery 1 |# fake
|stats  -0.86339000   1.93480000  -2.37290000  -1.08970000 |forgery 1 |# fake
```

Chapter 6: boston_train_cntk.txt (tab delimited)

```
|predictors  0.006320     65.20  4.09   15.30  396.90 |medval      24.00
|predictors  0.027310     78.90  4.97   17.80  396.90 |medval      21.60
|predictors  0.027290     61.10  4.97   17.80  392.83 |medval      34.70
|predictors  0.032370     45.80  6.06   18.70  394.63 |medval      33.40
|predictors  0.069050     54.20  6.06   18.70  396.90 |medval      36.20
```

```
|predictors  0.029850      58.70  6.06  18.70  394.12 |medval       28.70
|predictors  0.088290      66.60  5.56  15.20  395.60 |medval       22.90
|predictors  0.144550      96.10  5.95  15.20  396.90 |medval       27.10
|predictors  0.211240      100.00 6.08  15.20  386.63 |medval       16.50
|predictors  0.170040      85.90  6.59  15.20  386.71 |medval       18.90
|predictors  0.224890      94.30  6.35  15.20  392.52 |medval       15.00
|predictors  0.117470      82.90  6.23  15.20  396.90 |medval       18.90
|predictors  0.093780      39.00  5.45  15.20  390.50 |medval       21.70
|predictors  0.629760      61.80  4.71  21.00  396.90 |medval       20.40
|predictors  0.637960      84.50  4.46  21.00  380.02 |medval       18.20
|predictors  0.627390      56.50  4.50  21.00  395.62 |medval       19.90
|predictors  1.053930      29.30  4.50  21.00  386.85 |medval       23.10
|predictors  0.784200      81.70  4.26  21.00  386.75 |medval       17.50
|predictors  0.802710      36.60  3.80  21.00  288.99 |medval       20.20
|predictors  0.725800      69.50  3.80  21.00  390.95 |medval       18.20
|predictors  1.251790      98.10  3.80  21.00  376.57 |medval       13.60
|predictors  0.852040      89.20  4.01  21.00  392.53 |medval       19.60
|predictors  1.232470      91.70  3.98  21.00  396.90 |medval       15.20
|predictors  0.988430      100.00 4.10  21.00  394.54 |medval       14.50
|predictors  0.750260      94.10  4.40  21.00  394.33 |medval       15.60
|predictors  0.840540      85.70  4.45  21.00  303.42 |medval       13.90
|predictors  0.671910      90.30  4.68  21.00  376.88 |medval       16.60
|predictors  0.955770      88.80  4.45  21.00  306.38 |medval       14.80
|predictors  0.772990      94.40  4.45  21.00  387.94 |medval       18.40
|predictors  1.002450      87.30  4.24  21.00  380.23 |medval       21.00
|predictors  1.130810      94.10  4.23  21.00  360.17 |medval       12.70
|predictors  1.354720      100.00 4.18  21.00  376.73 |medval       14.50
|predictors  1.387990      82.00  3.99  21.00  232.60 |medval       13.20
|predictors  1.151720      95.00  3.79  21.00  358.77 |medval       13.10
|predictors  1.612820      96.90  3.76  21.00  248.31 |medval       13.50
|predictors  0.064170      68.20  3.36  19.20  396.90 |medval       18.90
|predictors  0.097440      61.40  3.38  19.20  377.56 |medval       20.00
|predictors  0.080140      41.50  3.93  19.20  396.90 |medval       21.00
|predictors  0.175050      30.20  3.85  19.20  393.43 |medval       24.70
|predictors  0.027630      21.80  5.40  18.30  395.63 |medval       30.80
|predictors  0.033590      15.80  5.40  18.30  395.62 |medval       34.90
|predictors  0.127440      2.90   5.72  17.90  385.41 |medval       26.60
|predictors  0.141500      6.60   5.72  17.90  383.37 |medval       25.30
|predictors  0.159360      6.50   5.72  17.90  394.46 |medval       24.70
|predictors  0.122690      40.00  5.72  17.90  389.39 |medval       21.20
|predictors  0.171420      33.80  5.10  17.90  396.90 |medval       19.30
|predictors  0.188360      33.30  5.10  17.90  396.90 |medval       20.00
|predictors  0.229270      85.50  5.69  17.90  392.74 |medval       16.60
|predictors  0.253870      95.30  5.87  17.90  396.90 |medval       14.40
|predictors  0.219770      62.00  6.09  17.90  396.90 |medval       19.40
|predictors  0.088730      45.70  6.81  16.80  395.56 |medval       19.70
|predictors  0.043370      63.00  6.81  16.80  393.97 |medval       20.50
|predictors  0.053600      21.10  6.81  16.80  396.90 |medval       25.00
|predictors  0.049810      21.40  6.81  16.80  396.90 |medval       23.40
|predictors  0.013600      47.60  7.32  21.10  396.90 |medval       18.90
|predictors  0.013110      21.90  8.70  17.90  395.93 |medval       35.40
|predictors  0.020550      35.70  9.19  17.30  396.90 |medval       24.70
|predictors  0.014320      40.50  8.32  15.10  392.90 |medval       31.60
|predictors  0.154450      29.20  7.81  19.70  390.68 |medval       23.30
|predictors  0.103280      47.20  6.93  19.70  396.90 |medval       19.60
|predictors  0.149320      66.20  7.23  19.70  395.11 |medval       18.70
|predictors  0.171710      93.40  6.82  19.70  378.08 |medval       16.00
|predictors  0.110270      67.80  7.23  19.70  396.90 |medval       22.20
```

