Deep Learning With Python

Develop Deep Learning Models on Theano and TensorFlow Using Keras

Jason Brownlee

MACHINE
LEARNING
MASTERY

Jason Brownlee

Deep Learning With Python

Develop Deep Learning Models On Theano And TensorFlow Using Keras

Deep Learning With Python

Edition: v1.4

Contents

Preface

Deep learning is a fascinating field. Artificial neural networks have been around for a long time, but something special has happened in recent years. The mixture of new faster hardware, new techniques and highly optimized open source libraries allow very large networks to be created with frightening ease.

This new wave of much larger and much deeper neural networks are also impressively skillful on a range of problems. I have watched over recent years as they tackle and handily become state-of-the-art across a range of difficult problem domains. Not least object recognition, speech recognition, sentiment classification, translation and more.

When a technique comes a long that does so well on such a broad set of problems, you have to pay attention. The problem is where do you start with deep learning? I created this book because I thought that there was no gentle way for Python machine learning practitioners to quickly get started developing deep learning models.

In developing the lessons in this book, I chose the best of breed Python deep learning library called Keras that abstracted away all of the complexity, ruthlessly leaving you an API containing only what you need to know to efficiently develop and evaluate neural network models.

This is the guide that I wish I had when I started apply deep learning to machine learning problems. I hope that you find it useful on your own projects and have as much fun applying deep learning as I did in creating this book for you.

Jason Brownlee
Melbourne, Australia
2016

Part I

Introduction

Chapter 1

Welcome

Welcome to Deep Learning With Python. This book is your guide to deep learning in Python. You will discover the Keras Python library for deep learning and how to use it to develop and evaluate deep learning models. In this book you will discover the techniques, recipes and skills in deep learning that you can then bring to your own machine learning projects.

Deep learning does have a lot of fascinating math under the covers, but you do not need to know it to be able to pick it up as a tool and wield it on important projects and deliver real value. From the applied perspective, deep learning is quite a shallow field and a motivated developer can quickly pick it up and start making very real and impactful contributions. This is my goal for you and this book is your ticket to that outcome.

1.1 Deep Learning The Wrong Way

If you ask a deep learning practitioner how to get started with neural networks and deep learning, what do they say? They say things like

- You must have a strong foundation in linear algebra.

- You must have a deep knowledge of traditional neural network techniques.

- You really must know about probability and statistics.

- You should really have a deep knowledge of machine learning.

- You probably need to be a PhD in computer science.

- You probably need 10 years of experience as a machine learning developer.

You can see that the "common sense" advice means that it is not until after you have completed years of study and experience that you are ready to actually start developing and evaluating machine learning model for your machine learning projects.

I think this advice is dead wrong.

1.2 Deep Learning With Python

The approach taken with this book and with all of Machine Learning Mastery is to flip the traditional approach. If you are interested in deep learning, start by developing and evaluating deep learning models. Then if you discover you really like it or have a knack for it, later you can step deeper and deeper into the background and theory, as you need it in order to serve you in developing better and more valuable results. This book is your ticket to jumping in and making a ruckus with deep learning.

I have used many of the top deep learning platforms and libraries and I chose what I think is the best-of-breed platform for getting started and very quickly developing powerful and even state-of-the-art deep learning models in the Keras deep learning library for Python. Unlike R, Python is a fully featured programming language allowing you to use the same libraries and code for model development as you can use in production. Unlike Java, Python has the SciPy stack for scientific computing and scikit-learn which is a professional grade machine library.

There are two top numerical platforms for developing deep learning models, they are Theano developed by the University of Montreal and TensorFlow developed at Google. Both were developed for use in Python and both can be leveraged by the super simple to use Keras library. Keras wraps the numerical computing complexity of Theano and TensorFlow providing a concise API that we will use to develop our own neural network and deep learning models.

You will develop your own and perhaps your first neural network and deep learning models while working through this book, and you will have the skills to bring this amazing new technology to your own projects. It is going to be a fun journey and I can't wait to start.

1.3 Book Organization

This book is broken down into three parts.

- **Lessons** where you learn about specific features of neural network models and or how to use specific aspects of the Keras API.

- **Projects** where you will pull together multiple lessons into an end-to-end project and deliver a result, providing a template your your own projects.

- **Recipes** where you can copy and paste the standalone code into your own project, including all of the code presented in this book.

1.3.1 Lessons and Projects

Lessons are discrete and are focused on one topic, designed for you to complete in one sitting. You can take as long as you need, from 20 minutes if you are racing through, to hours if you want to experiment with the code or ideas and improve upon the presented results. Your lessons are divided into five parts:

- Background.

- Multilayer Perceptrons.

- Advanced Multilayer Perceptrons and Keras.

- Convolutional Neural Networks.

- Recurrent Neural Networks.

1.3.2   Part 2: Background

In this part you will learn about the Theano, TensorFlow and Keras libraries that lay the foundation for your deep learning journey and about how you can leverage very cheap Amazon Web Service computing in order to develop and evaluate your own large models in the cloud. This part of the book includes the following lessons:

- Introduction to the Theano Numerical Library.

- Introduction to the TensorFlow Numerical Library.

- Introduction to the Keras Deep Learning Library.

The lessons will introduce you to the important foundational libraries that you need to install and use on your workstation. This is taken one step further in a project that shows how you can cheaply harness GPU cloud computing to develop and evaluate very large deep learning models.

- Project: Develop Large Models on GPUs Cheaply In the Cloud.

At the end of this part you will be ready to start developing models in Keras on your workstation or in the cloud.

1.3.3   Part 3: Multilayer Perceptrons

In this part you will learn about feedforward neural networks that may be deep or not and how to expertly develop your own networks and evaluate them efficiently using Keras. This part of the book includes the following lessons:

- Crash Course In Multilayer Perceptrons.

- Develop Your First Neural Network With Keras.

- Evaluate The Performance of Deep Learning Models.

- Use Keras Models With Scikit-Learn For General Machine Learning.

These important lessons are tied together with three foundation projects. These projects demonstrate how you can quickly and efficiently develop neural network models for tabular data and provide project templates that you can use on your own regression and classification machine learning problems. These projects include:

- Project: Multiclass Classification Problem.

- Project: Binary Classification Problem.

- Project: Regression Problem.

At the end of this part you will be ready to discover the finer points of deep learning using the Keras API.

1.3.4   Part 4: Advanced Multilayer Perceptrons

In this part you will learn about some of the more finer points of the Keras library and API for practical machine learning projects and some of the more important developments in applied neural networks that you need to know in order to deliver world class results. This part of the book includes the following lessons:

- Save Your Models For Later With Network Serialization.

- Keep The Best Models During Training With Checkpointing.

- Understand Model Behavior During Training By Plotting History.

- Reduce Overfitting With Dropout Regularization.

- Lift Performance With Learning Rate Schedules.

At the end of this part you will know how to confidently wield Keras on your own machine learning projects with a focus of the finer points of investigating model performance, persisting models for later use and gaining lifts in performance over baseline models.

1.3.5   Part 5: Convolutional Neural Networks

In this part you will receive a crash course in the dominant model for computer vision machine learning problems and some natural language problems and how you can best exploit the capabilities of the Keras API for your own projects. This part of the book includes the following lessons:

- Crash Course In Convolutional Neural Networks.

- Improve Model Performance With Image Augmentation.

The best way to learn about this impressive type of neural network model is to apply it. You will work through three larger projects and apply CNN to image data for object recognition and text data for sentiment classification.

- Project: Handwritten Digit Recognition.

- Project: Object Recognition in Photographs.

- Project: Movie Review Sentiment Classification.

After completing the lessons and projects in this part you will have the skills and the confidence of complete and working templates and recipes to tackle your own deep learning projects using convolutional neural networks.

1.3.6   Part 6: Recurrent Neural Networks

In this part you will receive a crash course in the dominant model for data with a sequence or time component and how you can best exploit the capabilities of the Keras API for your own projects. This part of the book includes the following lessons:

- Crash Course In Recurrent Neural Networks.

- Multilayer Perceptron Models for Time Series Problems.

- LSTM Models for Time Series Problems.

- Understanding State in LSTM Models for Sequence Prediction.

The best way to learn about this complex type of neural network model is to apply it. You will work through two larger projects and apply RNN to sequence classification and text generation.

- Project: Sequence Classification of Movie Reviews.

- Project: Text Generation With Alice in Wonderland.

After completing the lessons and projects in this part you will have the skills and the confidence of complete and working templates and recipes to tackle your own deep learning projects using recurrent neural networks.

1.3.7   Conclusions

The book concludes with some resources that you can use to learn more information about a specific topic or find help if you need it as you start to develop and evaluate your own deep learning models.

1.3.8   Recipes

Building up a catalog of code recipes is an important part of your deep learning journey. Each time you learn about a new technique or new problem type, you should write up a short code recipe that demonstrates it. This will give you a starting point to use on your next deep learning or machine learning project.

As part of this book you will receive a catalog of deep learning recipes. This includes recipes for all of the lessons presented in this book, as well as the complete code for all of the projects. You are strongly encouraged to add to and build upon this catalog of recipes as you expand your use and knowledge of deep learning in Python.

1.4   Requirements For This Book

1.4.1   Python and SciPy

You do not need to be a Python expert, but it would be helpful if you knew how to install and setup Python and SciPy. The lessons and projects assume that you have a Python and SciPy

environment available. This may be on your workstation or laptop, it may be in a VM or a Docker instance that you run, or it may be a server instance that you can configure in the cloud as taught in Part II of this book.

Technical Requirements: The technical requirements for the code and tutorials in this book are as follows:

- Python version 2 or 3 installed. This book was developed using Python version 2.7.11.

- SciPy and NumPy installed. This book was developed with SciPy version 0.17.0 and NumPy version 1.11.0.

- Matplotlib installed. This book was developed with Matplotlib version 1.5.1.

- Pandas installed. This book was developed with Pandas version 0.18.0.

- scikit-learn installed. This book was developed with scikit-learn 0.17.1.

You do not need to match the version exactly, but if you are having problems running a specific code example, please ensure that you update to the same or higher version as the library specified. You will be guided as to how to install the deep learning libraries Theano, TensorFlow and Keras in Part II of the book.

1.4.2 Machine Learning

You do not need to be a machine learning expert, but it would be helpful if you knew how to navigate a small machine learning problem using scikit-learn. Basic concepts like cross validation and one hot encoding used in lessons and projects are described, but only briefly. There are resources to go into these topics in more detail at the end of the book, but some knowledge of these areas might make things easier for you.

1.4.3 Deep Learning

You do not need to know the math and theory of deep learning algorithms, but it would be helpful to have some basic idea of the field. You will get a crash course in neural network terminology and models, but we will not go into much detail. Again, there will be resources for more information at the end of the book, but it might be helpful if you can start with some idea about neural networks.

Note: All tutorials can be completed on standard workstation hardware with a CPU. A GPU is not required. Some tutorials later in the book can be sped up significantly by running on the GPU and a suggestion is provided to consider using GPU hardware at the beginning of those sections. You can access GPU hardware easily and cheaply in the cloud and a step-by-step procedure is taught on how to do this in Chapter 5.

1.5 Your Outcomes From Reading This Book

This book will lead you from being a developer who is interested in deep learning with Python to a developer who has the resources and capabilities to work through a new dataset end-to-end using Python and develop accurate deep learning models. Specifically, you will know:

- How to develop and evaluate neural network models end-to-end.

- How to use more advanced techniques required for developing state-of-the-art deep learning models.

- How to build larger models for image and text data.

- How to use advanced image augmentation techniques in order to lift model performance.

- How to get help with deep learning in Python.

From here you can start to dive into the specifics of the functions, techniques and algorithms used with the goal of learning how to use them better in order to deliver more accurate predictive models, more reliably in less time. There are a few ways you can read this book. You can dip into the lessons and projects as your need or interests motivate you. Alternatively, you can work through the book end-to-end and take advantage of how the lessons and projects build in complexity and range. I recommend the latter approach.

To get the very most from this book, I recommend taking each lesson and project and build upon them. Attempt to improve the results, apply the method to a similar but different problem, and so on. Write up what you tried or learned and share it on your blog, social media or send me an email at `jason@MachineLearningMastery.com`. This book is really what you make of it and by putting in a little extra, you can quickly become a true force in applied deep learning.

1.6 What This Book is Not

This book solves a specific problem of getting you, a developer, up to speed applying deep learning to your own machine learning projects in Python. As such, this book was not intended to be everything to everyone and it is very important to calibrate your expectations. Specifically:

- **This is not a deep learning textbook**. We will not be getting into the basic theory of artificial neural networks or deep learning algorithms. You are also expected to have some familiarity with machine learning basics, or be able to pick them up yourself.

- **This is not an algorithm book**. We will not be working through the details of how specific machine deep learning algorithms work. You are expected to have some basic knowledge of deep learning algorithms or how to pick up this knowledge yourself.

- **This is not a Python programming book**. We will not be spending a lot of time on Python syntax and programming (e.g. basic programming tasks in Python). You are expected to already be familiar with Python or a developer who can pick up a new C-like language relatively quickly.

You can still get a lot out of this book if you are weak in one or two of these areas, but you may struggle picking up the language or require some more explanation of the techniques. If this is the case, see the Getting More Help chapter at the end of the book and seek out a good companion reference text.

1.7 Summary

It is a special time right now. The tools for applied deep learning have never been so good. The pace of change with neural networks and deep learning feels like it has never been so fast, spurred by the amazing results that the methods are showing in such a broad range of fields. This is the start of your journey into deep learning and I am excited for you. Take your time, have fun and I'm so excited to see where you can take this amazing new technology.

1.7.1 Next

Let's dive in. Next up is Part II where you will take a whirlwind tour of the foundation libraries for deep learning in Python, namely the numerical libraries Theano and TensorFlow and the library you will be using throughout this book called Keras.

Part II

Background

Chapter 2

Introduction to Theano

Theano is a Python library for fast numerical computation that can be run on the CPU or GPU. It is a key foundational library for deep learning in Python that you can use directly to create deep learning models. After completing this lesson, you will know:

- About the Theano library for Python.

- How a very simple symbolic expression can be defined, compiled and calculated.

- Where you can learn more about Theano.

Let's get started.

2.1 What is Theano?

Theano is an open source project released under the BSD license and was developed by the LISA (now MILA[1]) group at the University of Montreal, Quebec, Canada (home of Yoshua Bengio). It is named after a Greek mathematician. At it's heart Theano is a compiler for mathematical expressions in Python. It knows how to take your structures and turn them into very efficient code that uses NumPy, efficient native libraries like BLAS and native code to run as fast as possible on CPUs or GPUs.

It uses a host of clever code optimizations to squeeze as much performance as possible from your hardware. If you are into the nitty-gritty of mathematical optimizations in code, check out this interesting list[2]. The actual syntax of Theano expressions is symbolic, which can be off putting to beginners. Specifically, expression are defined in the abstract sense, compiled and later actually used to make calculations.

Theano was specifically designed to handle the types of computation required for large neural network algorithms used in deep learning. It was one of the first libraries of its kind (development started in 2007) and is considered an industry standard for deep learning research and development.

---

[1]http://mila.umontreal.ca/
[2]http://deeplearning.net/software/theano/optimizations.html#optimizations

2.2   How to Install Theano

Theano provides extensive installation instructions for the major operating systems: Windows, OS X and Linux. Read the Installing Theano guide for your platform[3]. Theano assumes a working Python 2 or Python 3 environment with SciPy. There are ways to make the installation easier, such as using Anaconda[4] to quickly setup Python and SciPy on your machine as well as using Docker images. With a working Python and SciPy environment, it is relatively straightforward to install Theano using pip, for example:

```
sudo pip install Theano
```

Listing 2.1: Install Theano with pip.

New releases of Theano may be announced and you will want to update to get any bug fixes and efficiency improvements. You can upgrade Theano using pip as follows:

```
sudo pip install --upgrade --no-deps theano
```

Listing 2.2: Upgrade Theano with pip.

You may want to use the bleeding edge version of Theano checked directly out of GitHub. This may be required for some wrapper libraries that make use of bleeding edge API changes. You can install Theano directly from a GitHub checkout as follows:

```
sudo pip install --upgrade --no-deps git+git://github.com/Theano/Theano.git
```

Listing 2.3: Upgrade Theano with pip from GitHub.

You are now ready to run Theano on your CPU, which is just fine for the development of small models. Large models may run slowly on the CPU. If you have a Nvidia GPU, you may want to look into configuring Theano to use your GPU. There is a wealth of documentation of the Theano homepage for further configuring the library.

> **Theano v0.8.2 is the latest at the time of writing and is used in this book.**

2.3   Simple Theano Example

In this section we demonstrate a simple Python script that gives you a flavor of Theano. In this example we define two symbolic floating point variables a and b. We define an expression that uses these variables ($c = a + b$). We then compile this symbolic expression into a function using Theano that we can use later. Finally, we use our compiled expression by plugging in some real values and performing the calculation using efficient compiled Theano code under the covers.

```
# Example of Theano library
import theano
from theano import tensor
# declare two symbolic floating-point scalars
a = tensor.dscalar()
b = tensor.dscalar()
```

---

[3]http://deeplearning.net/software/theano/install.html
[4]https://www.continuum.io/downloads

```
# create a simple symbolic expression
c = a + b
# convert the expression into a callable object that takes (a,b) and computes c
f = theano.function([a,b], c)
# bind 1.5 to 'a', 2.5 to 'b', and evaluate 'c'
result = f(1.5, 2.5)
print(result)
```

Listing 2.4: Example of Symbolic Arithmetic with Theano.

Running the example prints the output 4, which matches our expectation that $1.5 + 2.5 = 4.0$. This is a useful example as it gives you a flavor for how a symbolic expression can be defined, compiled and used. You can see how this may be scaled up to large vector and matrix operations required for deep learning.

2.4 Extensions and Wrappers for Theano

If you are new to deep learning you do not have to use Theano directly. In fact, you are highly encouraged to use one of many popular Python projects that make Theano a lot easier to use for deep learning. These projects provide data structures and behaviors in Python, specifically designed to quickly and reliably create deep learning models whilst ensuring that fast and efficient models are created and executed by Theano under the covers. The amount of Theano syntax exposed by the libraries varies.

Keras is a wrapper library that hides Theano completely and provides a very simple API to work with to create deep learning models. It hides Theano so well, that it can in fact run as a wrapper for another popular foundation framework called TensorFlow (discussed next).

2.5 More Theano Resources

Looking for some more resources on Theano? Take a look at some of the following.

- Theano Official Homepage
  http://deeplearning.net/software/theano/

- Theano GitHub Repository
  https://github.com/Theano/Theano/

- Theano: A CPU and GPU Math Compiler in Python (2010)
  http://www.iro.umontreal.ca/~lisa/pointeurs/theano_scipy2010.pdf

- List of Libraries Built on Theano
  https://github.com/Theano/Theano/wiki/Related-projects

- List of Theano configuration options
  http://deeplearning.net/software/theano/library/config.html

2.6   Summary

In this lesson you discovered the Theano Python library for efficient numerical computation. You learned:

- Theano is a foundation library used for deep learning research and development.

- Deep learning models can be developed directly in Theano if desired.

- The development and evaluation of deep learning models is easier with wrapper libraries like Keras.

2.6.1   Next

You now know about the Theano library for numerical computation in Python. In the next lesson you will discover the TensorFlow library released by Google that attempts to offer the same capabilities.

Chapter 3

Introduction to TensorFlow

TensorFlow is a Python library for fast numerical computing created and released by Google. It is a foundation library that can be used to create deep learning models directly or by using wrapper libraries that simplify the process built on top of TensorFlow. After completing this lesson you will know:

- About the TensorFlow library for Python.

- How to define, compile and evaluate a simple symbolic expression in TensorFlow.

- Where to go to get more information on the Library.

Let's get started.

Note: TensorFlow is not easily supported on Windows at the time of writing. It may be possible to get TensorFlow working on windows with Docker. TensorFlow is not required to complete the rest of this book, and if you are on the Windows platform you can skip this lesson.

3.1   What is TensorFlow?

TensorFlow is an open source library for fast numerical computing. It was created and is maintained by Google and released under the Apache 2.0 open source license. The API is nominally for the Python programming language, although there is access to the underlying C++ API. Unlike other numerical libraries intended for use in Deep Learning like Theano, TensorFlow was designed for use both in research and development and in production systems, not least RankBrain in Google search[1] and the fun DeepDream project[2]. It can run on single CPU systems, GPUs as well as mobile devices and large scale distributed systems of hundreds of machines.

3.2   How to Install TensorFlow

Installation of TensorFlow is straightforward if you already have a Python SciPy environment. TensorFlow works with Python 2.7 and Python 3.3+. With a working Python and SciPy

---

[1]https://en.wikipedia.org/wiki/RankBrain
[2]https://en.wikipedia.org/wiki/DeepDream

environment, it is relatively straightforward to install TensorFlow using pip There are a number of different distributions of TensorFlow, customized for different environments, therefore to install TensorFlow you can follow the Download and Setup instructions[3] on the TensorFlow website. , for example:

> **TensorFlow v0.9 is the latest at the time of writing and is used in this book.**

3.3 Your First Examples in TensorFlow

Computation is described in terms of data flow and operations in the structure of a directed graph.

- Nodes: Nodes perform computation and have zero or more inputs and outputs. Data that moves between nodes are known as tensors, which are multi-dimensional arrays of real values.

- Edges: The graph defines the flow of data, branching, looping and updates to state. Special edges can be used to synchronize behavior within the graph, for example waiting for computation on a number of inputs to complete.

- Operation: An operation is a named abstract computation which can take input attributes and produce output attributes. For example, you could define an add or multiply operation.

3.4 Simple TensorFlow Example

In this section we demonstrate a simple Python script that gives you a flavor of TensorFlow. In this example we define two symbolic floating point variables a and b. We define an expression that uses these variables ($c = a + b$). This is the same example used in the previous chapter that introduced Theano. We then compile this symbolic expression into a function using TensorFlow that we can use later. Finally, we use our complied expression by plugging in some real values and performing the calculation using efficient compiled TensorFlow code under the covers.

```python
# Example of TensorFlow library
import tensorflow as tf
# declare two symbolic floating-point scalars
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
# create a simple symbolic expression using the add function
add = tf.add(a, b)
# bind 1.5 to 'a', 2.5 to 'b', and evaluate 'c'
sess = tf.Session()
binding = {a: 1.5, b: 2.5}
c = sess.run(add, feed_dict=binding)
print(c)
```

Listing 3.1: Example of Symbolic Arithmetic with TensorFlow.

---

[3] https://www.tensorflow.org/versions/r0.9/get_started/os_setup.html

Running the example prints the output 4, which matches our expectation that $1.5 + 2.5 = 4.0$. This is a useful example as it gives you a flavor for how a symbolic expression can be defined, compiled and used. You can see how this may be scaled up to large vector and matrix operations required for deep learning.

3.5 More Deep Learning Models

Your TensorFlow installation comes with a number of Deep Learning models that you can use and experiment with directly. Firstly, you need to find out where TensorFlow was installed on your system. For example, you can use the following Python script:

```
python -c 'import os; import inspect; import tensorflow;
    print(os.path.dirname(inspect.getfile(tensorflow)))'
```

Listing 3.2: Print Install Directory for TensorFlow.

Change to this directory and take note of the **models/** subdirectory. Included are a number of deep learning models with tutorial-like comments, such as:

- Multi-threaded word2vec mini-batched skip-gram model.

- Multi-threaded word2vec unbatched skip-gram model.

- CNN for the CIFAR-10 network.

- Simple, end-to-end, LeNet-5-like convolutional MNIST model example.

- Sequence-to-sequence model with an attention mechanism.

Also check the examples directory as it contains an example using the MNIST dataset. There is also an excellent list of tutorials on the main TensorFlow website[4]. They show how to use different network types, different datasets and how to use the framework in various different ways. Finally, there is the TensorFlow playground[5] where you can experiment with small networks right in your web browser.

3.6 Summary

In this lesson you discovered the TensorFlow Python library for deep learning. You learned:

- TensorFlow is another efficient numerical library like Theano.

- Like Theano, deep learning models can be developed directly in TensorFlow if desired.

- Also like Theano, TensorFlow may be better leveraged by a wrapper library that abstracts the complexity and lower level details.

---

[4]https://www.tensorflow.org/versions/r0.9/tutorials/
[5]http://playground.tensorflow.org/

3.6.1 Next

You now know about the Theano and TensorFlow libraries for efficient numerical computation in Python. In the next lesson you will discover the Keras library that wraps both libraries and gives you a clean and simple API for developing and evaluating deep learning models.

Chapter 4

Introduction to Keras

Two of the top numerical platforms in Python that provide the basis for deep learning research and development are Theano and TensorFlow. Both are very powerful libraries, but both can be difficult to use directly for creating deep learning models. In this lesson you will discover the Keras Python library that provides a clean and convenient way to create a range of deep learning models on top of Theano or TensorFlow. After completing this lesson you will know:

- About the Keras Python library for deep learning.

- How to configure Keras for Theano or TensorFlow.

- The standard idiom for creating models with Keras.

Let's get started.

4.1   What is Keras?

Keras is a minimalist Python library for deep learning that can run on top of Theano or TensorFlow. It was developed to make developing deep learning models as fast and easy as possible for research and development. It runs on Python 2.7 or 3.5 and can seamlessly execute on GPUs and CPUs given the underlying frameworks. It is released under the permissive MIT license. Keras was developed and maintained by François Chollet, a Google engineer using four guiding principles:

- **Modularity**: A model can be understood as a sequence or a graph alone. All the concerns of a deep learning model are discrete components that can be combined in arbitrary ways.

- **Minimalism**: The library provides just enough to achieve an outcome, no frills and maximizing readability.

- **Extensibility**: New components are intentionally easy to add and use within the framework, intended for developers to trial and explore new ideas.

- **Python**: No separate model files with custom file formats. Everything is native Python.

4.2 How to Install Keras

Keras is relatively straightforward to install if you already have a working Python and SciPy environment. You must also have an installation of Theano or TensorFlow on your system. Keras can be installed easily using pip, as follows:

```
sudo pip install keras
```

Listing 4.1: Install Keras With Pip.

You can check your version of Keras on the command line using the following script:

```
python -c "import keras; print(keras.__version__)"
```

Listing 4.2: Print Keras Version.

Running the above script you will see:

```
1.0.5
```

Listing 4.3: Output of Printing Keras Version.

You can upgrade your installation of Keras using the same method:

```
sudo pip install --upgrade keras
```

Listing 4.4: Upgrade Keras With Pip.

Keras v1.0.5 is the latest at the time of writing and is used in this book.

4.3 Theano and TensorFlow Backends for Keras

Keras is a lightweight API and rather than providing an implementation of the required mathematical operations needed for deep learning it provides a consistent interface to efficient numerical libraries called *backends*. Assuming you have both Theano and TensorFlow installed, you can configure the backend used by Keras. The easiest way is by adding or editing the Keras configuration file in your home directory:

```
~/.keras/keras.json
```

Listing 4.5: Path to Keras Configuration File.

Which has the format:

```
{
    "image_dim_ordering": "th",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "theano"
}
```

Listing 4.6: Example Content of Keras Configuration File.

In this configuration file you can change the `backend` property from `theano` (the default) to `tensorflow`. Keras will then use the configuration the next time it is run. You can confirm the backend used by Keras using the following script on the command line:

```
python -c "from keras import backend; print(backend._BACKEND)"
```
Listing 4.7: Script to Print the Configured Keras Backend.

Running this with default configuration you will see:

```
Using Theano backend.
theano
```
Listing 4.8: Sample Output of Script to Print the Configured Keras Backend.

You can also specify the backend to use by Keras on the command line by specifying the `KERAS_BACKEND` environment variable, as follows:

```
KERAS_BACKEND=tensorflow python -c "from keras import backend; print(backend._BACKEND)"
```
Listing 4.9: Example of Using the Environment Variable to Change the Keras Backend.

Running this example prints:

```
Using TensorFlow backend.
tensorflow
```
Listing 4.10: Sample Output of Using the TensorFlow Backend.

4.4 Build Deep Learning Models with Keras

The focus of Keras is the idea of a model. The main type of model is a sequence of layers called a `Sequential` which is a linear stack of layers. You create a `Sequential` and add layers to it in the order that you wish for the computation to be performed. Once defined, you compile the model which makes use of the underlying framework to optimize the computation to be performed by your model. In this you can specify the loss function and the optimizer to be used.

Once compiled, the model must be fit to data. This can be done one batch of data at a time or by firing off the entire model training regime. This is where all the compute happens. Once trained, you can use your model to make predictions on new data. We can summarize the construction of deep learning models in Keras as follows:

1. **Define your model**. Create a `Sequential` model and add configured layers.

2. **Compile your model**. Specify loss function and optimizers and call the `compile()` function on the model.

3. **Fit your model**. Train the model on a sample of data by calling the `fit()` function on the model.

4. **Make predictions**. Use the model to generate predictions on new data by calling functions such as `evaluate()` or `predict()` on the model.

4.5 Summary

In this lesson you discovered the Keras Python library for deep learning research and development. You learned:

- Keras wraps both the TensorFlow and Theano libraries, abstracting their capabilities and hiding their complexity.

- Keras is designed for minimalism and modularity allowing you to very quickly define deep learning models.

- Keras deep learning models can be developed using an idiom of defining, compiling and fitting models that can then be evaluated or used to make predictions.

4.5.1 Next

You are now up to speed with the Python libraries for deep learning. In the next project you will discover step-by-step how you can develop and run very large deep learning models using these libraries in the cloud using GPU hardware at a fraction of the cost of purchasing your own hardware.

Chapter 5

Project: Develop Large Models on GPUs Cheaply In the Cloud

Large deep learning models require a lot of compute time to run. You can run them on your CPU but it can take hours or days to get a result. If you have access to a GPU on your desktop, you can drastically speed up the training time of your deep learning models. In this project you will discover how you can get access to GPUs to speed up the training of your deep learning models by using the Amazon Web Service (AWS) infrastructure. For less than a dollar per hour and often a lot cheaper you can use this service from your workstation or laptop. After working through this project you will know:

- How to create an account and log-in to Amazon Web Service.

- How to launch a server instance for deep learning.

- How to configure a server instance for faster deep learning on the GPU.

Let's get started.

5.1   Project Overview

The process is quite simple because most of the work has already been done for us. Below is an overview of the process.

- Setup Your AWS Account.

- Launch Your Server Instance.

- Login and Run Your Code.

- Close Your Server Instance.

Note, it costs money to use a virtual server instance on Amazon. The cost is low for ad hoc model development (e.g. less than one US dollar per hour), which is why this is so attractive, but it is not free. The server instance runs Linux. It is desirable although not required that you know how to navigate Linux or a Unix-like environment. We're just running our Python scripts, so no advanced skills are needed.

5.2 Setup Your AWS Account

You need an account on Amazon Web Services[1].

- 1. You can create account by the Amazon Web Services portal and click *Sign in to the Console*. From there you can sign in using an existing Amazon account or create a new account.
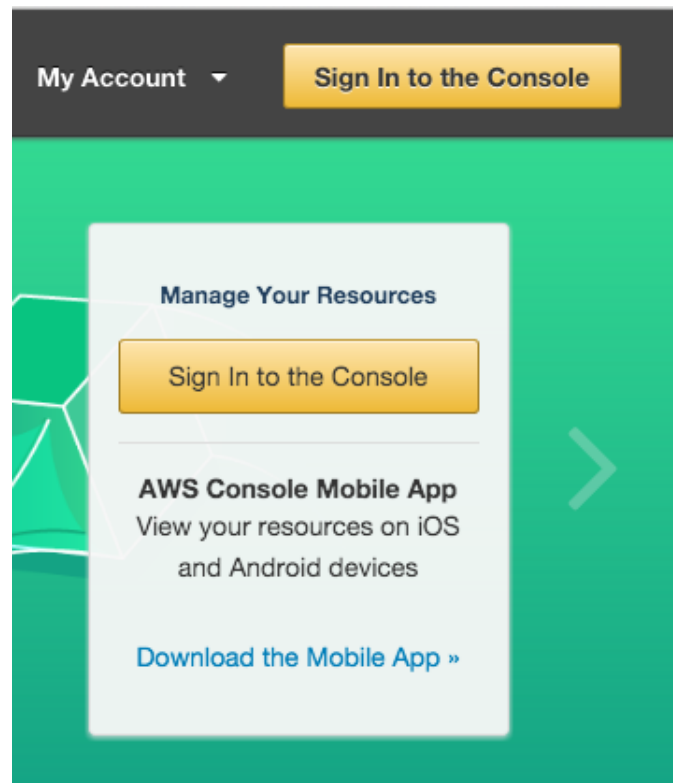
Figure 5.1: AWS Sign-in Button

- 2. You will need to provide your details as well as a valid credit card that Amazon can charge. The process is a lot quicker if you are already an Amazon customer and have your credit card on file.

---

[1]https://aws.amazon.com

Figure 5.2: AWS Sign-In Form

Once you have an account you can log into the Amazon Web Services console. You will see a range of different services that you can access.

5.3   Launch Your Server Instance

Now that you have an AWS account, you want to launch an EC2 virtual server instance on which you can run Keras. Launching an instance is as easy as selecting the image to load and starting the virtual server. Thankfully there is already an image available that has almost everything we need it has the cryptic name **ami-125b2c72** and was created for the Stanford CS231n class. Let's launch it as an instance.

- 1. Login to your AWS console[2] if you have not already.

---

[2] https://console.aws.amazon.com/console/home
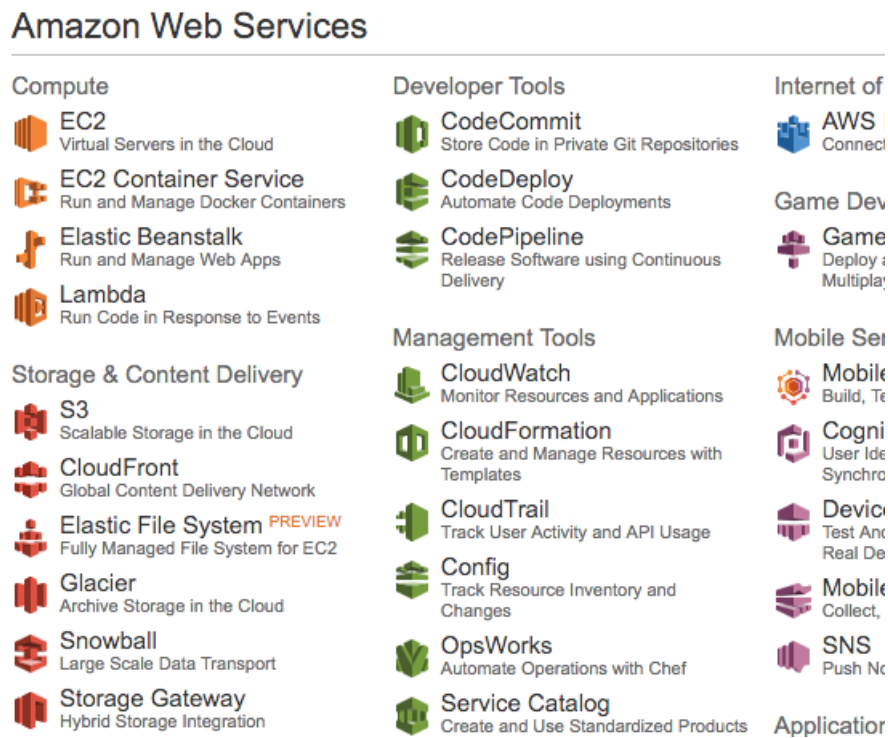
Figure 5.3: AWS Console

- 2. Click on EC2 for launching a new virtual server.

- 3. Select *N. California* from the drop-down in the top right hand corner. This is important otherwise you will not be able to find the image we plan to use.
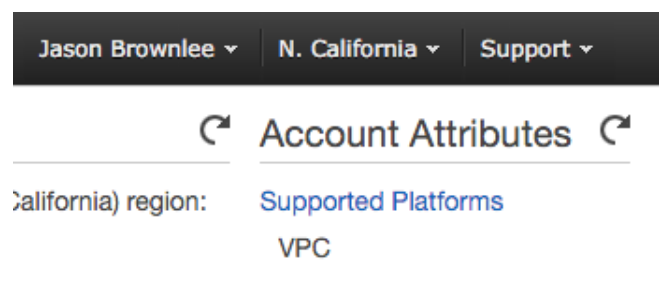
Figure 5.4: Select North California

- 4. Click the *Launch Instance* button.

- 5. Click *Community AMIs*. An AMI is an Amazon Machine Image. It is a frozen instance of a server that you can select and instantiate on a new virtual server.
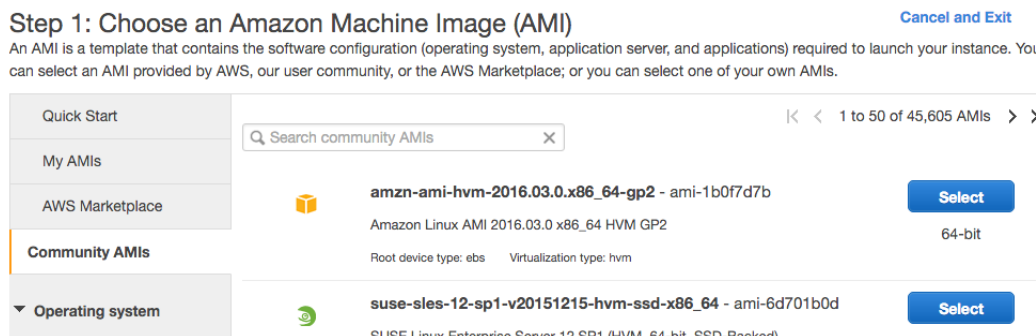
Figure 5.5: Community AMIs

- 6. Enter `ami-125b2c72` in the *Search community AMIs* search box and press enter. You should be presented with a single result.
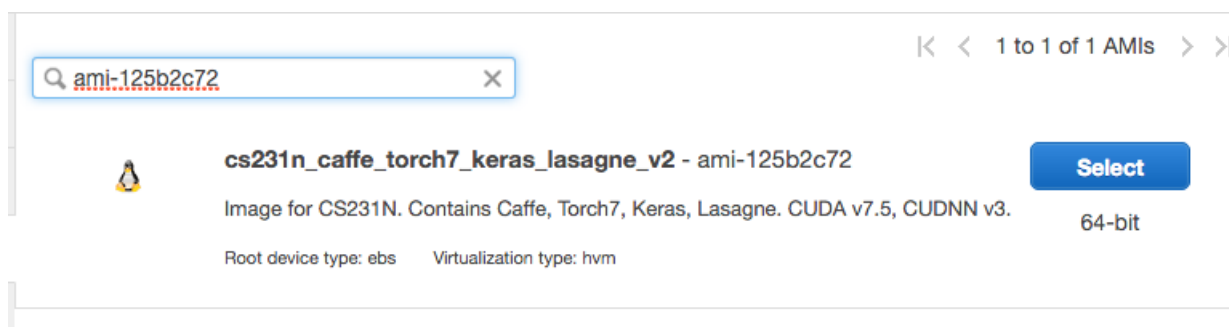
Figure 5.6: Select a Specific AMI

- 7. Click *Select* to choose the AMI in the search result.

- 8. Now you need to select the hardware on which to run the image. Scroll down and select the *g2.2xlarge* hardware. This includes a GPU that we can use to significantly increase the training speed of our models.
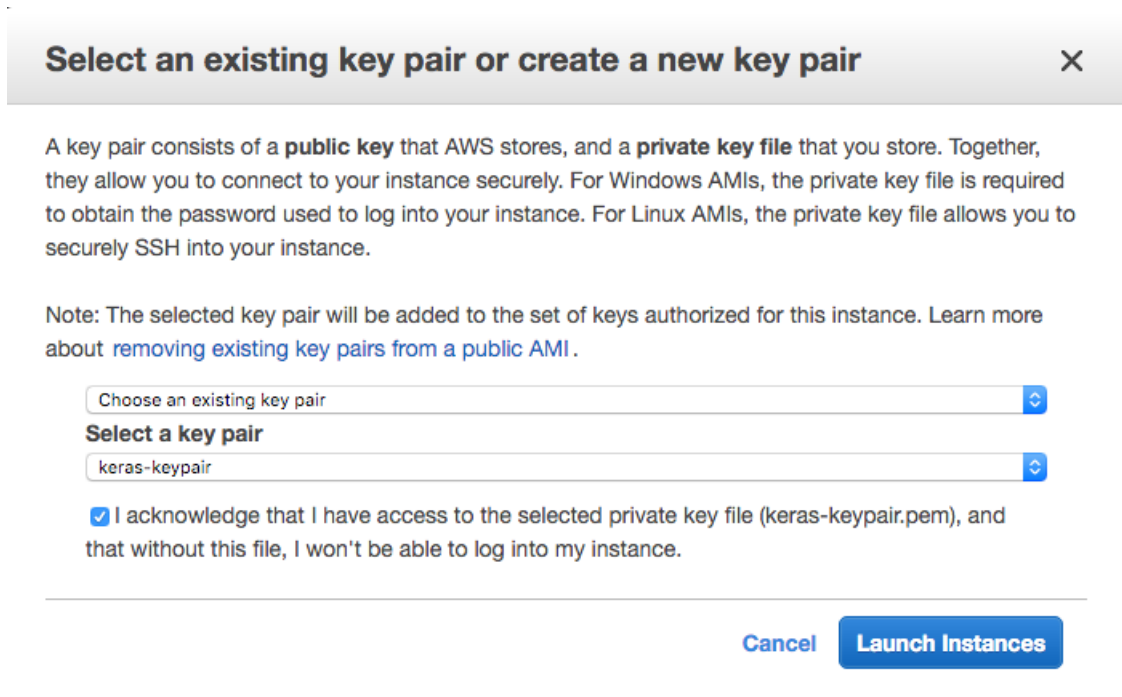
Figure 5.7: Select g2.2xlarge Hardware

- 9. Click *Review and Launch* to finalize the configuration of your server instance.

- 10. Click the *Launch* button.

- 11. Select Your Key Pair.

If you have a key pair because you have used EC2 before, select *Choose an existing key pair* and choose your key pair from the list. Then check *I acknowledge...*. If you do not have a key

pair, select the option *Create a new key pair* and enter a *Key pair name* such as keras-keypair. Click the *Download Key Pair* button.

Figure 5.8: Select Your Key Pair

- 12. Open a Terminal and change directory to where you downloaded your key pair.

- 13. If you have not already done so, restrict the access permissions on your key pair file. This is required as part of the SSH access to your server. For example, open a terminal on your workstation and type:

```
cd Downloads
chmod 600 keras-aws-keypair.pem
```

Listing 5.1: Change Permissions of Your Key Pair File.

- 14. Click *Launch Instances*. If this is your first time using AWS, Amazon may have to validate your request and this could take up to 2 hours (often just a few minutes).

- 15. Click *View Instances* to review the status of your instance.

Figure 5.9: Review Your Running Instance

Your server is now running and ready for you to log in.