```
|predictors  0.126500     43.40  7.98   19.70  395.58 |medval       25.00
|predictors  0.019510     59.50  9.22   18.60  393.24 |medval       33.00
|predictors  0.035840     17.80  6.61   16.10  396.90 |medval       23.50
|predictors  0.043790     31.10  6.61   16.10  396.90 |medval       19.40
|predictors  0.057890     21.40  6.50   18.90  396.21 |medval       22.00
|predictors  0.135540     36.80  6.50   18.90  396.90 |medval       17.40
|predictors  0.128160     33.00  6.50   18.90  396.90 |medval       20.90
|predictors  0.088260     6.60   5.29   19.20  383.73 |medval       24.20
|predictors  0.158760     17.50  5.29   19.20  376.94 |medval       21.70
|predictors  0.091640     7.80   5.29   19.20  390.91 |medval       22.80
|predictors  0.195390     6.20   5.29   19.20  377.17 |medval       23.40
|predictors  0.078960     6.00   4.25   18.70  394.92 |medval       24.10
|predictors  0.095120     45.00  4.50   18.70  383.23 |medval       21.40
|predictors  0.101530     74.50  4.05   18.70  373.66 |medval       20.00
|predictors  0.087070     45.80  4.09   18.70  386.96 |medval       20.80
|predictors  0.056460     53.70  5.01   18.70  386.40 |medval       21.20
|predictors  0.083870     36.60  4.50   18.70  396.06 |medval       20.30
```

Chapter 6: boston_test_cntk.txt (tab delimited)

```
|predictors  0.041130     33.50  5.40   19.00  396.90 |medval       28.00
|predictors  0.044620     70.40  5.40   19.00  395.63 |medval       23.90
|predictors  0.036590     32.20  5.40   19.00  396.90 |medval       24.80
|predictors  0.035510     46.70  5.40   19.00  390.64 |medval       22.90
|predictors  0.050590     48.00  4.78   18.50  396.90 |medval       23.90
|predictors  0.057350     56.10  4.44   18.50  392.30 |medval       26.60
|predictors  0.051880     45.10  4.43   18.50  395.99 |medval       22.50
|predictors  0.071510     56.80  3.75   18.50  395.15 |medval       22.20
|predictors  0.056600     86.30  3.42   17.80  396.90 |medval       23.60
|predictors  0.053020     63.10  3.41   17.80  396.06 |medval       28.70
|predictors  0.046840     66.10  3.09   17.80  392.18 |medval       22.60
|predictors  0.039320     73.90  3.09   17.80  393.55 |medval       22.00
|predictors  0.042030     53.60  3.67   18.20  395.01 |medval       22.90
|predictors  0.028750     28.90  3.67   18.20  396.33 |medval       25.00
|predictors  0.042940     77.30  3.62   18.20  396.90 |medval       20.60
|predictors  0.122040     57.80  3.50   18.00  357.98 |medval       28.40
|predictors  0.115040     69.60  3.50   18.00  391.83 |medval       21.40
|predictors  0.120830     76.00  3.50   18.00  396.90 |medval       38.70
|predictors  0.081870     36.90  3.50   18.00  393.53 |medval       43.80
|predictors  0.068600     62.50  3.50   18.00  396.90 |medval       33.20
```

Chapter 7: airline_train_cntk.txt (space delimited)

```
|prev 1.12 1.18 1.32 1.29 |next 1.21 |change 1.08035714
|prev 1.18 1.32 1.29 1.21 |next 1.35 |change 1.14406780
|prev 1.32 1.29 1.21 1.35 |next 1.48 |change 1.12121212
|prev 1.29 1.21 1.35 1.48 |next 1.48 |change 1.14728682
|prev 1.21 1.35 1.48 1.48 |next 1.36 |change 1.12396694
|prev 1.35 1.48 1.48 1.36 |next 1.19 |change 0.88148148
|prev 1.48 1.48 1.36 1.19 |next 1.04 |change 0.70270270
|prev 1.48 1.36 1.19 1.04 |next 1.18 |change 0.79729730
|prev 1.36 1.19 1.04 1.18 |next 1.15 |change 0.84558824
|prev 1.19 1.04 1.18 1.15 |next 1.26 |change 1.05882353
|prev 1.04 1.18 1.15 1.26 |next 1.41 |change 1.35576923
|prev 1.18 1.15 1.26 1.41 |next 1.35 |change 1.14406780
```

```
|prev 1.15 1.26 1.41 1.35 |next 1.25 |change 1.08695652
|prev 1.26 1.41 1.35 1.25 |next 1.49 |change 1.18253968
|prev 1.41 1.35 1.25 1.49 |next 1.70 |change 1.20567376
|prev 1.35 1.25 1.49 1.70 |next 1.70 |change 1.25925926
|prev 1.25 1.49 1.70 1.70 |next 1.58 |change 1.26400000
|prev 1.49 1.70 1.70 1.58 |next 1.33 |change 0.89261745
|prev 1.70 1.70 1.58 1.33 |next 1.14 |change 0.67058824
|prev 1.70 1.58 1.33 1.14 |next 1.40 |change 0.82352941
|prev 1.58 1.33 1.14 1.40 |next 1.45 |change 0.91772152
|prev 1.33 1.14 1.40 1.45 |next 1.50 |change 1.12781955
|prev 1.14 1.40 1.45 1.50 |next 1.78 |change 1.56140351
|prev 1.40 1.45 1.50 1.78 |next 1.63 |change 1.16428571
|prev 1.45 1.50 1.78 1.63 |next 1.72 |change 1.18620690
|prev 1.50 1.78 1.63 1.72 |next 1.78 |change 1.18666667
|prev 1.78 1.63 1.72 1.78 |next 1.99 |change 1.11797753
|prev 1.63 1.72 1.78 1.99 |next 1.99 |change 1.22085890
|prev 1.72 1.78 1.99 1.99 |next 1.84 |change 1.06976744
|prev 1.78 1.99 1.99 1.84 |next 1.62 |change 0.91011236
|prev 1.99 1.99 1.84 1.62 |next 1.46 |change 0.73366834
|prev 1.99 1.84 1.62 1.46 |next 1.66 |change 0.83417085
|prev 1.84 1.62 1.46 1.66 |next 1.71 |change 0.92934783
|prev 1.62 1.46 1.66 1.71 |next 1.80 |change 1.11111111
|prev 1.46 1.66 1.71 1.80 |next 1.93 |change 1.32191781