5.4   Login, Configure and Run

Now that you have launched your server instance, it is time to log in and start using it.

- 1. Click *View Instances* in your Amazon EC2 console if you have not done so already.

- 2. Copy the *Public IP* (down the bottom of the screen in Description) to your clipboard. In this example my IP address is `52.53.186.1`. **Do not use this IP address, it will not work as your server IP address will be different**.

- 3. Open a Terminal and change directory to where you downloaded your key pair. Login in to your server using SSH, for example:

```
ssh -i keras-aws-keypair.pem ubuntu@52.53.186.1
```
Listing 5.2: Log-in To Your AWS Instance.

- 4. If prompted, type `yes` and press enter.

You are now logged into your server.

```
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-76-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

  System information as of Fri Apr 29 20:40:02 UTC 2016

  System load: 0.44                Memory usage: 1%   Processes:       129
  Usage of /:  61.5% of 11.67GB    Swap usage:   0%   Users logged in: 0

  Graph this data and manage this system at:
    https://landscape.canonical.com/

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

4 packages can be updated.
4 updates are security updates.


Last login: Wed Jan 27 05:21:36 2016 from 128.12.246.6
ubuntu@ip-172-31-0-102:~$
```

Figure 5.10: Log in Screen for Your AWS Server

We need to make two small changes before we can start using Keras. This will just take a minute. You will have to do these changes each time you start the instance.

5.4.1 Update Keras

Update to a specific version of Keras known to work on this configuration, at the time of writing the latest version of Keras is version 1.0.5. We can specify this version as part of the upgrade of Keras via pip.

```
pip install --upgrade --no-deps keras==1.0.5
```

Listing 5.3: Update Keras Using Pip.

5.4.2 Configure Theano

Update your configuration of Theano (the Keras backend) to always use the GPU. First open the Theano configuration file in your favorite command line text editor, such as vi:

```
vi ~/.theanorc
```

Listing 5.4: Edit the Theano Configuration File.

Copy and paste the following configuration and save the file:

```
[global]
device = gpu
floatX = float32
optimizer_including = cudnn
allow_gc = False

[lib]
cnmem=.95
```

Listing 5.5: New Configuration For Theano.

This configures Theano to use the GPU instead of the CPU. Among some other minor configuration it ensures that not all GPU memory is used by Theano, avoiding memory errors if you start using larger datasets like CIFAR-10. We're done. You can confirm that Theano is working correctly by typing:

```
python -c "import theano; print(theano.sandbox.cuda.dnn.dnn_available())"
```

Listing 5.6: Script to Check Theano Configuration.

This command will output Theano configuration. You should see:

```
Using gpu device 0: GRID K520 (CNMeM is enabled)
True
```

Listing 5.7: Sample Output of Script to Check Theano Configuration.

You can also confirm that Keras is installed and is working correctly by typing:

```
python -c "import keras; print(keras.__version__)"
```

Listing 5.8: Script To Check Keras Configuration.

You should see:

```
Using Theano backend.
Using gpu device 0: GRID K520 (CNMeM is enabled)
1.0.5
```

Listing 5.9: Sample Output of Script to Check Keras Configuration.

You are now free to run your code.

5.5 Build and Run Models on AWS

This section offers some tips for running your code on AWS.

5.5.1 Copy Scripts and Data to AWS

You can get started quickly by copying your files to your running AWS instance. For example, you can copy the examples provided with this book to your AWS instance using the `scp` command as follows:

```
scp -i keras-aws-keypair.pem -r src ubuntu@52.53.186.1:~/
```

Listing 5.10: Example for Copying Sample Code to AWS.

This will copy the entire `src/` directory to your home directory on your AWS instance. You can easily adapt this example to get your larger datasets from your workstation onto your AWS instance. Note that Amazon may impose charges for moving very large amounts of data in and out of your AWS instance. Refer to Amazon documentation for relevant charges.

5.5.2 Run Models on AWS

You can run your scripts on your AWS instance as per normal:

```
python filename.py
```

Listing 5.11: Example of Running a Python script on AWS.

You are using AWS to create large neural network models that may take hours or days to train. As such, it is a better idea to run your scripts as a background job. This allows you to close your terminal and your workstation while your AWS instance continues to run your script. You can easily run your script as a background process as follows:

```
nohup /path/to/script >/path/to/script.log 2>&1 < /dev/null &
```

Listing 5.12: Run Script as a Background Process.

You can then check the status and results in your `script.log` file later.

5.6 Close Your EC2 Instance

When you are finished with your work you must close your instance. Remember you are charged by the amount of time that you use the instance. It is cheap, but you do not want to leave an instance on if you are not using it.

- 1. Log out of your instance at the terminal, for example you can type:

```
exit
```

Listing 5.13: Log-out of Server Instance.

- 2. Log in to your AWS account with your web browser.

- 3. Click EC2.

- 4. Click *Instances* from the left-hand side menu.
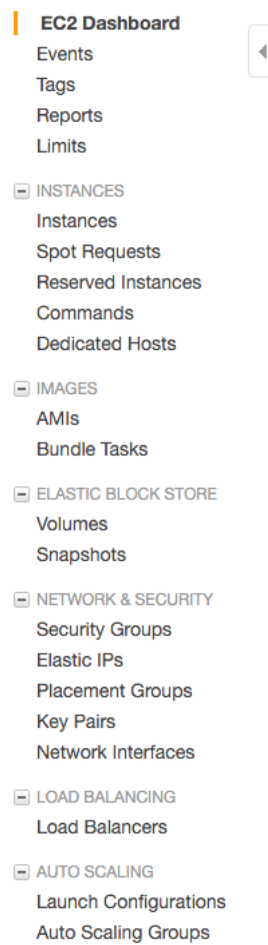
Figure 5.11: Review Your List of Running Instances

- 5. Select your running instance from the list (it may already be selected if you only have one running instance).

Figure 5.12: Select Your Running AWS Instance

- 6. Click the *Actions* button and select *Instance State* and choose *Terminate*. Confirm that you want to terminate your running instance.

It may take a number of seconds for the instance to close and to be removed from your list of instances.

5.7 Tips and Tricks for Using Keras on AWS

Below are some tips and tricks for getting the most out of using Keras on AWS instances.

- **Design a suite of experiments to run beforehand**. Experiments can take a long time to run and you are paying for the time you use. Make time to design a batch of experiments to run on AWS. Put each in a separate file and call them in turn from another script. This will allow you to answer multiple questions from one long run, perhaps overnight.

- **Always close your instance at the end of your experiments**. You do not want to be surprised with a very large AWS bill.

- **Try spot instances for a cheaper but less reliable option**. Amazon sell unused time on their hardware at a much cheaper price, but at the cost of potentially having your instance closed at any second. If you are learning or your experiments are not critical, this might be an ideal option for you. You can access spot instances from the *Spot Instance* option on the left hand side menu in your EC2 web console.

5.8 More Resources For Deep Learning on AWS

Below is a list of resources to learn more about AWS and developing deep learning models in the cloud.

- An introduction to Amazon Elastic Compute Cloud (EC2) if you are new to all of this.
  http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

- An introduction to Amazon Machine Images (AMI).
  http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

- AWS Tutorial for the Stanford CS231n Convolutional Neural Networks for Visual Recognition class (it is slightly out of date).
  http://cs231n.github.io/aws-tutorial/

- Learn more about how to configure Theano on the Theano Configuration page.
  http://deeplearning.net/software/theano/library/config.html

5.9 Summary

In this lesson you discovered how you can develop and evaluate your large deep learning models in Keras using GPUs on the Amazon Web Service. You learned:

- Amazon Web Services with their Elastic Compute Cloud offers an affordable way to run large deep learning models on GPU hardware.

- How to setup and launch an EC2 server for deep learning experiments.

- How to update the Keras version on the server and confirm that the system is working correctly.

- How to run Keras experiments on AWS instances in batch as background tasks.

5.9.1 Next

This concludes Part II and gives you the capability to install, configure and use the Python deep learning libraries on your workstation or in the cloud, leveraging GPU hardware. Next in Part III you will learn how to use the Keras API and develop your own neural network models.

Part III

Multilayer Perceptrons

Chapter 6

Crash Course In Multilayer Perceptrons

Artificial neural networks are a fascinating area of study, although they can be intimidating when just getting started. There is a lot of specialized terminology used when describing the data structures and algorithms used in the field. In this lesson you will get a crash course in the terminology and processes used in the field of Multilayer Perceptron artificial neural networks. After completing this lesson you will know:

- The building blocks of neural networks including neurons, weights and activation functions.

- How the building blocks are used in layers to create networks.

- How networks are trained from example data.

Let's get started.

6.1 Crash Course Overview

We are going to cover a lot of ground in this lesson. Here is an idea of what is ahead:

1. Multilayer Perceptrons.

2. Neurons, Weights and Activations.

3. Networks of Neurons.

4. Training Networks.

We will start off with an overview of Multilayer Perceptrons.

6.2   Multilayer Perceptrons

The field of artificial neural networks is often just called *Neural Networks* or *Multilayer Perceptrons* after perhaps the most useful type of neural network. A Perceptron is a single neuron model that was a precursor to larger neural networks. It is a field of study that investigates how simple models of biological brains can be used to solve difficult computational tasks like the predictive modeling tasks we see in machine learning. The goal is not to create realistic models of the brain, but instead to develop robust algorithms and data structures that we can use to model difficult problems.

The power of neural networks come from their ability to learn the representation in your training data and how to best relate it to the output variable that you want to predict. In this sense neural networks learn a mapping. Mathematically, they are capable of learning any mapping function and have been proven to be a universal approximation algorithm. The predictive capability of neural networks comes from the hierarchical or multilayered structure of the networks. The data structure can pick out (learn to represent) features at different scales or resolutions and combine them into higher-order features. For example from lines, to collections of lines to shapes.

6.3   Neurons

The building block for neural networks are artificial neurons. These are simple computational units that have weighted input signals and produce an output signal using an activation function.
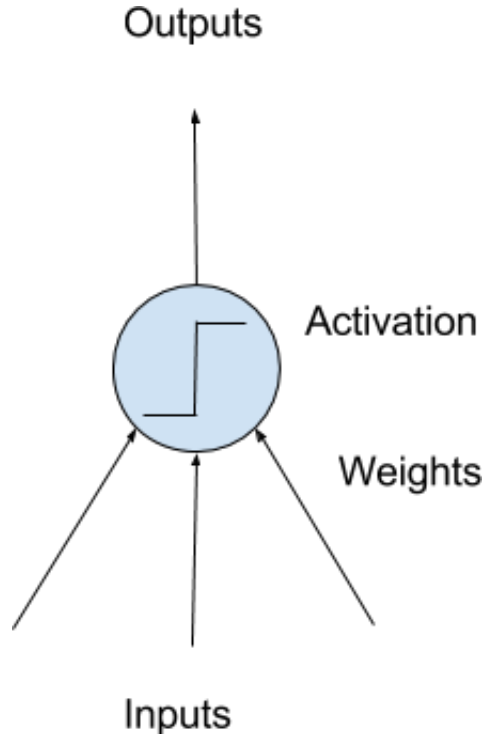
Figure 6.1: Model of a Simple Neuron

6.3.1 Neuron Weights

You may be familiar with linear regression, in which case the weights on the inputs are very much like the coefficients used in a regression equation. Like linear regression, each neuron also has a bias which can be thought of as an input that always has the value 1.0 and it too must be weighted. For example, a neuron may have two inputs in which case it requires three weights. One for each input and one for the bias.

Weights are often initialized to small random values, such as values in the range 0 to 0.3, although more complex initialization schemes can be used. Like linear regression, larger weights indicate increased complexity and fragility of the model. It is desirable to keep weights in the network small and regularization techniques can be used.

6.3.2 Activation

The weighted inputs are summed and passed through an activation function, sometimes called a transfer function. An activation function is a simple mapping of summed weighted input to the output of the neuron. It is called an activation function because it governs the threshold at which the neuron is activated and the strength of the output signal. Historically simple step activation functions were used where if the summed input was above a threshold, for example 0.5, then the neuron would output a value of 1.0, otherwise it would output a 0.0.

Traditionally nonlinear activation functions are used. This allows the network to combine the inputs in more complex ways and in turn provide a richer capability in the functions they can model. Nonlinear functions like the logistic function also called the sigmoid function were used that output a value between 0 and 1 with an s-shaped distribution, and the hyperbolic tangent function also called Tanh that outputs the same distribution over the range -1 to +1. More recently the rectifier activation function has been shown to provide better results.

6.4 Networks of Neurons

Neurons are arranged into networks of neurons. A row of neurons is called a layer and one network can have multiple layers. The architecture of the neurons in the network is often called the network topology.
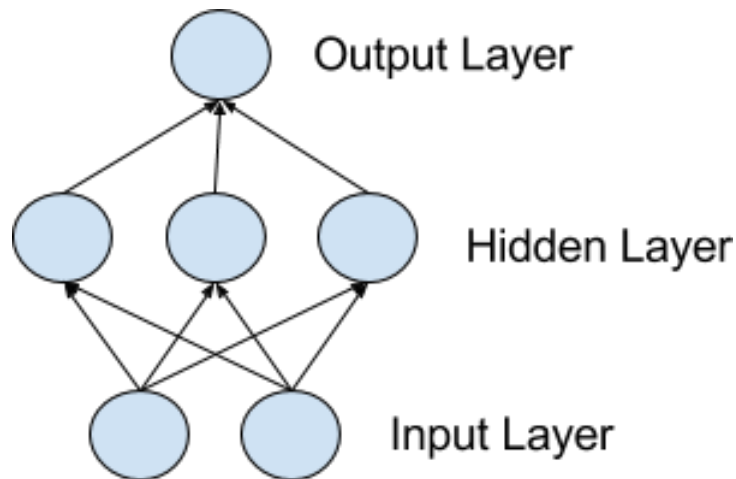
Figure 6.2: Model of a Simple Network

6.4.1 Input or Visible Layers

The bottom layer that takes input from your dataset is called the visible layer, because it is the exposed part of the network. Often a neural network is drawn with a visible layer with one neuron per input value or column in your dataset. These are not neurons as described above, but simply pass the input value though to the next layer.

6.4.2 Hidden Layers

Layers after the input layer are called hidden layers because they are not directly exposed to the input. The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value. Given increases in computing power and efficient libraries, very deep neural networks can be constructed. Deep learning can refer to having many hidden layers in your neural network. They are deep because they would have been unimaginably slow to train historically, but may take seconds or minutes to train using modern techniques and hardware.

6.4.3 Output Layer

The final hidden layer is called the output layer and it is responsible for outputting a value or vector of values that correspond to the format required for the problem. The choice of activation function in the output layer is strongly constrained by the type of problem that you are modeling. For example:

- A regression problem may have a single output neuron and the neuron may have no activation function.

- A binary classification problem may have a single output neuron and use a sigmoid activation function to output a value between 0 and 1 to represent the probability of predicting a value for the primary class. This can be turned into a crisp class value by using a threshold of 0.5 and snap values less than the threshold to 0 otherwise to 1.

- A multiclass classification problem may have multiple neurons in the output layer, one for each class (e.g. three neurons for the three classes in the famous iris flowers classification problem). In this case a softmax activation function may be used to output a probability of the network predicting each of the class values. Selecting the output with the highest probability can be used to produce a crisp class classification value.

6.5 Training Networks

Once configured, the neural network needs to be trained on your dataset.

6.5.1 Data Preparation

You must first prepare your data for training on a neural network. Data must be numerical, for example real values. If you have categorical data, such as a `sex` attribute with the values `male` and `female`, you can convert it to a real-valued representation called a one hot encoding. This is where one new column is added for each class value (two columns in the case of sex of male and female) and a 0 or 1 is added for each row depending on the class value for that row.

This same one hot encoding can be used on the output variable in classification problems with more than one class. This would create a binary vector from a single column that would be easy to directly compare to the output of the neuron in the network's output layer, that as described above, would output one value for each class. Neural networks require the input to be scaled in a consistent way. You can rescale it to the range between 0 and 1 called normalization. Another popular technique is to standardize it so that the distribution of each column has the mean of zero and the standard deviation of 1. Scaling also applies to image pixel data. Data such as words can be converted to integers, such as the frequency rank of the word in the dataset and other encoding techniques.

6.5.2 Stochastic Gradient Descent

The classical and still preferred training algorithm for neural networks is called stochastic gradient descent. This is where one row of data is exposed to the network at a time as input. The network processes the input upward activating neurons as it goes to finally produce an output value. This is called a forward pass on the network. It is the type of pass that is also used after the network is trained in order to make predictions on new data.

The output of the network is compared to the expected output and an error is calculated. This error is then propagated back through the network, one layer at a time, and the weights are updated according to the amount that they contributed to the error. This clever bit of math is called the Back Propagation algorithm. The process is repeated for all of the examples in your training data. One round of updating the network for the entire training dataset is called an epoch. A network may be trained for tens, hundreds or many thousands of epochs.

6.5.3 Weight Updates

The weights in the network can be updated from the errors calculated for each training example and this is called online learning. It can result in fast but also chaotic changes to the network.

Alternatively, the errors can be saved up across all of the training examples and the network can be updated at the end. This is called batch learning and is often more stable.

Because datasets are so large and because of computational efficiencies, the size of the batch, the number of examples the network is shown before an update is often reduced to a small number, such as tens or hundreds of examples. The amount that weights are updated is controlled by a configuration parameter called the learning rate. It is also called the step size and controls the step or change made to network weights for a given error. Often small learning rates are used such as 0.1 or 0.01 or smaller. The update equation can be complemented with additional configuration terms that you can set.

- **Momentum** is a term that incorporates the properties from the previous weight update to allow the weights to continue to change in the same direction even when there is less error being calculated.

- **Learning Rate Decay** is used to decrease the learning rate over epochs to allow the network to make large changes to the weights at the beginning and smaller fine tuning changes later in the training schedule.

6.5.4   Prediction

Once a neural network has been trained it can be used to make predictions. You can make predictions on test or validation data in order to estimate the skill of the model on unseen data. You can also deploy it operationally and use it to make predictions continuously. The network topology and the final set of weights is all that you need to save from the model. Predictions are made by providing the input to the network and performing a forward-pass allowing it to generate an output that you can use as a prediction.

6.6   Summary

In this lesson you discovered artificial neural networks for machine learning. You learned:

- How neural networks are not models of the brain but are instead computational models for solving complex machine learning problems.

- That neural networks are comprised of neurons that have weights and activation functions.

- The networks are organized into layers of neurons and are trained using stochastic gradient descent.

- That it is a good idea to prepare your data before training a neural network model.

6.6.1   Next

You now know the basics of neural network models. In the next section you will develop your very first Multilayer Perceptron model in Keras.

Chapter 7

Develop Your First Neural Network With Keras

Keras is a powerful and easy-to-use Python library for developing and evaluating deep learning models. It wraps the efficient numerical computation libraries Theano and TensorFlow and allows you to define and train neural network models in a few short lines of code. In this lesson you will discover how to create your first neural network model in Python using Keras. After completing this lesson you will know:

- How to load a CSV dataset ready for use with Keras.

- How to define and compile a Multilayer Perceptron model in Keras.

- How to evaluate a Keras model on a validation dataset.

Let's get started.

7.1 Tutorial Overview

There is not a lot of code required, but we are going to step over it slowly so that you will know how to create your own models in the future. The steps you are going to cover in this tutorial are as follows:

1. Load Data.

2. Define Model.

3. Compile Model.

4. Fit Model.

5. Evaluate Model.

6. Tie It All Together.

7.2 Pima Indians Onset of Diabetes Dataset

In this tutorial we are going to use the Pima Indians onset of diabetes dataset. This is a standard machine learning dataset available for free download from the UCI Machine Learning repository. It describes patient medical record data for Pima Indians and whether they had an onset of diabetes within five years. It is a binary classification problem (onset of diabetes as 1 or not as 0). The input variables that describe each patient are numerical and have varying scales. Below lists the eight attributes for the dataset:

1. Number of times pregnant.

2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test.

3. Diastolic blood pressure (mm Hg).

4. Triceps skin fold thickness (mm).

5. 2-Hour serum insulin (mu U/ml).

6. Body mass index.

7. Diabetes pedigree function.

8. Age (years).

9. Class, onset of diabetes within five years.

Given that all attributes are numerical makes it easy to use directly with neural networks that expect numerical inputs and output values, and ideal for our first neural network in Keras. This dataset will also be used for a number of additional lessons coming up in this book, so keep it handy. below is a sample of the dataset showing the first 5 rows of the 768 instances:

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
```

Listing 7.1: Sample of the Pima Indians Dataset.

The dataset file is available in your bundle of code recipes provided with this book. Alternatively, you can download the Pima Indian dataset from the UCI Machine Learning repository and place it in your local working directory, the same as your Python file[1]. Save it with the file name:

```
pima-indians-diabetes.csv
```

Listing 7.2: Pima Indians Dataset File.

---

[1]http://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data

The baseline accuracy if all predictions are made as *no onset of diabetes* is 65.1%. Top results on the dataset are in the range of 77.7% accuracy using 10-fold cross validation[2]. You can learn more about the dataset on the dataset home page on the UCI Machine Learning Repository[3].

7.3 Load Data

Whenever we work with machine learning algorithms that use a stochastic process (e.g. random numbers), it is a good idea to initialize the random number generator with a fixed seed value. This is so that you can run the same code again and again and get the same result. This is useful if you need to demonstrate a result, compare algorithms using the same source of randomness or to debug a part of your code. You can initialize the random number generator with any seed you like, for example:

```python
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

Listing 7.3: Load Libraries and Seed Random Number Generator.

Now we can load our Pima Indians dataset. You can now load the file directly using the NumPy function `loadtxt()`. There are eight input variables and one output variable (the last column). Once loaded we can split the dataset into input variables (X) and the output class variable (Y).

```python
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

Listing 7.4: Load The Dataset Using NumPy.

We have initialized our random number generator to ensure our results are reproducible and loaded our data. We are now ready to define our neural network model.

7.4 Define Model

Models in Keras are defined as a sequence of layers. We create a `Sequential` model and add layers one at a time until we are happy with our network topology. The first thing to get right is to ensure the input layer has the right number of inputs. This can be specified when creating the first layer with the `input_dim` argument and setting it to 8 for the 8 input variables.

How do we know the number of layers to use and their types? This is a very hard question. There are heuristics that we can use and often the best network structure is found through a process of trial and error experimentation. Generally, you need a network large enough

---

[2]http://www.is.umk.pl/projects/datasets.html#Diabetes
[3]http://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes

to capture the structure of the problem if that helps at all. In this example we will use a fully-connected network structure with three layers.

Fully connected layers are defined using the `Dense` class. We can specify the number of neurons in the layer as the first argument, the initialization method as the second argument as `init` and specify the activation function using the `activation` argument. In this case we initialize the network weights to a small random number generated from a uniform distribution (`uniform`), in this case between 0 and 0.05 because that is the default uniform weight initialization in Keras. Another traditional alternative would be `normal` for small random numbers generated from a Gaussian distribution.

We will use the rectifier (`relu`) activation function on the first two layers and the `sigmoid` activation function in the output layer. It used to be the case that sigmoid and tanh activation functions were preferred for all layers. These days, better performance is seen using the rectifier activation function. We use a sigmoid activation function on the output layer to ensure our network output is between 0 and 1 and easy to map to either a probability of class 1 or snap to a hard classification of either class with a default threshold of 0.5. We can piece it all together by adding each layer. The first hidden layer has 12 neurons and expects 8 input variables. The second hidden layer has 8 neurons and finally the output layer has 1 neuron to predict the class (onset of diabetes or not).

```python
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
```

Listing 7.5: Define the Neural Network Model in Keras.

Below provides a depiction of the network structure.
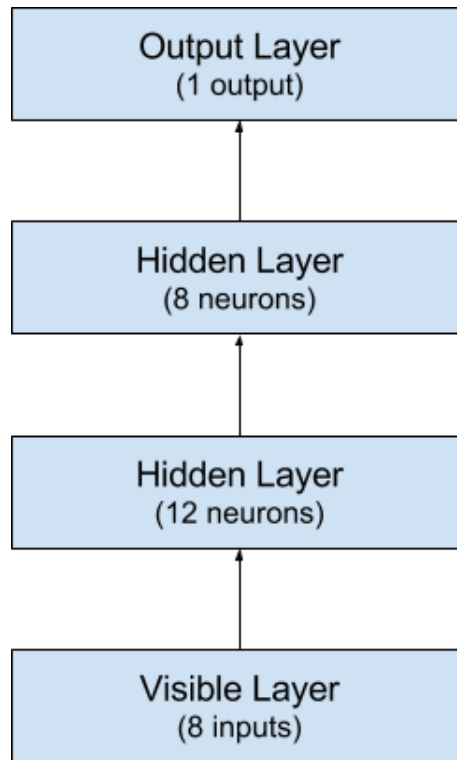
Figure 7.1: Visualization of Neural Network Structure.

7.5 Compile Model

Now that the model is defined, we can compile it. Compiling the model uses the efficient numerical libraries under the covers (the so-called backend) such as Theano or TensorFlow. The backend automatically chooses the best way to represent the network for training and making predictions to run on your hardware. When compiling, we must specify some additional properties required when training the network. Remember training a network means finding the best set of weights to make predictions for this problem.

We must specify the loss function to use to evaluate a set of weights, the optimizer used to search through different weights for the network and any optional metrics we would like to collect and report during training. In this case we will use logarithmic loss, which for a binary classification problem is defined in Keras as binary_crossentropy. We will also use the efficient gradient descent algorithm adam for no other reason that it is an efficient default. Learn more about the Adam optimization algorithm in the paper *Adam: A Method for Stochastic Optimization*[4]. Finally, because it is a classification problem, we will collect and report the classification accuracy as the metric.

```
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 7.6: Compile the Neural Network Model.

---

[4]http://arxiv.org/abs/1412.6980

7.6 Fit Model

We have defined our model and compiled it ready for efficient computation. Now it is time to execute the model on some data. We can train or fit our model on our loaded data by calling the `fit()` function on the model.

The training process will run for a fixed number of iterations through the dataset called epochs, that we must specify using the `nb_epoch` argument. We can also set the number of instances that are evaluated before a weight update in the network is performed called the batch size and set using the `batch_size` argument. For this problem we will run for a small number of epochs (150) and use a relatively small batch size of 10. Again, these can be chosen experimentally by trial and error.

```
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
```

Listing 7.7: Fit the Neural Network Model to the Dataset.

This is where the work happens on your CPU or GPU.

7.7 Evaluate Model

We have trained our neural network on the entire dataset and we can evaluate the performance of the network on the same dataset. This will only give us an idea of how well we have modeled the dataset (e.g. train accuracy), but no idea of how well the algorithm might perform on new data. We have done this for simplicity, but ideally, you could separate your data into train and test datasets for the training and evaluation of your model.

You can evaluate your model on your training dataset using the `evaluation()` function on your model and pass it the same input and output used to train the model. This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics you have configured, such as accuracy.

```
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Listing 7.8: Evaluate the Neural Network Model on the Dataset.

7.8 Tie It All Together

You have just seen how you can easily create your first neural network model in Keras. Let's tie it all together into a complete code example.

```
# Create your first MLP in Keras
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
```

```
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Listing 7.9: Complete Working Example of Your First Neural Network in Keras.

Running this example, you should see a message for each of the 150 epochs printing the loss and accuracy for each, followed by the final evaluation of the trained model on the training dataset. It takes about 10 seconds to execute on my workstation running on the CPU with a Theano backend.

```
...
Epoch 143/150
768/768 [==============================] - 0s - loss: 0.4614 - acc: 0.7878
Epoch 144/150
768/768 [==============================] - 0s - loss: 0.4508 - acc: 0.7969
Epoch 145/150
768/768 [==============================] - 0s - loss: 0.4580 - acc: 0.7747
Epoch 146/150
768/768 [==============================] - 0s - loss: 0.4627 - acc: 0.7812
Epoch 147/150
768/768 [==============================] - 0s - loss: 0.4531 - acc: 0.7943
Epoch 148/150
768/768 [==============================] - 0s - loss: 0.4656 - acc: 0.7734
Epoch 149/150
768/768 [==============================] - 0s - loss: 0.4566 - acc: 0.7839
Epoch 150/150
768/768 [==============================] - 0s - loss: 0.4593 - acc: 0.7839
768/768 [==============================] - 0s
acc: 79.56%
```

Listing 7.10: Output of Running Your First Neural Network in Keras.

7.9 Summary

In this lesson you discovered how to create your first neural network model using the powerful Keras Python library for deep learning. Specifically you learned the five key steps in using Keras to create a neural network or deep learning model, step-by-step including:

- How to load data.

- How to define a neural network model in Keras.

- How to compile a Keras model using the efficient numerical backend.

- How to train a model on data.

- How to evaluate a model on data.

7.9.1 Next

You now know how to develop a Multilayer Perceptron model in Keras. In the next section you will discover different ways that you can evaluate your models and estimate their performance on unseen data.

Chapter 8

Evaluate The Performance of Deep Learning Models

There are a lot of decisions to make when designing and configuring your deep learning models. Most of these decisions must be resolved empirically through trial and error and evaluating them on real data. As such, it is critically important to have a robust way to evaluate the performance of your neural network and deep learning models. In this lesson you will discover a few ways that you can use to evaluate model performance using Keras. After completing this lesson, you will know:

- How to evaluate a Keras model using an automatic verification dataset.

- How to evaluate a Keras model using a manual verification dataset.

- How to evaluate a Keras model using k-fold cross validation.

Let's get started.

8.1 Empirically Evaluate Network Configurations

There are a myriad of decisions you must make when designing and configuring your deep learning models. Many of these decisions can be resolved by copying the structure of other people's networks and using heuristics. Ultimately, the best technique is to actually design small experiments and empirically evaluate options using real data. This includes high-level decisions like the number, size and type of layers in your network. It also includes the lower level decisions like the choice of loss function, activation functions, optimization procedure and number of epochs.

Deep learning is often used on problems that have very large datasets. That is tens of thousands or hundreds of thousands of instances. As such, you need to have a robust test harness that allows you to estimate the performance of a given configuration on unseen data, and reliably compare the performance to other configurations.

8.2 Data Splitting

The large amount of data and the complexity of the models require very long training times. As such, it is typically to use a simple separation of data into training and test datasets or training

and validation datasets. Keras provides two convenient ways of evaluating your deep learning algorithms this way:

1. Use an automatic verification dataset.

2. Use a manual verification dataset.

8.2.1 Use a Automatic Verification Dataset

Keras can separate a portion of your training data into a validation dataset and evaluate the performance of your model on that validation dataset each epoch. You can do this by setting the `validation_split` argument on the `fit()` function to a percentage of the size of your training dataset. For example, a reasonable value might be 0.2 or 0.33 for 20% or 33% of your training data held back for validation. The example below demonstrates the use of using an automatic validation dataset on the Pima Indians onset of diabetes dataset (see Section 7.2).

```python
# MLP with automatic validation set
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, validation_split=0.33, nb_epoch=150, batch_size=10)
```

Listing 8.1: Evaluate A Neural Network Using an Automatic Validation Set.

Running the example, you can see that the verbose output on each epoch shows the loss and accuracy on both the training dataset and the validation dataset.

```
Epoch 145/150
514/514 [==============================] - 0s - loss: 0.4885 - acc: 0.7743 - val_loss:
    0.5016 - val_acc: 0.7638
Epoch 146/150
514/514 [==============================] - 0s - loss: 0.4862 - acc: 0.7704 - val_loss:
    0.5202 - val_acc: 0.7323
Epoch 147/150
514/514 [==============================] - 0s - loss: 0.4959 - acc: 0.7588 - val_loss:
    0.5012 - val_acc: 0.7598
Epoch 148/150
514/514 [==============================] - 0s - loss: 0.4966 - acc: 0.7665 - val_loss:
    0.5244 - val_acc: 0.7520
```

```
Epoch 149/150
514/514 [==============================] - 0s - loss: 0.4863 - acc: 0.7724 - val_loss:
    0.5074 - val_acc: 0.7717
Epoch 150/150
514/514 [==============================] - 0s - loss: 0.4884 - acc: 0.7724 - val_loss:
    0.5462 - val_acc: 0.7205
```

Listing 8.2: Output of Evaluating A Neural Network Using an Automatic Validation Set.

8.2.2   Use a Manual Verification Dataset

Keras also allows you to manually specify the dataset to use for validation during training. In this example we use the handy `train_test_split()` function from the Python scikit-learn machine learning library to separate our data into a training and test dataset. We use 67% for training and the remaining 33% of the data for validation. The validation dataset can be specified to the `fit()` function in Keras by the `validation_data` argument. It takes a tuple of the input and output datasets.

```python
# MLP with manual validation set
from keras.models import Sequential
from keras.layers import Dense
from sklearn.cross_validation import train_test_split
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# split into 67% for train and 33% for test
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33, random_state=seed)
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test,y_test), nb_epoch=150, batch_size=10)
```

Listing 8.3: Evaluate A Neural Network Using an Manual Validation Set.

Like before, running the example provides verbose output of training that includes the loss and accuracy of the model on both the training and validation datasets for each epoch.

```
...
Epoch 145/150
514/514 [==============================] - 0s - loss: 0.5001 - acc: 0.7685 - val_loss:
    0.5617 - val_acc: 0.7087
Epoch 146/150
514/514 [==============================] - 0s - loss: 0.5041 - acc: 0.7529 - val_loss:
    0.5423 - val_acc: 0.7362
```

```
Epoch 147/150
514/514 [==============================] - 0s - loss: 0.4936 - acc: 0.7685 - val_loss:
    0.5426 - val_acc: 0.7283
Epoch 148/150
514/514 [==============================] - 0s - loss: 0.4957 - acc: 0.7685 - val_loss:
    0.5430 - val_acc: 0.7362
Epoch 149/150
514/514 [==============================] - 0s - loss: 0.4953 - acc: 0.7685 - val_loss:
    0.5403 - val_acc: 0.7323
Epoch 150/150
514/514 [==============================] - 0s - loss: 0.4941 - acc: 0.7743 - val_loss:
    0.5452 - val_acc: 0.7323
```

Listing 8.4: Output of Evaluating A Neural Network Using an Manual Validation Set.

8.3   Manual k-Fold Cross Validation

The gold standard for machine learning model evaluation is k-fold cross validation. It provides a robust estimate of the performance of a model on unseen data. It does this by splitting the training dataset into k subsets and takes turns training models on all subsets except one which is held out, and evaluating model performance on the held out validation dataset. The process is repeated until all subsets are given an opportunity to be the held out validation set. The performance measure is then averaged across all models that are created.

Cross validation is often not used for evaluating deep learning models because of the greater computational expense. For example k-fold cross validation is often used with 5 or 10 folds. As such, 5 or 10 models must be constructed and evaluated, greatly adding to the evaluation time of a model. Nevertheless, when the problem is small enough or if you have sufficient compute resources, k-fold cross validation can give you a less biased estimate of the performance of your model.

In the example below we use the handy `StratifiedKFold` class[1] from the scikit-learn Python machine learning library to split up the training dataset into 10 folds. The folds are stratified, meaning that the algorithm attempts to balance the number of instances of each class in each fold. The example creates and evaluates 10 models using the 10 splits of the data and collects all of the scores. The verbose output for each epoch is turned off by passing `verbose=0` to the `fit()` and `evaluate()` functions on the model. The performance is printed for each model and it is stored. The average and standard deviation of the model performance is then printed at the end of the run to provide a robust estimate of model accuracy.

```python
# MLP for Pima Indians Dataset with 10-fold cross validation
from keras.models import Sequential
from keras.layers import Dense
from sklearn.cross_validation import StratifiedKFold
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
```

---

[1] http://scikit-learn.org/stable/modules/generated/sklearn.cross_validation.StratifiedKFold.html

```python
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# define 10-fold cross validation test harness
kfold = StratifiedKFold(y=Y, n_folds=10, shuffle=True, random_state=seed)
cvscores = []
for i, (train, test) in enumerate(kfold):
  # create model
  model = Sequential()
  model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
  model.add(Dense(8, init='uniform', activation='relu'))
  model.add(Dense(1, init='uniform', activation='sigmoid'))
  # Compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  # Fit the model
  model.fit(X[train], Y[train], nb_epoch=150, batch_size=10, verbose=0)
  # evaluate the model
  scores = model.evaluate(X[test], Y[test], verbose=0)
  print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
  cvscores.append(scores[1] * 100)

print("%.2f%% (+/- %.2f%%)" % (numpy.mean(cvscores), numpy.std(cvscores)))
```

Listing 8.5: Evaluate A Neural Network Using scikit-learn.

Running the example will take less than a minute and will produce the following output:

```
acc: 77.92%
acc: 79.22%
acc: 76.62%
acc: 77.92%
acc: 75.32%
acc: 74.03%
acc: 77.92%
acc: 71.43%
acc: 71.05%
acc: 75.00%
75.64% (+/- 2.67%)
```

Listing 8.6: Output of Evaluating A Neural Network Using scikit-learn.

Notice that we had to re-create the model each loop to then fit and evaluate it with the data for the fold. In the next lesson we will look at how we can use Keras models natively with the scikit-learn machine learning library.

8.4   Summary

In this lesson you discovered the importance of having a robust way to estimate the performance of your deep learning models on unseen data. You learned three ways that you can estimate the performance of your deep learning models in Python using the Keras library:

- Automatically splitting a training dataset into train and validation datasets.

- Manually and explicitly defining a training and validation dataset.

- Evaluating performance using k-fold cross validation, the gold standard technique.

8.4.1 Next

You now know how to evaluate your models and estimate their performance. In the next lesson you will discover how you can best integrate your Keras models with the scikit-learn machine learning library.

Chapter 9

Use Keras Models With Scikit-Learn For General Machine Learning

The scikit-learn library is the most popular library for general machine learning in Python. In this lesson you will discover how you can use deep learning models from Keras with the scikit-learn library in Python. After completing this lesson you will know:

- How to wrap a Keras model for use with the scikit-learn machine learning library.

- How to easily evaluate Keras models using cross validation in scikit-learn.

- How to tune Keras model hyperparameters using grid search in scikit-learn.

Let's get started.

9.1   Overview

Keras is a popular library for deep learning in Python, but the focus of the library is deep learning. In fact it strives for minimalism, focusing on only what you need to quickly and simply define and build deep learning models. The scikit-learn library in Python is built upon the SciPy stack for efficient numerical computation. It is a fully featured library for general purpose machine learning and provides many utilities that are useful in the development of deep learning models. Not least:

- Evaluation of models using resampling methods like k-fold cross validation.

- Efficient search and evaluation of model hyperparameters.

The Keras library provides a convenient wrapper for deep learning models to be used as classification or regression estimators in scikit-learn. In the next sections we will work through examples of using the `KerasClassifier` wrapper for a classification neural network created in Keras and used in the scikit-learn library. The test problem is the Pima Indians onset of diabetes classification dataset (see Section 7.2).

9.2 Evaluate Models with Cross Validation

The `KerasClassifier` and `KerasRegressor` classes in Keras take an argument `build_fn` which is the name of the function to call to create your model. You must define a function called whatever you like that defines your model, compiles it and returns it. In the example below we define a function `create_model()` that create a simple multilayer neural network for the problem.

We pass this function name to the `KerasClassifier` class by the `build_fn` argument. We also pass in additional arguments of `nb_epoch=150` and `batch_size=10`. These are automatically bundled up and passed on to the `fit()` function which is called internally by the `KerasClassifier` class. In this example we use the scikit-learn `StratifiedKFold` to perform 10-fold stratified cross validation. This is a resampling technique that can provide a robust estimate of the performance of a machine learning model on unseen data. We use the scikit-learn function `cross_val_score()` to evaluate our model using the cross validation scheme and print the results.

```python
# MLP for Pima Indians Dataset with 10-fold cross validation via sklearn
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.cross_validation import StratifiedKFold
from sklearn.cross_validation import cross_val_score
import numpy

# Function to create model, required for KerasClassifier
def create_model():
  # create model
  model = Sequential()
  model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
  model.add(Dense(8, init='uniform', activation='relu'))
  model.add(Dense(1, init='uniform', activation='sigmoid'))
  # Compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model

# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, nb_epoch=150, batch_size=10, verbose=0)
# evaluate using 10-fold cross validation
kfold = StratifiedKFold(y=Y, n_folds=10, shuffle=True, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Listing 9.1: Evaluate A Neural Network Using scikit-learn.

Running the example displays the skill of the model for each epoch. A total of 10 models are created and evaluated and the final average accuracy is displayed.

```
0.756442244065
```

Listing 9.2: Output of Evaluate A Neural Network Using scikit-learn.

You can see that when the Keras model is wrapped that estimating model accuracy can be greatly streamlined, compared to the manual enumeration of cross validation folds performed in the previous lesson.

9.3 Grid Search Deep Learning Model Parameters

The previous example showed how easy it is to wrap your deep learning model from Keras and use it in functions from the scikit-learn library. In this example we go a step further. We already know we can provide arguments to the `fit()` function. The function that we specify to the `build_fn` argument when creating the `KerasClassifier` wrapper can also take arguments. We can use these arguments to further customize the construction of the model.

In this example we use a grid search to evaluate different configurations for our neural network model and report on the combination that provides the best estimated performance. The `create_model()` function is defined to take two arguments `optimizer` and `init`, both of which must have default values. This will allow us to evaluate the effect of using different optimization algorithms and weight initialization schemes for our network. After creating our model, we define arrays of values for the parameter we wish to search, specifically:

- Optimizers for searching different weight values.

- Initializers for preparing the network weights using different schemes.

- Number of epochs for training the model for different number of exposures to the training dataset.

- Batches for varying the number of samples before weight updates.

The options are specified into a dictionary and passed to the configuration of the `GridSearchCV` scikit-learn class. This class will evaluate a version of our neural network model for each combination of parameters ($2 \times 3 \times 3 \times 3$) for the combinations of optimizers, initializations, epochs and batches). Each combination is then evaluated using the default of 3-fold stratified cross validation.

That is a lot of models and a lot of computation. This is not a scheme that you want to use lightly because of the time it will take to compute. It may be useful for you to design small experiments with a smaller subset of your data that will complete in a reasonable time. This experiment is reasonable in this case because of the small network and the small dataset (less than 1,000 instances and 9 attributes). Finally, the performance and combination of configurations for the best model are displayed, followed by the performance of all combinations of parameters.