|prev 1.66 1.71 1.80 1.93 |next 1.81 |change 1.09036145
|prev 1.71 1.80 1.93 1.81 |next 1.83 |change 1.07017544
|prev 1.80 1.93 1.81 1.83 |next 2.18 |change 1.21111111
|prev 1.93 1.81 1.83 2.18 |next 2.30 |change 1.19170984
|prev 1.81 1.83 2.18 2.30 |next 2.42 |change 1.33701657
|prev 1.83 2.18 2.30 2.42 |next 2.09 |change 1.14207650
|prev 2.18 2.30 2.42 2.09 |next 1.91 |change 0.87614679
|prev 2.30 2.42 2.09 1.91 |next 1.72 |change 0.74782609
|prev 2.42 2.09 1.91 1.72 |next 1.94 |change 0.80165289
|prev 2.09 1.91 1.72 1.94 |next 1.96 |change 0.93779904
|prev 1.91 1.72 1.94 1.96 |next 1.96 |change 1.02617801
|prev 1.72 1.94 1.96 1.96 |next 2.36 |change 1.37209302
|prev 1.94 1.96 1.96 2.36 |next 2.35 |change 1.21134021
|prev 1.96 1.96 2.36 2.35 |next 2.29 |change 1.16836735
|prev 1.96 2.36 2.35 2.29 |next 2.43 |change 1.23979592
|prev 2.36 2.35 2.29 2.43 |next 2.64 |change 1.11864407
|prev 2.35 2.29 2.43 2.64 |next 2.72 |change 1.15744681
|prev 2.29 2.43 2.64 2.72 |next 2.37 |change 1.03493450
|prev 2.43 2.64 2.72 2.37 |next 2.11 |change 0.86831276
|prev 2.64 2.72 2.37 2.11 |next 1.80 |change 0.68181818
|prev 2.72 2.37 2.11 1.80 |next 2.01 |change 0.73897059
|prev 2.37 2.11 1.80 2.01 |next 2.04 |change 0.86075949
|prev 2.11 1.80 2.01 2.04 |next 1.88 |change 0.89099526
|prev 1.80 2.01 2.04 1.88 |next 2.35 |change 1.30555556
|prev 2.01 2.04 1.88 2.35 |next 2.27 |change 1.12935323
|prev 2.04 1.88 2.35 2.27 |next 2.34 |change 1.14705882
|prev 1.88 2.35 2.27 2.34 |next 2.64 |change 1.40425532
|prev 2.35 2.27 2.34 2.64 |next 3.02 |change 1.28510638
|prev 2.27 2.34 2.64 3.02 |next 2.93 |change 1.29074890
|prev 2.34 2.64 3.02 2.93 |next 2.59 |change 1.10683761
|prev 2.64 3.02 2.93 2.59 |next 2.29 |change 0.86742424
|prev 3.02 2.93 2.59 2.29 |next 2.03 |change 0.67218543
|prev 2.93 2.59 2.29 2.03 |next 2.29 |change 0.78156997
|prev 2.59 2.29 2.03 2.29 |next 2.42 |change 0.93436293
|prev 2.29 2.03 2.29 2.42 |next 2.33 |change 1.01746725
```

```
|prev 2.03 2.29 2.42 2.33 |next 2.67 |change 1.31527094
|prev 2.29 2.42 2.33 2.67 |next 2.69 |change 1.17467249
|prev 2.42 2.33 2.67 2.69 |next 2.70 |change 1.11570248
|prev 2.33 2.67 2.69 2.70 |next 3.15 |change 1.35193133
|prev 2.67 2.69 2.70 3.15 |next 3.64 |change 1.36329588
|prev 2.69 2.70 3.15 3.64 |next 3.47 |change 1.28996283
|prev 2.70 3.15 3.64 3.47 |next 3.12 |change 1.15555556
|prev 3.15 3.64 3.47 3.12 |next 2.74 |change 0.86984127
|prev 3.64 3.47 3.12 2.74 |next 2.37 |change 0.65109890
|prev 3.47 3.12 2.74 2.37 |next 2.78 |change 0.80115274
|prev 3.12 2.74 2.37 2.78 |next 2.84 |change 0.91025641
|prev 2.74 2.37 2.78 2.84 |next 2.77 |change 1.01094891
|prev 2.37 2.78 2.84 2.77 |next 3.17 |change 1.33755274
|prev 2.78 2.84 2.77 3.17 |next 3.13 |change 1.12589928
|prev 2.84 2.77 3.17 3.13 |next 3.18 |change 1.11971831
|prev 2.77 3.17 3.13 3.18 |next 3.74 |change 1.35018051
|prev 3.17 3.13 3.18 3.74 |next 4.13 |change 1.30283912
|prev 3.13 3.18 3.74 4.13 |next 4.05 |change 1.29392971
|prev 3.18 3.74 4.13 4.05 |next 3.55 |change 1.11635220
|prev 3.74 4.13 4.05 3.55 |next 3.06 |change 0.81818182
|prev 4.13 4.05 3.55 3.06 |next 2.71 |change 0.65617433
|prev 4.05 3.55 3.06 2.71 |next 3.06 |change 0.75555556
|prev 3.55 3.06 2.71 3.06 |next 3.15 |change 0.88732394
|prev 3.06 2.71 3.06 3.15 |next 3.01 |change 0.98366013
|prev 2.71 3.06 3.15 3.01 |next 3.56 |change 1.31365314
|prev 3.06 3.15 3.01 3.56 |next 3.48 |change 1.13725490
|prev 3.15 3.01 3.56 3.48 |next 3.55 |change 1.12698413
|prev 3.01 3.56 3.48 3.55 |next 4.22 |change 1.40199336
|prev 3.56 3.48 3.55 4.22 |next 4.65 |change 1.30617978
|prev 3.48 3.55 4.22 4.65 |next 4.67 |change 1.34195402
|prev 3.55 4.22 4.65 4.67 |next 4.04 |change 1.13802817
|prev 4.22 4.65 4.67 4.04 |next 3.47 |change 0.82227488
|prev 4.65 4.67 4.04 3.47 |next 3.05 |change 0.65591398
|prev 4.67 4.04 3.47 3.05 |next 3.36 |change 0.71948608
|prev 4.04 3.47 3.05 3.36 |next 3.40 |change 0.84158416
|prev 3.47 3.05 3.36 3.40 |next 3.18 |change 0.91642651
|prev 3.05 3.36 3.40 3.18 |next 3.62 |change 1.18688525
|prev 3.36 3.40 3.18 3.62 |next 3.48 |change 1.03571429
|prev 3.40 3.18 3.62 3.48 |next 3.63 |change 1.06764706
|prev 3.18 3.62 3.48 3.63 |next 4.35 |change 1.36792453
|prev 3.62 3.48 3.63 4.35 |next 4.91 |change 1.35635359
|prev 3.48 3.63 4.35 4.91 |next 5.05 |change 1.45114943
|prev 3.63 4.35 4.91 5.05 |next 4.04 |change 1.11294766
|prev 4.35 4.91 5.05 4.04 |next 3.59 |change 0.82528736
|prev 4.91 5.05 4.04 3.59 |next 3.10 |change 0.63136456
|prev 5.05 4.04 3.59 3.10 |next 3.37 |change 0.66732673
|prev 4.04 3.59 3.10 3.37 |next 3.60 |change 0.89108911
|prev 3.59 3.10 3.37 3.60 |next 3.42 |change 0.95264624
|prev 3.10 3.37 3.60 3.42 |next 4.06 |change 1.30967742
|prev 3.37 3.60 3.42 4.06 |next 3.96 |change 1.17507418
|prev 3.60 3.42 4.06 3.96 |next 4.20 |change 1.16666667
|prev 3.42 4.06 3.96 4.20 |next 4.72 |change 1.38011696
|prev 4.06 3.96 4.20 4.72 |next 5.48 |change 1.34975369
|prev 3.96 4.20 4.72 5.48 |next 5.59 |change 1.41161616
|prev 4.20 4.72 5.48 5.59 |next 4.63 |change 1.10238095
|prev 4.72 5.48 5.59 4.63 |next 4.07 |change 0.86228814
|prev 5.48 5.59 4.63 4.07 |next 3.62 |change 0.66058394
|prev 5.59 4.63 4.07 3.62 |next 4.05 |change 0.72450805
```