```python
# MLP for Pima Indians Dataset with grid search via sklearn
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.grid_search import GridSearchCV
```

```python
import numpy

# Function to create model, required for KerasClassifier
def create_model(optimizer='rmsprop', init='glorot_uniform'):
  # create model
  model = Sequential()
  model.add(Dense(12, input_dim=8, init=init, activation='relu'))
  model.add(Dense(8, init=init, activation='relu'))
  model.add(Dense(1, init=init, activation='sigmoid'))
  # Compile model
  model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
  return model

# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, verbose=0)
# grid search epochs, batch size and optimizer
optimizers = ['rmsprop', 'adam']
init = ['glorot_uniform', 'normal', 'uniform']
epochs = numpy.array([50, 100, 150])
batches = numpy.array([5, 10, 20])
param_grid = dict(optimizer=optimizers, nb_epoch=epochs, batch_size=batches, init=init)
grid = GridSearchCV(estimator=model, param_grid=param_grid)
grid_result = grid.fit(X, Y)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
for params, mean_score, scores in grid_result.grid_scores_:
    print("%f (%f) with: %r" % (scores.mean(), scores.std(), params))
```

Listing 9.3: Grid Search Neural Network Parameters Using scikit-learn.

This might take about 5 minutes to complete on your workstation executed on the CPU. running the example shows the results below. We can see that the grid search discovered that using a uniform initialization scheme, `rmsprop` optimizer, 150 epochs and a batch size of 5 achieved the best cross validation score of approximately 75% on this problem.

```
Best: 0.750000 using {'init': 'normal', 'optimizer': 'rmsprop', 'nb_epoch': 150,
    'batch_size': 5}
0.662760 (0.038450) with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch':
    50, 'batch_size': 5}
0.665365 (0.004872) with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 50,
    'batch_size': 5}
0.669271 (0.028940) with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch':
    100, 'batch_size': 5}
0.709635 (0.034987) with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 100,
    'batch_size': 5}
0.699219 (0.022097) with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch':
    150, 'batch_size': 5}
0.725260 (0.008027) with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 150,
```

```
    'batch_size': 5}
...
```

Listing 9.4: Output of Grid Search Neural Network Parameters Using scikit-learn.

9.4 Summary

In this lesson you discovered how you can wrap your Keras deep learning models and use them in the scikit-learn general machine learning library. You learned:

- Specifically how to wrap Keras models so that they can be used with the scikit-learn machine learning library.

- How to use a wrapped Keras model as part of evaluating model performance in scikit-learn.

- How to perform hyperparameter tuning in scikit-learn using a wrapped Keras model.

You can see that using scikit-learn for standard machine learning operations such as model evaluation and model hyperparameter optimization can save a lot of time over implementing these schemes yourself.

9.4.1 Next

You now know how to best integrate your Keras models into the scikit-learn machine learning library. Now it is time to put your new skills to the test. Over the next few chapters you will practice developing neural network models in Keras end-to-end, starting with a multiclass classification problem next.

Chapter 10

Project: Multiclass Classification Of Flower Species

In this project tutorial you will discover how you can use Keras to develop and evaluate neural network models for multiclass classification problems. After completing this step-by-step tutorial, you will know:

- How to load data from CSV and make it available to Keras.

- How to prepare multiclass classification data for modeling with neural networks.

- How to evaluate Keras neural network models with scikit-learn.

Let's get started.

10.1   Iris Flowers Classification Dataset

In this tutorial we will use the standard machine learning problem called the iris flowers dataset. This dataset is well studied and is a good problem for practicing on neural networks because all of the 4 input variables are numeric and have the same scale in centimeters. Each instance describes the properties of an observed flower measurements and the output variable is specific iris species. The attributes for this dataset can be summarized as follows:

1. Sepal length in centimeters.

2. Sepal width in centimeters.

3. Petal length in centimeters.

4. Petal width in centimeters.

5. Class.

This is a multiclass classification problem, meaning that there are more than two classes to be predicted, in fact there are three flower species. This is an important type of problem on which to practice with neural networks because the three class values require specialized handling. Below is a sample of the first five of the 150 instances:

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
```

Listing 10.1: Sample of the Iris Flowers Dataset.

The iris flower dataset is a well studied problem and a such we can expect to achieve a model accuracy in the range of 95% to 97%. This provides a good target to aim for when developing our models. The dataset is provided in the bundle of sample code provided with this book. You can also download the iris flowers dataset from the UCI Machine Learning repository[1] and place it in your current working directory with the filename `iris.csv`. You can learn more about the iris flower classification dataset on the UCI Machine Learning Repository page[2].

10.2   Import Classes and Functions

We can begin by importing all of the classes and functions we will need in this tutorial. This includes both the functionality we require from Keras, but also data loading from Pandas as well as data preparation and model evaluation from scikit-learn.

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from keras.utils import np_utils
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
```

Listing 10.2: Import Classes and Functions.

10.3   Initialize Random Number Generator

Next we need to initialize the random number generator to a constant value. This is important to ensure that the results we achieve from this model can be achieved again precisely. It ensures that the stochastic process of training a neural network model can be reproduced.

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

Listing 10.3: Initialize Random Number Generator.

---

[1]http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
[2]https://archive.ics.uci.edu/ml/datasets/Iris

10.4 Load The Dataset

The dataset can be loaded directly. Because the output variable contains strings, it is easiest to load the data using pandas. We can then split the attributes (columns) into input variables (X) and output variables (Y).

```
# load dataset
dataframe = pandas.read_csv("iris.csv", header=None)
dataset = dataframe.values
X = dataset[:,0:4].astype(float)
Y = dataset[:,4]
```
Listing 10.4: Load Dataset And Separate Into Input and Output Variables.

10.5 Encode The Output Variable

The output variable contains three different string values. When modeling multiclass classification problems using neural networks, it is good practice to reshape the output attribute from a vector that contains values for each class value to be a matrix with a boolean for each class value and whether or not a given instance has that class value or not. This is called one hot encoding or creating dummy variables from a categorical variable. For example, in this problem the three class values are `Iris-setosa`, `Iris-versicolor` and `Iris-virginica`. If we had the three observations:

```
Iris-setosa
Iris-versicolor
Iris-virginica
```
Listing 10.5: Three Classes In The Iris Dataset.

We can turn this into a one-hot encoded binary matrix for each data instance that would look as follows:

```
Iris-setosa, Iris-versicolor, Iris-virginica
1,        0,          0
0,        1,          0
0,        0,          1
```
Listing 10.6: One Hot Encoding of The Classes In The Iris Dataset.

We can do this by first encoding the strings consistently to integers using the scikit-learn class `LabelEncoder`. Then convert the vector of integers to a one hot encoding using the Keras function `to_categorical()`.

```
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = np_utils.to_categorical(encoded_Y)
```
Listing 10.7: One Hot Encoding Of Iris Dataset Output Variable.

10.6 Define The Neural Network Model

The Keras library provides wrapper classes to allow you to use neural network models developed with Keras in scikit-learn as we saw in the previous lesson. There is a `KerasClassifier` class in Keras that can be used as an Estimator in scikit-learn, the base type of model in the library. The `KerasClassifier` takes the name of a function as an argument. This function must return the constructed neural network model, ready for training.

Below is a function that will create a baseline neural network for the iris classification problem. It creates a simple fully connected network with one hidden layer that contains 4 neurons, the same number of inputs (it could be any number of neurons). The hidden layer uses a rectifier activation function which is a good practice. Because we used a one-hot encoding for our iris dataset, the output layer must create 3 output values, one for each class. The output value with the largest value will be taken as the class predicted by the model. The network topology of this simple one-layer neural network can be summarized as:

```
4 inputs -> [4 hidden nodes] -> 3 outputs
```

Listing 10.8: Example Network Structure.

Note that we use a sigmoid activation function in the output layer. This is to ensure the output values are in the range of 0 and 1 and may be used as predicted probabilities. Finally, the network uses the efficient ADAM gradient descent optimization algorithm with a logarithmic loss function, which is called `categorical_crossentropy` in Keras.

```python
# define baseline model
def baseline_model():
  # create model
  model = Sequential()
  model.add(Dense(4, input_dim=4, init='normal', activation='relu'))
  model.add(Dense(3, init='normal', activation='sigmoid'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
```

Listing 10.9: Define and Compile the Neural Network Model.

We can now create our `KerasClassifier` for use in scikit-learn. We can also pass arguments in the construction of the `KerasClassifier` class that will be passed on to the `fit()` function internally used to train the neural network. Here, we pass the number of epochs `nb_epoch` as 200 and `batch_size` as 5 to use when training the model. Debugging is also turned off when training by setting `verbose` to 0.

```python
estimator = KerasClassifier(build_fn=baseline_model, nb_epoch=200, batch_size=5, verbose=0)
```

Listing 10.10: Create Wrapper For Neural Network Model For Use in scikit-learn.

10.7 Evaluate The Model with k-Fold Cross Validation

We can now evaluate the neural network model on our training data. The scikit-learn library has excellent capability to evaluate models using a suite of techniques. The gold standard for evaluating machine learning models is k-fold cross validation. First we can define the model

evaluation procedure. Here, we set the number of folds to be 10 (an excellent default) and to shuffle the data before partitioning it.

```python
kfold = KFold(n=len(X), n_folds=10, shuffle=True, random_state=seed)
```

Listing 10.11: Prepare Cross Validation.

Now we can evaluate our model (estimator) on our dataset (X and dummy_y) using a 10-fold cross validation procedure (kfold). Evaluating the model only takes approximately 10 seconds and returns an object that describes the evaluation of the 10 constructed models for each of the splits of the dataset.

```python
results = cross_val_score(estimator, X, dummy_y, cv=kfold)
print("Accuracy: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 10.12: Evaluate the Neural Network Model.

The full code listing is provided below for completeness.

```python
# Multiclass Classification with the Iris Flowers Dataset
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from keras.utils import np_utils
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("iris.csv", header=None)
dataset = dataframe.values
X = dataset[:,0:4].astype(float)
Y = dataset[:,4]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = np_utils.to_categorical(encoded_Y)
# define baseline model
def baseline_model():
  # create model
  model = Sequential()
  model.add(Dense(4, input_dim=4, init='normal', activation='relu'))
  model.add(Dense(3, init='normal', activation='sigmoid'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
estimator = KerasClassifier(build_fn=baseline_model, nb_epoch=200, batch_size=5, verbose=0)
kfold = KFold(n=len(X), n_folds=10, shuffle=True, random_state=seed)
results = cross_val_score(estimator, X, dummy_y, cv=kfold)
print("Accuracy: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 10.13: Multilayer Perceptron Model for Iris Flowers Problem.

The results are summarized as both the mean and standard deviation of the model accuracy on the dataset. This is a reasonable estimation of the performance of the model on unseen data. It is also within the realm of known top results for this problem.

```
Accuracy: 95.33% (4.27%)
```

Listing 10.14: Estimated Accuracy of Neural Network Model on the Iris Dataset.

10.8   Summary

In this lesson you discovered how to develop and evaluate a neural network using the Keras Python library for deep learning. By completing this tutorial, you learned:

- How to load data and make it available to Keras.

- How to prepare multiclass classification data for modeling using one hot encoding.

- How to use Keras neural network models with scikit-learn.

- How to define a neural network using Keras for multiclass classification.

- How to evaluate a Keras neural network model using scikit-learn with k-fold cross validation.

10.8.1   Next

This was your first end-to-end project using Keras on a standalone dataset. In the next tutorial you will develop neural network models for a binary classification problem and tune them to get increases in model performance.

Chapter 11

Project: Binary Classification Of Sonar Returns

In this project tutorial you will discover how to effectively use the Keras library in your machine learning project by working through a binary classification project step-by-step. After completing this step-by-step tutorial, you will know:

- How to load training data and make it available to Keras.

- How to design and train a neural network for tabular data.

- How to evaluate the performance of a neural network model in Keras on unseen data.

- How to perform data preparation to improve skill when using neural networks.

- How to tune the topology and configuration of neural networks in Keras.

Let's get started.

11.1   Sonar Object Classification Dataset

The dataset we will use in this tutorial is the Sonar dataset. This is a dataset that describes sonar chirp returns bouncing off different surfaces. The 60 input variables are the strength of the returns at different angles. It is a binary classification problem that requires a model to differentiate rocks from metal cylinders.

It is a well understood dataset. All of the variables are continuous and generally in the range of 0 to 1. The output variable is a string `M` for mine and `R` for rock, which will need to be converted to integers 1 and 0. The dataset contains 208 observations. The dataset is in the bundle of source code provided with this book. Alternatively, you can download the dataset and place it in your working directory with the filename `sonar.csv`[1].

A benefit of using this dataset is that it is a standard benchmark problem. This means that we have some idea of the expected skill of a good model. Using cross validation, a neural network should be able to achieve performance around 84% with an upper bound on accuracy

---

[1]https://archive.ics.uci.edu/ml/machine-learning-databases/undocumented/connectionist-bench/sonar/sonar.all-data

for custom models at around 88%[2]. You can learn more about this dataset on the UCI Machine Learning repository[3].

11.2 Baseline Neural Network Model Performance

Let's create a baseline model and result for this problem. We will start off by importing all of the classes and functions we will need.

```python
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.cross_validation import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.cross_validation import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

Listing 11.1: Import Classes and Functions.

Next, we can initialize the random number generator to ensure that we always get the same results when executing this code. This will help if we are debugging.

```python
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

Listing 11.2: Initialize The Random Number Generator.

Now we can load the dataset using Pandas and split the columns into 60 input variables (X) and 1 output variable (Y). We use Pandas to load the data because it easily handles strings (the output variable), whereas attempting to load the data directly using NumPy would be more difficult.

```python
# load dataset
dataframe = pandas.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
```

Listing 11.3: Load The Dataset And Separate Into Input and Output Variables.

The output variable is string values. We must convert them into integer values 0 and 1. We can do this using the `LabelEncoder` class from scikit-learn. This class will model the encoding required using the entire dataset via the `fit()` function, then apply the encoding to create a new output variable using the `transform()` function.

```python
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
```

---

[2]http://www.is.umk.pl/projects/datasets.html#Sonar
[3]https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks)

```
encoded_Y = encoder.transform(Y)
```

Listing 11.4: Label Encode Output Variable.

We are now ready to create our neural network model using Keras. We are going to use scikit-learn to evaluate the model using stratified k-fold cross validation. This is a resampling technique that will provide an estimate of the performance of the model. To use Keras models with scikit-learn, we must use the `KerasClassifier` wrapper. This class takes a function that creates and returns our neural network model. It also takes arguments that it will pass along to the call to `fit()` such as the number of epochs and the batch size. Let's start off by defining the function that creates our baseline model. Our model will have a single fully connected hidden layer with the same number of neurons as input variables. This is a good default starting point when creating neural networks on a new problem.

The weights are initialized using a small Gaussian random number. The Rectifier activation function is used. The output layer contains a single neuron in order to make predictions. It uses the sigmoid activation function in order to produce a probability output in the range of 0 to 1 that can easily and automatically be converted to crisp class values. Finally, we are using the logarithmic loss function (`binary_crossentropy`) during training, the preferred loss function for binary classification problems. The model also uses the efficient Adam optimization algorithm for gradient descent and accuracy metrics will be collected when the model is trained.

```python
# baseline model
def create_baseline():
  # create model
  model = Sequential()
  model.add(Dense(60, input_dim=60, init='normal', activation='relu'))
  model.add(Dense(1, init='normal', activation='sigmoid'))
  # Compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
```

Listing 11.5: Define and Compile Baseline Model.

Now it is time to evaluate this model using stratified cross validation in the scikit-learn framework. We pass the number of training epochs to the `KerasClassifier`, again using reasonable default values. Verbose output is also turned off given that the model will be created 10 times for the 10-fold cross validation being performed.

```python
# evaluate model with standardized dataset
estimator = KerasClassifier(build_fn=create_baseline, nb_epoch=100, batch_size=5, verbose=0)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, random_state=seed)
results = cross_val_score(estimator, X, encoded_Y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 11.6: Fit And Evaluate Baseline Model.

The full code listing is provided below for completeness.

```python
# Binary Classification with Sonar Dataset: Baseline
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.cross_validation import cross_val_score
```

```python
from sklearn.preprocessing import LabelEncoder
from sklearn.cross_validation import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# baseline model
def create_baseline():
  # create model
  model = Sequential()
  model.add(Dense(60, input_dim=60, init='normal', activation='relu'))
  model.add(Dense(1, init='normal', activation='sigmoid'))
  # Compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
# evaluate model with standardized dataset
estimator = KerasClassifier(build_fn=create_baseline, nb_epoch=100, batch_size=5, verbose=0)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, random_state=seed)
results = cross_val_score(estimator, X, encoded_Y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 11.7: Multilayer Perceptron Model for Sonar Problem.

Running this code produces the following output showing the mean and standard deviation of the estimated accuracy of the model on unseen data.

```
Baseline: 81.68% (5.67%)
```

Listing 11.8: Sample Output From Fitting And Evaluating The Baseline Model.

This is an excellent score without doing any hard work.

11.3 Improve Performance With Data Preparation

It is a good practice to prepare your data before modeling. Neural network models are especially suitable to having consistent input values, both in scale and distribution. An effective data preparation scheme for tabular data when building neural network models is standardization. This is where the data is rescaled such that the mean value for each attribute is 0 and the standard deviation is 1. This preserves Gaussian and Gaussian-like distributions whilst normalizing the central tendencies for each attribute.

We can use scikit-learn to perform the standardization of our Sonar dataset using the StandardScaler class. Rather than performing the standardization on the entire dataset, it is good practice to train the standardization procedure on the training data within the pass of a

cross validation run and to use the trained standardization instance to prepare the unseen test fold. This makes standardization a step in model preparation in the cross validation process and it prevents the algorithm having knowledge of unseen data during evaluation, knowledge that might be passed from the data preparation scheme like a crisper distribution.

We can achieve this in scikit-learn using a `Pipeline` class. The pipeline is a wrapper that executes one or more models within a pass of the cross validation procedure. Here, we can define a pipeline with the `StandardScaler` followed by our neural network model.

```python
# Binary Classification with Sonar Dataset: Standardized
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.cross_validation import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.cross_validation import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# baseline model
def create_baseline():
  # create model
  model = Sequential()
  model.add(Dense(60, input_dim=60, init='normal', activation='relu'))
  model.add(Dense(1, init='normal', activation='sigmoid'))
  # Compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
# evaluate baseline model with standardized dataset
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_baseline, nb_epoch=100,
    batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, random_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Standardized: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 11.9: Update Experiment To Use Data Standardization.

Running this example provides the results below. We do see a small but very nice lift in the

mean accuracy.

```
Standardized: 84.07% (6.23%)
```

Listing 11.10: Sample Output From Update Using Data Standardization.

11.4 Tuning Layers and Neurons in The Model

There are many things to tune on a neural network, such as the weight initialization, activation functions, optimization procedure and so on. One aspect that may have an outsized effect is the structure of the network itself called the network topology. In this section we take a look at two experiments on the structure of the network: making it smaller and making it larger. These are good experiments to perform when tuning a neural network on your problem.

11.4.1 Evaluate a Smaller Network

I suspect that there is a lot of redundancy in the input variables for this problem. The data describes the same signal from different angles. Perhaps some of those angles are more relevant than others. We can force a type of feature extraction by the network by restricting the representational space in the first hidden layer.

In this experiment we take our baseline model with 60 neurons in the hidden layer and reduce it by half to 30. This will put pressure on the network during training to pick out the most important structure in the input data to model. We will also standardize the data as in the previous experiment with data preparation and try to take advantage of the small lift in performance.

```python
# Binary Classification with Sonar Dataset: Standardized Smaller
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.cross_validation import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.cross_validation import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# smaller model
def create_smaller():
```

```
  # create model
  model = Sequential()
  model.add(Dense(30, input_dim=60, init='normal', activation='relu'))
  model.add(Dense(1, init='normal', activation='sigmoid'))
  # Compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_smaller, nb_epoch=100,
    batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, random_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Smaller: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 11.11: Update To Use a Smaller Network Topology.

Running this example provides the following result. We can see that we have a very slight boost in the mean estimated accuracy and an important reduction in the standard deviation (average spread) of the accuracy scores for the model. This is a great result because we are doing slightly better with a network half the size, which in turn takes half the time to train.

```
Smaller: 84.61% (4.65%)
```

Listing 11.12: Sample Output From Using A Smaller Network Topology.

11.4.2 Evaluate a Larger Network

A neural network topology with more layers offers more opportunity for the network to extract key features and recombine them in useful nonlinear ways. We can evaluate whether adding more layers to the network improves the performance easily by making another small tweak to the function used to create our model. Here, we add one new layer (one line) to the network that introduces another hidden layer with 30 neurons after the first hidden layer. Our network now has the topology:

```
60 inputs -> [60 -> 30] -> 1 output
```

Listing 11.13: Summary of New Network Topology.

The idea here is that the network is given the opportunity to model all input variables before being bottlenecked and forced to halve the representational capacity, much like we did in the experiment above with the smaller network. Instead of squeezing the representation of the inputs themselves, we have an additional hidden layer to aid in the process.

```
# Binary Classification with Sonar Dataset: Standardized Larger
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.cross_validation import cross_val_score
from sklearn.preprocessing import LabelEncoder
```

```
from sklearn.cross_validation import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# larger model
def create_larger():
  # create model
  model = Sequential()
  model.add(Dense(60, input_dim=60, init='normal', activation='relu'))
  model.add(Dense(30, init='normal', activation='relu'))
  model.add(Dense(1, init='normal', activation='sigmoid'))
  # Compile model
  model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_larger, nb_epoch=100,
    batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, random_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Larger: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 11.14: Update To Use a Larger Network Topology.

Running this example produces the results below. We can see that we do get a nice lift in the model performance, achieving near state-of-the-art results with very little effort indeed.

```
Larger: 86.47% (3.82%)
```

Listing 11.15: Sample Output From Using A Larger Network Topology.

With further tuning of aspects like the optimization algorithm and the number of training epochs, it is expected that further improvements are possible. What is the best score that you can achieve on this dataset?

11.5 Summary

In this lesson you discovered how you can work through a binary classification problem step-by-step with Keras, specifically:

- How to load and prepare data for use in Keras.

- How to create a baseline neural network model.

- How to evaluate a Keras model using scikit-learn and stratified k-fold cross validation.

- How data preparation schemes can lift the performance of your models.

- How experiments adjusting the network topology can lift model performance.

11.5.1   Next

You now know how to develop neural network models in Keras for multiclass and binary classification problems. In the next tutorial you will work through a project to develop neural network models for a regression problem.

Chapter 12

Project: Regression Of Boston House Prices

In this project tutorial you will discover how to develop and evaluate neural network models using Keras for a regression problem. After completing this step-by-step tutorial, you will know:

- How to load a CSV dataset and make it available to Keras.

- How to create a neural network model with Keras for a regression problem.

- How to use scikit-learn with Keras to evaluate models using cross validation.

- How to perform data preparation in order to improve skill with Keras models.

- How to tune the network topology of models with Keras.

Let's get started.

12.1   Boston House Price Dataset

The problem that we will look at in this tutorial is the Boston house price dataset. The dataset describes properties of houses in Boston suburbs and is concerned with modeling the price of houses in those suburbs in thousands of dollars. As such, this is a regression predictive modeling problem. There are 13 input variables that describe the properties of a given Boston suburb. The full list of attributes in this dataset are as follows:

1. CRIM: per capita crime rate by town.

2. ZN: proportion of residential land zoned for lots over 25,000 sq.ft.

3. INDUS: proportion of non-retail business acres per town.

4. CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).

5. NOX: nitric oxides concentration (parts per 10 million).

6. RM: average number of rooms per dwelling.

7. `AGE`: proportion of owner-occupied units built prior to 1940.

8. `DIS`: weighted distances to five Boston employment centers.

9. `RAD`: index of accessibility to radial highways.

10. `TAX`: full-value property-tax rate per $10,000.

11. `PTRATIO`: pupil-teacher ratio by town.

12. `B`: $1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town.

13. `LSTAT`: % lower status of the population.

14. `MEDV`: Median value of owner-occupied homes in $1000s.

This is a well studied problem in machine learning. It is convenient to work with because all of the input and output attributes are numerical and there are 506 instances to work with. A sample of the first 5 rows of the 506 in the dataset is provided below:

```
0.00632 18.00   2.310 0  0.5380 6.5750 65.20 4.0900   1 296.0 15.30 396.90  4.98 24.00
0.02731  0.00   7.070 0  0.4690 6.4210 78.90 4.9671   2 242.0 17.80 396.90  9.14 21.60
0.02729  0.00   7.070 0  0.4690 7.1850 61.10 4.9671   2 242.0 17.80 392.83  4.03 34.70
0.03237  0.00   2.180 0  0.4580 6.9980 45.80 6.0622   3 222.0 18.70 394.63  2.94 33.40
0.06905  0.00   2.180 0  0.4580 7.1470 54.20 6.0622   3 222.0 18.70 396.90  5.33 36.20
```

Listing 12.1: Sample of the Boston House Price Dataset.

The dataset is available in the bundle of source code provided with this book. Alternatively, you can download this dataset and save it to your current working directly with the file name `housing.csv`[1]. Reasonable performance for models evaluated using Mean Squared Error (MSE) are around 20 in squared thousands of dollars (or $4,500 if you take the square root). This is a nice target to aim for with our neural network model. You can learn more about the Boston house price dataset on the UCI Machine Learning Repository[2].

12.2 Develop a Baseline Neural Network Model

In this section we will create a baseline neural network model for the regression problem. Let's start off by importing all of the functions and objects we will need for this tutorial.

```python
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

Listing 12.2: Import Classes and Functions.

---

[1]https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data
[2]https://archive.ics.uci.edu/ml/datasets/Housing

We can now load our dataset from a file in the local directory. The dataset is in fact not in CSV format on the UCI Machine Learning Repository, the attributes are instead separated by whitespace. We can load this easily using the Pandas library. We can then split the input (X) and output (Y) attributes so that they are easier to model with Keras and scikit-learn.

```python
# load dataset
dataframe = pandas.read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
```

Listing 12.3: Load Dataset and Separate Into Input and Output Variables.

We can create Keras models and evaluate them with scikit-learn by using handy wrapper objects provided by the Keras library. This is desirable, because scikit-learn excels at evaluating models and will allow us to use powerful data preparation and model evaluation schemes with very few lines of code. The Keras wrapper class require a function as an argument. This function that we must define is responsible for creating the neural network model to be evaluated.

Below we define the function to create the baseline model to be evaluated. It is a simple model that has a single fully connected hidden layer with the same number of neurons as input attributes (13). The network uses good practices such as the rectifier activation function for the hidden layer. No activation function is used for the output layer because it is a regression problem and we are interested in predicting numerical values directly without transform.

The efficient ADAM optimization algorithm is used and a mean squared error loss function is optimized. This will be the same metric that we will use to evaluate the performance of the model. It is a desirable metric because by taking the square root of an error value it gives us a result that we can directly understand in the context of the problem with the units in thousands of dollars.

```python
# define base mode
def baseline_model():
  # create model
  model = Sequential()
  model.add(Dense(13, input_dim=13, init='normal', activation='relu'))
  model.add(Dense(1, init='normal'))
  # Compile model
  model.compile(loss='mean_squared_error', optimizer='adam')
  return model
```

Listing 12.4: Define and Compile a Baseline Neural Network Model.

The Keras wrapper object for use in scikit-learn as a regression estimator is called `KerasRegressor`. We create an instance and pass it both the name of the function to create the neural network model as well as some parameters to pass along to the `fit()` function of the model later, such as the number of epochs and batch size. Both of these are set to sensible defaults. We also initialize the random number generator with a constant random seed, a process we will repeat for each model evaluated in this tutorial. This is to ensure we compare models consistently and that the results are reproducible.

```python
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

```
# evaluate model with standardized dataset
estimator = KerasRegressor(build_fn=baseline_model, nb_epoch=100, batch_size=5, verbose=0)
```

Listing 12.5: Initialize Random Number Generator and Prepare Model Wrapper for scikit-learn.

The final step is to evaluate this baseline model. We will use 10-fold cross validation to evaluate the model.

```
kfold = KFold(n=len(X), n_folds=10, random_state=seed)
results = cross_val_score(estimator, X, Y, cv=kfold)
print("Baseline: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 12.6: Evaluate Baseline Model.

The full code listing is provided below for completeness.

```
# Regression Example With Boston Dataset: Baseline
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = pandas.read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
# define base model
def baseline_model():
  # create model
  model = Sequential()
  model.add(Dense(13, input_dim=13, init='normal', activation='relu'))
  model.add(Dense(1, init='normal'))
  # Compile model
  model.compile(loss='mean_squared_error', optimizer='adam')
  return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# evaluate model with standardized dataset
estimator = KerasRegressor(build_fn=baseline_model, nb_epoch=100, batch_size=5, verbose=0)
kfold = KFold(n=len(X), n_folds=10, random_state=seed)
results = cross_val_score(estimator, X, Y, cv=kfold)
print("Baseline: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 12.7: Multilayer Perceptron Model for Boston House Problem.

Running this code gives us an estimate of the model's performance on the problem for unseen data. The result reports the mean squared error including the average and standard deviation (average variance) across all 10 folds of the cross validation evaluation.

```
Baseline: 38.04 (28.15) MSE
```

Listing 12.8: Sample Output From Evaluating the Baseline Model.

12.3   Lift Performance By Standardizing The Dataset

An important concern with the Boston house price dataset is that the input attributes all vary in their scales because they measure different quantities. It is almost always good practice to prepare your data before modeling it using a neural network model. Continuing on from the above baseline model, we can re-evaluate the same model using a standardized version of the input dataset.

We can use scikit-learn's `Pipeline` framework[3] to perform the standardization during the model evaluation process, within each fold of the cross validation. This ensures that there is no data leakage from each testset cross validation fold into the training data. The code below creates a scikit-learn `Pipeline` that first standardizes the dataset then creates and evaluates the baseline neural network model.

```python
# Regression Example With Boston Dataset: Standardized
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = pandas.read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
# define base model
def baseline_model():
  # create model
  model = Sequential()
  model.add(Dense(13, input_dim=13, init='normal', activation='relu'))
  model.add(Dense(1, init='normal'))
  # Compile model
  model.compile(loss='mean_squared_error', optimizer='adam')
  return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# evaluate model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=baseline_model, nb_epoch=50,
    batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
```

---

[3]http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

```
kfold = KFold(n=len(X), n_folds=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Standardized: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 12.9: Update To Use a Standardized Dataset.

Running the example provides an improved performance over the baseline model without standardized data, dropping the error by 10 thousand squared dollars.

```
Standardized: 28.24 (26.25) MSE
```

Listing 12.10: Sample Output From Evaluating the Model on The Standardized Dataset.

A further extension of this section would be to similarly apply a rescaling to the output variable such as normalizing it to the range of 0 to 1 and use a Sigmoid or similar activation function on the output layer to narrow output predictions to the same range.

12.4 Tune The Neural Network Topology

There are many concerns that can be optimized for a neural network model. Perhaps the point of biggest leverage is the structure of the network itself, including the number of layers and the number of neurons in each layer. In this section we will evaluate two additional network topologies in an effort to further improve the performance of the model. We will look at both a deeper and a wider network topology.

12.4.1 Evaluate a Deeper Network Topology

One way to improve the performance of a neural network is to add more layers. This might allow the model to extract and recombine higher order features embedded in the data. In this section we will evaluate the effect of adding one more hidden layer to the model. This is as easy as defining a new function that will create this deeper model, copied from our baseline model above. We can then insert a new line after the first hidden layer. In this case with about half the number of neurons. Our network topology now looks like:

```
13 inputs -> [13 -> 6] -> 1 output
```

Listing 12.11: Summary of Deeper Network Topology.

We can evaluate this network topology in the same way as above, whilst also using the standardization of the dataset that above was shown to improve performance.

```
# Regression Example With Boston Dataset: Standardized and Larger
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = pandas.read_csv("housing.csv", delim_whitespace=True, header=None)
```

```
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
# define the model
def larger_model():
  # create model
  model = Sequential()
  model.add(Dense(13, input_dim=13, init='normal', activation='relu'))
  model.add(Dense(6, init='normal', activation='relu'))
  model.add(Dense(1, init='normal'))
  # Compile model
  model.compile(loss='mean_squared_error', optimizer='adam')
  return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# evaluate model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=larger_model, nb_epoch=50, batch_size=5,
    verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n=len(X), n_folds=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Larger: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 12.12: Evaluate the Larger Neural Network Model.

Running this model does show a further improvement in performance from 28 down to 24 thousand squared dollars.

```
Larger: 24.60 (25.65) MSE
```

Listing 12.13: Sample Output From Evaluating the Deeper Model.

12.4.2   Evaluate a Wider Network Topology

Another approach to increasing the representational capacity of the model is to create a wider network. In this section we evaluate the effect of keeping a shallow network architecture and nearly doubling the number of neurons in the one hidden layer. Again, all we need to do is define a new function that creates our neural network model. Here, we have increased the number of neurons in the hidden layer compared to the baseline model from 13 to 20. The topology for our wider network can be summarized as follows:

```
13 inputs -> [20] -> 1 output
```

Listing 12.14: Summary of Wider Network Topology.

We can evaluate the wider network topology using the same scheme as above.

```
# Regression Example With Boston Dataset: Standardized and Wider
import numpy
import pandas
from keras.models import Sequential
```

```python
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.cross_validation import cross_val_score
from sklearn.cross_validation import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# load dataset
dataframe = pandas.read_csv("housing.csv", delim_whitespace=True, header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:13]
Y = dataset[:,13]
# define wider model
def wider_model():
	# create model
	model = Sequential()
	model.add(Dense(20, input_dim=13, init='normal', activation='relu'))
	model.add(Dense(1, init='normal'))
	# Compile model
	model.compile(loss='mean_squared_error', optimizer='adam')
	return model
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# evaluate model with standardized dataset
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasRegressor(build_fn=wider_model, nb_epoch=100, batch_size=5,
	verbose=0)))
pipeline = Pipeline(estimators)
kfold = KFold(n=len(X), n_folds=10, random_state=seed)
results = cross_val_score(pipeline, X, Y, cv=kfold)
print("Wider: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Listing 12.15: Evaluate the Wider Neural Network Model.

Building the model does see a further drop in error to about 21 thousand squared dollars. This is not a bad result for this problem.

```
Wider: 21.64 (23.75) MSE
```

Listing 12.16: Sample Output From Evaluating the Wider Model.

It would have been hard to guess that a wider network would outperform a deeper network on this problem. The results demonstrate the importance of empirical testing when it comes to developing neural network models.

12.5 Summary

In this lesson you discovered the Keras deep learning library for modeling regression problems. Through this tutorial you learned how to develop and evaluate neural network models, including:

- How to load data and develop a baseline model.

- How to lift performance using data preparation techniques like standardization.

- How to design and evaluate networks with different varying topologies on a problem.

12.5.1   Next

This concludes Part III of the book and leaves you with the skills to develop neural network models on standard machine learning datasets. Next in Part IV you will learn how to get more from your neural network models with some advanced techniques and use some of the more advanced features of the Keras library.

Part IV

Advanced Multilayer Perceptrons and Keras

Chapter 13

Save Your Models For Later With Serialization

Given that deep learning models can take hours, days and even weeks to train, it is important to know how to save and load them from disk. In this lesson you will discover how you can save your Keras models to file and load them up again to make predictions. After completing this lesson you will know:

- How to save and load Keras model weights to HDF5 formatted files.

- How to save and load Keras model structure to JSON files.

- How to save and load Keras model structure to YAML files.

Let's get started.

13.1 Tutorial Overview

Keras separates the concerns of saving your model architecture and saving your model weights. Model weights are saved to HDF5 format. This is a grid format that is ideal for storing multi-dimensional arrays of numbers. The model structure can be described and saved (and loaded) using two different formats: JSON and YAML.

Each example will also demonstrate saving and loading your model weights to HDF5 formatted files. The examples will use the same simple network trained on the Pima Indians onset of diabetes binary classification dataset (see Section 7.2).

13.1.1 HDF5 Format

The Hierarchical Data Format or HDF5 for short is a flexible data storage format and is convenient for storing large arrays of real values, as we have in the weights of neural networks. You may need to install Python support for the HDF5 file format. You can do this using your preferred Python package management system such as Pip:

```
sudo pip install h5py
```
Listing 13.1: Install Python Support For the HDF5 File Format via Pip.

13.2 Save Your Neural Network Model to JSON

JSON is a simple file format for describing data hierarchically. Keras provides the ability to describe any model using JSON format with a `to_json()` function. This can be saved to file and later loaded via the `model_from_json()` function that will create a new model from the JSON specification.

The weights are saved directly from the model using the `save_weights()` function and later loaded using the symmetrical `load_weights()` function. The example below trains and evaluates a simple model on the Pima Indians dataset. The model structure is then converted to JSON format and written to `model.json` in the local directory. The network weights are written to `model.h5` in the local directory.

The model and weight data is loaded from the saved files and a new model is created. It is important to compile the loaded model before it is used. This is so that predictions made using the model can use the appropriate efficient computation from the Keras backend. The model is evaluated in the same way printing the same evaluation score.

```python
# MLP for Pima Indians Dataset Serialize to JSON and HDF5
from keras.models import Sequential
from keras.layers import Dense
from keras.models import model_from_json
import numpy
import os
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10, verbose=0)
# evaluate the model
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

# serialize model to JSON
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")

# later...

# load json and create model
```

```python
json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")

# evaluate loaded model on test data
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
score = loaded_model.evaluate(X, Y, verbose=0)
print "%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100)
```

Listing 13.2: Serialize Model To JSON Format.

Running this example provides the output below. It shows first the accuracy of the trained model, the saving of the model to disk in JSON format, the loading of the model and finally the re-evaluation of the loaded model achieving the same accuracy.

```
acc: 79.56%
Saved model to disk
Loaded model from disk
acc: 79.56%
```

Listing 13.3: Sample Output From Serializing Model To JSON Format.

The JSON format of the model looks like the following:

```json
{
   "class_name":"Sequential",
   "config":[
     {
        "class_name":"Dense",
        "config":{
           "W_constraint":null,
           "b_constraint":null,
           "name":"dense_1",
           "output_dim":12,
           "activity_regularizer":null,
           "trainable":true,
           "init":"uniform",
           "input_dtype":"float32",
           "input_dim":8,
           "b_regularizer":null,
           "W_regularizer":null,
           "activation":"relu",
           "batch_input_shape":[
              null,
              8
           ]
        }
     },
     {
        "class_name":"Dense",
        "config":{
           "W_constraint":null,
           "b_constraint":null,
```

```
            "name":"dense_2",
            "activity_regularizer":null,
            "trainable":true,
            "init":"uniform",
            "input_dim":null,
            "b_regularizer":null,
            "W_regularizer":null,
            "activation":"relu",
            "output_dim":8
         }
      },
      {
         "class_name":"Dense",
         "config":{
            "W_constraint":null,
            "b_constraint":null,
            "name":"dense_3",
            "activity_regularizer":null,
            "trainable":true,
            "init":"uniform",
            "input_dim":null,
            "b_regularizer":null,
            "W_regularizer":null,
            "activation":"sigmoid",
            "output_dim":1
         }
      }
   ]
}
```

Listing 13.4: Sample JSON Model File.

13.3 Save Your Neural Network Model to YAML

This example is much the same as the above JSON example, except the YAML format is used
for the model specification. The model is described using YAML, saved to file `model.yaml` and
later loaded into a new model via the `model_from_yaml()` function. Weights are handled in the
same way as above in HDF5 format as `model.h5`.

```python
# MLP for Pima Indians Dataset serialize to YAML and HDF5
from keras.models import Sequential
from keras.layers import Dense
from keras.models import model_from_yaml
import numpy
import os
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
```

```python
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10, verbose=0)
# evaluate the model
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

# serialize model to YAML
model_yaml = model.to_yaml()
with open("model.yaml", "w") as yaml_file:
    yaml_file.write(model_yaml)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")

# later...

# load YAML and create model
yaml_file = open('model.yaml', 'r')
loaded_model_yaml = yaml_file.read()
yaml_file.close()
loaded_model = model_from_yaml(loaded_model_yaml)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")

# evaluate loaded model on test data
loaded_model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
score = loaded_model.evaluate(X, Y, verbose=0)
print "%s: %.2f%%" % (loaded_model.metrics_names[1], score[1]*100)
```

Listing 13.5: Serialize Model To YAML Format.

Running the example displays the output below. Again, the example demonstrates the accuracy of the model, the model serialization, deserialization and re-evaluation achieving the same results.

```
acc: 79.56%
Saved model to disk
Loaded model from disk
acc: 79.56%
```

Listing 13.6: Sample Output From Serializing Model To YAML Format.

The model described in YAML format looks like the following:

```
class_name: Sequential
config:
- class_name: Dense
  config:
    W_constraint: null
    W_regularizer: null
```

```
      activation: relu
      activity_regularizer: null
      b_constraint: null
      b_regularizer: null
      batch_input_shape: !!python/tuple [null, 8]
      init: uniform
      input_dim: 8
      input_dtype: float32
      name: dense_1
      output_dim: 12
      trainable: true
- class_name: Dense
  config: {W_constraint: null, W_regularizer: null, activation: relu, activity_regularizer:
      null,
    b_constraint: null, b_regularizer: null, init: uniform, input_dim: null, name: dense_2,
    output_dim: 8, trainable: true}
- class_name: Dense
  config: {W_constraint: null, W_regularizer: null, activation: sigmoid,
      activity_regularizer: null,
    b_constraint: null, b_regularizer: null, init: uniform, input_dim: null, name: dense_3,
    output_dim: 1, trainable: true}
```

Listing 13.7: Sample YAML Model File.

13.4 Summary

Saving and loading models is an important capability for transplanting a deep learning model from research and development to operations. In this lesson you discovered how to serialize your Keras deep learning models. You learned:

- How to save model weights to HDF5 formatted files and load them again later.

- How to save Keras model definitions to JSON files and load them again.

- How to save Keras model definitions to YAML files and load them again.

13.4.1 Next

You now know how to serialize your deep learning models in Keras. Next you will discover the importance of checkpointing your models during long training periods and how to load those checkpointed models in order to make predictions.

Chapter 14

Keep The Best Models During Training With Checkpointing

Deep learning models can take hours, days or even weeks to train and if a training run is stopped unexpectedly, you can lose a lot of work. In this lesson you will discover how you can checkpoint your deep learning models during training in Python using the Keras library. After completing this lesson you will know:

- The importance of checkpointing neural network models when training.

- How to checkpoint each improvement to a model during training.

- How to checkpoint the very best model observed during training.

Let's get started.

14.1   Checkpointing Neural Network Models

Application checkpointing is a fault tolerance technique for long running processes. It is an approach where a snapshot of the state of the system is taken in case of system failure. If there is a problem, not all is lost. The checkpoint may be used directly, or used as the starting point for a new run, picking up where it left off. When training deep learning models, the checkpoint captures the weights of the model. These weights can be used to make predictions as-is, or used as the basis for ongoing training.

The Keras library provides a checkpointing capability by a callback API. The `ModelCheckpoint` callback class allows you to define where to checkpoint the model weights, how the file should be named and under what circumstances to make a checkpoint of the model. The API allows you to specify which metric to monitor, such as loss or accuracy on the training or validation dataset. You can specify whether to look for an improvement in maximizing or minimizing the score. Finally, the filename that you use to store the weights can include variables like the epoch number or metric. The `ModelCheckpoint` instance can then be passed to the training process when calling the `fit()` function on the model. Note, you may need to install the `h5py` library (see Section 13.1.1).

14.2 Checkpoint Neural Network Model Improvements

A good use of checkpointing is to output the model weights each time an improvement is observed during training. The example below creates a small neural network for the Pima Indians onset of diabetes binary classification problem (see Section 7.2). The example uses 33% of the data for validation.

Checkpointing is setup to save the network weights only when there is an improvement in classification accuracy on the validation dataset (`monitor='val_acc'` and `mode='max'`). The weights are stored in a file that includes the score in the filename `weights-improvement-val_acc=.2f.hdf5`.

```python
# Checkpoint the weights when validation accuracy improves
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint
import matplotlib.pyplot as plt
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# checkpoint
filepath="weights-improvement-{epoch:02d}-{val_acc:.2f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_acc', verbose=1, save_best_only=True,
    mode='max')
callbacks_list = [checkpoint]
# Fit the model
model.fit(X, Y, validation_split=0.33, nb_epoch=150, batch_size=10,
    callbacks=callbacks_list, verbose=0)
```

Listing 14.1: Checkpoint Model Improvements.

Running the example produces the output below, truncated for brevity. In the output you can see cases where an improvement in the model accuracy on the validation dataset resulted in a new weight file being written to disk.

```
...
Epoch 00134: val_acc did not improve
Epoch 00135: val_acc did not improve
Epoch 00136: val_acc did not improve
Epoch 00137: val_acc did not improve
Epoch 00138: val_acc did not improve
Epoch 00139: val_acc did not improve
```

```
Epoch 00140: val_acc improved from 0.83465 to 0.83858, saving model to
    weights-improvement-140-0.84.hdf5
Epoch 00141: val_acc did not improve
Epoch 00142: val_acc did not improve
Epoch 00143: val_acc did not improve
Epoch 00144: val_acc did not improve
Epoch 00145: val_acc did not improve
Epoch 00146: val_acc improved from 0.83858 to 0.84252, saving model to
    weights-improvement-146-0.84.hdf5
Epoch 00147: val_acc did not improve
Epoch 00148: val_acc improved from 0.84252 to 0.84252, saving model to
    weights-improvement-148-0.84.hdf5
Epoch 00149: val_acc did not improve
```

Listing 14.2: Sample Output From Checkpoint Model Improvements.

You will also see a number of files in your working directory containing the network weights in HDF5 format. For example:

```
...
weights-improvement-74-0.81.hdf5
weights-improvement-81-0.82.hdf5
weights-improvement-91-0.82.hdf5
weights-improvement-93-0.83.hdf5
```

Listing 14.3: Sample Model Checkpoint Files.

This is a very simple checkpointing strategy. It may create a lot of unnecessary checkpoint files if the validation accuracy moves up and down over training epochs. Nevertheless, it will ensure that you have a snapshot of the best model discovered during your run.

14.3   Checkpoint Best Neural Network Model Only

A simpler checkpoint strategy is to save the model weights to the same file, if and only if the validation accuracy improves. This can be done easily using the same code from above and changing the output filename to be fixed (not include score or epoch information). In this case, model weights are written to the file `weights.best.hdf5` only if the classification accuracy of the model on the validation dataset improves over the best seen so far.

```
# Checkpoint the weights for best model on validation accuracy
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint
import matplotlib.pyplot as plt
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
```

```
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# checkpoint
filepath="weights.best.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='val_acc', verbose=1, save_best_only=True,
    mode='max')
callbacks_list = [checkpoint]
# Fit the model
model.fit(X, Y, validation_split=0.33, nb_epoch=150, batch_size=10,
    callbacks=callbacks_list, verbose=0)
```

Listing 14.4: Checkpoint Best Model Only.

Running this example provides the following output (truncated for brevity):

```
...
Epoch 00136: val_acc did not improve
Epoch 00137: val_acc did not improve
Epoch 00138: val_acc did not improve
Epoch 00139: val_acc did not improve
Epoch 00140: val_acc improved from 0.83465 to 0.83858, saving model to weights.best.hdf5
Epoch 00141: val_acc did not improve
Epoch 00142: val_acc did not improve
Epoch 00143: val_acc did not improve
Epoch 00144: val_acc did not improve
Epoch 00145: val_acc did not improve
Epoch 00146: val_acc improved from 0.83858 to 0.84252, saving model to weights.best.hdf5
Epoch 00147: val_acc did not improve
Epoch 00148: val_acc improved from 0.84252 to 0.84252, saving model to weights.best.hdf5
Epoch 00149: val_acc did not improve
```

Listing 14.5: Sample Output From Checkpoint The Best Model.

You should see the weight file in your local directory.

```
weights.best.hdf5
```

Listing 14.6: Sample Best Model Checkpoint File.

14.4   Loading a Saved Neural Network Model

Now that you have seen how to checkpoint your deep learning models during training, you need to review how to load and use a checkpointed model. The checkpoint only includes the model weights. It assumes you know the network structure. This too can be serialize to file in JSON or YAML format. In the example below, the model structure is known and the best weights are loaded from the previous experiment, stored in the working directory in the `weights.best.hdf5` file. The model is then used to make predictions on the entire dataset.

```
# How to load and use weights from a checkpoint
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint
```

```python
import matplotlib.pyplot as plt
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# load weights
model.load_weights("weights.best.hdf5")
# Compile model (required to make predictions)
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print("Created model and loaded weights from file")
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# estimate accuracy on whole dataset using loaded weights
scores = model.evaluate(X, Y, verbose=0)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Listing 14.7: Load and Evaluate a Model Checkpoint.

Running the example produces the following output:

```
Created model and loaded weights from file
acc: 77.73%
```

Listing 14.8: Sample Output From Loading and Evaluating a Model Checkpoint.

14.5 Summary

In this lesson you have discovered the importance of checkpointing deep learning models for long training runs. You learned:

- How to use Keras to checkpoint each time an improvement to the model is observed.

- How to only checkpoint the very best model observed during training.

- How to load a checkpointed model from file and use it later to make predictions.

14.5.1 Next

You now know how to checkpoint your deep learning models in Keras during long training schemes. In the next lesson you will discover how to collect, inspect and plot metrics collected about your model during training.

Chapter 15

Understand Model Behavior During Training By Plotting History

You can learn a lot about neural networks and deep learning models by observing their performance over time during training. In this lesson you will discover how you can review and visualize the performance of deep learning models over time during training in Python with Keras. After completing this lesson you will know:

- How to inspect the history metrics collected during training.

- How to plot accuracy metrics on training and validation datasets during training.

- How to plot model loss metrics on training and validation datasets during training.

Let's get started.

15.1   Access Model Training History in Keras

Keras provides the capability to register callbacks when training a deep learning model. One of the default callbacks that is registered when training all deep learning models is the `History` callback. It records training metrics for each epoch. This includes the loss and the accuracy (for classification problems) as well as the loss and accuracy for the validation dataset, if one is set.

The history object is returned from calls to the `fit()` function used to train the model. Metrics are stored in a dictionary in the `history` member of the object returned. For example, you can list the metrics collected in a history object using the following snippet of code after a model is trained:

```python
# list all data in history
print(history.history.keys())
```

Listing 15.1: Output Recorded History Metric Names.

For example, for a model trained on a classification problem with a validation dataset, this might produce the following listing:

```python
['acc', 'loss', 'val_acc', 'val_loss']
```

Listing 15.2: Sample Output From Recorded History Metric Names.

We can use the data collected in the history object to create plots. The plots can provide an indication of useful things about the training of the model, such as:

- It's speed of convergence over epochs (slope).

- Whether the model may have already converged (plateau of the line).

- Whether the model may be over-learning the training data (inflection for validation line).

And more.

15.2   Visualize Model Training History in Keras

We can create plots from the collected history data. In the example below we create a small network to model the Pima Indians onset of diabetes binary classification problem (see Section 7.2). The example collects the history, returned from training the model and creates two charts:

1. A plot of accuracy on the training and validation datasets over training epochs.

2. A plot of loss on the training and validation datasets over training epochs.

```
# Visualize training history
from keras.models import Sequential
from keras.layers import Dense
import matplotlib.pyplot as plt
import numpy
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
history = model.fit(X, Y, validation_split=0.33, nb_epoch=150, batch_size=10, verbose=0)
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
```

```
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

Listing 15.3: Evaluate a Model and Plot Training History.

The plots are provided below. The history for the validation dataset is labeled test by convention as it is indeed a test dataset for the model. From the plot of accuracy we can see that the model could probably be trained a little more as the trend for accuracy on both datasets is still rising for the last few epochs. We can also see that the model has not yet over-learned the training dataset, showing comparable skill on both datasets.
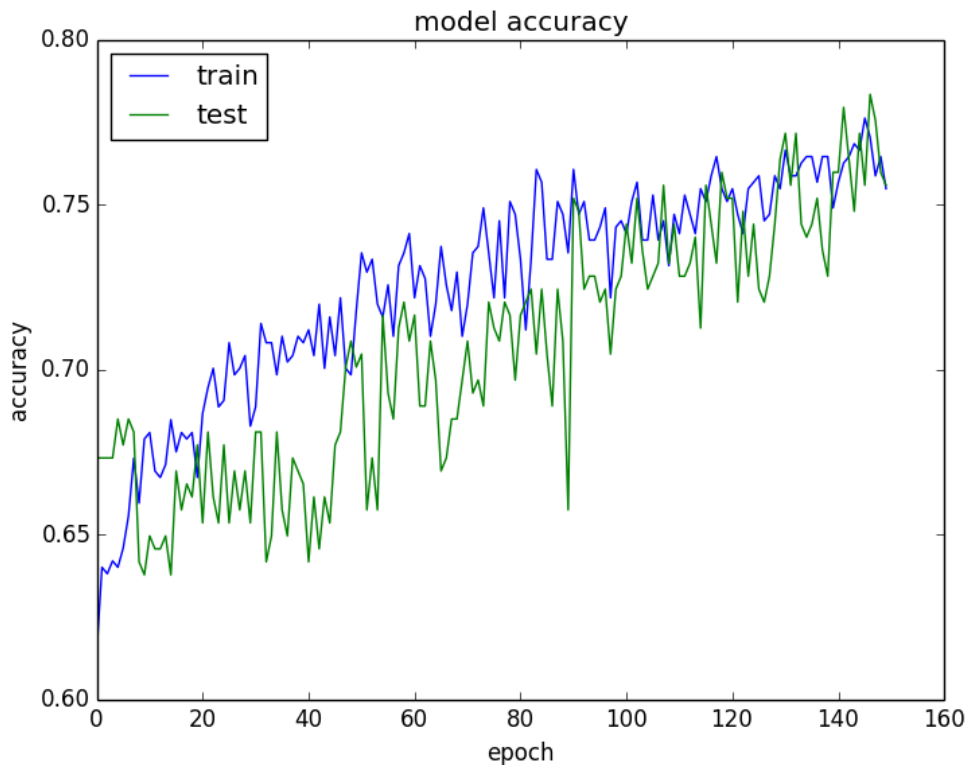
Figure 15.1: Plot of Model Accuracy on Train and Validation Datasets

From the plot of loss, we can see that the model has comparable performance on both train and validation datasets (labeled test). If these parallel plots start to depart consistently, it might be a sign to stop training at an earlier epoch.
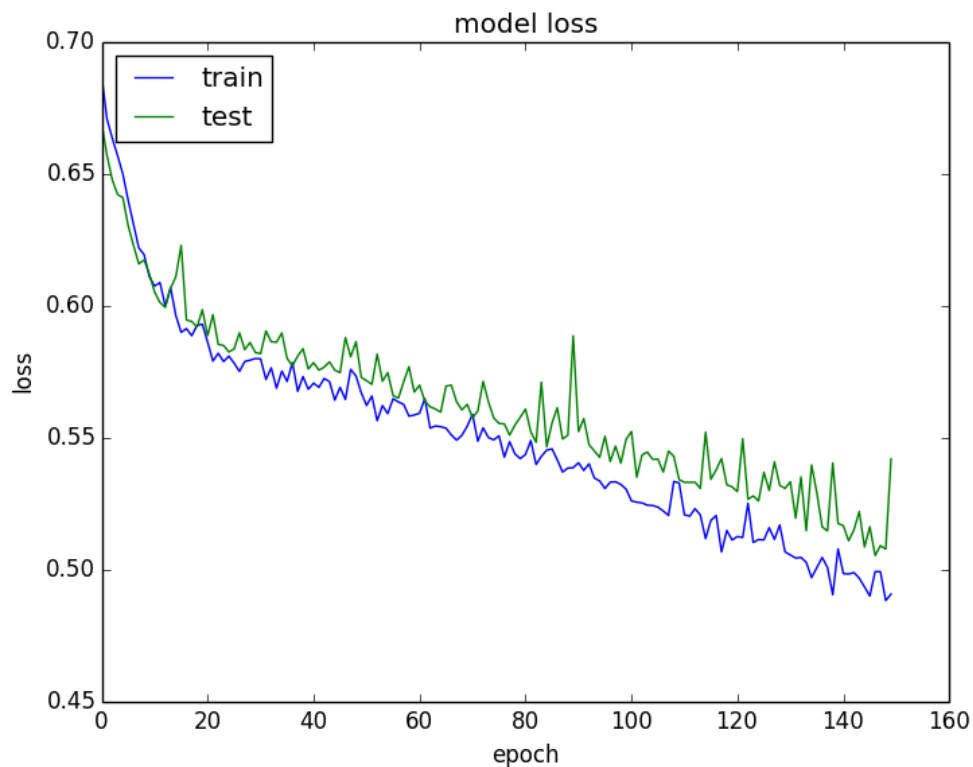
Figure 15.2: Plot of Model Loss on Training and Validation Datasets

15.3 Summary

In this lesson you discovered the importance of collecting and reviewing metrics during the training of your deep learning models. You learned:

- How to inspect a history object returned from training to discover the metrics that were collected.

- How to extract model accuracy information for training and validation datasets and plot the data.

- How to extract and plot the model loss information calculated from training and validation datasets.

15.3.1 Next

A simple yet very powerful technique for decreasing the amount of overfitting of your model to training data is called dropout. In the next lesson you will discover the dropout technique, how to apply it to visible and hidden layers in Keras and best practices for using it on your own problems.

Chapter 16

Reduce Overfitting With Dropout Regularization

A simple and powerful regularization technique for neural networks and deep learning models is dropout. In this lesson you will discover the dropout regularization technique and how to apply it to your models in Python with Keras. After completing this lesson you will know:

- How the dropout regularization technique works.

- How to use dropout on your input and hidden layers.

- How to use dropout on your hidden layers.

Let's get started.

16.1   Dropout Regularization For Neural Networks

Dropout is a regularization technique for neural network models proposed by Srivastava, et al. in their 2014 paper Dropout: A Simple Way to Prevent Neural Networks from Overfitting[1]. Dropout is a technique where randomly selected neurons are ignored during training. They are *dropped-out* randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

As a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features providing some specialization. Neighboring neurons become to rely on this specialization, which if taken too far can result in a fragile model too specialized to the training data. This reliant on context for a neuron during training is referred to as *complex co-adaptations*. You can imagine that if neurons are randomly dropped out of the network during training, that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.

The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to overfit the training data.

---

[1]http://jmlr.org/papers/v15/srivastava14a.html

16.2   Dropout Regularization in Keras

Dropout is easily implemented by randomly selecting nodes to be dropped-out with a given probability (e.g. 20%) each weight update cycle. This is how Dropout is implemented in Keras. Dropout is only used during the training of a model and is not used when evaluating the skill of the model. Next we will explore a few different ways of using Dropout in Keras.

The examples will use the Sonar dataset binary classification dataset (learn more in Section 11.1). We will evaluate the developed models using scikit-learn with 10-fold cross validation, in order to better tease out differences in the results. There are 60 input values and a single output value and the input values are standardized before being used in the network. The baseline neural network model has two hidden layers, the first with 60 units and the second with 30. Stochastic gradient descent is used to train the model with a relatively low learning rate and momentum. The full baseline model is listed below.

```python
# Baseline Model on the Sonar Dataset
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.wrappers.scikit_learn import KerasClassifier
from keras.constraints import maxnorm
from keras.optimizers import SGD
from sklearn.cross_validation import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.cross_validation import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.grid_search import GridSearchCV
from sklearn.pipeline import Pipeline
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)

# baseline
def create_baseline():
  # create model
  model = Sequential()
  model.add(Dense(60, input_dim=60, init='normal', activation='relu'))
  model.add(Dense(30, init='normal', activation='relu'))
  model.add(Dense(1, init='normal', activation='sigmoid'))
  # Compile model
  sgd = SGD(lr=0.01, momentum=0.8, decay=0.0, nesterov=False)
  model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
  return model
```

```
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_baseline, nb_epoch=300,
    batch_size=16, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, random_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Baseline: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 16.1: Baseline Neural Network For The Sonar Dataset.

Running the example for the baseline model without drop-out generates an estimated classification accuracy of 82%.

```
Baseline: 82.68% (3.90%)
```

Listing 16.2: Sample Output From Baseline Neural Network For The Sonar Dataset.

16.3 Using Dropout on the Visible Layer

Dropout can be applied to input neurons called the visible layer. In the example below we add a new Dropout layer between the input (or visible layer) and the first hidden layer. The dropout rate is set to 20%, meaning one in five inputs will be randomly excluded from each update cycle.

Additionally, as recommended in the original paper on dropout, a constraint is imposed on the weights for each hidden layer, ensuring that the maximum norm of the weights does not exceed a value of 3. This is done by setting the `W_constraint` argument on the `Dense` class when constructing the layers. The learning rate was lifted by one order of magnitude and the momentum was increased to 0.9. These increases in the learning rate were also recommended in the original dropout paper. Continuing on from the baseline example above, the code below exercises the same network with input dropout.

```
# Example of Dropout on the Sonar Dataset: Visible Layer
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.wrappers.scikit_learn import KerasClassifier
from keras.constraints import maxnorm
from keras.optimizers import SGD
from sklearn.cross_validation import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.cross_validation import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.grid_search import GridSearchCV
from sklearn.pipeline import Pipeline
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("sonar.csv", header=None)
```

```
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)

# dropout in the input layer with weight constraint
def create_model():
  # create model
  model = Sequential()
  model.add(Dropout(0.2, input_shape=(60,)))
  model.add(Dense(60, init='normal', activation='relu', W_constraint=maxnorm(3)))
  model.add(Dense(30, init='normal', activation='relu', W_constraint=maxnorm(3)))
  model.add(Dense(1, init='normal', activation='sigmoid'))
  # Compile model
  sgd = SGD(lr=0.1, momentum=0.9, decay=0.0, nesterov=False)
  model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
  return model

numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_model, nb_epoch=300,
    batch_size=16, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, random_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Visible: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 16.3: Example of Using Dropout on the Visible Layer.

Running the example with dropout in the visible layer provides a nice lift in classification accuracy to 86%.

```
Visible: 86.04% (6.33%)
```

Listing 16.4: Sample Output From Example of Using Dropout on the Visible Layer.

16.4 Using Dropout on Hidden Layers

Dropout can be applied to hidden neurons in the body of your network model. In the example below dropout is applied between the two hidden layers and between the last hidden layer and the output layer. Again a dropout rate of 20% is used as is a weight constraint on those layers.

```
# Example of Dropout on the Sonar Dataset: Hidden Layer
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.wrappers.scikit_learn import KerasClassifier
from keras.constraints import maxnorm
```

```python
from keras.optimizers import SGD
from sklearn.cross_validation import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.cross_validation import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.grid_search import GridSearchCV
from sklearn.pipeline import Pipeline
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)

# dropout in hidden layers with weight constraint
def create_model():
	# create model
	model = Sequential()
	model.add(Dense(60, input_dim=60, init='normal', activation='relu',
		W_constraint=maxnorm(3)))
	model.add(Dropout(0.2))
	model.add(Dense(30, init='normal', activation='relu', W_constraint=maxnorm(3)))
	model.add(Dropout(0.2))
	model.add(Dense(1, init='normal', activation='sigmoid'))
	# Compile model
	sgd = SGD(lr=0.1, momentum=0.9, decay=0.0, nesterov=False)
	model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
	return model

numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_model, nb_epoch=300,
	batch_size=16, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(y=encoded_Y, n_folds=10, shuffle=True, random_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Hidden: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Listing 16.5: Example of Using Dropout on Hidden Layers.

We can see that for this problem and for the chosen network configuration that using dropout in the hidden layers did not lift performance. In fact, performance was worse than the baseline. It is possible that additional training epochs are required or that further tuning is required to the learning rate.

```
Hidden: 83.09% (7.63%)
```

Listing 16.6: Sample Output From Example of Using Dropout on the Hidden Layers.

16.5 Tips For Using Dropout

The original paper on Dropout provides experimental results on a suite of standard machine learning problems. As a result they provide a number of useful heuristics to consider when using dropout in practice:

- Generally use a small dropout value of 20%-50% of neurons with 20% providing a good starting point. A probability too low has minimal effect and a value too high results in under-learning by the network.

- Use a larger network. You are likely to get better performance when dropout is used on a larger network, giving the model more of an opportunity to learn independent representations.

- Use dropout on input (visible) as well as hidden layers. Application of dropout at each layer of the network has shown good results.

- Use a large learning rate with decay and a large momentum. Increase your learning rate by a factor of 10 to 100 and use a high momentum value of 0.9 or 0.99.

- Constrain the size of network weights. A large learning rate can result in very large network weights. Imposing a constraint on the size of network weights such as max-norm regularization with a size of 4 or 5 has been shown to improve results.

16.6 Summary

In this lesson you discovered the dropout regularization technique for deep learning models. You learned:

- What dropout is and how it works.

- How you can use dropout on your own deep learning models.

- Tips for getting the best results from dropout on your own models.

16.6.1 Next

Another important technique for improving the performance of your neural network models is to adapt the learning rate during training. In the next lesson you will discover different learning rate schedules and how you can apply them with Keras to your own problems.

Chapter 17

Lift Performance With Learning Rate Schedules

Training a neural network or large deep learning model is a difficult optimization task. The classical algorithm to train neural networks is called stochastic gradient descent. It has been well established that you can achieve increased performance and faster training on some problems by using a learning rate that changes during training. In this lesson you will discover how you can use different learning rate schedules for your neural network models in Python using the Keras deep learning library. After completing this lesson you will know:

- The benefit of learning rate schedules on lifting model performance during training.

- How to configure and evaluate a time-based learning rate schedule.

- How to configure and evaluate a drop-based learning rate schedule.

Let's get started.

17.1   Learning Rate Schedule For Training Models

Adapting the learning rate for your stochastic gradient descent optimization procedure can increase performance and reduce training time. Sometimes this is called learning rate annealing or adaptive learning rates. Here we will call this approach a learning rate schedule, were the default schedule is to use a constant learning rate to update network weights for each training epoch.

The simplest and perhaps most used adaptation of learning rates during training are techniques that reduce the learning rate over time. These have the benefit of making large changes at the beginning of the training procedure when larger learning rate values are used, and decreasing the learning rate such that a smaller rate and therefore smaller training updates are made to weights later in the training procedure. This has the effect of quickly learning good weights early and fine tuning them later. Two popular and easy to use learning rate schedules are as follows:

- Decrease the learning rate gradually based on the epoch.

- Decrease the learning rate using punctuated large drops at specific epochs.

Next, we will look at how you can use each of these learning rate schedules in turn with Keras.

17.2 Ionosphere Classification Dataset

The Ionosphere binary classification problem is used as a demonstration in this lesson. The dataset describes radar returns where the target was free electrons in the ionosphere. It is a binary classification problem where positive cases (`g` for good) show evidence of some type of structure in the ionosphere and negative cases (`b` for bad) do not. It is a good dataset for practicing with neural networks because all of the inputs are small numerical values of the same scale. There are 34 attributes and 351 observations.

State-of-the-art results on this dataset achieve an accuracy of approximately 94% to 98% accuracy using 10-fold cross validation[1]. The dataset is available within the code bundle provided with this book. Alternatively, you can download it directly from the UCI Machine Learning repository[2]. Place the data file in your working directory with the filename `ionosphere.csv`. You can learn more about the ionosphere dataset on the UCI Machine Learning Repository website[3].

17.3 Time-Based Learning Rate Schedule

Keras has a time-based learning rate schedule built in. The stochastic gradient descent optimization algorithm implementation in the `SGD` class has an argument called decay. This argument is used in the time-based learning rate decay schedule equation as follows:

$$LearningRate = LearningRate \times \frac{1}{1 + decay \times epoch}$$

Figure 17.1: Calculate Learning Rate For Time-Based Decay.

When the decay argument is zero (the default), this has no effect on the learning rate (e.g. 0.1).

```
LearningRate = 0.1 * 1/(1 + 0.0 * 1)
LearningRate = 0.1
```

Listing 17.1: Example Calculating Learning Rate Without Decay.

When the decay argument is specified, it will decrease the learning rate from the previous epoch by the given fixed amount. For example, if we use the initial learning rate value of 0.1 and the decay of 0.001, the first 5 epochs will adapt the learning rate as follows:

```
Epoch Learning Rate
1   0.1
```

---

[1]http://www.is.umk.pl/projects/datasets.html#Ionosphere
[2]http://archive.ics.uci.edu/ml/machine-learning-databases/ionosphere/ionosphere.data
[3]https://archive.ics.uci.edu/ml/datasets/Ionosphere

```
2   0.0999000999
3   0.0997006985
4   0.09940249103
5   0.09900646517
```

Listing 17.2: Output of Calculating Learning Rate With Decay.

Extending this out to 100 epochs will produce the following graph of learning rate (y-axis) versus epoch (x-axis):
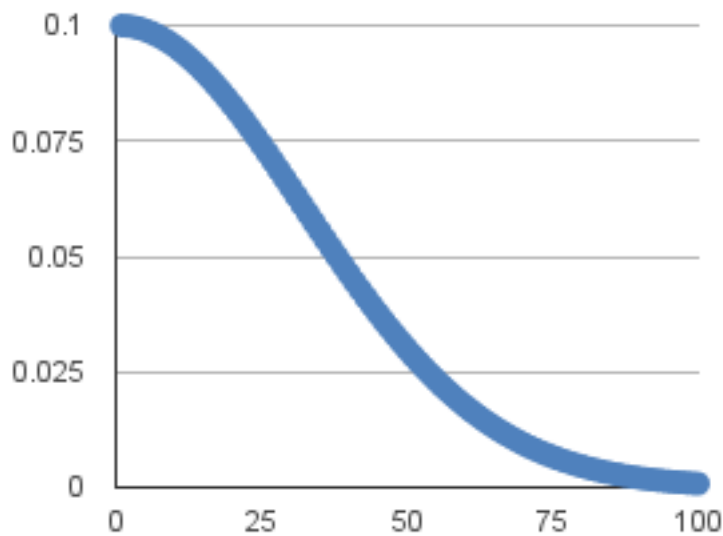
Figure 17.2: Time-Based Learning Rate Schedule

You can create a nice default schedule by setting the decay value as follows:

```
Decay = LearningRate / Epochs
Decay = 0.1 / 100
Decay = 0.001
```

Listing 17.3: Example of A Good Default Decay Rate.

The example below demonstrates using the time-based learning rate adaptation schedule in Keras. A small neural network model is constructed with a single hidden layer with 34 neurons and using the rectifier activation function. The output layer has a single neuron and uses the sigmoid activation function in order to output probability-like values. The learning rate for stochastic gradient descent has been set to a higher value of 0.1. The model is trained for 50 epochs and the decay argument has been set to 0.002, calculated as $\frac{0.1}{50}$. Additionally, it can be a good idea to use momentum when using an adaptive learning rate. In this case we use a momentum value of 0.8. The complete example is listed below.

```
# Time Based Learning Rate Decay
```

```python
import pandas
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from sklearn.preprocessing import LabelEncoder
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("ionosphere.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:34].astype(float)
Y = dataset[:,34]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
Y = encoder.transform(Y)
# create model
model = Sequential()
model.add(Dense(34, input_dim=34, init='normal', activation='relu'))
model.add(Dense(1, init='normal', activation='sigmoid'))
# Compile model
epochs = 50
learning_rate = 0.1
decay_rate = learning_rate / epochs
momentum = 0.8
sgd = SGD(lr=learning_rate, momentum=momentum, decay=decay_rate, nesterov=False)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
# Fit the model
model.fit(X, Y, validation_split=0.33, nb_epoch=epochs, batch_size=28, verbose=2)
```

Listing 17.4: Example of Time-Based Learning Rate Decay.

The model is trained on 67% of the dataset and evaluated using a 33% validation dataset. Running the example shows a classification accuracy of 99.14%. This is higher than the baseline of 95.69% without the learning rate decay or momentum.

```
Epoch 46/50
0s - loss: 0.0570 - acc: 0.9830 - val_loss: 0.0867 - val_acc: 0.9914
Epoch 47/50
0s - loss: 0.0584 - acc: 0.9830 - val_loss: 0.0808 - val_acc: 0.9914
Epoch 48/50
0s - loss: 0.0610 - acc: 0.9872 - val_loss: 0.0653 - val_acc: 0.9828
Epoch 49/50
0s - loss: 0.0591 - acc: 0.9830 - val_loss: 0.0821 - val_acc: 0.9914
Epoch 50/50
0s - loss: 0.0598 - acc: 0.9872 - val_loss: 0.0739 - val_acc: 0.9914
```

Listing 17.5: Sample Output of Time-Based Learning Rate Decay.

17.4 Drop-Based Learning Rate Schedule

Another popular learning rate schedule used with deep learning models is to systematically drop the learning rate at specific times during training. Often this method is implemented by dropping the learning rate by half every fixed number of epochs. For example, we may have an initial learning rate of 0.1 and drop it by a factor of 0.5 every 10 epochs. The first 10 epochs of training would use a value of 0.1, in the next 10 epochs a learning rate of 0.05 would be used, and so on. If we plot out the learning rates for this example out to 100 epochs you get the graph below showing learning rate (*y*-axis) versus epoch (*x*-axis).
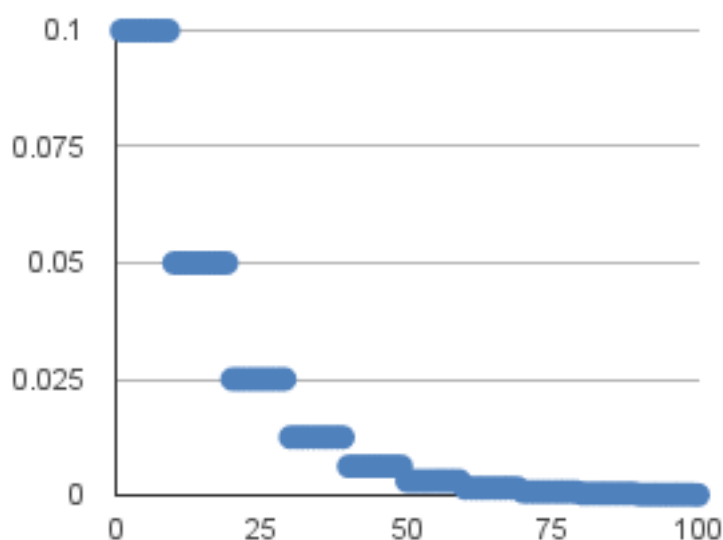
Figure 17.3: Drop Based Learning Rate Schedule

We can implement this in Keras using a the `LearningRateScheduler` callback[4] when fitting the model. The `LearningRateScheduler` callback allows us to define a function to call that takes the epoch number as an argument and returns the learning rate to use in stochastic gradient descent. When used, the learning rate specified by stochastic gradient descent is ignored. In the code below, we use the same example before of a single hidden layer network on the Ionosphere dataset. A new `step_decay()` function is defined that implements the equation:

$$LearningRate = InitialLearningRate \times DropRate^{floor(\frac{1+Epoch}{EpochDrop})}$$

Figure 17.4: Calculate Learning Rate Using a Drop Schedule.

---

[4]http://keras.io/callbacks/

Where `InitialLearningRate` is the learning rate at the beginning of the run, `EpochDrop` is how often the learning rate is dropped in epochs and `DropRate` is how much to drop the learning rate each time it is dropped.

```python
# Drop-Based Learning Rate Decay
import pandas
import pandas
import numpy
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from sklearn.preprocessing import LabelEncoder
from keras.callbacks import LearningRateScheduler

# learning rate schedule
def step_decay(epoch):
  initial_lrate = 0.1
  drop = 0.5
  epochs_drop = 10.0
  lrate = initial_lrate * math.pow(drop, math.floor((1+epoch)/epochs_drop))
  return lrate

# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load dataset
dataframe = pandas.read_csv("ionosphere.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:34].astype(float)
Y = dataset[:,34]
# encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
Y = encoder.transform(Y)
# create model
model = Sequential()
model.add(Dense(34, input_dim=34, init='normal', activation='relu'))
model.add(Dense(1, init='normal', activation='sigmoid'))
# Compile model
sgd = SGD(lr=0.0, momentum=0.9, decay=0.0, nesterov=False)
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
# learning schedule callback
lrate = LearningRateScheduler(step_decay)
callbacks_list = [lrate]
# Fit the model
model.fit(X, Y, validation_split=0.33, nb_epoch=50, batch_size=28,
    callbacks=callbacks_list, verbose=2)
```

Listing 17.6: Example of Drop-Based Learning Rate Decay.

Running the example results in a classification accuracy of 99.14% on the validation dataset, again an improvement over the baseline for the model on this dataset.

```
0s - loss: 0.0546 - acc: 0.9830 - val_loss: 0.0705 - val_acc: 0.9914
Epoch 46/50
```

```
0s - loss: 0.0542 - acc: 0.9830 - val_loss: 0.0676 - val_acc: 0.9914
Epoch 47/50
0s - loss: 0.0538 - acc: 0.9830 - val_loss: 0.0668 - val_acc: 0.9914
Epoch 48/50
0s - loss: 0.0539 - acc: 0.9830 - val_loss: 0.0708 - val_acc: 0.9914
Epoch 49/50
0s - loss: 0.0539 - acc: 0.9830 - val_loss: 0.0674 - val_acc: 0.9914
Epoch 50/50
0s - loss: 0.0531 - acc: 0.9830 - val_loss: 0.0694 - val_acc: 0.9914
```

Listing 17.7: Sample Output of Time-Based Learning Rate Decay.

17.5 Tips for Using Learning Rate Schedules

This section lists some tips and tricks to consider when using learning rate schedules with neural networks.

- **Increase the initial learning rate**. Because the learning rate will decrease, start with a larger value to decrease from. A larger learning rate will result in a lot larger changes to the weights, at least in the beginning, allowing you to benefit from fine tuning later.

- **Use a large momentum**. Using a larger momentum value will help the optimization algorithm to continue to make updates in the right direction when your learning rate shrinks to small values.

- **Experiment with different schedules**. It will not be clear which learning rate schedule to use so try a few with different configuration options and see what works best on your problem. Also try schedules that change exponentially and even schedules that respond to the accuracy of your model on the training or test datasets.

17.6 Summary

In this lesson you discovered learning rate schedules for training neural network models. You learned:

- The benefits of using learning rate schedules during training to lift model performance.

- How to configure and use a time-based learning rate schedule in Keras.

- How to develop your own drop-based learning rate schedule in Keras.

17.6.1 Next

This concludes the lessons for Part IV. Now you know how to use more advanced features of Keras and more advanced techniques to get improved performance from your neural network models. Next, in Part V, you will discover a new type of model called the convolutional neural network that is achieving state-of-the-art results in computer vision and natural language processing problems.

Part V

Convolutional Neural Networks

Chapter 18

Crash Course In Convolutional Neural Networks

Convolutional Neural Networks are a powerful artificial neural network technique. These networks preserve the spatial structure of the problem and were developed for object recognition tasks such as handwritten digit recognition. They are popular because people are achieving state-of-the-art results on difficult computer vision and natural language processing tasks. In this lesson you will discover Convolutional Neural Networks for deep learning, also called ConvNets or CNNs. After completing this crash course you will know:

- The building blocks used in CNNs such as convolutional layers and pool layers.

- How the building blocks fit together with a short worked example.

- Best practices for configuring CNNs on your own object recognition tasks.

Let's get started.

18.1   The Case for Convolutional Neural Networks

Given a dataset of gray scale images with the standardized size of 32×32 pixels each, a traditional feedforward neural network would require 1,024 input weights (plus one bias). This is fair enough, but the flattening of the image matrix of pixels to a long vector of pixel values looses all of the spatial structure in the image. Unless all of the images are perfectly resized, the neural network will have great difficulty with the problem.

Convolutional Neural Networks expect and preserve the spatial relationship between pixels by learning internal feature representations using small squares of input data. Feature are learned and used across the whole image, allowing for the objects in the images to be shifted or translated in the scene and still detectable by the network. It is this reason why the network is so useful for object recognition in photographs, picking out digits, faces, objects and so on with varying orientation. In summary, below are some of the benefits of using convolutional neural networks:

- They use fewer parameters (weights) to learn than a fully connected network.

- They are designed to be invariant to object position and distortion in the scene.

- They automatically learn and generalize features from the input domain.

18.2   Building Blocks of Convolutional Neural Networks

There are three types of layers in a Convolutional Neural Network:

1. Convolutional Layers.

2. Pooling Layers.

3. Fully-Connected Layers.

18.3   Convolutional Layers

Convolutional layers are comprised of filters and feature maps.

18.3.1   Filters

The filters are essentially the *neurons* of the layer. They have both weighted inputs and generate an output value like a neuron. The input size is a fixed square called a patch or a receptive field. If the convolutional layer is an input layer, then the input patch will be pixel values. If they deeper in the network architecture, then the convolutional layer will take input from a feature map from the previous layer.

18.3.2   Feature Maps

The feature map is the output of one filter applied to the previous layer. A given filter is drawn across the entire previous layer, moved one pixel at a time. Each position results in an activation of the neuron and the output is collected in the feature map. You can see that if the receptive field is moved one pixel from activation to activation, then the field will overlap with the previous activation by (field width - 1) input values.

The distance that filter is moved across the input from the previous layer each activation is referred to as the stride. If the size of the previous layer is not cleanly divisible by the size of the filters receptive field and the size of the stride then it is possible for the receptive field to attempt to read off the edge of the input feature map. In this case, techniques like zero padding can be used to invent mock inputs with zero values for the receptive field to read.

18.4   Pooling Layers

The pooling layers down-sample the previous layers feature map. Pooling layers follow a sequence of one or more convolutional layers and are intended to consolidate the features learned and expressed in the previous layers feature map. As such, pooling may be consider a technique to compress or generalize feature representations and generally reduce the overfitting of the training data by the model.

They too have a receptive field, often much smaller than the convolutional layer. Also, the stride or number of inputs that the receptive field is moved for each activation is often equal to the size of the receptive field to avoid any overlap. Pooling layers are often very simple, taking the average or the maximum of the input value in order to create its own feature map.

18.5 Fully Connected Layers

Fully connected layers are the normal flat feedforward neural network layer. These layers may have a nonlinear activation function or a softmax activation in order to output probabilities of class predictions. Fully connected layers are used at the end of the network after feature extraction and consolidation has been performed by the convolutional and pooling layers. They are used to create final nonlinear combinations of features and for making predictions by the network.

18.6 Worked Example

You now know about convolutional, pooling and fully connected layers. Let's make this more concrete by working through how these three layers may be connected together.

18.6.1 Image Input Data

Let's assume we have a dataset of gray scale images. Each image has the same size of 32 pixels wide and 32 pixels high, and pixel values are between 0 and 255, e.g. a matrix of $32 \times 32 \times 1$ or 1,024 pixel values. Image input data is expressed as a 3-dimensional matrix of *width* $\times$ *height* $\times$ *channels*. If we were using color images in our example, we would have 3 channels for the red, green and blue pixel values, e.g. $32 \times 32 \times 3$.

18.6.2 Convolutional Layer

We define a convolutional layer with 10 filters and a receptive field 5 pixels wide and 5 pixels high and a stride length of 1. Because each filter can only get input from (i.e. *see*) 5×5 (25) pixels at a time, we can calculate that each will require $25 + 1$ input weights (plus 1 for the bias input). Dragging the 5×5 receptive field across the input image data with a stride width of 1 will result in a feature map of 28×28 output values or 784 distinct activations per image.

We have 10 filters, so that is 10 different 28×28 feature maps or 7,840 outputs that will be created for one image. Finally, we know we have 26 inputs per filter, 10 filters and 28×28 output values to calculate per filter, therefore we have a total of $26 \times 10 \times 28 \times 28$ or 203,840 connections in our convolutional layer, we want to phrase it using traditional neural network nomenclature. Convolutional layers also make use of a nonlinear transfer function as part of activation and the rectifier activation function is the popular default to use.

18.6.3 Pool Layer

We define a pooling layer with a receptive field with a width of 2 inputs and a height of 2 inputs. We also use a stride of 2 to ensure that there is no overlap. This results in feature maps that are one half the size of the input feature maps. From 10 different 28×28 feature maps as input to 10 different 14×14 feature maps as output. We will use a `max()` operation for each receptive field so that the activation is the maximum input value.

18.6.4 Fully Connected Layer

Finally, we can flatten out the square feature maps into a traditional flat fully connected layer. We can define the fully connected layer with 200 hidden neurons, each with $10 \times 14 \times 14$ input connections, or $1,960 + 1$ weights per neuron. That is a total of 392,200 connections and weights to learn in this layer. We can use a sigmoid or softmax transfer function to output probabilities of class values directly.

18.7 Convolutional Neural Networks Best Practices

Now that we know about the building blocks for a convolutional neural network and how the layers hang together, we can review some best practices to consider when applying them.

- **Input Receptive Field Dimensions**: The default is 2D for images, but could be 1D such as for words in a sentence or 3D for video that adds a time dimension.

- **Receptive Field Size**: The patch should be as small as possible, but large enough to *see* features in the input data. It is common to use 3×3 on small images and 5×5 or 7×7 and more on larger image sizes.

- **Stride Width**: Use the default stride of 1. It is easy to understand and you don't need padding to handle the receptive field falling off the edge of your images. This could be increased to 2 or larger for larger images.

- **Number of Filters**: Filters are the feature detectors. Generally fewer filters are used at the input layer and increasingly more filters used at deeper layers.

- **Padding**: Set to zero and called zero padding when reading non-input data. This is useful when you cannot or do not want to standardize input image sizes or when you want to use receptive field and stride sizes that do not neatly divide up the input image size.

- **Pooling**: Pooling is a destructive or generalization process to reduce overfitting. Receptive field size is almost always set to 2×2 with a stride of 2 to discard 75% of the activations from the output of the previous layer.

- **Data Preparation**: Consider standardizing input data, both the dimensions of the images and pixel values.

- **Pattern Architecture**: It is common to pattern the layers in your network architecture. This might be one, two or some number of convolutional layers followed by a pooling layer. This structure can then be repeated one or more times. Finally, fully connected layers are often only used at the output end and may be stacked one, two or more deep.

- **Dropout**: CNNs have a habit of overfitting, even with pooling layers. Dropout should be used such as between fully connected layers and perhaps after pooling layers.