```
|prev 4.63 4.07 3.62 4.05 |next 4.17 |change 0.90064795
|prev 4.07 3.62 4.05 4.17 |next 3.91 |change 0.96068796
|prev 3.62 4.05 4.17 3.91 |next 4.19 |change 1.15745856
|prev 4.05 4.17 3.91 4.19 |next 4.61 |change 1.13827160
|prev 4.17 3.91 4.19 4.61 |next 4.72 |change 1.13189448
|prev 3.91 4.19 4.61 4.72 |next 5.35 |change 1.36828645
|prev 4.19 4.61 4.72 5.35 |next 6.22 |change 1.48448687
|prev 4.61 4.72 5.35 6.22 |next 6.06 |change 1.31453362
|prev 4.72 5.35 6.22 6.06 |next 5.08 |change 1.07627119
|prev 5.35 6.22 6.06 5.08 |next 4.61 |change 0.86168224
|prev 6.22 6.06 5.08 4.61 |next 3.90 |change 0.62700965
|prev 6.06 5.08 4.61 3.90 |next 4.32 |change 0.71287129
```

Chapter 7: Raw unemployment claims data (comma-delimited)

```
1/1/2005,8016
2/1/2005,6218
3/1/2005,6902
4/1/2005,6844
5/1/2005,6165
6/1/2005,6609
7/1/2005,6104
8/1/2005,6088
9/1/2005,5750
10/1/2005,6757
11/1/2005,6742
12/1/2005,7101
1/1/2006,8429
2/1/2006,6344
3/1/2006,6587
4/1/2006,6372
5/1/2006,5870
6/1/2006,5990
7/1/2006,5729
8/1/2006,6444
9/1/2006,5119
10/1/2006,6293
11/1/2006,6527
12/1/2006,6518
1/1/2007,7987
2/1/2007,5808
3/1/2007,5853
4/1/2007,7172
5/1/2007,5949
6/1/2007,5869
7/1/2007,5837
8/1/2007,6110
9/1/2007,5298
10/1/2007,7265
11/1/2007,7037
12/1/2007,7280
1/1/2008,9062
2/1/2008,7553
3/1/2008,7448
4/1/2008,8140
5/1/2008,6963
6/1/2008,7695
```

```
7/1/2008,8009
8/1/2008,7287
9/1/2008,10361
10/1/2008,12823
11/1/2008,12664
12/1/2008,15652
1/1/2009,18196
2/1/2009,16741
3/1/2009,16646
4/1/2009,16061
5/1/2009,13903
6/1/2009,14311
7/1/2009,14469
8/1/2009,12115
9/1/2009,13495
10/1/2009,13871
11/1/2009,13637
12/1/2009,15430
1/1/2010,14022
2/1/2010,11854
3/1/2010,12337
4/1/2010,11892
5/1/2010,9975
6/1/2010,11250
7/1/2010,10428
8/1/2010,10322
9/1/2010,10445
10/1/2010,10683
11/1/2010,10251
12/1/2010,13110
1/1/2011,12859
2/1/2011,9744
3/1/2011,10395
4/1/2011,10298
5/1/2011,9361
6/1/2011,9393
7/1/2011,9047
8/1/2011,8596
9/1/2011,8410
10/1/2011,9656
11/1/2011,9929
12/1/2011,10142
1/1/2012,11597
2/1/2012,9229
3/1/2012,8930
4/1/2012,9285
5/1/2012,8243
6/1/2012,8574
7/1/2012,9295
8/1/2012,7867
9/1/2012,7668
10/1/2012,9526
11/1/2012,9299
12/1/2012,9316
1/1/2013,10452
2/1/2013,7885
3/1/2013,8019
4/1/2013,8904
```

```
5/1/2013,7597
6/1/2013,7543
7/1/2013,7878
8/1/2013,7041
9/1/2013,7069
10/1/2013,10207
11/1/2013,7532
12/1/2013,9057
1/1/2014,9799
2/1/2014,7630
3/1/2014,7577
4/1/2014,8256
5/1/2014,6675
6/1/2014,7007
7/1/2014,7169
8/1/2014,6488
9/1/2014,7509
10/1/2014,7925
11/1/2014,6993
12/1/2014,7978
```