18.8 Summary

In this lesson you discovered convolutional neural networks. You learned:

- Why CNNs are needed to preserve spatial structure in your input data and the benefits they provide.

- The building blocks of CNN including convolutional, pooling and fully connected layers.

- How the layers in a CNN hang together.

- Best practices when applying CNN to your own problems.

18.8.1 Next

You now know about convolutional neural networks. In the next section you will discover how to develop your first convolutional neural network in Keras for a handwriting digit recognition problem.

Chapter 19

Project: Handwritten Digit Recognition

A popular demonstration of the capability of deep learning techniques is object recognition in image data. The *hello world* of object recognition for machine learning and deep learning is the MNIST dataset for handwritten digit recognition. In this project you will discover how to develop a deep learning model to achieve near state-of-the-art performance on the MNIST handwritten digit recognition task in Python using the Keras deep learning library. After completing this step-by-step tutorial, you will know:

- How to load the MNIST dataset in Keras and develop a baseline neural network model for the problem.

- How to implement and evaluate a simple Convolutional Neural Network for MNIST.

- How to implement a close to state-of-the-art deep learning model for MNIST.

Let's get started.

Note: You may want to speed up the computation for this tutorial by using GPU rather than CPU hardware, such as the process described in Chapter 5. This is a suggestion, not a requirement. The tutorial will work just fine on the CPU.

19.1   Handwritten Digit Recognition Dataset

The MNIST problem is a dataset developed by Yann LeCun, Corinna Cortes and Christopher Burges for evaluating machine learning models on the handwritten digit classification problem[1]. The dataset was constructed from a number of scanned document datasets available from the National Institute of Standards and Technology (NIST). This is where the name for the dataset comes from, as the Modified NIST or MNIST dataset.

Images of digits were taken from a variety of scanned documents, normalized in size and centered. This makes it an excellent dataset for evaluating models, allowing the developer to focus on the machine learning with very little data cleaning or preparation required. Each image is a 28×28 pixel square (784 pixels total). A standard split of the dataset is used to

---

[1]http://yann.lecun.com/exdb/mnist/

evaluate and compare models, where 60,000 images are used to train a model and a separate set of 10,000 images are used to test it.

It is a digit recognition task. As such there are 10 digits (0 to 9) or 10 classes to predict. Results are reported using prediction error, which is nothing more than the inverted classification accuracy. Excellent results achieve a prediction error of less than 1%. State-of-the-art prediction error of approximately 0.2% can be achieved with large Convolutional Neural Networks. There is a listing of the state-of-the-art results and links to the relevant papers on the MNIST and other datasets on Rodrigo Benenson's webpage[2].

19.2   Loading the MNIST dataset in Keras

The Keras deep learning library provides a convenience method for loading the MNIST dataset. The dataset is downloaded automatically the first time this function is called and is stored in your home directory in `~/.keras/datasets/mnist.pkl.gz` as a 15 megabyte file. This is very handy for developing and testing deep learning models. To demonstrate how easy it is to load the MNIST dataset, we will first write a little script to download and visualize the first 4 images in the training dataset.

```python
# Plot ad hoc mnist instances
from keras.datasets import mnist
import matplotlib.pyplot as plt
# load (downloaded if needed) the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# plot 4 images as gray scale
plt.subplot(221)
plt.imshow(X_train[0], cmap=plt.get_cmap('gray'))
plt.subplot(222)
plt.imshow(X_train[1], cmap=plt.get_cmap('gray'))
plt.subplot(223)
plt.imshow(X_train[2], cmap=plt.get_cmap('gray'))
plt.subplot(224)
plt.imshow(X_train[3], cmap=plt.get_cmap('gray'))
# show the plot
plt.show()
```

Listing 19.1: Load the MNIST Dataset in Keras.

You can see that downloading and loading the MNIST dataset is as easy as calling the `mnist.load_data()` function. Running the above example, you should see the image below.

---

[2]http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html
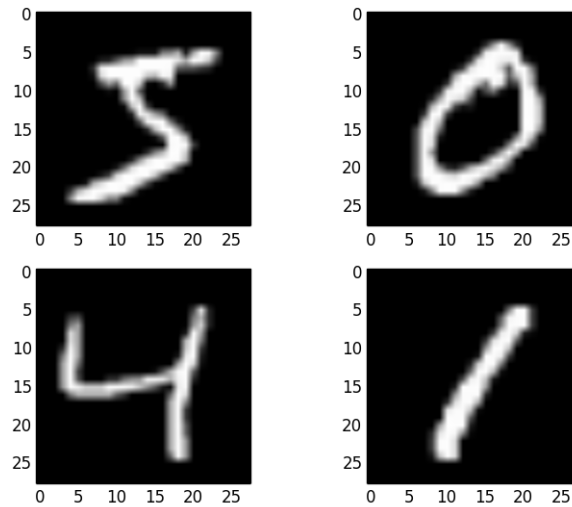
Figure 19.1: Examples from the MNIST dataset

19.3 Baseline Model with Multilayer Perceptrons

Do we really need a complex model like a convolutional neural network to get the best results with MNIST? You can get good results using a very simple neural network model with a single hidden layer. In this section we will create a simple Multilayer Perceptron model that achieves an error rate of 1.74%. We will use this as a baseline for comparison to more complex convolutional neural network models. Let's start off by importing the classes and functions we will need.

```
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.utils import np_utils
```

Listing 19.2: Import Classes and Functions.

It is always a good idea to initialize the random number generator to a constant to ensure that the results of your script are reproducible.

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

Listing 19.3: Initialize The Random Number Generator.

Now we can load the MNIST dataset using the Keras helper function.

```
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Listing 19.4: Load the MNIST Dataset.

The training dataset is structured as a 3-dimensional array of instance, image width and image height. For a Multilayer Perceptron model we must reduce the images down into a vector of pixels. In this case the 28×28 sized images will be 784 pixel input vectors. We can do this transform easily using the `reshape()` function on the NumPy array. The pixel values are integers, so we cast them to floating point values so that we can normalize them easily in the next step.

```python
# flatten 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
```

Listing 19.5: Prepare MNIST Dataset For Modeling.

The pixel values are gray scale between 0 and 255. It is almost always a good idea to perform some scaling of input values when using neural network models. Because the scale is well known and well behaved, we can very quickly normalize the pixel values to the range 0 and 1 by dividing each value by the maximum of 255.

```python
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
```

Listing 19.6: Normalize Pixel Values.

Finally, the output variable is an integer from 0 to 9. This is a multiclass classification problem. As such, it is good practice to use a one hot encoding of the class values, transforming the vector of class integers into a binary matrix. We can easily do this using the built-in `np_utils.to_categorical()` helper function in Keras.

```python
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Listing 19.7: One Hot Encode The Output Variable.

We are now ready to create our simple neural network model. We will define our model in a function. This is handy if you want to extend the example later and try and get a better score.

```python
# define baseline model
def baseline_model():
  # create model
  model = Sequential()
  model.add(Dense(num_pixels, input_dim=num_pixels, init='normal', activation='relu'))
  model.add(Dense(num_classes, init='normal', activation='softmax'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
```

Listing 19.8: Define and Compile the Baseline Model.

The model is a simple neural network with one hidden layer with the same number of neurons as there are inputs (784). A rectifier activation function is used for the neurons in the hidden layer. A softmax activation function is used on the output layer to turn the outputs into probability-like values and allow one class of the 10 to be selected as the model's output

prediction. Logarithmic loss is used as the loss function (called `categorical_crossentropy` in Keras) and the efficient ADAM gradient descent algorithm is used to learn the weights. A summary of the network structure is provided below:
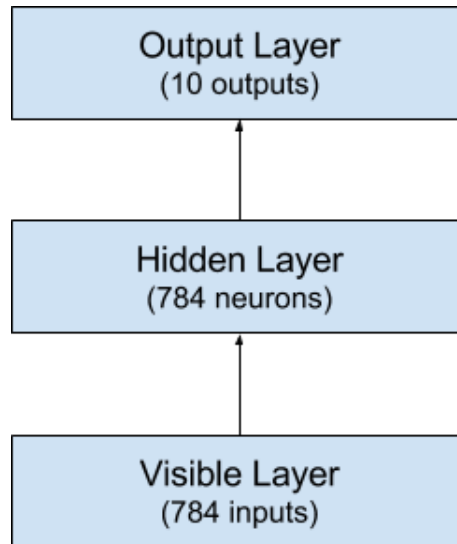
Figure 19.2: Summary of Multilayer Perceptron Network Structure.

We can now fit and evaluate the model. The model is fit over 10 epochs with updates every 200 images. The test data is used as the validation dataset, allowing you to see the skill of the model as it trains. A verbose value of 2 is used to reduce the output to one line for each training epoch. Finally, the test dataset is used to evaluate the model and a classification error rate is printed.

```
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
    verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

Listing 19.9: Evaluate the Baseline Model.

The full code listing is provided below for completeness.

```
# Baseline MLP for MNIST dataset
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.utils import np_utils
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```python
# flatten 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# define baseline model
def baseline_model():
  # create model
  model = Sequential()
  model.add(Dense(num_pixels, input_dim=num_pixels, init='normal', activation='relu'))
  model.add(Dense(num_classes, init='normal', activation='softmax'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
    verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Baseline Error: %.2f%%" % (100-scores[1]*100))
```

Listing 19.10: Multilayer Perceptron Model for MNIST Problem.

Running the example might take a few minutes when run on a CPU. You should see the output below. This simple network defined in very few lines of code achieves a respectable error rate of 1.73%.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
7s - loss: 0.2791 - acc: 0.9203 - val_loss: 0.1420 - val_acc: 0.9579
Epoch 2/10
7s - loss: 0.1122 - acc: 0.9679 - val_loss: 0.0992 - val_acc: 0.9699
Epoch 3/10
8s - loss: 0.0724 - acc: 0.9790 - val_loss: 0.0784 - val_acc: 0.9745
Epoch 4/10
8s - loss: 0.0510 - acc: 0.9853 - val_loss: 0.0777 - val_acc: 0.9774
Epoch 5/10
8s - loss: 0.0367 - acc: 0.9898 - val_loss: 0.0628 - val_acc: 0.9792
Epoch 6/10
8s - loss: 0.0264 - acc: 0.9931 - val_loss: 0.0641 - val_acc: 0.9797
Epoch 7/10
8s - loss: 0.0185 - acc: 0.9958 - val_loss: 0.0604 - val_acc: 0.9810
Epoch 8/10
10s - loss: 0.0148 - acc: 0.9967 - val_loss: 0.0621 - val_acc: 0.9811
Epoch 9/10
8s - loss: 0.0109 - acc: 0.9978 - val_loss: 0.0607 - val_acc: 0.9817
Epoch 10/10
8s - loss: 0.0073 - acc: 0.9987 - val_loss: 0.0595 - val_acc: 0.9827
```

```
Baseline Error: 1.73%
```

Listing 19.11: Sample Output From Evaluating the Baseline Model.

19.4 Simple Convolutional Neural Network for MNIST

Now that we have seen how to load the MNIST dataset and train a simple Multilayer Perceptron model on it, it is time to develop a more sophisticated convolutional neural network or CNN model. Keras does provide a lot of capability for creating convolutional neural networks. In this section we will create a simple CNN for MNIST that demonstrates how to use all of the aspects of a modern CNN implementation, including Convolutional layers, Pooling layers and Dropout layers. The first step is to import the classes and functions needed.

```python
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
```

Listing 19.12: Import classes and functions.

Again, we always initialize the random number generator to a constant seed value for reproducibility of results.

```python
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

Listing 19.13: Seed Random Number Generator.

Next we need to load the MNIST dataset and reshape it so that it is suitable for use training a CNN. In Keras, the layers used for two-dimensional convolutions expect pixel values with the dimensions `[channels][width][height]`. In the case of RGB, the first dimension channels would be 3 for the red, green and blue components and it would be like having 3 image inputs for every color image. In the case of MNIST where the channels values are gray scale, the pixel dimension is set to 1.

```python
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][channels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')
```

Listing 19.14: Load Dataset and Separate Into Train and Test Sets.

As before, it is a good idea to normalize the pixel values to the range 0 and 1 and one hot encode the output variable.

```python
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
```

```
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Listing 19.15: Normalize and One Hot Encode Data.

Next we define our neural network model. Convolutional neural networks are more complex than standard Multilayer Perceptrons, so we will start by using a simple structure to begin with that uses all of the elements for state-of-the-art results. Below summarizes the network architecture.

1. The first hidden layer is a convolutional layer called a `Convolution2D`. The layer has 32 feature maps, which with the size of 5×5 and a rectifier activation function. This is the input layer, expecting images with the structure outline above.

2. Next we define a pooling layer that takes the maximum value called `MaxPooling2D`. It is configured with a pool size of 2×2.

3. The next layer is a regularization layer using dropout called `Dropout`. It is configured to randomly exclude 20% of neurons in the layer in order to reduce overfitting.

4. Next is a layer that converts the 2D matrix data to a vector called `Flatten`. It allows the output to be processed by standard fully connected layers.

5. Next a fully connected layer with 128 neurons and rectifier activation function is used.

6. Finally, the output layer has 10 neurons for the 10 classes and a softmax activation function to output probability-like predictions for each class.

As before, the model is trained using logarithmic loss and the ADAM gradient descent algorithm. A depiction of the network structure is provided below.
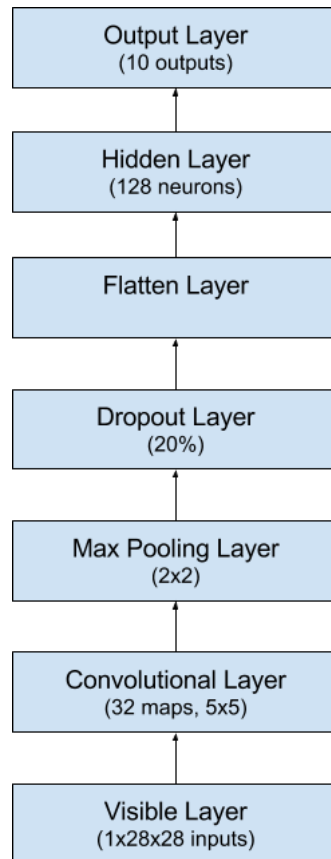
Figure 19.3: Summary of Convolutional Neural Network Structure.

```python
def baseline_model():
  # create model
  model = Sequential()
  model.add(Convolution2D(32, 5, 5, border_mode='valid', input_shape=(1, 28, 28),
      activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.2))
  model.add(Flatten())
  model.add(Dense(128, activation='relu'))
  model.add(Dense(num_classes, activation='softmax'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
```

Listing 19.16: Define and Compile CNN Model.

We evaluate the model the same way as before with the Multilayer Perceptron. The CNN is fit over 10 epochs with a batch size of 200.

```python
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
    verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
```

```
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

Listing 19.17: Fit and Evaluate The CNN Model.

The full code listing is provided below for completeness.

```
# Simple CNN for the MNIST Dataset
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][channels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# define a simple CNN model
def baseline_model():
  # create model
  model = Sequential()
  model.add(Convolution2D(32, 5, 5, border_mode='valid', input_shape=(1, 28, 28),
      activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.2))
  model.add(Flatten())
  model.add(Dense(128, activation='relu'))
  model.add(Dense(num_classes, activation='softmax'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
    verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("CNN Error: %.2f%%" % (100-scores[1]*100))
```

Listing 19.18: CNN Model for MNIST Problem.

Running the example, the accuracy on the training and validation test is printed each epoch

and at the end of the classification error rate is printed. Epochs may take 60 seconds to run on the CPU, or about 10 minutes in total depending on your hardware. You can see that the network achieves an error rate of 1.00, which is better than our simple Multilayer Perceptron model above.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
52s - loss: 0.2412 - acc: 0.9318 - val_loss: 0.0756 - val_acc: 0.9763
Epoch 2/10
52s - loss: 0.0728 - acc: 0.9781 - val_loss: 0.0527 - val_acc: 0.9825
Epoch 3/10
53s - loss: 0.0497 - acc: 0.9852 - val_loss: 0.0393 - val_acc: 0.9852
Epoch 4/10
53s - loss: 0.0414 - acc: 0.9868 - val_loss: 0.0436 - val_acc: 0.9852
Epoch 5/10
53s - loss: 0.0324 - acc: 0.9898 - val_loss: 0.0376 - val_acc: 0.9871
Epoch 6/10
53s - loss: 0.0281 - acc: 0.9910 - val_loss: 0.0421 - val_acc: 0.9866
Epoch 7/10
52s - loss: 0.0227 - acc: 0.9928 - val_loss: 0.0319 - val_acc: 0.9895
Epoch 8/10
52s - loss: 0.0198 - acc: 0.9938 - val_loss: 0.0357 - val_acc: 0.9885
Epoch 9/10
52s - loss: 0.0158 - acc: 0.9951 - val_loss: 0.0333 - val_acc: 0.9887
Epoch 10/10
52s - loss: 0.0144 - acc: 0.9953 - val_loss: 0.0322 - val_acc: 0.9900
CNN Error: 1.00%
```

Listing 19.19: Sample Output From Evaluating the CNN Model.

19.5 Larger Convolutional Neural Network for MNIST

Now that we have seen how to create a simple CNN, let's take a look at a model capable of close to state-of-the-art results. We import the classes and functions then load and prepare the data the same as in the previous CNN example. This time we define a larger CNN architecture with additional convolutional, max pooling layers and fully connected layers. The network topology can be summarized as follows.

1. Convolutional layer with 30 feature maps of size 5×5.

2. Pooling layer taking the max over 2×2 patches.

3. Convolutional layer with 15 feature maps of size 3×3.

4. Pooling layer taking the max over 2×2 patches.

5. Dropout layer with a probability of 20%.

6. Flatten layer.

7. Fully connected layer with 128 neurons and rectifier activation.

8. Fully connected layer with 50 neurons and rectifier activation.

9. Output layer.

A depictions of this larger network structure is provided below.
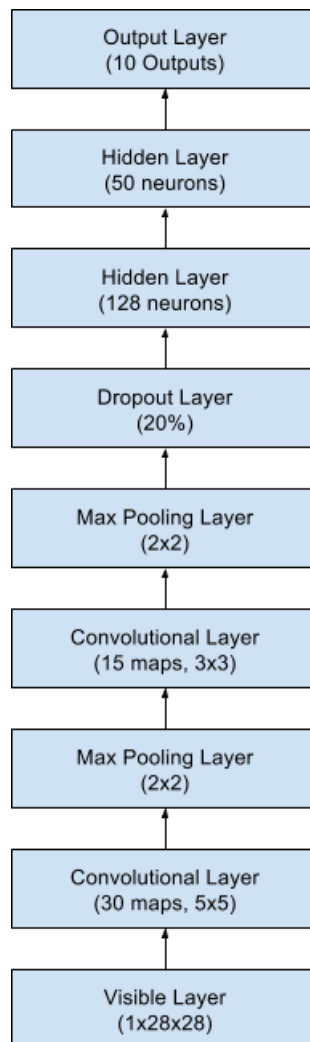
Figure 19.4: Summary of the Larger Convolutional Neural Network Structure.

Like the previous two experiments, the model is fit over 10 epochs with a batch size of 200.

```python
# Larger CNN for the MNIST Dataset
import numpy
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
```

```python
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28).astype('float32')
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# define the larger model
def larger_model():
  # create model
  model = Sequential()
  model.add(Convolution2D(30, 5, 5, border_mode='valid', input_shape=(1, 28, 28),
      activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Convolution2D(15, 3, 3, activation='relu'))
  model.add(MaxPooling2D(pool_size=(2, 2)))
  model.add(Dropout(0.2))
  model.add(Flatten())
  model.add(Dense(128, activation='relu'))
  model.add(Dense(50, activation='relu'))
  model.add(Dense(num_classes, activation='softmax'))
  # Compile model
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  return model
# build the model
model = larger_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=10, batch_size=200,
    verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Large CNN Error: %.2f%%" % (100-scores[1]*100))
```

Listing 19.20: Larger CNN for the MNIST Problem.

Running the example prints accuracy on the training and validation datasets each epoch and a final classification error rate. The model takes about 60 seconds to run per epoch on a modern CPU. This slightly larger model achieves the respectable classification error rate of 0.83%.

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
59s - loss: 0.3791 - acc: 0.8793 - val_loss: 0.0812 - val_acc: 0.9742
Epoch 2/10
59s - loss: 0.0929 - acc: 0.9708 - val_loss: 0.0467 - val_acc: 0.9846
Epoch 3/10
59s - loss: 0.0679 - acc: 0.9787 - val_loss: 0.0374 - val_acc: 0.9878
Epoch 4/10
59s - loss: 0.0539 - acc: 0.9828 - val_loss: 0.0321 - val_acc: 0.9893
Epoch 5/10
59s - loss: 0.0462 - acc: 0.9858 - val_loss: 0.0282 - val_acc: 0.9897
Epoch 6/10
60s - loss: 0.0396 - acc: 0.9874 - val_loss: 0.0278 - val_acc: 0.9902
```

```
Epoch 7/10
60s - loss: 0.0365 - acc: 0.9884 - val_loss: 0.0229 - val_acc: 0.9921
Epoch 8/10
59s - loss: 0.0328 - acc: 0.9895 - val_loss: 0.0287 - val_acc: 0.9901
Epoch 9/10
60s - loss: 0.0306 - acc: 0.9903 - val_loss: 0.0224 - val_acc: 0.9923
Epoch 10/10
59s - loss: 0.0268 - acc: 0.9915 - val_loss: 0.0240 - val_acc: 0.9917
Large CNN Error: 0.83%
```

Listing 19.21: Sample Output From Evaluating the Larger CNN Model.

This is not an optimized network topology. Nor is this a reproduction of a network topology from a recent paper. There is a lot of opportunity for you to tune and improve upon this model. What is the best classification error rate you can achieve?

19.6 Summary

In this lesson you discovered the MNIST handwritten digit recognition problem and deep learning models developed in Python using the Keras library that are capable of achieving excellent results. Working through this tutorial you learned:

- How to load the MNIST dataset in Keras and generate plots of the dataset.

- How to reshape the MNIST dataset and develop a simple but well performing Multilayer Perceptron model for the problem.

- How to use Keras to create convolutional neural network models for MNIST.

- How to develop and evaluate larger CNN models for MNIST capable of near world class results.

19.6.1 Next

You now know how to develop and improve convolutional neural network models in Keras. A powerful technique for improving the performance of CNN models is to use data augmentation. In the next section you will discover the data augmentation API in Keras and how the different image augmentation techniques affect the MNIST images.

Chapter 20

Improve Model Performance With Image Augmentation

Data preparation is required when working with neural network and deep learning models. Increasingly data augmentation is also required on more complex object recognition tasks. In this lesson you will discover how to use data preparation and data augmentation with your image datasets when developing and evaluating deep learning models in Python with Keras. After completing this lesson, you will know:

- About the image augmentation API provide by Keras and how to use it with your models.

- How to perform feature standardization.

- How to perform ZCA whitening of your images.

- How to augment data with random rotations, shifts and flips of images.

- How to save augmented image data to disk.

Let's get started.

20.1 Keras Image Augmentation API

Like the rest of Keras, the image augmentation API is simple and powerful. Keras provides the `ImageDataGenerator` class that defines the configuration for image data preparation and augmentation. This includes capabilities such as:

- Feature-wise standardization.

- ZCA whitening.

- Random rotation, shifts, shear and flips.

- Dimension reordering.

- Save augmented images to disk.

An augmented image generator can be created as follows:

```
datagen = ImageDataGenerator()
```

Listing 20.1: Create a ImageDataGenerator.

Rather than performing the operations on your entire image dataset in memory, the API is designed to be iterated by the deep learning model fitting process, creating augmented image data for you just-in-time. This reduces your memory overhead, but adds some additional time cost during model training. After you have created and configured your `ImageDataGenerator`, you must fit it on your data. This will calculate any statistics required to actually perform the transforms to your image data. You can do this by calling the `fit()` function on the data generator and pass it your training dataset.

```
datagen.fit(train)
```

Listing 20.2: Fit the ImageDataGenerator.

The data generator itself is in fact an iterator, returning batches of image samples when requested. We can configure the batch size and prepare the data generator and get batches of images by calling the `flow()` function.

```
X_batch, y_batch = datagen.flow(train, train, batch_size=32)
```

Listing 20.3: Configure the Batch Size for the ImageDataGenerator.

Finally we can make use of the data generator. Instead of calling the `fit()` function on our model, we must call the `fit_generator()` function and pass in the data generator and the desired length of an epoch as well as the total number of epochs on which to train.

```
fit_generator(datagen, samples_per_epoch=len(train), nb_epoch=100)
```

Listing 20.4: Fit a Model Using the ImageDataGenerator.

You can learn more about the Keras image data generator API in the Keras documentation[1].

20.2   Point of Comparison for Image Augmentation

Now that you know how the image augmentation API in Keras works, let's look at some examples. We will use the MNIST handwritten digit recognition task in these examples (learn more in Section 19.1). To begin, let's take a look at the first 9 images in the training dataset.

```
# Plot of images as baseline for comparison
from keras.datasets import mnist
from matplotlib import pyplot
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# create a grid of 3x3 images
for i in range(0, 9):
  pyplot.subplot(330 + 1 + i)
  pyplot.imshow(X_train[i], cmap=pyplot.get_cmap('gray'))

# show the plot
```

---

[1]http://keras.io/preprocessing/image/

```
pyplot.show()
```

Listing 20.5: Load and Plot the MNIST dataset.

Running this example provides the following image that we can use as a point of comparison with the image preparation and augmentation tasks in the examples below.

Figure 20.1: Samples from the MNIST dataset.

20.3   Feature Standardization

It is also possible to standardize pixel values across the entire dataset. This is called feature standardization and mirrors the type of standardization often performed for each column in a tabular dataset. This is different to sample standardization described in the previous section as pixel values are standardized across all samples (all images in the dataset). In this case each image is considered a feature. You can perform feature standardization by setting the `featurewise_center` and `featurewise_std_normalization` arguments on the `ImageDataGenerator` class.

```
# Standardize images across the dataset, mean=0, stdev=1
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load data
```

```python
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
datagen = ImageDataGenerator(featurewise_center=True, featurewise_std_normalization=True)
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
  # create a grid of 3x3 images
  for i in range(0, 9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap('gray'))
  # show the plot
  pyplot.show()
  break
```

Listing 20.6: Example of Feature Standardization.

Running this example you can see that the effect on the actual images, seemingly darkening and lightening different digits.
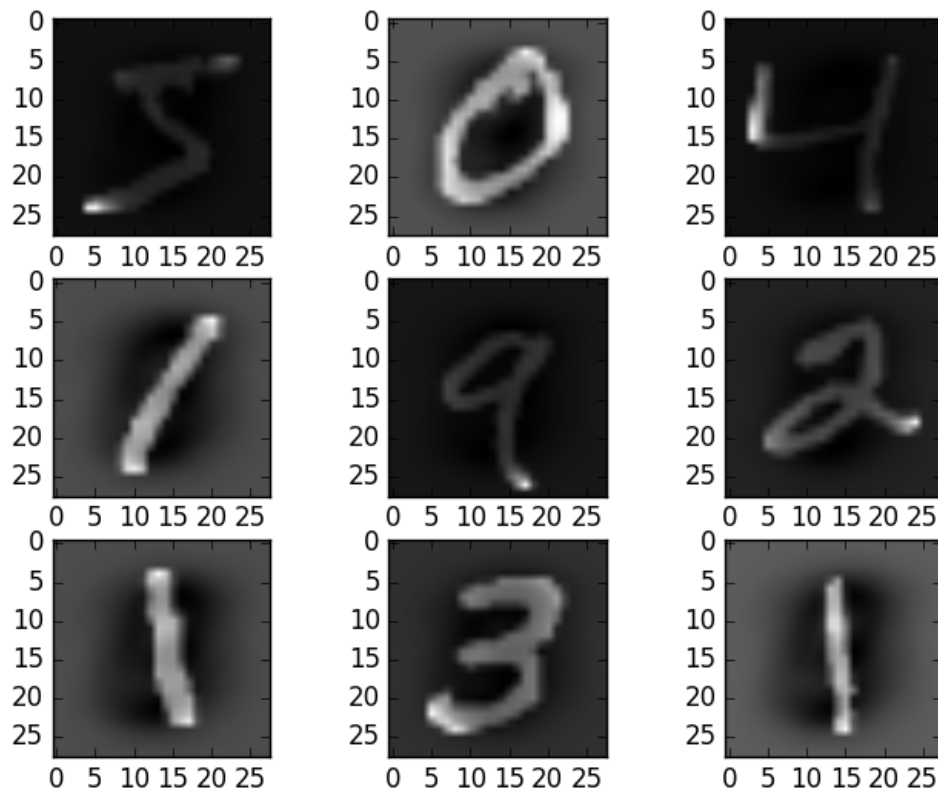
Figure 20.2: Standardized Feature MNIST Images.

20.4   ZCA Whitening

A whitening transform of an image is a linear algebra operation that reduces the redundancy in the matrix of pixel images. Less redundancy in the image is intended to better highlight the structures and features in the image to the learning algorithm. Typically, image whitening is performed using the Principal Component Analysis (PCA) technique. More recently, an alternative called ZCA (learn more in Appendix A of this tech report[2]) shows better results and results in transformed images that keeps all of the original dimensions and unlike PCA, resulting transformed images still look like their originals. You can perform a ZCA whitening transform by setting the zca_whitening argument to True.

```python
# ZCA whitening
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
```

[2]http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

```
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
datagen = ImageDataGenerator(zca_whitening=True)
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
  # create a grid of 3x3 images
  for i in range(0, 9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap('gray'))
  # show the plot
  pyplot.show()
  break
```

Listing 20.7: Example of ZCA Whitening.

Running the example, you can see the same general structure in the images and how the outline of each digit has been highlighted.
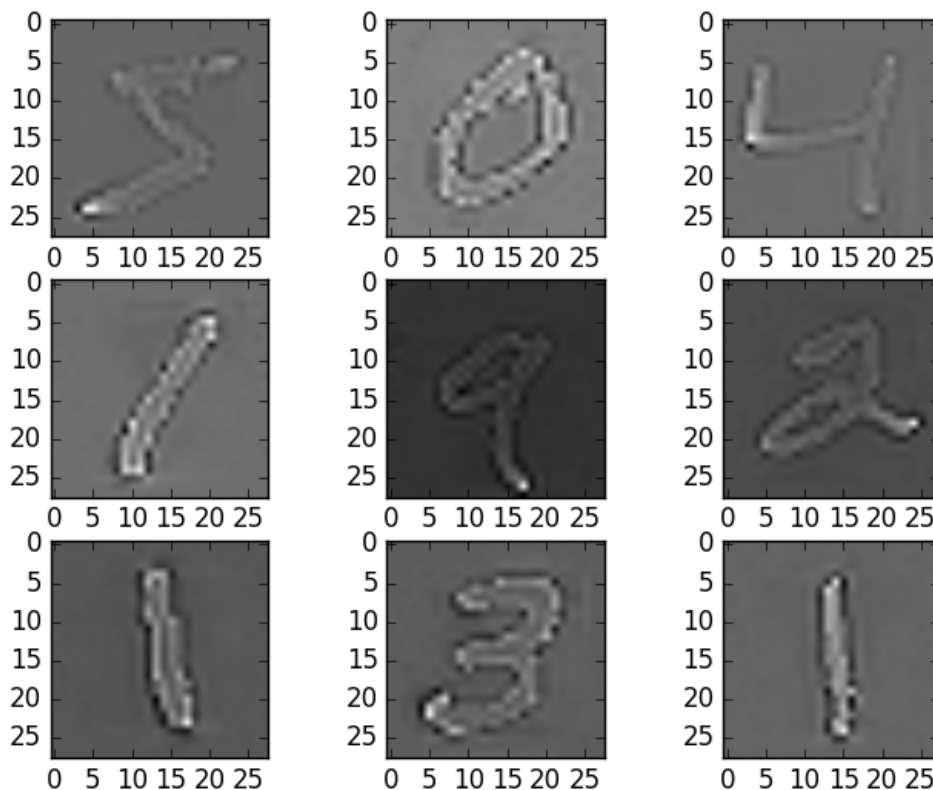
Figure 20.3: ZCA Whitening of MNIST Images.

20.5 Random Rotations

Sometimes images in your sample data may have varying and different rotations in the scene. You can train your model to better handle rotations of images by artificially and randomly rotating images from your dataset during training. The example below creates random rotations of the MNIST digits up to 90 degrees by setting the `rotation_range` argument.

```python
# Random Rotations
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
datagen = ImageDataGenerator(rotation_range=90)
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
  # create a grid of 3x3 images
  for i in range(0, 9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap('gray'))
  # show the plot
  pyplot.show()
  break
```

Listing 20.8: Example of Random Image Rotations.

Running the example, you can see that images have been rotated left and right up to a limit of 90 degrees. This is not helpful on this problem because the MNIST digits have a normalized orientation, but this transform might be of help when learning from photographs where the objects may have different orientations.
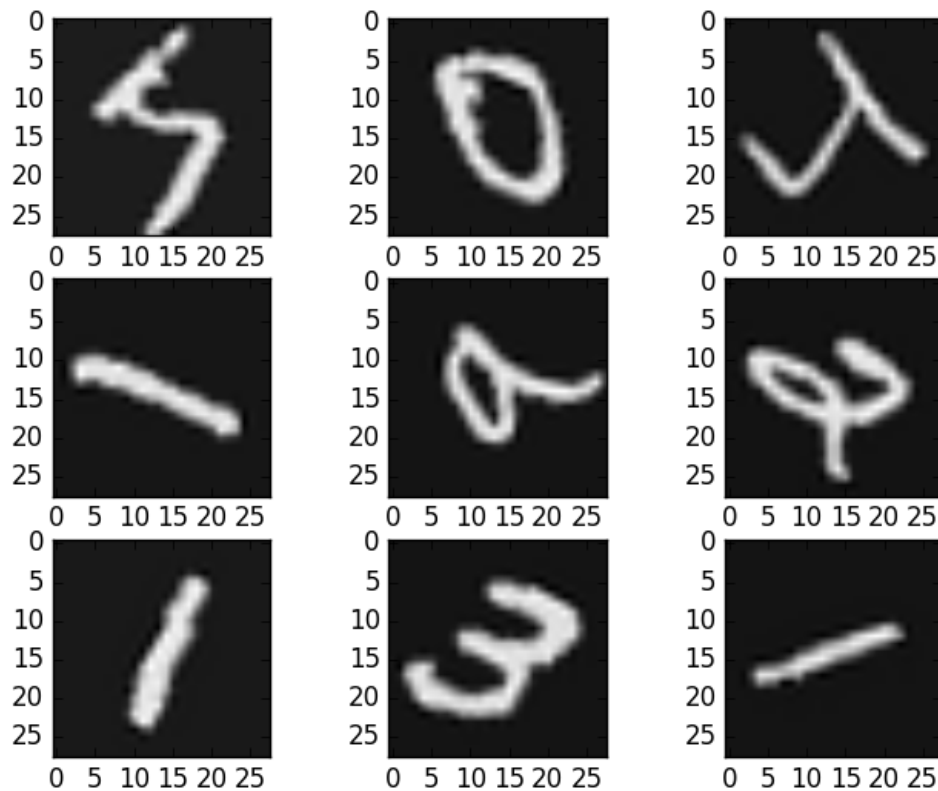
Figure 20.4: Random Rotations of MNIST Images.

20.6 Random Shifts

Objects in your images may not be centered in the frame. They may be off-center in a variety of different ways. You can train your deep learning network to expect and currently handle off-center objects by artificially creating shifted versions of your training data. Keras supports separate horizontal and vertical random shifting of training data by the `width_shift_range` and `height_shift_range` arguments.

```
# Random Shifts
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
shift = 0.2
```

```
datagen = ImageDataGenerator(width_shift_range=shift, height_shift_range=shift)
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
  # create a grid of 3x3 images
  for i in range(0, 9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap('gray'))
  # show the plot
  pyplot.show()
  break
```

Listing 20.9: Example of Random Image Shifts.

Running this example creates shifted versions of the digits. Again, this is not required for MNIST as the handwritten digits are already centered, but you can see how this might be useful on more complex problem domains.
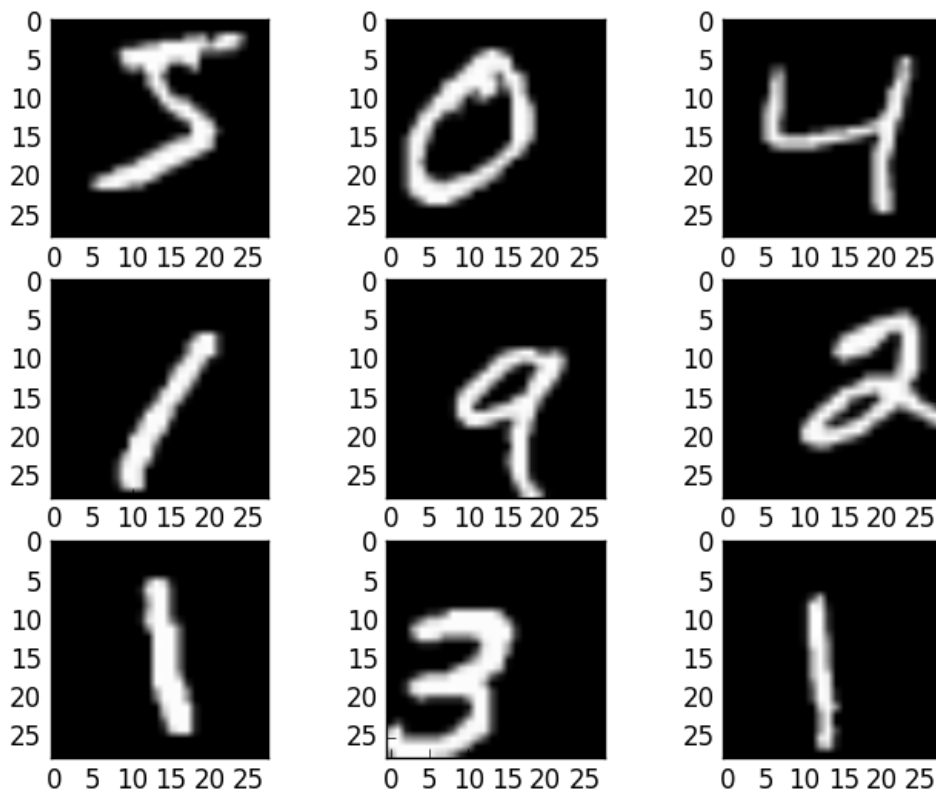
Figure 20.5: Random Shifted MNIST Images.

20.7 Random Flips

Another augmentation to your image data that can improve performance on large and complex problems is to create random flips of images in your training data. Keras supports random flipping along both the vertical and horizontal axes using the `vertical_flip` and `horizontal_flip` arguments.

```python
# Random Flips
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True)
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9):
  # create a grid of 3x3 images
  for i in range(0, 9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap('gray'))
  # show the plot
  pyplot.show()
  break
```

Listing 20.10: Example of Random Image Flips.

Running this example you can see flipped digits. Flipping digits in MNIST is not useful as they will always have the correct left and right orientation, but this may be useful for problems with photographs of objects in a scene that can have a varied orientation.
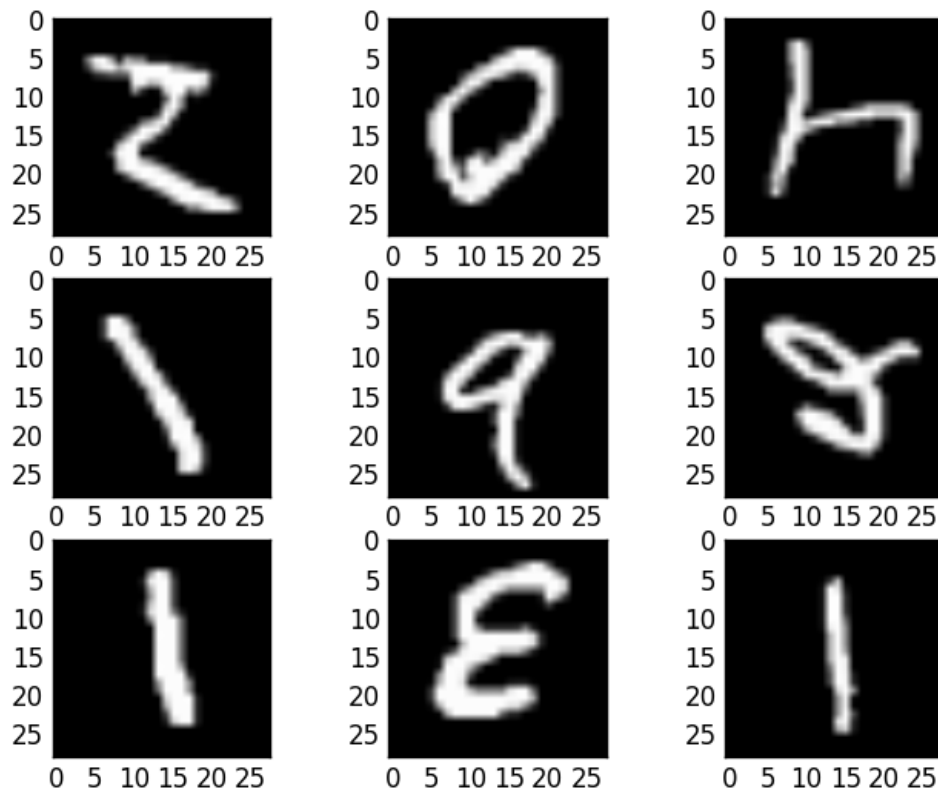
Figure 20.6: Randomly Flipped MNIST Images.

20.8   Saving Augmented Images to File

The data preparation and augmentation is performed just-in-time by Keras. This is efficient in terms of memory, but you may require the exact images used during training. For example, perhaps you would like to use them with a different software package later or only generate them once and use them on multiple different deep learning models or configurations.

Keras allows you to save the images generated during training. The directory, filename prefix and image file type can be specified to the `flow()` function before training. Then, during training, the generated images will be written to file. The example below demonstrates this and writes 9 images to a `images` subdirectory with the prefix `aug` and the file type of PNG.

```
# Save augmented images to file
from keras.datasets import mnist
from keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot
import os
# load data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# reshape to be [samples][pixels][width][height]
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
```

```python
# convert from int to float
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# define data preparation
datagen = ImageDataGenerator()
# fit parameters from data
datagen.fit(X_train)
# configure batch size and retrieve one batch of images
os.makedirs('images')
for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=9, save_to_dir='images',
    save_prefix='aug', save_format='png'):
  # create a grid of 3x3 images
  for i in range(0, 9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(X_batch[i].reshape(28, 28), cmap=pyplot.get_cmap('gray'))
  # show the plot
  pyplot.show()
  break
```

Listing 20.11: Save Augmented Images To File.

Running the example you can see that images are only written when they are generated.
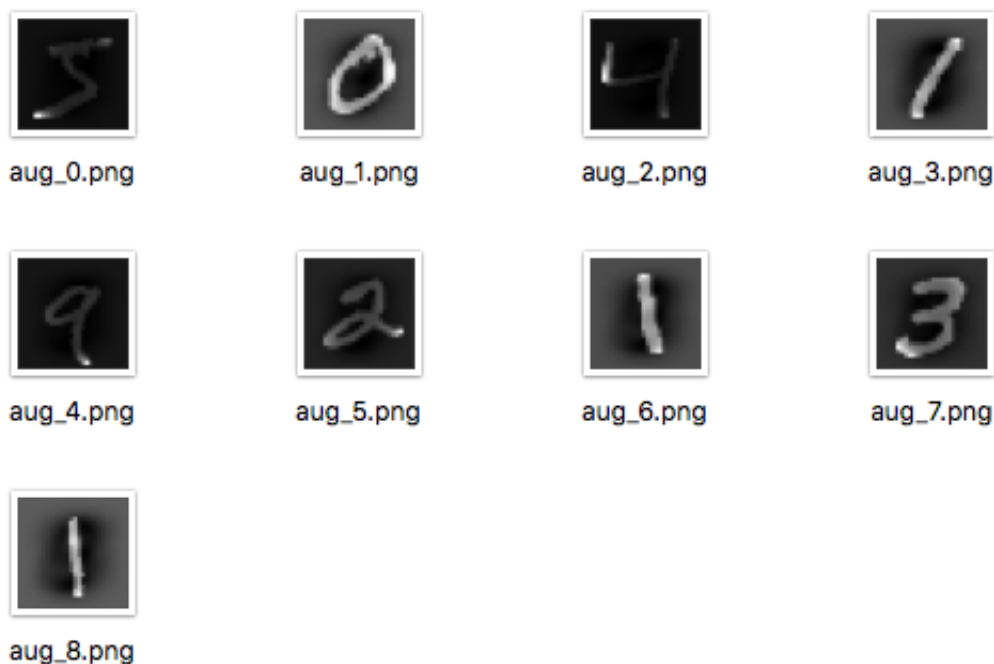
Figure 20.7: Randomly Flipped MNIST Images.

20.9   Tips For Augmenting Image Data with Keras

Image data is unique in that you can review the transformed copies of the data and quickly get an idea of how the model may be perceive it by your model. Below are some times for getting the most from image data preparation and augmentation for deep learning.

- **Review Dataset**. Take some time to review your dataset in great detail. Look at the images. Take note of image preparation and augmentations that might benefit the training process of your model, such as the need to handle different shifts, rotations or flips of objects in the scene.

- **Review Augmentations**. Review sample images after the augmentation has been performed. It is one thing to intellectually know what image transforms you are using, it is a very different thing to look at examples. Review images both with individual augmentations you are using as well as the full set of augmentations you plan to use in aggregate. You may see ways to simplify or further enhance your model training process.

- **Evaluate a Suite of Transforms**. Try more than one image data preparation and augmentation scheme. Often you can be surprised by results of a data preparation scheme you did not think would be beneficial.

20.10 Summary

In this lesson you discovered image data preparation and augmentation. You discovered a range of techniques that you can use easily in Python with Keras for deep learning models. You learned about:

- The `ImageDataGenerator` API in Keras for generating transformed images just-in-time.

- Feature-wise pixel standardization.

- The ZCA whitening transform.

- Random rotations, shifts and flips of images.

- How to save transformed images to file for later reuse.

20.10.1 Next

You now know how to develop convolutional neural networks and use the image augmentation API in Keras. In the next chapter you will work through developing larger and deeper models for a more complex object recognition task using Keras.

Chapter 21

Project Object Recognition in Photographs

A difficult problem where traditional neural networks fall down is called object recognition. It is where a model is able to identify objects in images. In this lesson you will discover how to develop and evaluate deep learning models for object recognition in Keras. After completing this step-by-step tutorial, you will know:

- About the CIFAR-10 object recognition dataset and how to load and use it in Keras.

- How to create a simple Convolutional Neural Network for object recognition.

- How to lift performance by creating deeper Convolutional Neural Networks.

Let's get started.

Note: You may want to speed up the computation for this tutorial by using GPU rather than CPU hardware, such as the process described in Chapter 5. This is a suggestion, not a requirement. The tutorial will work just fine on the CPU.

21.1   Photograph Object Recognition Dataset

The problem of automatically identifying objects in photographs is difficult because of the near infinite number of permutations of objects, positions, lighting and so on. It's a really hard problem. This is a well studied problem in computer vision and more recently an important demonstration of the capability of deep learning. A standard computer vision and deep learning dataset for this problem was developed by the Canadian Institute for Advanced Research (CIFAR).

The CIFAR-10 dataset consists of 60,000 photos divided into 10 classes (hence the name CIFAR-10)[1]. Classes include common objects such as airplanes, automobiles, birds, cats and so on. The dataset is split in a standard way, where 50,000 images are used for training a model and the remaining 10,000 for evaluating its performance. The photos are in color with red, green and blue channels, but are small measuring 32×32 pixel squares.

---

[1] http://www.cs.toronto.edu/~kriz/cifar.html

State-of-the-art results can be achieved using very large convolutional neural networks. You can learn about state-of-the-art results on CIFAR-10 on Rodrigo Benenson's webpage[2]. Model performance is reported in classification accuracy, with very good performance above 90% with human performance on the problem at 94% and state-of-the-art results at 96% at the time of writing.

21.2 Loading The CIFAR-10 Dataset in Keras

The CIFAR-10 dataset can easily be loaded in Keras. Keras has the facility to automatically download standard datasets like CIFAR-10 and store them in the `~/.keras/datasets` directory using the `cifar10.load_data()` function. This dataset is large at 163 megabytes, so it may take a few minutes to download. Once downloaded, subsequent calls to the function will load the dataset ready for use.

The dataset is stored as Python pickled training and test sets, ready for use in Keras. Each image is represented as a three dimensional matrix, with dimensions for red, green, blue, width and height. We can plot images directly using the Matplotlib Python plotting library.

```python
# Plot ad hoc CIFAR10 instances
from keras.datasets import cifar10
from matplotlib import pyplot
from scipy.misc import toimage
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# create a grid of 3x3 images
for i in range(0, 9):
  pyplot.subplot(330 + 1 + i)
  pyplot.imshow(toimage(X_train[i]))
# show the plot
pyplot.show()
```

Listing 21.1: Load And Plot Sample CIFAR-10 Images.

Running the code create a 3×3 plot of photographs. The images have been scaled up from their small 32×32 size, but you can clearly see trucks horses and cars. You can also see some distortion in the images that have been forced to the square aspect ratio.

---

[2]http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

Figure 21.1: Small Sample of CIFAR-10 Images.

21.3   Simple CNN for CIFAR-10

The CIFAR-10 problem is best solved using a convolutional neural network (CNN). We can quickly start off by importing all of the classes and functions we will need in this example.

```
# Simple CNN model for CIFAR-10
import numpy
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.constraints import maxnorm
from keras.optimizers import SGD
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
```

Listing 21.2: Load Classes and Functions.

As is good practice, we next initialize the random number seed with a constant to ensure the results are reproducible.

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

Listing 21.3: Initialize Random Number Generator.

Next we can load the CIFAR-10 dataset.

```
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

Listing 21.4: Load the CIFAR-10 Dataset.

The pixel values are in the range of 0 to 255 for each of the red, green and blue channels. It is good practice to work with normalized data. Because the input values are well understood, we can easily normalize to the range 0 to 1 by dividing each value by the maximum observation which is 255. Note, the data is loaded as integers, so we must cast it to floating point values in order to perform the division.

```
# normalize inputs from 0-255 to 0.0-1.0
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0
```

Listing 21.5: Normalize the CIFAR-10 Dataset.

The output variables are defined as a vector of integers from 0 to 1 for each class. We can use a one hot encoding to transform them into a binary matrix in order to best model the classification problem. We know there are 10 classes for this problem, so we can expect the binary matrix to have a width of 10.

```
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Listing 21.6: One Hot Encode The Output Variable.

Let's start off by defining a simple CNN structure as a baseline and evaluate how well it performs on the problem. We will use a structure with two convolutional layers followed by max pooling and a flattening out of the network to fully connected layers to make predictions. Our baseline network structure can be summarized as follows:

1. Convolutional input layer, 32 feature maps with a size of 3×3, a rectifier activation function and a weight constraint of max norm set to 3.

2. Dropout set to 20%.

3. Convolutional layer, 32 feature maps with a size of 3×3, a rectifier activation function and a weight constraint of max norm set to 3.

4. Max Pool layer with the size 2×2.

5. Flatten layer.

6. Fully connected layer with 512 units and a rectifier activation function.

7. Dropout set to 50%.

8. Fully connected output layer with 10 units and a softmax activation function.

A logarithmic loss function is used with the stochastic gradient descent optimization algorithm configured with a large momentum and weight decay, starting with a learning rate of 0.01. A visualization of the network structure is provided below.
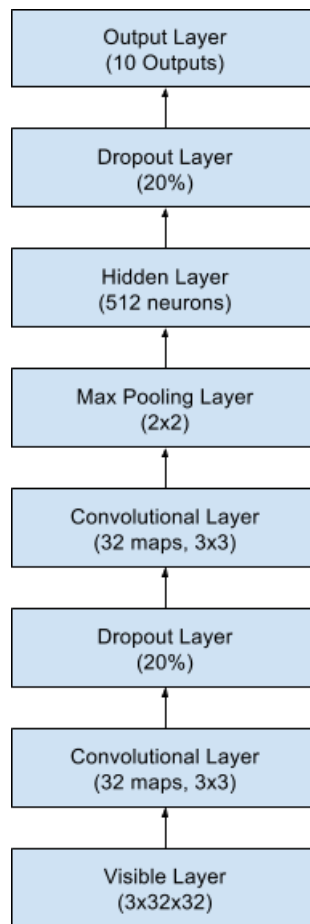
Figure 21.2: Summary of the Convolutional Neural Network Structure.

```python
# Create the model
model = Sequential()
model.add(Convolution2D(32, 3, 3, input_shape=(3, 32, 32), border_mode='same',
    activation='relu', W_constraint=maxnorm(3)))
model.add(Dropout(0.2))
model.add(Convolution2D(32, 3, 3, activation='relu', border_mode='same',
    W_constraint=maxnorm(3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(512, activation='relu', W_constraint=maxnorm(3)))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
# Compile model
epochs = 25
```

```
lrate = 0.01
decay = lrate/epochs
sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
print(model.summary())
```

Listing 21.7: Define and Compile the CNN Model.

We fit this model with 25 epochs and a batch size of 32. A small number of epochs was chosen to help keep this tutorial moving. Normally the number of epochs would be one or two orders of magnitude larger for this problem. Once the model is fit, we evaluate it on the test dataset and print out the classification accuracy.

```
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=epochs,
    batch_size=32, verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 21.8: Evaluate the Accuracy of the CNN Model.

The full code listing is provided below for completeness.

```
# Simple CNN model for the CIFAR-10 Dataset
import numpy
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.constraints import maxnorm
from keras.optimizers import SGD
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# normalize inputs from 0-255 to 0.0-1.0
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# Create the model
model = Sequential()
model.add(Convolution2D(32, 3, 3, input_shape=(3, 32, 32), border_mode='same',
    activation='relu', W_constraint=maxnorm(3)))
model.add(Dropout(0.2))
model.add(Convolution2D(32, 3, 3, activation='relu', border_mode='same',
    W_constraint=maxnorm(3)))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(512, activation='relu', W_constraint=maxnorm(3)))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
# Compile model
epochs = 25
lrate = 0.01
decay = lrate/epochs
sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
print(model.summary())
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=epochs,
    batch_size=32, verbose=2)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 21.9: CNN Model for CIFAR-10 Problem.

The classification accuracy and loss is printed each epoch on both the training and test datasets. The model is evaluated on the test set and achieves an accuracy of 69.76%, which is good but not excellent.

```
Epoch 21/25
24s - loss: 0.3583 - acc: 0.8721 - val_loss: 1.0083 - val_acc: 0.6936
Epoch 22/25
24s - loss: 0.3360 - acc: 0.8818 - val_loss: 0.9994 - val_acc: 0.6991
Epoch 23/25
24s - loss: 0.3215 - acc: 0.8854 - val_loss: 1.0237 - val_acc: 0.6994
Epoch 24/25
24s - loss: 0.3040 - acc: 0.8930 - val_loss: 1.0289 - val_acc: 0.7016
Epoch 25/25
24s - loss: 0.2879 - acc: 0.8990 - val_loss: 1.0600 - val_acc: 0.6976
Accuracy: 69.76%
```

Listing 21.10: Sample Output for the CNN Model.

We can improve the accuracy significantly by creating a much deeper network. This is what we will look at in the next section.

21.4 Larger CNN for CIFAR-10

We have seen that a simple CNN performs poorly on this complex problem. In this section we look at scaling up the size and complexity of our model. Let's design a deep version of the simple CNN above. We can introduce an additional round of convolutions with many more feature maps. We will use the same pattern of Convolutional, Dropout, Convolutional and Max Pooling layers.

This pattern will be repeated 3 times with 32, 64, and 128 feature maps. The effect will be an increasing number of feature maps with a smaller and smaller size given the max pooling layers. Finally an additional and larger Dense layer will be used at the output end of the network in an attempt to better translate the large number feature maps to class values. We can summarize a new network architecture as follows:

1. Convolutional input layer, 32 feature maps with a size of 3×3 and a rectifier activation function.

2. Dropout layer at 20%.

3. Convolutional layer, 32 feature maps with a size of 3×3 and a rectifier activation function.

4. Max Pool layer with size 2×2.

5. Convolutional layer, 64 feature maps with a size of 3×3 and a rectifier activation function.

6. Dropout layer at 20%.

7. Convolutional layer, 64 feature maps with a size of 3×3 and a rectifier activation function.

8. Max Pool layer with size 2×2.

9. Convolutional layer, 128 feature maps with a size of 3×3 and a rectifier activation function.

10. Dropout layer at 20%.

11. Convolutional layer, 128 feature maps with a size of 3×3 and a rectifier activation function.

12. Max Pool layer with size 2×2.

13. Flatten layer.

14. Dropout layer at 20%.

15. Fully connected layer with 1,024 units and a rectifier activation function.

16. Dropout layer at 20%.

17. Fully connected layer with 512 units and a rectifier activation function.

18. Dropout layer at 20%.

19. Fully connected output layer with 10 units and a softmax activation function.

This is a larger network and a bit unwieldy to visualize. We can fit and evaluate this model using the same procedure above and the same number of epochs but a larger batch size of 64, found through some minor experimentation.

```python
# Large CNN model for the CIFAR-10 Dataset
import numpy
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.constraints import maxnorm
from keras.optimizers import SGD
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D
from keras.utils import np_utils
```

```python
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
# normalize inputs from 0-255 to 0.0-1.0
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
# Create the model
model = Sequential()
model.add(Convolution2D(32, 3, 3, input_shape=(3, 32, 32), activation='relu',
    border_mode='same'))
model.add(Dropout(0.2))
model.add(Convolution2D(32, 3, 3, activation='relu', border_mode='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Convolution2D(64, 3, 3, activation='relu', border_mode='same'))
model.add(Dropout(0.2))
model.add(Convolution2D(64, 3, 3, activation='relu', border_mode='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Convolution2D(128, 3, 3, activation='relu', border_mode='same'))
model.add(Dropout(0.2))
model.add(Convolution2D(128, 3, 3, activation='relu', border_mode='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(1024, activation='relu', W_constraint=maxnorm(3)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu', W_constraint=maxnorm(3)))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))
# Compile model
epochs = 25
lrate = 0.01
decay = lrate/epochs
sgd = SGD(lr=lrate, momentum=0.9, decay=decay, nesterov=False)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
print(model.summary())
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=epochs,
    batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 21.11: Large CNN Model for CIFAR-10 Problem.

Running this example prints the classification accuracy and loss on the training and test datasets each epoch. The estimate of classification accuracy for the final model is 78.77% which is nearly 10 points better than our simpler model.

```
Epoch 21/25
50000/50000 [==============================] - 34s - loss: 0.5154 - acc: 0.8150 - val_loss:
    0.6428 - val_acc: 0.7801
Epoch 22/25
50000/50000 [==============================] - 34s - loss: 0.4968 - acc: 0.8228 - val_loss:
    0.6338 - val_acc: 0.7854
Epoch 23/25
50000/50000 [==============================] - 34s - loss: 0.4851 - acc: 0.8283 - val_loss:
    0.6350 - val_acc: 0.7849
Epoch 24/25
50000/50000 [==============================] - 34s - loss: 0.4663 - acc: 0.8340 - val_loss:
    0.6376 - val_acc: 0.7823
Epoch 25/25
50000/50000 [==============================] - 34s - loss: 0.4544 - acc: 0.8389 - val_loss:
    0.6289 - val_acc: 0.7877
Accuracy: 78.77%
```

Listing 21.12: Sample Output for Fitting the Larger CNN Model.

21.5 Extensions To Improve Model Performance

We have achieved good results on this very difficult problem, but we are still a good way from achieving world class results. Below are some ideas that you can try to extend upon the model and improve model performance.

- **Train for More Epochs**. Each model was trained for a very small number of epochs, 25. It is common to train large convolutional neural networks for hundreds or thousands of epochs. I would expect that performance gains can be achieved by significantly raising the number of training epochs.

- **Image Data Augmentation**. The objects in the image vary in their position. Another boost in model performance can likely be achieved by using some data augmentation. Methods such as standardization and random shifts and horizontal image flips may be beneficial.

- **Deeper Network Topology**. The larger network presented is deep, but larger networks could be designed for the problem. This may involve more feature maps closer to the input and perhaps less aggressive pooling. Additionally, standard convolutional network topologies that have been shown useful may be adopted and evaluated on the problem.

What accuracy can you achieve on this problem?

21.6 Summary

In this lesson you discovered how to create deep learning models in Keras for object recognition in photographs. After working through this tutorial you learned:

- About the CIFAR-10 dataset and how to load it in Keras and plot ad hoc examples from the dataset.

- How to train and evaluate a simple Convolutional Neural Network on the problem.

- How to expand a simple convolutional neural network into a deep convolutional neural network in order to boost performance on the difficult problem.

- How to use data augmentation to get a further boost on the difficult object recognition problem.

21.6.1 Next

The CIFAR-10 dataset does present a difficult challenge and you now know how to develop much larger convolutional neural networks. A clever feature of this type of network is that they can be used to learn the spatial structure in other domains such as in sequences of words. In the next chapter you will work through the application of a one-dimensional convolutional neural network to a natural language processing problem of sentiment classification.

Chapter 22

Project: Predict Sentiment From Movie Reviews

Sentiment analysis is a natural language processing problem where text is understood and the underlying intent is predicted. In this lesson you will discover how you can predict the sentiment of movie reviews as either positive or negative in Python using the Keras deep learning library. After completing this step-by-step tutorial, you will know:

- About the IMDB sentiment analysis problem for natural language processing and how to load it in Keras.

- How to use word embedding in Keras for natural language problems.

- How to develop and evaluate a Multilayer Perceptron model for the IMDB problem.

- How to develop a one-dimensional convolutional neural network model for the IMDB problem.

Let's get started.

22.1 Movie Review Sentiment Classification Dataset

The dataset used in this project is the Large Movie Review Dataset often referred to as the IMDB dataset[1]. The Large Movie Review Dataset (often referred to as the IMDB dataset) contains 25,000 highly-polar movie reviews (good or bad) for training and the same amount again for testing. The problem is to determine whether a given moving review has a positive or negative sentiment.

The data was collected by Stanford researchers and was used in a 2011 paper where a split of 50-50 of the data was used for training and test[2]. An accuracy of 88.89% was achieved.

---

[1]http://ai.stanford.edu/~amaas/data/sentiment/
[2]http://ai.stanford.edu/~amaas/papers/wvSent_acl2011.pdf

22.2 Load the IMDB Dataset With Keras

Keras provides access to the IMDB dataset built-in[3]. The `imdb.load_data()` function allows you to load the dataset in a format that is ready for use in neural network and deep learning models. The words have been replaced by integers that indicate the absolute popularity of the word in the dataset. The sentences in each review are therefore comprised of a sequence of integers.

Calling `imdb.load_data()` the first time will download the IMDB dataset to your computer and store it in your home directory under `~/.keras/datasets/imdb.pkl` as a 32 megabyte file. Usefully, the `imdb.load_data()` function provides additional arguments including the number of top words to load (where words with a lower integer are marked as zero in the returned data), the number of top words to skip (to avoid the `the`'s), the maximum length of reviews to support and the split of the dataset into training and test sets. Let's load the dataset and calculate some properties of it. We will start off by loading some libraries and loading the entire IMDB dataset as a training dataset.

```python
import numpy
from keras.datasets import imdb
from matplotlib import pyplot
# load the dataset
(X_train, y_train), (X_test, y_test) = imdb.load_data(test_split=0)
```
Listing 22.1: Load the IMDB Dataset.

Next we can display the shape of the training dataset.

```python
# summarize size
print("Training data: ")
print(X_train.shape)
print(y_train.shape)
```
Listing 22.2: Display The Shape of the IMDB Dataset.

Running this snippet, we can see that there are 25,000 records.

```
Training data:
(25000,)
(25000,)
```
Listing 22.3: Output of the Shape of the IMDB Dataset.

We can also print the unique class values.

```python
# Summarize number of classes
print("Classes: ")
print(numpy.unique(y_train))
```
Listing 22.4: Display The Classes in the IMDB Dataset.

We can see that it is a binary classification problem for good and bad sentiment in the review.

```
Classes:
[0 1]
```
Listing 22.5: Output of the Classes of the IMDB Dataset.

---

[3]http://keras.io/datasets/

Next we can get an idea of the total number of unique words in the dataset.

```
# Summarize number of words
print("Number of words: ")
print(len(numpy.unique(numpy.hstack(X_train))))
```

Listing 22.6: Display The Number of Unique Words in the IMDB Dataset.

Interestingly, we can see that there are just over 100,000 words across the entire dataset.

```
Number of words:
102099
```

Listing 22.7: Output of the Classes of Unique Words in the IMDB Dataset.

Finally, we can get an idea of the average review length.

```
# Summarize review length
print("Review length: ")
result = map(len, X_train)
print("Mean %.2f words (%f)" % (numpy.mean(result), numpy.std(result)))
# plot review length as a boxplot and histogram
pyplot.subplot(121)
pyplot.boxplot(result)
pyplot.subplot(122)
pyplot.hist(result)
pyplot.show()
```

Listing 22.8: Plot the distribution of Review Lengths.

We can see that the average review has just under 300 words with a standard deviation of just over 200 words.

```
Review length:
Mean 285.84 words (212.622320)
```

Listing 22.9: Output of the Distribution Summary for Review Length.

The full code listing is provided below for completeness.

```
# Load and Plot the IMDB dataset
import numpy
from keras.datasets import imdb
from matplotlib import pyplot
# load the dataset
(X_train, y_train), (X_test, y_test) = imdb.load_data(test_split=0)
# summarize size
print("Training data: ")
print(X_train.shape)
print(y_train.shape)
# Summarize number of classes
print("Classes: ")
print(numpy.unique(y_train))
# Summarize number of words
print("Number of words: ")
print(len(numpy.unique(numpy.hstack(X_train))))
# Summarize review length
print("Review length: ")
result = map(len, X_train)
```

```
print("Mean %.2f words (%f)" % (numpy.mean(result), numpy.std(result)))
# plot review length as a boxplot and histogram
pyplot.subplot(121)
pyplot.boxplot(result)
pyplot.subplot(122)
pyplot.hist(result)
pyplot.show()
```

Listing 22.10: Summarize the IMDB Dataset.

Looking at the box and whisker plot and the histogram for the review lengths in words, we can probably see an exponential distribution that we can probably cover the mass of the distribution with a clipped length of 400 to 500 words.
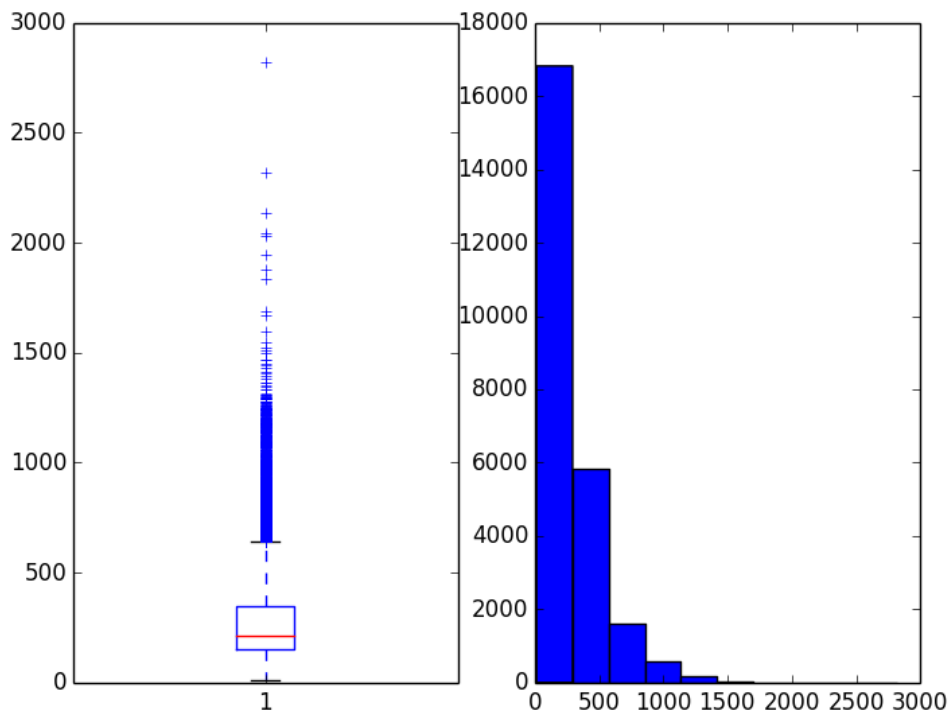
Figure 22.1: Review Length in Words for IMDB Dataset.

22.3 Word Embeddings

A recent breakthrough in the field of natural language processing is called word embedding. This is a technique where words are encoded as real-valued vectors in a high dimensional space, where the similarity between words in terms of meaning translates to closeness in the vector space. Discrete words are mapped to vectors of continuous numbers. This is useful when working with natural language problems with neural networks as we require numbers as input values.

Keras provides a convenient way to convert positive integer representations of words into a

word embedding by an `Embedding` layer[4]. The layer takes arguments that define the mapping including the maximum number of expected words also called the vocabulary size (e.g. the largest integer value that will be seen as an input). The layer also allows you to specify the dimensionality for each word vector, called the output dimension.

We would like to use a word embedding representation for the IMDB dataset. Let's say that we are only interested in the first 5,000 most used words in the dataset. Therefore our vocabulary size will be 5,000. We can choose to use a 32-dimensional vector to represent each word. Finally, we may choose to cap the maximum review length at 500 words, truncating reviews longer than that and padding reviews shorter than that with 0 values. We would load the IMDB dataset as follows:

```
imdb.load_data(nb_words=5000, test_split=0.33)
```

Listing 22.11: Only Load the Top 5,000 words in the IMDB Review.

We would then use the Keras utility to truncate or pad the dataset to a length of 500 for each observation using the `sequence.pad_sequences()` function.

```
X_train = sequence.pad_sequences(X_train, maxlen=500)
X_test = sequence.pad_sequences(X_test, maxlen=500)
```

Listing 22.12: Pad Reviews in the IMDB Dataset.

Finally, later on, the first layer of our model would be an word embedding layer created using the `Embedding` class as follows:

```
Embedding(5000, 32, input_length=500)
```

Listing 22.13: Define a Word Embedding Representation.

The output of this first layer would be a matrix with the size 32×500 for a given movie review training or test pattern in integer format. Now that we know how to load the IMDB dataset in Keras and how to use a word embedding representation for it, let's develop and evaluate some models.

22.4 Simple Multilayer Perceptron Model

We can start off by developing a simple Multilayer Perceptron model with a single hidden layer. The word embedding representation is a true innovation and we will demonstrate what would have been considered world class results in 2011 with a relatively simple neural network. Let's start off by importing the classes and functions required for this model and initializing the random number generator to a constant value to ensure we can easily reproduce the results.

```
# MLP for the IMDB problem
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
# fix random seed for reproducibility
```

---

[4]http://keras.io/layers/embeddings/

```
seed = 7
numpy.random.seed(seed)
```

Listing 22.14: Load Classes and Functions and Seed Random Number Generator.

Next we will load the IMDB dataset. We will simplify the dataset as discussed during the section on word embeddings. Only the top 5,000 words will be loaded. We will also use a 67%/33% split of the dataset into training and test. This is a good standard split methodology.

```
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
test_split = 0.33
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=top_words,
    test_split=test_split)
```

Listing 22.15: Load and sPlit the IMDB Dataset.

We will bound reviews at 500 words, truncating longer reviews and zero-padding shorter reviews.

```
max_words = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_words)
X_test = sequence.pad_sequences(X_test, maxlen=max_words)
```

Listing 22.16: Pad IMDB Reviews to a Fixed Length.

Now we can create our model. We will use an `Embedding` layer as the input layer, setting the vocabulary to 5,000, the word vector size to 32 dimensions and the `input_length` to 500. The output of this first layer will be a 32×500 sized matrix as discussed in the previous section. We will flatten the Embedding layers output to one dimension, then use one dense hidden layer of 250 units with a rectifier activation function. The output layer has one neuron and will use a sigmoid activation to output values of 0 and 1 as predictions. The model uses logarithmic loss and is optimized using the efficient ADAM optimization procedure.

```
# create the model
model = Sequential()
model.add(Embedding(top_words, 32, input_length=max_words))
model.add(Flatten())
model.add(Dense(250, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
```

Listing 22.17: Define a Multilayer Perceptron Model.

We can fit the model and use the test set as validation while training. This model overfits very quickly so we will use very few training epochs, in this case just 2. There is a lot of data so we will use a batch size of 128. After the model is trained, we evaluate it's accuracy on the test dataset.

```
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=2, batch_size=128,
    verbose=1)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 22.18: Fit and Evaluate the Multilayer Perceptron Model.

The full code listing is provided below for completeness.

```python
# MLP for the IMDB problem
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
test_split = 0.33
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=top_words,
    test_split=test_split)
max_words = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_words)
X_test = sequence.pad_sequences(X_test, maxlen=max_words)
# create the model
model = Sequential()
model.add(Embedding(top_words, 32, input_length=max_words))
model.add(Flatten())
model.add(Dense(250, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=2, batch_size=128,
    verbose=1)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 22.19: Multilayer Perceptron Model for the IMDB Dataset.

Running this example fits the model and summarizes the estimated performance. We can see that this very simple model achieves a score of 86.27% which is in the neighborhood of the original paper, with very little effort.

```
Accuracy: 86.27%
```

Listing 22.20: Output from Evaluating the Multilayer Perceptron Model.

I'm sure we can do better if we trained this network, perhaps using a larger embedding and adding more hidden layers. Let's try a different network type.

22.5 One-Dimensional Convolutional Neural Network

Convolutional neural networks were designed to honor the spatial structure in image data whilst being robust to the position and orientation of learned objects in the scene. This same principle can be used on sequences, such as the one-dimensional sequence of words in a movie review. The same properties that make the CNN model attractive for learning to recognize objects in

images can help to learn structure in paragraphs of words, namely the techniques invariance to the specific position of features.

Keras supports one dimensional convolutions and pooling by the `Convolution1D` and `MaxPooling1D` classes respectively. Again, let's import the classes and functions needed for this example and initialize our random number generator to a constant value so that we can easily reproduce results.

```python
# CNN for the IMDB problem
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Convolution1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

Listing 22.21: Import Classes and Functions and Seed Random Number Generator.

We can also load and prepare our IMDB dataset as we did before.

```python
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
test_split = 0.33
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=top_words,
    test_split=test_split)
# pad dataset to a maximum review length in words
max_words = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_words)
X_test = sequence.pad_sequences(X_test, maxlen=max_words)
```

Listing 22.22: Load, Split and Pad IMDB Dataset.

We can now define our convolutional neural network model. This time, after the `Embedding` input layer, we insert a `Convolution1D` layer. This convolutional layer has 32 feature maps and reads embedded word representations 3 vector elements of the word embedding at a time. The convolutional layer is followed by a `MaxPooling1D` layer with a length and stride of 2 that halves the size of the feature maps from the convolutional layer. The rest of the network is the same as the neural network above.

```python
# create the model
model = Sequential()
model.add(Embedding(top_words, 32, input_length=max_words))
model.add(Convolution1D(nb_filter=32, filter_length=3, border_mode='same',
    activation='relu'))
model.add(MaxPooling1D(pool_length=2))
model.add(Flatten())
model.add(Dense(250, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
```

Listing 22.23: Define the CNN Model.

We also fit the network the same as before.

```python
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=2, batch_size=128,
    verbose=1)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 22.24: Fit and Evaluate the CNN Model.

Running the example, we are first presented with a summary of the network structure (not shown here). We can see our convolutional layer preserves the dimensionality of our `Embedding` input layer of 32 dimensional input with a maximum of 500 words. The pooling layer compresses this representation by halving it. Running the example offers a small but welcome improvement over the neural network model above with an accuracy of nearly 88.5%.

The full code listing is provided below for completeness.

```python
# CNN for the IMDB problem
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Convolution1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
test_split = 0.33
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=top_words,
    test_split=test_split)
# pad dataset to a maximum review length in words
max_words = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_words)
X_test = sequence.pad_sequences(X_test, maxlen=max_words)
# create the model
model = Sequential()
model.add(Embedding(top_words, 32, input_length=max_words))
model.add(Convolution1D(nb_filter=32, filter_length=3, border_mode='same',
    activation='relu'))
model.add(MaxPooling1D(pool_length=2))
model.add(Flatten())
model.add(Dense(250, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=2, batch_size=128,
    verbose=1)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
```

```
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 22.25: CNN Model for the IMDB Dataset.

```
Accuracy: 88.48%
```

Listing 22.26: Output from Evaluating the CNN Model.

Again, there is a lot of opportunity for further optimization, such as the use of deeper and/or larger convolutional layers. One interesting idea is to set the max pooling layer to use an input length of 500. This would compress each feature map to a single 32 length vector and may boost performance.

22.6 Summary

In this lesson you discovered the IMDB sentiment analysis dataset for natural language processing. You learned how to develop deep learning models for sentiment analysis including:

- How to load and review the IMDB dataset within Keras.

- How to develop a large neural network model for sentiment analysis.

- How to develop a one-dimensional convolutional neural network model for sentiment analysis.

This tutorial concludes Part V and your introduction to convolutional neural networks in Keras. Next in Part VI we will discover a different type of neural network intended to learn and predict sequences called recurrent neural networks.

Part VI

Recurrent Neural Networks

Chapter 23

Crash Course In Recurrent Neural Networks

There is another type of neural network that is dominating difficult machine learning problems that involve sequences of inputs called recurrent neural networks. Recurrent neural networks have connections that have loops, adding feedback and memory to the networks over time. This memory allows this type of network to learn and generalize across sequences of inputs rather than individual patterns.

A powerful type of Recurrent Neural Network called the Long Short-Term Memory Network has been shown to be particularly effective when stacked into a deep configuration, achieving state-of-the-art results on a diverse array of problems from language translation to automatic captioning of images and videos. In this lesson you will get a crash course in recurrent neural networks for deep learning, acquiring just enough understanding to start using LSTM networks in Python with Keras. After reading this lesson, you will know:

- The limitations of Multilayer Perceptrons that are addressed by recurrent neural networks.

- The problems that must be addressed to make Recurrent Neural networks useful.

- The details of the Long Short-Term Memory networks used in applied deep learning.

Let's get started.

23.1   Support For Sequences in Neural Networks

There are some problem types that are best framed involving either a sequence as an input or an output. For example, consider a univariate time series problem, like the price of a stock over time. This dataset can be framed as a prediction problem for a classical feedforward Multilayer Perceptron network by defining a windows size (e.g. 5) and training the network to learn to make short term predictions from the fixed sized window of inputs.

This would work, but is very limited. The window of inputs adds memory to the problem, but is limited to just a fixed number of points and must be chosen with sufficient knowledge of the problem. A naive window would not capture the broader trends over minutes, hours and days that might be relevant to making a prediction. From one prediction to the next, the network only knows about the specific inputs it is provided. Univariate time series prediction

is important, but there are even more interesting problems that involve sequences. Consider the following taxonomy of sequence problems that require a mapping of an input to an output (taken from Andrej Karpathy[1]).

- **One-to-Many**: sequence output, for image captioning.

- **Many-to-One**: sequence input, for sentiment classification.

- **Many-to-Many**: sequence in and out, for machine translation.

- **Synchronized Many-to-Many**: synced sequences in and out, for video classification.

We can also see that a one-to-one example of input to output would be an example of a classical feedforward neural network for a prediction task like image classification. Support for sequences in neural networks is an important class of problem and one where deep learning has recently shown impressive results State-of-the art results have been using a type of network specifically designed for sequence problems called recurrent neural networks.

23.2 Recurrent Neural Networks

Recurrent Neural Networks or RNNs are a special type of neural network designed for sequence problems. Given a standard feedforward Multilayer Perceptron network, a recurrent neural network can be thought of as the addition of loops to the architecture. For example, in a given layer, each neuron may pass its signal latterly (sideways) in addition to forward to the next layer. The output of the network may feedback as an input to the network with the next input vector. And so on.

The recurrent connections add state or memory to the network and allow it to learn broader abstractions from the input sequences. The field of recurrent neural networks is well established with popular methods. For the techniques to be effective on real problems, two major issues needed to be resolved for the network to be useful.

1. How to train the network with Back propagation.

2. How to stop gradients vanishing or exploding during training.

23.2.1 How to Train Recurrent Neural Networks

The staple technique for training feedforward neural networks is to back propagate error and update the network weights. Back propagation breaks down in a recurrent neural network, because of the recurrent or loop connections. This was addressed with a modification of the Back propagation technique called Back propagation Through Time or BPTT.

Instead of performing back propagation on the recurrent network as stated, the structure of the network is unrolled, where copies of the neurons that have recurrent connections are created. For example a single neuron with a connection to itself ($A \rightarrow A$) could be represented as two neurons with the same weight values ($A \rightarrow B$). This allows the cyclic graph of a recurrent neural network to be turned into an acyclic graph like a classic feedforward neural network, and Back propagation can be applied.

---

[1]http://karpathy.github.io/2015/05/21/rnn-effectiveness/

23.2.2   How to Have Stable Gradients During Training

When Back propagation is used in very deep neural networks and in unrolled recurrent neural networks, the gradients that are calculated in order to update the weights can become unstable. They can become very large numbers called exploding gradients or very small numbers called the vanishing gradient problem. These large numbers in turn are used to update the weights in the network, making training unstable and the network unreliable.

This problem is alleviated in deep Multilayer Perceptron networks through the use of the Rectifier transfer function, and even more exotic but now less popular approaches of using unsupervised pre-training of layers. In recurrent neural network architectures, this problem has been alleviated using a new type of architecture called the Long Short-Term Memory Networks that allows deep recurrent networks to be trained.

23.3   Long Short-Term Memory Networks

The Long Short-Term Memory or LSTM network is a recurrent neural network that is trained using Back propagation Through Time and overcomes the vanishing gradient problem. As such it can be used to create large (stacked) recurrent networks, that in turn can be used to address difficult sequence problems in machine learning and achieve state-of-the-art results. Instead of neurons, LSTM networks have memory blocks that are connected into layers.

A block has components that make it smarter than a classical neuron and a memory for recent sequences. A block contains gates that manage the block's state and output. A unit operates upon an input sequence and each gate within a unit uses the sigmoid activation function to control whether they are triggered or not, making the change of state and addition of information flowing through the unit conditional. There are three types of gates within a memory unit:

- **Forget Gate**: conditionally decides what information to discard from the unit.

- **Input Gate**: conditionally decides which values from the input to update the memory state.

- **Output Gate**: conditionally decides what to output based on input and the memory of the unit.

Each unit is like a mini state machine where the gates of the units have weights that are learned during the training procedure. You can see how you may achieve a sophisticated learning and memory from a layer of LSTMs, and it is not hard to imagine how higher-order abstractions may be layered with multiple such layers.

23.4   Summary

In this lesson you discovered sequence problems and recurrent neural networks that can be used to address them. Specifically, you learned:

- The limitations of classical feedforward neural networks and how recurrent neural networks can overcome these problems.

- The practical problems in training recurrent neural networks and how they are overcome.

- The Long Short-Term Memory network used to create deep recurrent neural networks.

23.4.1 Next

In this lesson you took a crash course in recurrent neural networks. In the next lesson you will discover how you can tackle simple time series prediction problems with Multilayer Perceptron models, a precursor to working with recurrent networks.

Chapter 24

Time Series Prediction with Multilayer Perceptrons

Time Series prediction is a difficult problem both to frame and to address with machine learning. In this lesson you will discover how to develop neural network models for time series prediction in Python using the Keras deep learning library. After reading this lesson you will know:

- About the airline passengers univariate time series prediction problem.

- How to phrase time series prediction as a regression problem and develop a neural network model for it.

- How to frame time series prediction with a time lag and develop a neural network model for it.

Let's get started.

24.1   Problem Description: Time Series Prediction

The problem we are going to look at in this lesson is the international airline passengers prediction problem. This is a problem where given a year and a month, the task is to predict the number of international airline passengers in units of 1,000. The data ranges from January 1949 to December 1960 or 12 years, with 144 observations. The dataset is available for free from the DataMarket webpage as a CSV download[1] with the filename `international-airline-passengers.csv`. Below is a sample of the first few lines of the file.

```
"Month","International airline passengers: monthly totals in thousands. Jan 49 ? Dec 60"
"1949-01",112
"1949-02",118
"1949-03",132
"1949-04",129
"1949-05",121
```

Listing 24.1: Sample Output From Evaluating the Baseline Model.

---

[1]https://goo.gl/fpwEXk

We can load this dataset easily using the Pandas library. We are not interested in the date, given that each observation is separated by the same interval of one month. Therefore when we load the dataset we can exclude the first column. The downloaded dataset also has footer information that we can exclude with the `skipfooter` argument to `pandas.read_csv()` set to 3 for the 3 footer lines. Once loaded we can easily plot the whole dataset. The code to load and plot the dataset is listed below.

```python
import pandas
import matplotlib.pyplot as plt
dataset = pandas.read_csv('international-airline-passengers.csv', usecols=[1],
    engine='python', skipfooter=3)
plt.plot(dataset)
plt.show()
```

Listing 24.2: Load and Plot the Time Series dataset.

You can see an upward trend in the plot. You can also see some periodicity to the dataset that probably corresponds to the northern hemisphere summer holiday period.
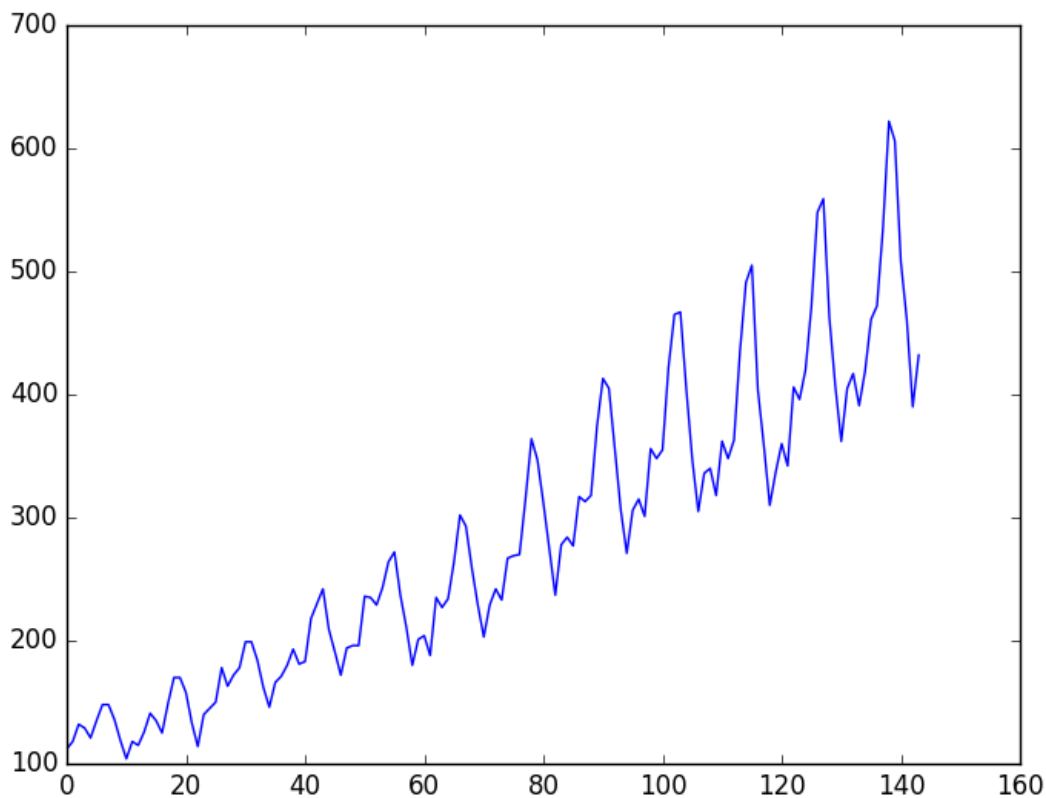
Figure 24.1: Plot of the Airline Passengers Dataset

We are going to keep things simple and work with the data as-is. Normally, it is a good idea to investigate various data preparation techniques to rescale the data and to make it stationary.

24.2 Multilayer Perceptron Regression

We will phrase the time series prediction problem as a regression problem. That is, given the number of passengers (in units of thousands) this month, what is the number of passengers next month. We can write a simple function to convert our single column of data into a two-column dataset. The first column containing this month's (`t`) passenger count and the second column containing next month's (`t+1`) passenger count, to be predicted. Before we get started, let's first import all of the functions and classes we intend to use.

```python
import numpy
import matplotlib.pyplot as plt
import pandas
from keras.models import Sequential
from keras.layers import Dense
# fix random seed for reproducibility
numpy.random.seed(7)
```

Listing 24.3: Import Classes and Functions.

We can also use the code from the previous section to load the dataset as a Pandas dataframe. We can then extract the NumPy array from the dataframe and convert the integer values to floating point values which are more suitable for modeling with a neural network.

```python
# load the dataset
dataframe = pandas.read_csv('international-airline-passengers.csv', usecols=[1],
    engine='python', skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype('float32')
```

Listing 24.4: Load the Time Series Dataset.

After we model our data and estimate the skill of our model on the training dataset, we need to get an idea of the skill of the model on new unseen data. For a normal classification or regression problem we would do this using k-fold cross validation. With time series data, the sequence of values is important. A simple method that we can use is to split the ordered dataset into train and test datasets. The code below calculates the index of the split point and separates the data into the training datasets with 67% of the observations that we can use to train our model, leaving the remaining 33% for testing the model.

```python
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
print(len(train), len(test))
```

Listing 24.5: Split Dataset into Train and Test.

Now we can define a function to create a new dataset as described above. The function takes two arguments, the dataset which is a NumPy array that we want to convert into a dataset and the **look_back** which is the number of previous time steps to use as input variables to predict the next time period, in this case, defaulted to 1. This default will create a dataset where X is the number of passengers at a given time (`t`) and Y is the number of passengers at the next time (`t+1`). It can be configured and we will look at constructing a differently shaped dataset in the next section.

```
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
  dataX, dataY = [], []
  for i in range(len(dataset)-look_back-1):
    a = dataset[i:(i+look_back), 0]
    dataX.append(a)
    dataY.append(dataset[i + look_back, 0])
  return numpy.array(dataX), numpy.array(dataY)
```

Listing 24.6: Function to Prepare Dataset For Modeling.

Let's take a look at the effect of this function on the first few rows of the dataset.

```
X    Y
112  118
118  132
132  129
129  121
121  135
```

Listing 24.7: Sample of the Prepared Dataset.

If you compare these first 5 rows to the original dataset sample listed in the previous section, you can see the X=t and Y=t+1 pattern in the numbers. Let's use this function to prepare the train and test datasets ready for modeling.

```
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
```

Listing 24.8: Call Function to Prepare Dataset For Modeling.

We can now fit a Multilayer Perceptron model to the training data. We use a simple network with 1 input, 1 hidden layer with 8 neurons and an output layer. The model is fit using mean squared error, which if we take the square root gives us an error score in the units of the dataset. I tried a few rough parameters and settled on the configuration below, but by no means is the network listed optimized.

```
# create and fit Multilayer Perceptron model
model = Sequential()
model.add(Dense(8, input_dim=look_back, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, nb_epoch=200, batch_size=2, verbose=2)
```

Listing 24.9: Define and Fit Multilayer Perceptron Model.

Once the model is fit, we can estimate the performance of the model on the train and test datasets. This will give us a point of comparison for new models.

```
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: ', trainScore)
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: ', testScore)
```

Listing 24.10: Evaluate the Fit Model.

Finally, we can generate predictions using the model for both the train and test dataset to get a visual indication of the skill of the model. Because of how the dataset was prepared, we must shift the predictions so that they align on the x-axis with the original dataset. Once prepared, the data is plotted, showing the original dataset in blue, the predictions for the train dataset in green the predictions on the unseen test dataset in red.

```python
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)

# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict

# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict

# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 24.11: Generate and Plot Predictions.

We can see that the model did an excellent job of fitting both the training and the test datasets.
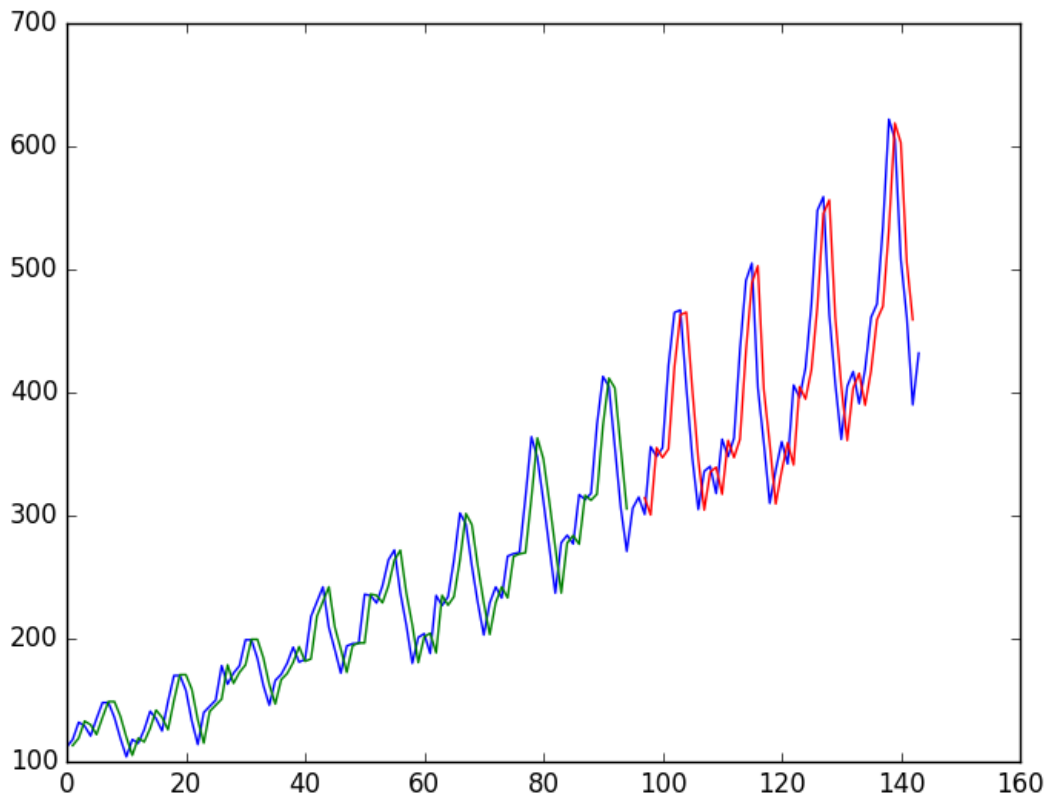
Figure 24.2: Number of Passengers Predicted Using a Simple Multilayer Perceptron Model. Blue=Whole Dataset, Green=Training, Red=Predictions.

For completeness, below is the entire code listing.

```python
# Multilayer Perceptron to Predict International Airline Passengers (t+1, given t)
import numpy
import matplotlib.pyplot as plt
import pandas
from keras.models import Sequential
from keras.layers import Dense
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
  dataX, dataY = [], []
  for i in range(len(dataset)-look_back-1):
    a = dataset[i:(i+look_back), 0]
    dataX.append(a)
    dataY.append(dataset[i + look_back, 0])
  return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = pandas.read_csv('international-airline-passengers.csv', usecols=[1],
    engine='python', skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype('float32')
```

```python
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
print(len(train), len(test))
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# create and fit Multilayer Perceptron model
model = Sequential()
model.add(Dense(8, input_dim=look_back, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, nb_epoch=200, batch_size=2, verbose=2)
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: ', trainScore)
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: ', testScore)
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 24.12: Multilayer Perceptron Model for the t+1 Model.

Running the model produces the following output.

```
...
Epoch 195/200
0s - loss: 551.1626
Epoch 196/200
0s - loss: 542.7755
Epoch 197/200
0s - loss: 539.6731
Epoch 198/200
0s - loss: 539.1133
Epoch 199/200
0s - loss: 539.8144
Epoch 200/200
0s - loss: 539.8541
('Train Score: ', 531.45189520653253)
('Test Score: ', 2353.351849099864)
```

Listing 24.13: Sample output of the Multilayer Perceptron Model for the t+1 Model.

Taking the square root of the performance estimates, we can see that the model has an average error of 23 passengers (in thousands) on the training dataset and 48 passengers (in thousands) on the test dataset.

24.3 Multilayer Perceptron Using the Window Method

We can also phrase the problem so that multiple recent time steps can be used to make the prediction for the next time step. This is called the window method, and the size of the window is a parameter that can be tuned for each problem. For example, given the current time (`t`) we want to predict the value at the next time in the sequence (`t+1`), we can use the current time (`t`) as well as the two prior times (`t-1` and `t-2`). When phrased as a regression problem the input variables are `t-2`, `t-1`, `t` and the output variable is `t+1`.

The `create_dataset()` function we wrote in the previous section allows us to create this formulation of the time series problem by increasing the `look_back` argument from 1 to 3. A sample of the dataset with this formulation looks as follows:

```
X1  X2  X3  Y
112 118 132 129
118 132 129 121
132 129 121 135
129 121 135 148
121 135 148 148
```

Listing 24.14: Sample Dataset of the Window Formulation of the Problem.

We can re-run the example in the previous section with the larger window size. The whole code listing with just the window size change is listed below for completeness.

```python
# Multilayer Perceptron to Predict International Airline Passengers (t+1, given t, t-1, t-2)
import numpy
import matplotlib.pyplot as plt
import pandas
from keras.models import Sequential
from keras.layers import Dense
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
  dataX, dataY = [], []
  for i in range(len(dataset)-look_back-1):
    a = dataset[i:(i+look_back), 0]
    dataX.append(a)
    dataY.append(dataset[i + look_back, 0])
  return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = pandas.read_csv('international-airline-passengers.csv', usecols=[1],
    engine='python', skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype('float32')
# split into train and test sets
```

```
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
print(len(train), len(test))
# reshape dataset
look_back = 10
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# create and fit Multilayer Perceptron model
model = Sequential()
model.add(Dense(8, input_dim=look_back, activation='relu'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, nb_epoch=200, batch_size=2, verbose=2)
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: ', trainScore)
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: ', testScore)
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 24.15: Multilayer Perceptron Model for the Window Model.

Running the example provides the following output.

```
...
Epoch 194/200
0s - loss: 497.1802
Epoch 195/200
0s - loss: 455.6591
Epoch 196/200
0s - loss: 477.5430
Epoch 197/200
0s - loss: 495.3042
Epoch 198/200
0s - loss: 479.2633
Epoch 199/200
0s - loss: 473.2058
Epoch 200/200
0s - loss: 460.4878
('Train Score: ', 454.45418648097825)
('Test Score: ', 1954.8213112571023)
```

Listing 24.16: Sample output of the Multilayer Perceptron Model for the Window Model.

We can see that the error was reduced compared to that of the previous section. Again, the window size and the network architecture were not tuned, this is just a demonstration of how to frame a prediction problem. Taking the square root of the performance scores we can see the average error on the training dataset was 21 passengers (in thousands per month) and the average error on the unseen test set was 44 passengers (in thousands per month).
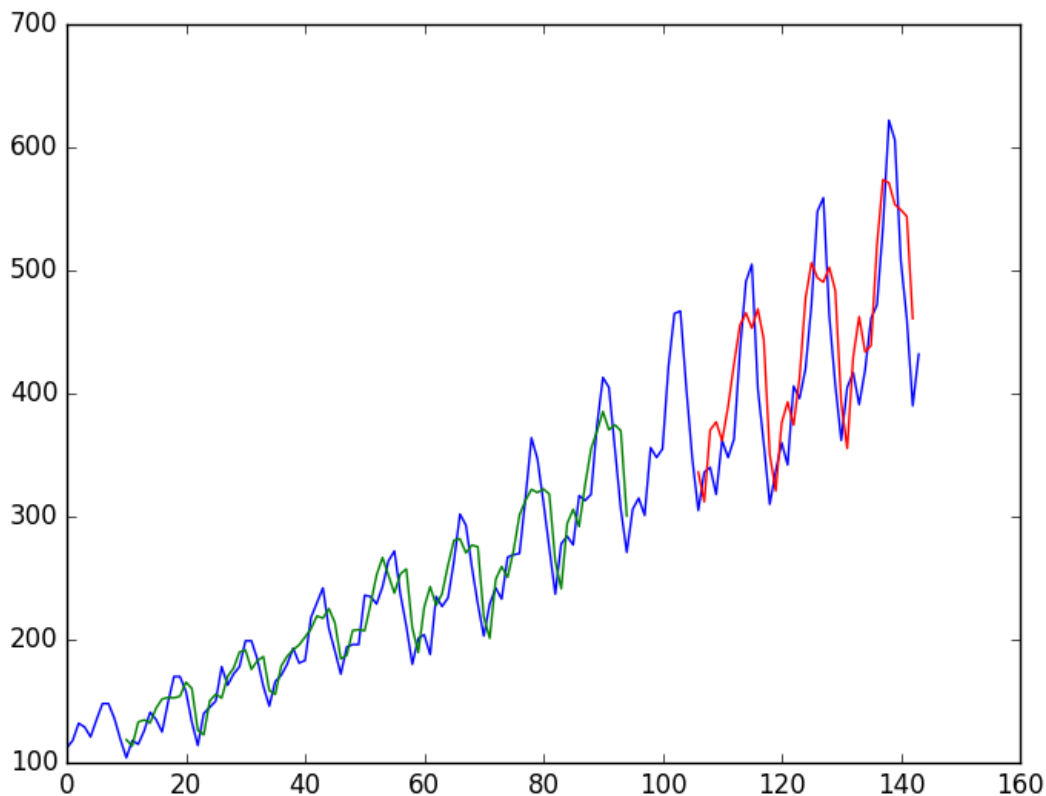
Figure 24.3: Prediction of the Number of Passengers using a Simple Multilayer Perceptron Model With Time Lag. Blue=Whole Dataset, Green=Training, Red=Predictions.

24.4 Summary

In this lesson you discovered how to develop a neural network model for a time series prediction problem using the Keras deep learning library. After working through this tutorial you now know:

- About the international airline passenger prediction time series dataset.

- How to frame time series prediction problems as a regression problems and develop a neural network model.

- How use the window approach to frame a time series prediction problem and develop a neural network model.

24.4.1   Next

In this lesson you learned how you can develop simple Multilayer Perceptron models for times series prediction. In the next lesson you will learn how we can build on this understanding and use LSTM recurrent neural networks for time series prediction.

Chapter 25

Time Series Prediction with LSTM Recurrent Neural Networks

Time series prediction problems are a difficult type of predictive modeling problem. Unlike regression predictive modeling, time series also adds the complexity of a sequence dependence among the input variables. A powerful type of neural network designed to handle sequence dependence are called recurrent neural networks. The Long Short-Term Memory Networks or LSTM network is a type of recurrent neural network used in deep learning because very large architectures can be successfully trained.

In this lesson you will discover how to develop LSTM networks in Python using the Keras deep learning library to address a demonstration time series prediction problem. After completing this tutorial you will know how to implement and develop LSTM networks for your own time series prediction problems and other more general sequence problems. You will know:

- How to develop LSTM networks for a time series prediction problem framed as regression.

- How to develop LSTM networks for a time series prediction problem using a window for both features and time steps.

- How to develop and make predictions using LSTM networks that maintain state (memory) across very long sequences.

We will develop a number of LSTMs for a standard time series prediction problem. The problem and the chosen configuration for the LSTM networks are for demonstration purposes only, they are not optimized. These examples will show you exactly how you can develop your own LSTM networks for time series predictive modeling problems. Let's get started.

25.1 LSTM Network For Regression

The problem we are going to look at in this lesson is the international airline passengers prediction problem, described in Section 24.1. We can phrase the problem as a regression problem, as was done in the previous chapter. That is, given the number of passengers (in units of thousands) this month, what is the number of passengers next month. This example will reuse the same data loading and preparation from the previous chapter, specifically the use of the `create_dataset()` function.

185

LSTMs are sensitive to the scale of the input data, specifically when the sigmoid (default) or tanh activation functions are used. It can be a good practice to rescale the data to the range of 0-to-1, also called normalizing. We can easily normalize the dataset using the `MinMaxScaler` preprocessing class from the scikit-learn library.

```
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
```

Listing 25.1: Normalize the Dataset.

The LSTM network expects the input data (`X`) to be provided with a specific array structure in the form of: `[samples, time steps, features]`. Our prepared data is in the form: `[samples, features]` and we are framing the problem as one time step for each sample. We can transform the prepared train and test input data into the expected structure using `numpy.reshape()` as follows:

```
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

Listing 25.2: Reshape the Prepared Dataset for the LSTM Layers.

We are now ready to design and fit our LSTM network for this problem. The network has a visible layer with 1 input, a hidden layer with 4 LSTM blocks or neurons and an output layer that makes a single value prediction. The default sigmoid activation function is used for the LSTM memory blocks. The network is trained for 100 epochs and a batch size of 1 is used.

```
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_dim=look_back))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, nb_epoch=100, batch_size=1, verbose=2)
```

Listing 25.3: Define and Fit the LSTM network.

For completeness, below is the entire code example.

```
# LSTM for international airline passengers problem with regression framing
import numpy
import matplotlib.pyplot as plt
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
  dataX, dataY = [], []
  for i in range(len(dataset)-look_back-1):
    a = dataset[i:(i+look_back), 0]
    dataX.append(a)
    dataY.append(dataset[i + look_back, 0])
  return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
```

```python
# load the dataset
dataframe = pandas.read_csv('international-airline-passengers.csv', usecols=[1],
    engine='python', skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_dim=look_back))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, nb_epoch=100, batch_size=1, verbose=2)
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: ', scaler.inverse_transform(numpy.array([[trainScore]])))
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: ', scaler.inverse_transform(numpy.array([[testScore]])))
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 25.4: LSTM Model on the t+1 Problem.

Running the model produces the following output.

```
...
Epoch 95/100
0s - loss: 0.0020
Epoch 96/100
0s - loss: 0.0020
Epoch 97/100
```

```
0s - loss: 0.0020
Epoch 98/100
0s - loss: 0.0020
Epoch 99/100
0s - loss: 0.0021
Epoch 100/100
0s - loss: 0.0021
('Train Score: ', array([[ 105.03351108]]))
('Test Score: ', array([[ 108.83114314]]))
```

Listing 25.5: Sample Output From the LSTM Model on the t+1 Problem.

We can see that the model did an excellent job of fitting both the training and the test datasets.
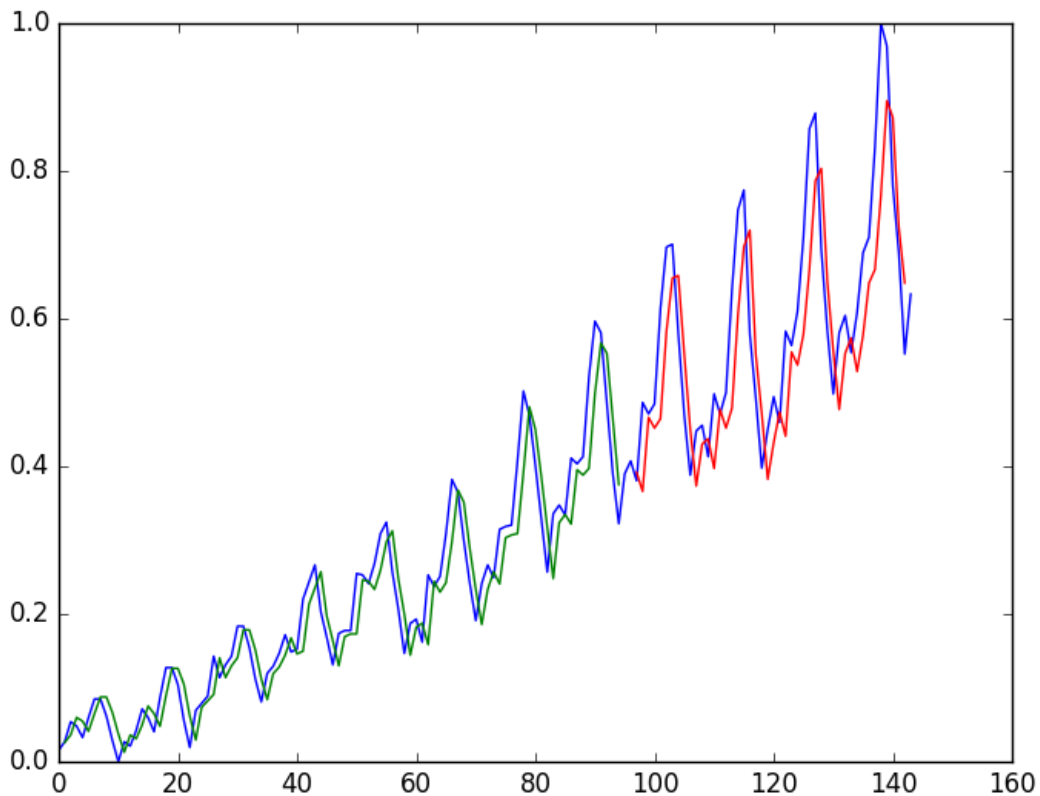
Figure 25.1: LSTM Trained on Regression Formulation of Passenger Prediction Problem

Taking the square root of the performance estimates, we can see that the model has an average error of about 10 passengers (in thousands) on the training dataset and about 10 passengers (in thousands) on the test dataset. Not bad at all.

25.2 LSTM For Regression Using the Window Method

We can also phrase the problem so that multiple recent time steps can be used to make the prediction for the next time step. This is called a window and the size of the window is a parameter that can be tuned for each problem. For example, given the current time (`t`) we want to predict the value at the next time in the sequence (`t+1`), we can use the current time (`t`) as well as the two prior times (`t-1` and `t-2`) as input variables. When phrased as a regression problem the input variables are `t-2`, t-1, `t` and the output variable is `t+1`.

The `create_dataset()` function we created in the previous section allows us to create this formulation of the time series problem by increasing the `look_back` argument from 1 to 3. A sample of the dataset with this formulation looks as follows:

```
X1 X2  X3  Y
112 118 132 129
118 132 129 121
132 129 121 135
129 121 135 148
121 135 148 148
```

Listing 25.6: Sample Dataset of the Window Formulation of the Problem.

We can re-run the example in the previous section with the larger window size. The whole code listing with just the window size change is listed below for completeness.

```python
# LSTM for international airline passengers problem with window regression framing
import numpy
import matplotlib.pyplot as plt
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
  dataX, dataY = [], []
  for i in range(len(dataset)-look_back-1):
    a = dataset[i:(i+look_back), 0]
    dataX.append(a)
    dataY.append(dataset[i + look_back, 0])
  return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = pandas.read_csv('international-airline-passengers.csv', usecols=[1],
    engine='python', skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
```

```
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_dim=look_back))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, nb_epoch=100, batch_size=1, verbose=2)
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: ', scaler.inverse_transform(numpy.array([[trainScore]])))
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: ', scaler.inverse_transform(numpy.array([[testScore]])))
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 25.7: LSTM for the Window Model.

Running the example provides the following output.

```
...
Epoch 95/100
0s - loss: 0.0020
Epoch 96/100
0s - loss: 0.0019
Epoch 97/100
0s - loss: 0.0020
Epoch 98/100
0s - loss: 0.0020
Epoch 99/100
0s - loss: 0.0020
Epoch 100/100
0s - loss: 0.0019
('Train Score: ', array([[ 105.00297435]]))
('Test Score: ', array([[ 108.59656682]]))
```

Listing 25.8: Sample Output From the LSTM Model on the Window Problem.

We can see that the error was reduced slightly compared to that of the previous section.

The window size and the network architecture were not tuned, this is just a demonstration of how to frame a prediction problem.
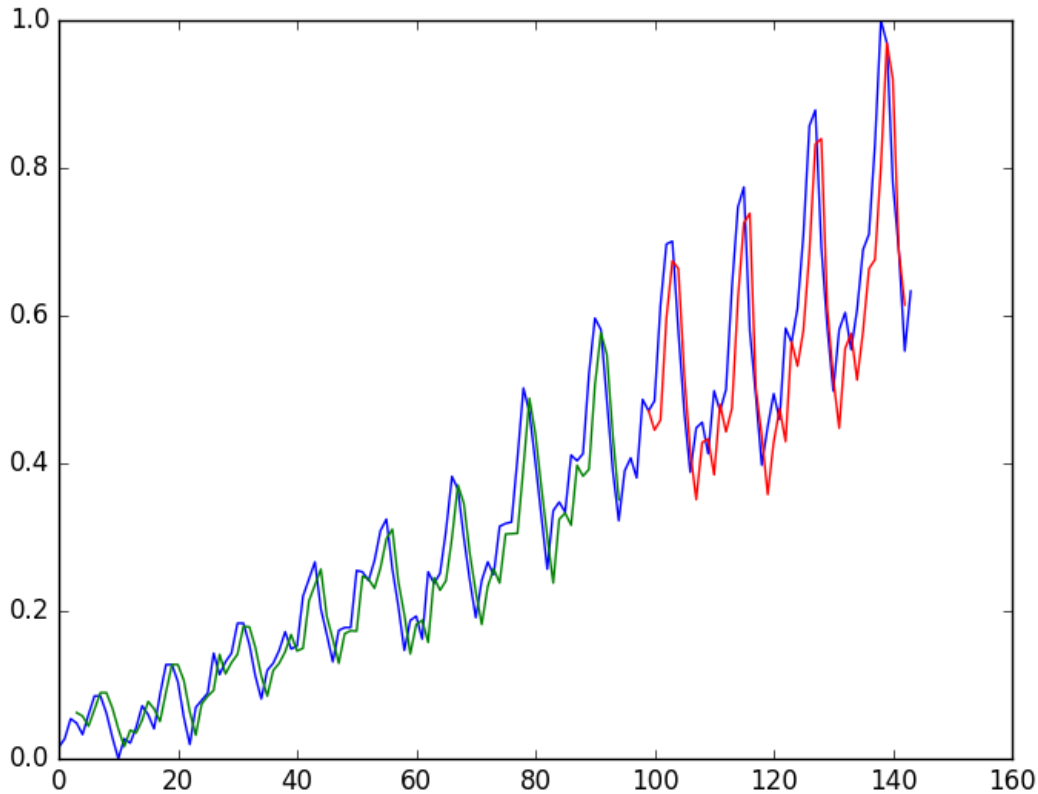
Figure 25.2: LSTM Trained on Window Formulation of Passenger Prediction Problem.

25.3 LSTM For Regression with Time Steps

You may have noticed that the data preparation for the LSTM network includes time steps. Some sequence problems may have a varied number of time steps per sample. For example, you may have measurements of a physical machine leading up to a point of failure or a point of surge. Each incident would be a sample, the observations that lead up to the event would be the time steps and the variables observed would be the features. Time steps provides another way to phrase our time series problem. Like above in the window example, we can take prior time steps in our time series as inputs to predict the output at the next time step.

Instead of phrasing the past observations as separate input features, we can use them as time steps of the one input feature, which is indeed a more accurate framing of the problem. We can do this using the same data representation as in the previous window-based example, except when we reshape the data we set the columns to be the time steps dimension and change the features dimension back to 1. For example:

```
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
```

```
testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))
```

Listing 25.9: Reshape the Prepared Dataset for the LSTM Layers with Time Steps.

The entire code listing is provided below for completeness.

```
# LSTM for international airline passengers problem with time step regression framing
import numpy
import matplotlib.pyplot as plt
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
  dataX, dataY = [], []
  for i in range(len(dataset)-look_back-1):
    a = dataset[i:(i+look_back), 0]
    dataX.append(a)
    dataY.append(dataset[i + look_back, 0])
  return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = pandas.read_csv('international-airline-passengers.csv', usecols=[1],
    engine='python', skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_dim=1))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, nb_epoch=100, batch_size=1, verbose=2)
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, verbose=0)
print('Train Score: ', scaler.inverse_transform(numpy.array([[trainScore]])))
testScore = model.evaluate(testX, testY, verbose=0)
print('Test Score: ', scaler.inverse_transform(numpy.array([[testScore]])))
# generate predictions for training
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
```

```
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 25.10: LSTM for the Time Steps Model.

Running the example provides the following output.

```
...
Epoch 95/100
0s - loss: 0.0023
Epoch 96/100
0s - loss: 0.0022
Epoch 97/100
0s - loss: 0.0022
Epoch 98/100
0s - loss: 0.0022
Epoch 99/100
0s - loss: 0.0023
Epoch 100/100
0s - loss: 0.0022
('Train Score: ', array([[ 105.11716663]]))
('Test Score: ', array([[ 110.4869926]]))
```

Listing 25.11: Sample Output From the LSTM Model on the Time Step Problem.

We can see that the results are generally on par with the previous example, although the structure of the input data makes more sense.
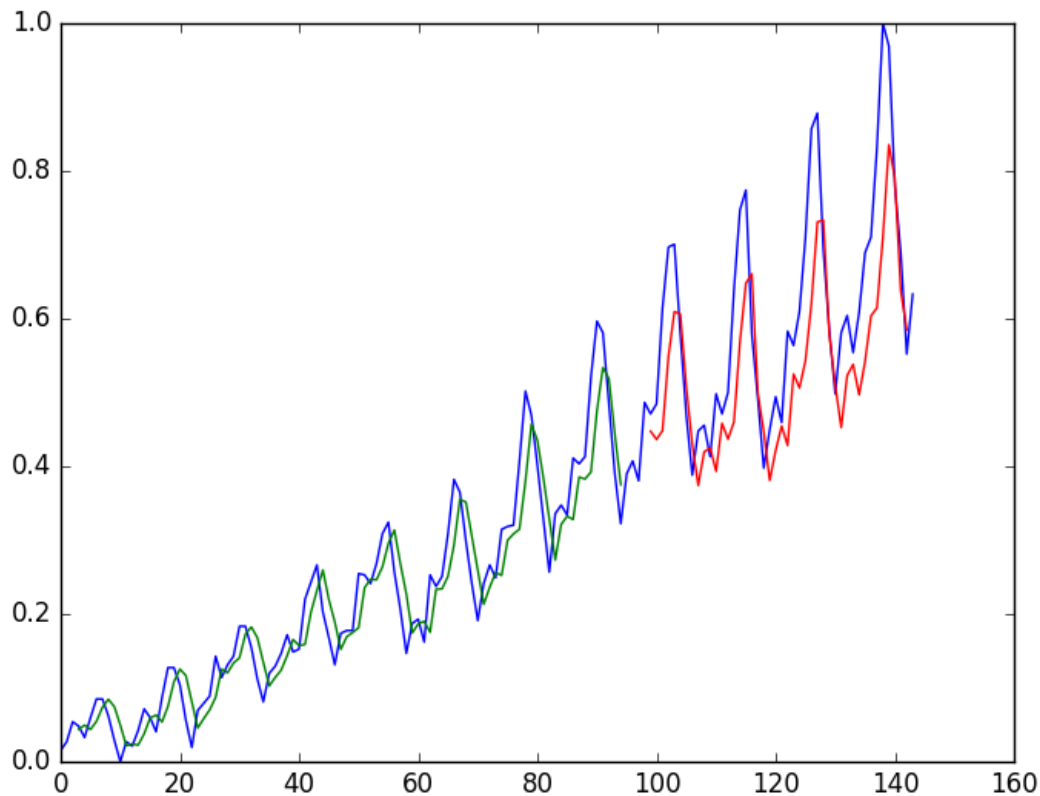
Figure 25.3: LSTM Trained on Time Step Formulation of Passenger Prediction Problem.

25.4 LSTM With Memory Between Batches

The LSTM network has memory which is capable of remembering across long sequences. Normally, the state within the network is reset after each training batch when fitting the model, as well as each call to `model.predict()` or `model.evaluate()`. We can gain finer control over when the internal state of the LSTM network is cleared in Keras by making the LSTM layer *stateful*. This means that it can build state over the entire training sequence and even maintain that state if needed to make predictions.

It requires that the training data not be shuffled when fitting the network. It also requires explicit resetting of the network state after each exposure to the training data (epoch) by calls to `model.reset_states()`. This means that we must create our own outer loop of epochs and within each epoch call `model.fit()` and `model.reset_states()`, for example:

```
for i in range(100):
  model.fit(trainX, trainY, nb_epoch=1, batch_size=batch_size, verbose=2, shuffle=False)
  model.reset_states()
```

Listing 25.12: Manually Reset LSTM State Between Epochs.

Finally, when the LSTM layer is constructed, the `stateful` parameter must be set to `True` and instead of specifying the input dimensions, we must hard code the number of samples in a

batch, number of time steps in a sample and number of features in a time step by setting the `batch_input_shape` parameter. For example:

```
model.add(LSTM(4, batch_input_shape=(batch_size, time_steps, features), stateful=True))
```

Listing 25.13: Set the Stateful Parameter on the LSTM Layer.

This same batch size must then be used later when evaluating the model and making predictions. For example:

```
model.predict(trainX, batch_size=batch_size)
```

Listing 25.14: Set Batch Size when making Predictions.

We can adapt the previous time step example to use a stateful LSTM. The full code listing is provided below.

```
# LSTM for international airline passengers problem with memory
import numpy
import matplotlib.pyplot as plt
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
  dataX, dataY = [], []
  for i in range(len(dataset)-look_back-1):
    a = dataset[i:(i+look_back), 0]
    dataX.append(a)
    dataY.append(dataset[i + look_back, 0])
  return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = pandas.read_csv('international-airline-passengers.csv', usecols=[1],
    engine='python', skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))
# create and fit the LSTM network
batch_size = 1
model = Sequential()
```

```
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
for i in range(100):
  model.fit(trainX, trainY, nb_epoch=1, batch_size=batch_size, verbose=2, shuffle=False)
  model.reset_states()
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, batch_size=batch_size, verbose=0)
print('Train Score: ', scaler.inverse_transform(numpy.array([[trainScore]])))
testScore = model.evaluate(testX, testY, batch_size=batch_size, verbose=0)
print('Test Score: ', scaler.inverse_transform(numpy.array([[testScore]])))
# generate predictions for training
trainPredict = model.predict(trainX, batch_size=batch_size)
testPredict = model.predict(testX, batch_size=batch_size)
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 25.15: LSTM for the Manual Management of State.

Running the example provides the following output:

```
...
0s - loss: 0.0019
Epoch 1/1
0s - loss: 0.0019
Epoch 1/1
0s - loss: 0.0019
Epoch 1/1
0s - loss: 0.0019
Epoch 1/1
0s - loss: 0.0019
('Train Score: ', array([[ 104.93952676]]))
('Test Score: ', array([[ 109.87094205]]))
```

Listing 25.16: Sample Output From the Stateful LSTM Model.

Again, we can see very similar skill to the prior model.
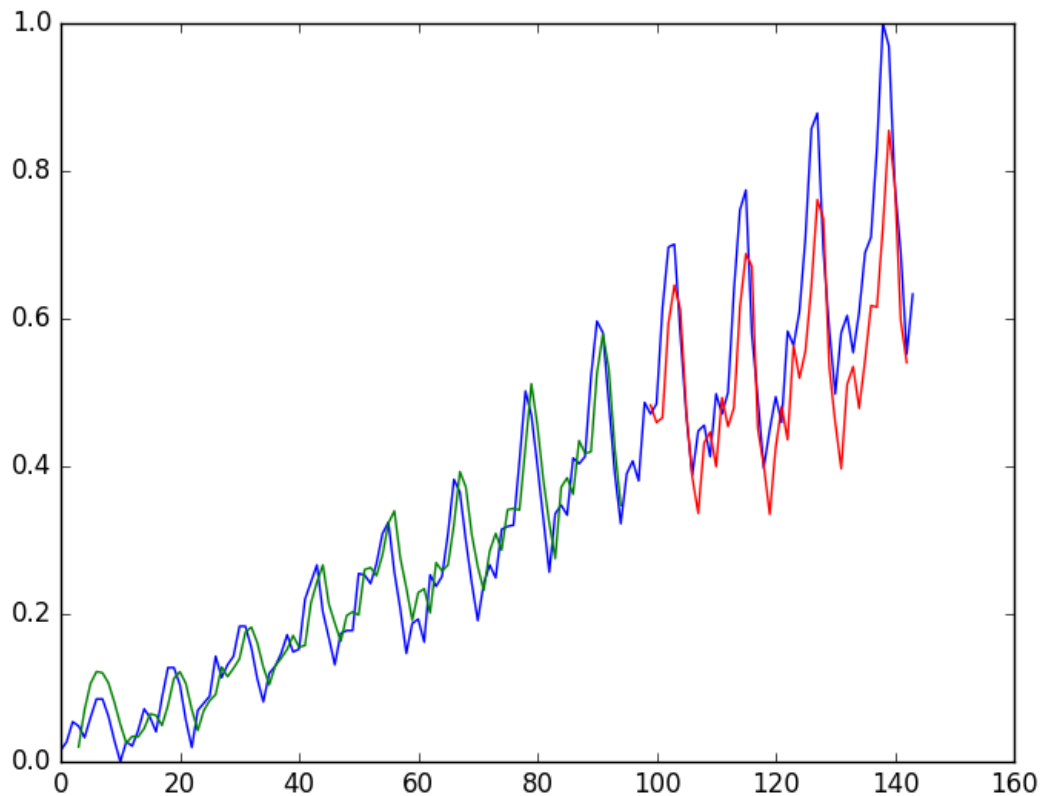
Figure 25.4: Stateful LSTM Trained on Regression Formulation of Passenger Prediction Problem.

25.5   Stacked LSTMs With Memory Between Batches

Finally, we will take a look at one of the big benefits of LSTMs, the fact that they can be successfully trained when stacked into deep network architectures. LSTM networks can be stacked in Keras in the same way that other layer types can be stacked. One addition to the configuration that is required is that an LSTM layer prior to each subsequent LSTM layer must return the sequence. This can be done by setting the `return_sequences` parameter on the layer to `True`. We can extend the stateful LSTM in the previous section to have two layers, as follows:

```
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True,
    return_sequences=True))
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True))
```

Listing 25.17: Define a Stacked LSTM Model.

The entire code listing is provided below for completeness.

```
# Stacked LSTM for international airline passengers problem with memory
import numpy
import matplotlib.pyplot as plt
import pandas
from keras.models import Sequential
```

```python
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
# convert an array of values into a dataset matrix
def create_dataset(dataset, look_back=1):
  dataX, dataY = [], []
  for i in range(len(dataset)-look_back-1):
    a = dataset[i:(i+look_back), 0]
    dataX.append(a)
    dataY.append(dataset[i + look_back, 0])
  return numpy.array(dataX), numpy.array(dataY)
# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
dataframe = pandas.read_csv('international-airline-passengers.csv', usecols=[1],
    engine='python', skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype('float32')
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
# reshape into X=t and Y=t+1
look_back = 3
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
testX = numpy.reshape(testX, (testX.shape[0], testX.shape[1], 1))
# create and fit the LSTM network
batch_size = 1
model = Sequential()
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True,
    return_sequences=True))
model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
for i in range(100):
  model.fit(trainX, trainY, nb_epoch=1, batch_size=batch_size, verbose=2, shuffle=False)
  model.reset_states()
# Estimate model performance
trainScore = model.evaluate(trainX, trainY, batch_size=batch_size, verbose=0)
model.reset_states()
print('Train Score: ', scaler.inverse_transform(numpy.array([[trainScore]])))
testScore = model.evaluate(testX, testY, batch_size=batch_size, verbose=0)
model.reset_states()
print('Test Score: ', scaler.inverse_transform(numpy.array([[testScore]])))
# generate predictions for training
trainPredict = model.predict(trainX, batch_size=batch_size)
model.reset_states()
testPredict = model.predict(testX, batch_size=batch_size)
model.reset_states()
# shift train predictions for plotting
```

```
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(dataset)
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
plt.show()
```

Listing 25.18: Stacked LSTM Model.

Running the example produces the following output.

```
...
Epoch 1/1
0s - loss: 0.0020
Epoch 1/1
0s - loss: 0.0020
Epoch 1/1
0s - loss: 0.0020
Epoch 1/1
0s - loss: 0.0020
Epoch 1/1
0s - loss: 0.0019
('Train Score: ', array([[ 105.15641323]]))
('Test Score: ', array([[ 118.18096732]]))
```

Listing 25.19: Sample Output From the Stacked LSTM Model.

Again, we can see that the results are generally equivalent to the previous examples.
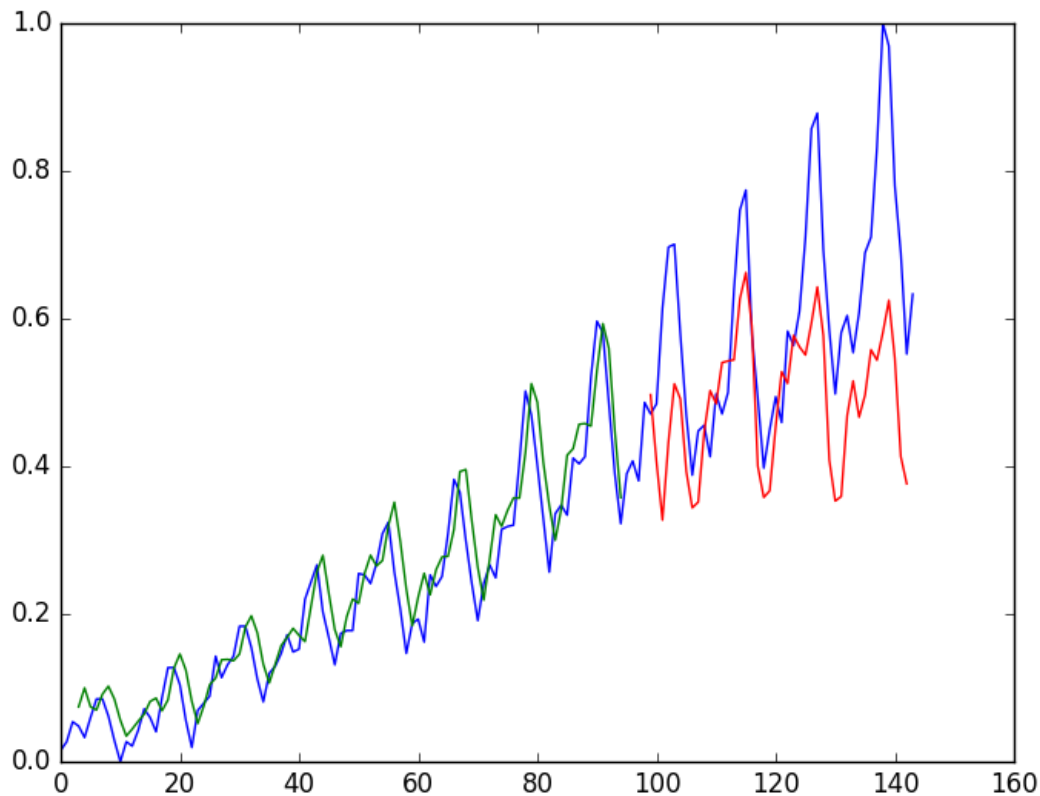
Figure 25.5: Stacked Stateful LSTMs Trained on Regression Formulation of Passenger Prediction Problem.

25.6 Summary

In this lesson you discovered how to develop LSTM recurrent neural networks for time series prediction in Python with the Keras deep learning network. Specifically, you learned:
    About the international airline passenger time series prediction problem.

- How to create an LSTM for a regression and a window formulation of the time series problem.

- How to create an LSTM with a time step formulation of the time series problem.

- How to create an LSTM with state and stacked LSTMs with state to learn long sequences.

25.6.1 Next

In this lesson you learned about how to use LSTM recurrent neural networks for time series prediction. Up next you will use your new found skill with LSTM networks to work on a sequence classification problem.

Chapter 26

Project: Sequence Classification of Movie Reviews

Sequence classification is a predictive modeling problem where you have some sequence of inputs over space or time and the task is to predict a category for the sequence. What makes this problem difficult is that the sequences can vary in length, be comprised of a very large vocabulary of input symbols and may require the model to learn the long term context or dependencies between symbols in the input sequence. In this project you will discover how you can develop LSTM recurrent neural network models for sequence classification problems in Python using the Keras deep learning library. After completing this project you will know:

- How to develop an LSTM model for a sequence classification problem.

- How to reduce overfitting in your LSTM models through the use of dropout.

- How to combine LSTM models with Convolutional Neural Networks that excel at learning spatial relationships.

Let's get started.

26.1   Simple LSTM for Sequence Classification

The problem that we will use to demonstrate sequence learning in this tutorial is the IMDB movie review sentiment classification problem, introduced in Section 22.1. We can quickly develop a small LSTM for the IMDB problem and achieve good accuracy. Let's start off by importing the classes and functions required for this model and initializing the random number generator to a constant value to ensure we can easily reproduce the results.

```
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
from theano.tensor.shared_randomstreams import RandomStreams
# fix random seed for reproducibility
```

```
numpy.random.seed(7)
srng = RandomStreams(7)
```

Listing 26.1: Import Classes and Functions and Seed Random Number Generator.

We need to load the IMDB dataset. We are constraining the dataset to the top 5,000 words. We also split the dataset into train (67%) and test (33%) sets.

```
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
test_split = 0.33
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=top_words,
    test_split=test_split)
```

Listing 26.2: Load and Split the Dataset.

Next, we need to truncate and pad the input sequences so that they are all the same length for modeling. The model will learn the zero values carry no information so indeed the sequences are not the same length in terms of content, but same length vectors is required to perform the computation in Keras.

```
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
```

Listing 26.3: Left-Pad Sequences to all be the Same Length.

We can now define, compile and fit our LSTM model. The first layer is the Embedded layer that uses 32 length vectors to represent each word. The next layer is the LSTM layer with 100 memory units (smart neurons). Finally, because this is a classification problem we use a Dense output layer with a single neuron and a sigmoid activation function to make 0 or 1 predictions for the two classes (good and bad) in the problem. Because it is a binary classification problem, log loss is used as the loss function (`binary_crossentropy` in Keras). The efficient ADAM optimization algorithm is used. The model is fit for only 2 epochs because it quickly overfits the problem. A large batch size of 64 reviews is used to space out weight updates.

```
# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, validation_data=(X_test, y_test), nb_epoch=3, batch_size=64)
```

Listing 26.4: Define and Fit LSTM Model for the IMDB Dataset.

Once fit, we estimate the performance of the model on unseen reviews.

```
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 26.5: Evaluate the Fit Model.

For completeness, here is the full code listing for this LSTM network on the IMDB dataset.

```python
# LSTM for sequence classification in the IMDB dataset
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
from theano.tensor.shared_randomstreams import RandomStreams
# fix random seed for reproducibility
numpy.random.seed(7)
srng = RandomStreams(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
test_split = 0.33
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=top_words,
    test_split=test_split)
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, nb_epoch=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 26.6: Simple LSTM Model for the IMDB Dataset.

Running this example produces the following output.

```
Epoch 1/3
16750/16750 [==============================] - 107s - loss: 0.5570 - acc: 0.7149
Epoch 2/3
16750/16750 [==============================] - 107s - loss: 0.3530 - acc: 0.8577
Epoch 3/3
16750/16750 [==============================] - 107s - loss: 0.2559 - acc: 0.9019
Accuracy: 86.79%
```

Listing 26.7: Output from Running the Simple LSTM Model.

You can see that this simple LSTM with little tuning achieves near state-of-the-art results on the IMDB problem. Importantly, this is a template that you can use to apply LSTM networks to your own sequence classification problems. Now, let's look at some extensions of this simple model that you may also want to bring to your own problems.

26.2   LSTM For Sequence Classification With Dropout

Recurrent Neural networks like LSTM generally have the problem of overfitting. Dropout can be applied between layers using the Dropout Keras layer. We can do this easily by adding new Dropout layers between the Embedding and LSTM layers and the LSTM and Dense output layers. We can also add dropout to the input on the Embedded layer by using the dropout parameter. For example:

```python
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length,
    dropout=0.2))
model.add(Dropout(0.2))
model.add(LSTM(100))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
```

Listing 26.8: Define LSTM Model with Dropout Layers.

The full code listing example above with the addition of Dropout layers is as follows:

```python
# LSTM with Dropout for sequence classification in the IMDB dataset
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
from theano.tensor.shared_randomstreams import RandomStreams
# fix random seed for reproducibility
numpy.random.seed(7)
srng = RandomStreams(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
test_split = 0.33
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=top_words,
    test_split=test_split)
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length,
    dropout=0.2))
model.add(Dropout(0.2))
model.add(LSTM(100))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, nb_epoch=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
```

```
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 26.9: LSTM Model with Dropout Layers for the IMDB Dataset.

Running this example provides the following output.

```
Epoch 1/3
16750/16750 [==============================] - 108s - loss: 0.5802 - acc: 0.6898
Epoch 2/3
16750/16750 [==============================] - 108s - loss: 0.4112 - acc: 0.8232
Epoch 3/3
16750/16750 [==============================] - 108s - loss: 0.3825 - acc: 0.8365
Accuracy: 85.56%
```

Listing 26.10: Output from Running the LSTM Model with Dropout Layers.

We can see dropout having the desired impact on training with a slightly slower trend in convergence and in this case a lower final accuracy. The model could probably use a few more epochs of training and may achieve a higher skill (try it an see). Alternately, dropout can be applied to the input and recurrent connections of the memory units with the LSTM precisely and separately. Keras provides this capability with parameters on the LSTM layer, the `dropout_W` for configuring the input dropout and `dropout_U` for configuring the recurrent dropout. For example, we can modify the first example to add dropout to the input and recurrent connections as follows:

```
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length,
    dropout=0.2))
model.add(LSTM(100, dropout_W=0.2, dropout_U=0.2))
model.add(Dense(1, activation='sigmoid'))
```

Listing 26.11: Define LSTM Model with Dropout on Gates.

The full code listing with more precise LSTM dropout is listed below for completeness.

```
# LSTM with dropout for sequence classification in the IMDB dataset
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
from theano.tensor.shared_randomstreams import RandomStreams
# fix random seed for reproducibility
numpy.random.seed(7)
srng = RandomStreams(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
test_split = 0.33
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=top_words,
    test_split=test_split)
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
```

```
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length,
    dropout=0.2))
model.add(LSTM(100, dropout_W=0.2, dropout_U=0.2))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, nb_epoch=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 26.12: LSTM Model with Dropout on Gates for the IMDB Dataset.

Running this example provides the following output.

```
Epoch 1/3
16750/16750 [==============================] - 112s - loss: 0.6623 - acc: 0.5935
Epoch 2/3
16750/16750 [==============================] - 113s - loss: 0.5159 - acc: 0.7484
Epoch 3/3
16750/16750 [==============================] - 113s - loss: 0.4502 - acc: 0.7981
Accuracy: 82.82%
```

Listing 26.13: Output from Running the LSTM Model with Dropout on the Gates.

We can see that the LSTM specific dropout has a more pronounced effect on the convergence of the network than the layer-wise dropout. As above, the number of epochs was kept constant and could be increased to see if the skill of the model can be further lifted. Dropout is a powerful technique for combating overfitting in your LSTM models and it is a good idea to try both methods, but you may bet better results with the gate-specific dropout provided in Keras.

26.3 LSTM and CNN For Sequence Classification

Convolutional neural networks excel at learning the spatial structure in input data. The IMDB review data does have a one-dimensional spatial structure in the sequence of words in reviews and the CNN may be able to pick out invariant features for good and bad sentiment. This learned spatial features may then be learned as sequences by an LSTM layer. We can easily add a one-dimensional CNN and max pooling layers after the Embedding layer which then feed the consolidated features to the LSTM. We can use a smallish set of 32 features with a small filter length of 3. The pooling layer can use the standard length of 2 to halve the feature map size. For example, we would create the model as follows:

```
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(Convolution1D(nb_filter=32, filter_length=3, border_mode='same',
    activation='relu'))
model.add(MaxPooling1D(pool_length=2))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
```

Listing 26.14: Define LSTM and CNN Model.

The full code listing with a CNN and LSTM layers is listed below for completeness.

```python
# LSTM and CNN for sequence classification in the IMDB dataset
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers.convolutional import Convolution1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
from theano.tensor.shared_randomstreams import RandomStreams
# fix random seed for reproducibility
numpy.random.seed(7)
srng = RandomStreams(7)
# load the dataset but only keep the top n words, zero the rest
top_words = 5000
test_split = 0.33
(X_train, y_train), (X_test, y_test) = imdb.load_data(nb_words=top_words,
    test_split=test_split)
# truncate and pad input sequences
max_review_length = 500
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
# create the model
embedding_vecor_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length, input_length=max_review_length))
model.add(Convolution1D(nb_filter=32, filter_length=3, border_mode='same',
    activation='relu'))
model.add(MaxPooling1D(pool_length=2))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
model.fit(X_train, y_train, nb_epoch=3, batch_size=64)
# Final evaluation of the model
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 26.15: LSTM and CNN Model for the IMDB Dataset.

Running this example provides the following output.

```
Epoch 1/3
16750/16750 [==============================] - 58s - loss: 0.5186 - acc: 0.7263
Epoch 2/3
16750/16750 [==============================] - 58s - loss: 0.2946 - acc: 0.8825
Epoch 3/3
16750/16750 [==============================] - 58s - loss: 0.2291 - acc: 0.9126
Accuracy: 86.36%
```

Listing 26.16: Output from Running the LSTM and CNN Model.

We can see that we achieve similar results to the first example although with less weights and faster training time. I would expect that even better results could be achieved if this example was further extended to use dropout.

26.4 Summary

In this project you discovered how to develop LSTM network models for sequence classification predictive modeling problems. Specifically, you learned:

- How to develop a simple single layer LSTM model for the IMDB movie review sentiment classification problem.

- How to extend your LSTM model with layer-wise and LSTM-specific dropout to reduce overfitting.

- How to combine the spatial structure learning properties of a Convolutional Neural Network with the sequence learning of an LSTM.

26.4.1 Next

In completing this project you learned how you can use LSTM recurrent neural networks for sequence classification. In the next lesson you will build upon your understanding of LSTM networks and better understand how the Keras API maintains state for simple sequence prediction problems.

Chapter 27

Understanding Stateful LSTM Recurrent Neural Networks

A powerful and popular recurrent neural network is the long short-term model network or LSTM. It is widely used because the architecture overcomes the vanishing and exposing gradient problem that plagues all recurrent neural networks, allowing very large and very deep networks to be created. Like other recurrent neural networks, LSTM networks maintain state, and the specifics of how this is implemented in Keras framework can be confusing. In this lesson you will discover exactly how state is maintained in LSTM networks by the Keras deep learning library. After reading this lesson you will know:

- How to develop a naive LSTM network for a sequence prediction problem.

- How to carefully manage state through batches and features with an LSTM network.

- Hot to manually manage state in an LSTM network for stateful prediction.

Let's get started.

27.1   Problem Description: Learn the Alphabet

In this tutorial we are going to develop and contrast a number of different LSTM recurrent neural network models. The context of these comparisons will be a simple sequence prediction problem of learning the alphabet. That is, given a letter of the alphabet, predict the next letter of the alphabet. This is a simple sequence prediction problem that once understood can be generalized to other sequence prediction problems like time series prediction and sequence classification. Let's prepare the problem with some Python code that we can reuse from example to example. Firstly, let's import all of the classes and functions we plan to use in this tutorial.

```python
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
from theano.tensor.shared_randomstreams import RandomStreams
```

Listing 27.1: Import Classes and Functions.

Next, we can seed the random number generator to ensure that the results are the same each time the code is executed.

```
# fix random seed for reproducibility
numpy.random.seed(7)
srng = RandomStreams(7)
```

Listing 27.2: Seed the Random Number Generators.

We can now define our dataset, the alphabet. We define the alphabet in uppercase characters for readability. Neural networks model numbers, so we need to map the letters of the alphabet to integer values. We can do this easily by creating a dictionary (map) of the letter index to the character. We can also create a reverse lookup for converting predictions back into characters to be used later.

```
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
```

Listing 27.3: Define the Alphabet Dataset.

Now we need to create our input and output pairs on which to train our neural network. We can do this by defining an input sequence length, then reading sequences from the input alphabet sequence. For example we use an input length of 1. Starting at the beginning of the raw input data, we can read off the first letter A and the next letter as the prediction B. We move along one character and repeat until we reach a prediction of Z.

```
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print seq_in, '->', seq_out
```

Listing 27.4: Create Patterns from Dataset.

We also print out the input pairs for sanity checking. Running the code to this point will produce the following output, summarizing input sequences of length 1 and a single output character.

```
A -> B
B -> C
C -> D
D -> E
E -> F
F -> G
G -> H
H -> I
I -> J
J -> K
```

```
K -> L
L -> M
M -> N
N -> O
O -> P
P -> Q
Q -> R
R -> S
S -> T
T -> U
U -> V
V -> W
W -> X
X -> Y
Y -> Z
```

Listing 27.5: Sample Alphabet Training Patterns.

We need to reshape the NumPy array into a format expected by the LSTM networks, that is `[samples, time steps, features]`.

```
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (X.shape[0], seq_length, 1))
```

Listing 27.6: Reshape Training Patterns for LSTM Layer.

Once reshaped, we can then normalize the input integers to the range 0-to-1, the range of the sigmoid activation functions used by the LSTM network.

```
# normalize
X = X / float(len(alphabet))
```

Listing 27.7: Normalize Training Patterns.

Finally, we can think of this problem as a sequence classification task, where each of the 26 letters represents a different class. As such, we can convert the output (y) to a one hot encoding, using the Keras built-in function `to_categorical()`.

```
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
```

Listing 27.8: One Hot Encode Output Patterns.

We are now ready to fit different LSTM models.

27.2 LSTM for Learning One-Char to One-Char Mapping

Let's start off by designing a simple LSTM to learn how to predict the next character in the alphabet given the context of just one character. We will frame the problem as a random collection of one-letter input to one-letter output pairs. As we will see this is a difficult framing of the problem for the LSTM to learn. Let's define an LSTM network with 32 units and a single output neuron with a softmax activation function for making predictions. Because this is a multiclass classification problem, we can use the log loss function (called `categorical_crossentropy` in

Keras), and optimize the network using the ADAM optimization function. The model is fit over 500 epochs with a batch size of 1.

```
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, nb_epoch=500, batch_size=1, verbose=2)
```

Listing 27.9: Define and Fit LSTM Network Model.

After we fit the model we can evaluate and summarize the performance on the entire training dataset.

```
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 27.10: Evaluate the Fit LSTM Network Model.

We can then re-run the training data through the network and generate predictions, converting both the input and output pairs back into their original character format to get a visual idea of how well the network learned the problem.

```
# demonstrate some model predictions
for pattern in dataX:
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
  print seq_in, "->", result
```

Listing 27.11: Make Predictions Using the Fit LSTM Network.

The entire code listing is provided below for completeness.

```
# Naive LSTM to learn one-char to one-char mapping
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
from theano.tensor.shared_randomstreams import RandomStreams
# fix random seed for reproducibility
numpy.random.seed(7)
srng = RandomStreams(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
```

```python
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print seq_in, '->', seq_out
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (X.shape[0], seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, nb_epoch=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for pattern in dataX:
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
  print seq_in, "->", result
```

Listing 27.12: LSTM Network for one-char to one-char Mapping.

Running this example produces the following output.

```
Model Accuracy: 84.00%
['A'] -> B
['B'] -> C
['C'] -> D
['D'] -> E
['E'] -> F
['F'] -> G
['G'] -> H
['H'] -> I
['I'] -> J
['J'] -> K
['K'] -> L
['L'] -> M
['M'] -> N
['N'] -> O
['O'] -> P
['P'] -> Q
['Q'] -> R
['R'] -> S
['S'] -> T
['T'] -> U
['U'] -> W
```

```
['V'] -> Y
['W'] -> Z
['X'] -> Z
['Y'] -> Z
```

Listing 27.13: Output from the one-char to one-char Mapping.

We can see that this problem is indeed difficult for the network to learn. The reason is, the poor LSTM units do not have any context to work with. Each input-output pattern is shown to the network in a random order and the state of the network is reset after each pattern (each batch where each batch contains one pattern). This is abuse of the LSTM network architecture, treating it like a standard Multilayer Perceptron. Next, let's try a different framing of the problem in order to provide more sequence to the network from which to learn.

27.3 LSTM for a Feature Window to One-Char Mapping

A popular approach to adding more context to data for Multilayer Perceptrons is to use the window method. This is where previous steps in the sequence are provided as additional input features to the network. We can try the same trick to provide more context to the LSTM network. Here, we increase the sequence length from 1 to 3, for example:

```
# prepare the dataset of input to output pairs encoded as integers
seq_length = 3
```

Listing 27.14: Increase Sequence Length.

Which creates training patterns like:

```
ABC -> D
BCD -> E
CDE -> F
```

Listing 27.15: Sample of Longer Input Sequence Length.

Each element in the sequence is then provided as a new input feature to the network. This requires a modification of how the input sequences reshaped in the data preparation step:

```
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (X.shape[0], 1, seq_length))
```

Listing 27.16: Reshape Input so Sequence is Features.

It also requires a modification for how the sample patterns are reshaped when demonstrating predictions from the model.

```
x = numpy.reshape(pattern, (1, 1, len(pattern)))
```

Listing 27.17: Reshape Input for Predictions so Sequence is Features.

The entire code listing is provided below for completeness.

```
# Naive LSTM to learn three-char window to one-char mapping
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
```

```python
from keras.utils import np_utils
from theano.tensor.shared_randomstreams import RandomStreams
# fix random seed for reproducibility
numpy.random.seed(7)
srng = RandomStreams(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 3
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print seq_in, '->', seq_out
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (X.shape[0], 1, seq_length))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, nb_epoch=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for pattern in dataX:
  x = numpy.reshape(pattern, (1, 1, len(pattern)))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
  print seq_in, "->", result
```

Listing 27.18: LSTM Network for three-char features to one-char Mapping.

Running this example provides the following output.

```
Model Accuracy: 86.96%
['A', 'B', 'C'] -> D
['B', 'C', 'D'] -> E
['C', 'D', 'E'] -> F
['D', 'E', 'F'] -> G
['E', 'F', 'G'] -> H
['F', 'G', 'H'] -> I
['G', 'H', 'I'] -> J
```

```
['H', 'I', 'J'] -> K
['I', 'J', 'K'] -> L
['J', 'K', 'L'] -> M
['K', 'L', 'M'] -> N
['L', 'M', 'N'] -> O
['M', 'N', 'O'] -> P
['N', 'O', 'P'] -> Q
['O', 'P', 'Q'] -> R
['P', 'Q', 'R'] -> S
['Q', 'R', 'S'] -> T
['R', 'S', 'T'] -> U
['S', 'T', 'U'] -> V
['T', 'U', 'V'] -> Y
['U', 'V', 'W'] -> Z
['V', 'W', 'X'] -> Z
['W', 'X', 'Y'] -> Z
```

Listing 27.19: Output from the three-char Features to one-char Mapping.

We can see a small lift in performance that may or may not be real. This is a simple problem that we were still not able to learn with LSTMs even with the window method. Again, this is a misuse of the LSTM network by a poor framing of the problem. Indeed, the sequences of letters are time steps of one feature rather than one time step of separate features. We have given more context to the network, but not more sequence as it expected. In the next section, we will give more context to the network in the form of time steps.

27.4 LSTM for a Time Step Window to One-Char Mapping

In Keras, the intended use of LSTMs is to provide context in the form of time steps, rather than windowed features like with other network types. We can take our first example and simply change the sequence length from 1 to 3.

```
seq_length = 3
```

Listing 27.20: Increase Sequence Length.

Again, this creates input-output pairs that look like:

```
ABC -> D
BCD -> E
CDE -> F
DEF -> G
```

Listing 27.21: Sample of Longer Input Sequence Length.

The difference is that the reshaping of the input data takes the sequence as a time step sequence of one feature, rather than a single time step of multiple features.

```
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (X.shape[0], seq_length, 1))
```

Listing 27.22: Reshape Input so Sequence is Time Steps.

This is the intended use of providing sequence context to your LSTM in Keras. The full code example is provided below for completeness.

```python
# Naive LSTM to learn three-char time steps to one-char mapping
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
from theano.tensor.shared_randomstreams import RandomStreams
# fix random seed for reproducibility
numpy.random.seed(7)
srng = RandomStreams(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 3
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print seq_in, '->', seq_out
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (X.shape[0], seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, nb_epoch=500, batch_size=1, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for pattern in dataX:
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
  print seq_in, "->", result
```

Listing 27.23: LSTM Network for three-char Time Steps to one-char Mapping.

Running this example provides the following output.

```
Model Accuracy: 100.00%
['A', 'B', 'C'] -> D
['B', 'C', 'D'] -> E
['C', 'D', 'E'] -> F
['D', 'E', 'F'] -> G
['E', 'F', 'G'] -> H
['F', 'G', 'H'] -> I
['G', 'H', 'I'] -> J
['H', 'I', 'J'] -> K
['I', 'J', 'K'] -> L
['J', 'K', 'L'] -> M
['K', 'L', 'M'] -> N
['L', 'M', 'N'] -> O
['M', 'N', 'O'] -> P
['N', 'O', 'P'] -> Q
['O', 'P', 'Q'] -> R
['P', 'Q', 'R'] -> S
['Q', 'R', 'S'] -> T
['R', 'S', 'T'] -> U
['S', 'T', 'U'] -> V
['T', 'U', 'V'] -> W
['U', 'V', 'W'] -> X
['V', 'W', 'X'] -> Y
['W', 'X', 'Y'] -> Z
```

Listing 27.24: Output from the three-char Time Steps to one-char Mapping.

We can see that the model learns the problem perfectly as evidenced by the model evaluation and the example predictions. But it has learned a simpler problem. Specifically, it has learned to predict the next letter from a sequence of three letters in the alphabet. It can be shown any random sequence of three letters from the alphabet and predict the next letter.

It cannot actually enumerate the alphabet. I expect that a larger enough Multilayer Perceptron network might be able to learn the same mapping using the window method. The LSTM networks are stateful. They should be able to learn the whole alphabet sequence, but by default the Keras implementation resets the network state after each training batch.

27.5 LSTM State Maintained Between Samples Within A Batch

The Keras implementation of LSTMs resets the state of the network after each batch. This suggests that if we had a batch size large enough to hold all input patterns and if all the input patterns were ordered sequentially, that the LSTM could use the context of the sequence within the batch to better learn the sequence. We can demonstrate this easily by modifying the first example for learning a one-to-one mapping and increasing the batch size from 1 to the size of the training dataset. Additionally, Keras shuffles the training dataset before each training epoch. To ensure the training data patterns remain sequential, we can disable this shuffling.

```
model.fit(X, y, nb_epoch=5000, batch_size=len(dataX), verbose=2, shuffle=False)
```

Listing 27.25: Increase Batch Size to Cover Entire Dataset.

The network will learn the mapping of characters using the within-batch sequence, but this context will not be available to the network when making predictions. We can evaluate both the ability of the network to make predictions randomly and in sequence. The full code example is provided below for completeness.

```python
# Naive LSTM to learn one-char to one-char mapping with all data in each batch
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
from keras.preprocessing.sequence import pad_sequences
from theano.tensor.shared_randomstreams import RandomStreams
# fix random seed for reproducibility
numpy.random.seed(7)
srng = RandomStreams(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print seq_in, '->', seq_out
# convert list of lists to array and pad sequences if needed
X = pad_sequences(dataX, maxlen=seq_length, dtype='float32')
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (X.shape[0], seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
model = Sequential()
model.add(LSTM(16, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, nb_epoch=5000, batch_size=len(dataX), verbose=2, shuffle=False)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for pattern in dataX:
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
```

```
  print seq_in, "->", result
# demonstrate predicting random patterns
print "Test a Random Pattern:"
for i in range(0,20):
  pattern_index = numpy.random.randint(len(dataX))
  pattern = dataX[pattern_index]
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
  print seq_in, "->", result
```

Listing 27.26: LSTM Network for one-char to one-char Mapping Within Batch.

Running the example provides the following output.

```
Model Accuracy: 100.00%
['A'] -> B
['B'] -> C
['C'] -> D
['D'] -> E
['E'] -> F
['F'] -> G
['G'] -> H
['H'] -> I
['I'] -> J
['J'] -> K
['K'] -> L
['L'] -> M
['M'] -> N
['N'] -> O
['O'] -> P
['P'] -> Q
['Q'] -> R
['R'] -> S
['S'] -> T
['T'] -> U
['U'] -> V
['V'] -> W
['W'] -> X
['X'] -> Y
['Y'] -> Z
Test a Random Pattern:
['T'] -> U
['V'] -> W
['M'] -> N
['Q'] -> R
['D'] -> E
['V'] -> W
['T'] -> U
['U'] -> V
['J'] -> K
['F'] -> G
['N'] -> O
['B'] -> C
```

```
['M'] -> N
['F'] -> G
['F'] -> G
['P'] -> Q
['A'] -> B
['K'] -> L
['W'] -> X
['E'] -> F
```

Listing 27.27: Output from the one-char to one-char Mapping Within Batch.

As we expected, the network is able to use the within-sequence context to learn the alphabet, achieving 100% accuracy on the training data. Importantly, the network can make accurate predictions for the next letter in the alphabet for randomly selected characters. Very impressive.

27.6 Stateful LSTM for a One-Char to One-Char Mapping

We have seen that we can break-up our raw data into fixed size sequences and that this representation can be learned by the LSTM, but only to learn random mappings of 3 characters to 1 character. We have also seen that we can pervert batch size to offer more sequence to the network, but only during training. Ideally, we want to expose the network to the entire sequence and let it learn the inter-dependencies, rather than us define those dependencies explicitly in the framing of the problem.

We can do this in Keras by making the LSTM layers stateful and manually resetting the state of the network at the end of the epoch, which is also the end of the training sequence. This is truly how the LSTM networks are intended to be used. We find that by allowing the network itself to learn the dependencies between the characters, that we need a smaller network (half the number of units) and fewer training epochs (almost half). We first need to define our LSTM layer as stateful. In so doing, we must explicitly specify the batch size as a dimension on the input shape. This also means that when we evaluate the network or make predictions, we must also specify and adhere to this same batch size. This is not a problem now as we are using a batch size of 1. This could introduce difficulties when making predictions when the batch size is not one as predictions will need to be made in batch and in sequence.

```
batch_size = 1
model.add(LSTM(16, batch_input_shape=(batch_size, X.shape[1], X.shape[2]), stateful=True))
```

Listing 27.28: Define a Stateful LSTM Layer.

An important difference in training the stateful LSTM is that we train it manually one epoch at a time and reset the state after each epoch. We can do this in a for loop. Again, we do not shuffle the input, preserving the sequence in which the input training data was created.

```
for i in range(300):
  model.fit(X, y, nb_epoch=1, batch_size=batch_size, verbose=2, shuffle=False)
  model.reset_states()
```

Listing 27.29: Manually Manage LSTM State For Each Epoch.

As mentioned, we specify the batch size when evaluating the performance of the network on the entire training dataset.

```
# summarize performance of the model
scores = model.evaluate(X, y, batch_size=batch_size, verbose=0)
model.reset_states()
print("Model Accuracy: %.2f%%" % (scores[1]*100))
```

Listing 27.30: Evaluate Model Using Pre-defined Batch Size.

Finally, we can demonstrate that the network has indeed learned the entire alphabet. We can seed it with the first letter `A`, request a prediction, feed the prediction back in as an input, and repeat the process all the way to `Z`.

```
# demonstrate some model predictions
seed = [char_to_int[alphabet[0]]]
for i in range(0, len(alphabet)-1):
  x = numpy.reshape(seed, (1, len(seed), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  print int_to_char[seed[0]], "->", int_to_char[index]
  seed = [index]
model.reset_states()
```

Listing 27.31: Seed Network and Make Predictions from A to Z.

We can also see if the network can make predictions starting from an arbitrary letter.

```
# demonstrate a random starting point
letter = "K"
seed = [char_to_int[letter]]
print "New start: ", letter
for i in range(0, 5):
  x = numpy.reshape(seed, (1, len(seed), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  print int_to_char[seed[0]], "->", int_to_char[index]
  seed = [index]
model.reset_states()
```

Listing 27.32: Seed Network with a Random Letter and a Sequence of Predictions.

The entire code listing is provided below for completeness.

```
# Stateful LSTM to learn one-char to one-char mapping
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
from theano.tensor.shared_randomstreams import RandomStreams
# fix random seed for reproducibility
numpy.random.seed(7)
srng = RandomStreams(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
```

```python
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
seq_length = 1
dataX = []
dataY = []
for i in range(0, len(alphabet) - seq_length, 1):
  seq_in = alphabet[i:i + seq_length]
  seq_out = alphabet[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
  print seq_in, '->', seq_out
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (X.shape[0], seq_length, 1))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
batch_size = 1
model = Sequential()
model.add(LSTM(16, batch_input_shape=(batch_size, X.shape[1], X.shape[2]), stateful=True))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
for i in range(300):
  model.fit(X, y, nb_epoch=1, batch_size=batch_size, verbose=2, shuffle=False)
  model.reset_states()
# summarize performance of the model
scores = model.evaluate(X, y, batch_size=batch_size, verbose=0)
model.reset_states()
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
seed = [char_to_int[alphabet[0]]]
for i in range(0, len(alphabet)-1):
  x = numpy.reshape(seed, (1, len(seed), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  print int_to_char[seed[0]], "->", int_to_char[index]
  seed = [index]
model.reset_states()
# demonstrate a random starting point
letter = "K"
seed = [char_to_int[letter]]
print "New start: ", letter
for i in range(0, 5):
  x = numpy.reshape(seed, (1, len(seed), 1))
  x = x / float(len(alphabet))
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  print int_to_char[seed[0]], "->", int_to_char[index]
  seed = [index]
model.reset_states()
```

Listing 27.33: Stateful LSTM Network for one-char to one-char Mapping.

Running the example provides the following output.

```
Model Accuracy: 100.00%
A -> B
B -> C
C -> D
D -> E
E -> F
F -> G
G -> H
H -> I
I -> J
J -> K
K -> L
L -> M
M -> N
N -> O
O -> P
P -> Q
Q -> R
R -> S
S -> T
T -> U
U -> V
V -> W
W -> X
X -> Y
Y -> Z
New start: K
K -> B
B -> C
C -> D
D -> E
E -> F
```

Listing 27.34: Output from the Stateful LSTM for one-char to one-char Mapping.

We can see that the network has memorized the entire alphabet perfectly. It used the context of the samples themselves and learned whatever dependency it needed to predict the next character in the sequence. We can also see that if we seed the network with the first letter, that it can correctly rattle off the rest of the alphabet. We can also see that it has only learned the full alphabet sequence and that from a cold start. When asked to predict the next letter from K that it predicts B and falls back into regurgitating the entire alphabet. To truly predict K the state of the network would need to be warmed up iteratively fed the letters from A to J. This tells us that we could achieve the same effect with a *stateless* LSTM by preparing training data like:

```
---a -> b
--ab -> c
-abc -> d
abcd -> e
```

Listing 27.35: Sample of Equivalent Training Data for Non-Stateful LSTM Layers.

Where the input sequence is fixed at 25 (a-to-y to predict z) and patterns are prefixed with zero-padding. Finally, this raises the question of training an LSTM network using variable length input sequences to predict the next character.

27.7 LSTM with Variable Length Input to One-Char Output

In the previous section we discovered that the Keras *stateful* LSTM was really only a short cut to replaying the first n-sequences, but didn't really help us learn a generic model of the alphabet. In this section we explore a variation of the *stateless* LSTM that learns random subsequences of the alphabet and an effort to build a model that can be given arbitrary letters or subsequences of letters and predict the next letter in the alphabet.

Firstly, we are changing the framing of the problem. To simplify we will define a maximum input sequence length and set it to a small value like 5 to speed up training. This defines the maximum length of subsequences of the alphabet will be drawn for training. In extensions, this could just as set to the full alphabet (26) or longer if we allow looping back to the start of the sequence. We also need to define the number of random sequences to create, in this case 1,000. This too could be more or less. I expect less patterns are actually required.

```python
# prepare the dataset of input to output pairs encoded as integers
num_inputs = 1000
max_len = 5
dataX = []
dataY = []
for i in range(num_inputs):
  start = numpy.random.randint(len(alphabet)-2)
  end = numpy.random.randint(start, min(start+max_len,len(alphabet)-1))
  sequence_in = alphabet[start:end+1]
  sequence_out = alphabet[end + 1]
  dataX.append([char_to_int[char] for char in sequence_in])
  dataY.append(char_to_int[sequence_out])
  print sequence_in, '->', sequence_out
```

Listing 27.36: Create Dataset of Variable Length Input Sequences.

Running this code in the broader context will create input patterns that look like the following:

```
PQRST -> U
W -> X
O -> P
OPQ -> R
IJKLM -> N
QRSTU -> V
ABCD -> E
X -> Y
GHIJ -> K
```

Listing 27.37: Sample of Variable Length Input Sequences.

The input sequences vary in length between 1 and `max_len` and therefore require zero padding. Here, we use left-hand-side (prefix) padding with the Keras built in `pad_sequences()` function.

```python
X = pad_sequences(dataX, maxlen=max_len, dtype='float32')
```

Listing 27.38: Left-Pad Variable Length Input Sequences.

The trained model is evaluated on randomly selected input patterns. This could just as easily be new randomly generated sequences of characters. I also believe this could also be a

linear sequence seeded with `A` with outputs fed back in as single character inputs. The full code listing is provided below for completeness.

```python
# LSTM with Variable Length Input Sequences to One Character Output
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.utils import np_utils
from keras.preprocessing.sequence import pad_sequences
from theano.tensor.shared_randomstreams import RandomStreams
# fix random seed for reproducibility
numpy.random.seed(7)
srng = RandomStreams(7)
# define the raw dataset
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# create mapping of characters to integers (0-25) and the reverse
char_to_int = dict((c, i) for i, c in enumerate(alphabet))
int_to_char = dict((i, c) for i, c in enumerate(alphabet))
# prepare the dataset of input to output pairs encoded as integers
num_inputs = 1000
max_len = 5
dataX = []
dataY = []
for i in range(num_inputs):
  start = numpy.random.randint(len(alphabet)-2)
  end = numpy.random.randint(start, min(start+max_len,len(alphabet)-1))
  sequence_in = alphabet[start:end+1]
  sequence_out = alphabet[end + 1]
  dataX.append([char_to_int[char] for char in sequence_in])
  dataY.append(char_to_int[sequence_out])
  print sequence_in, '->', sequence_out
# convert list of lists to array and pad sequences if needed
X = pad_sequences(dataX, maxlen=max_len, dtype='float32')
# reshape X to be [samples, time steps, features]
X = numpy.reshape(X, (X.shape[0], max_len, 1))
# normalize
X = X / float(len(alphabet))
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# create and fit the model
batch_size = 1
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], 1)))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, nb_epoch=500, batch_size=batch_size, verbose=2)
# summarize performance of the model
scores = model.evaluate(X, y, verbose=0)
print("Model Accuracy: %.2f%%" % (scores[1]*100))
# demonstrate some model predictions
for i in range(20):
  pattern_index = numpy.random.randint(len(dataX))
  pattern = dataX[pattern_index]
  x = pad_sequences([pattern], maxlen=max_len, dtype='float32')
  x = numpy.reshape(x, (1, max_len, 1))
```

```
x = x / float(len(alphabet))
prediction = model.predict(x, verbose=0)
index = numpy.argmax(prediction)
result = int_to_char[index]
seq_in = [int_to_char[value] for value in pattern]
print seq_in, "->", result
```

Listing 27.39: LSTM Network for Variable Length Sequences to one-char Mapping.

Running this code produces the following output:

```
Model Accuracy: 98.90%
['Q', 'R'] -> S
['W', 'X'] -> Y
['W', 'X'] -> Y
['C', 'D'] -> E
['E'] -> F
['S', 'T', 'U'] -> V
['G', 'H', 'I', 'J', 'K'] -> L
['O', 'P', 'Q', 'R', 'S'] -> T
['C', 'D'] -> E
['O'] -> P
['N', 'O', 'P'] -> Q
['D', 'E', 'F', 'G', 'H'] -> I
['X'] -> Y
['K'] -> L
['M'] -> N
['R'] -> T
['K'] -> L
['E', 'F', 'G'] -> H
['Q'] -> R
['Q', 'R', 'S'] -> T
```

Listing 27.40: Output for the LSTM Network for Variable Length Sequences to one-char Mapping.

We can see that although the model did not learn the alphabet perfectly from the randomly generated subsequences, it did very well. The model was not tuned and may require more training or a larger network, or both (an exercise for the reader). This is a good natural extension to the *all sequential input examples in each batch* alphabet model learned above in that it can handle ad hoc queries, but this time of arbitrary sequence length (up to the max length).

27.8   Summary

In this lesson you discovered LSTM recurrent neural networks in Keras and how they manage state. Specifically, you learned:

- How to develop a naive LSTM network for one-character to one-character prediction.

- How to configure a naive LSTM to learn a sequence across time steps within a sample.

- How to configure an LSTM to learn a sequence across samples by manually managing state.

27.8.1 Next

In this lesson you developed your understanding for how LSTM networks maintain state for simple sequence prediction problems. Up next you will use your understanding of LSTM networks to develop larger text generation models.

Chapter 28

Project: Text Generation With Alice in Wonderland

Recurrent neural networks can also be used as generative models. This means that in addition to being used for predictive models (making predictions) they can learn the sequences of a problem and then generate entirely new plausible sequences for the problem domain. Generative models like this are useful not only to study how well a model has learned a problem, but to learn more about the problem domain itself. In this project you will discover how to create a generative model for text, character-by-character using LSTM recurrent neural networks in Python with Keras. After completing this project you will know:

- Where to download a free corpus of text that you can use to train text generative models.

- How to frame the problem of text sequences to a recurrent neural network generative model.

- How to develop an LSTM to generate plausible text sequences for a given problem.

Let's get started.

Note: You may want to speed up the computation for this tutorial by using GPU rather than CPU hardware, such as the process described in Chapter 5. This is a suggestion, not a requirement. The tutorial will work just fine on the CPU.

28.1   Problem Description: Text Generation

Many of the classical texts are no longer protected under copyright. This means that you can download all of the text for these books for free and use them in experiments, like creating generative models. Perhaps the best place to get access to free books that are no longer protected by copyright is Project Gutenberg[1]. In this tutorial we are going to use a favorite book from childhood as the dataset: Alice's Adventures in Wonderland by Lewis Carroll[2].

We are going to learn the dependencies between characters and the conditional probabilities of characters in sequences so that we can in turn generate wholly new and original sequences of characters. This tutorial is a lot of fun and I recommend repeating these experiments with

---

[1]https://www.gutenberg.org/
[2]https://www.gutenberg.org/ebooks/11

other books from Project Gutenberg. These experiments are not limited to text, you can also experiment with other ASCII data, such as computer source code, marked up documents in LaTeX, HTML or Markdown and more.

You can download the complete text in ASCII format (Plain Text UTF-8)[3] for this book for free and place it in your working directory with the filename `wonderland.txt`. Now we need to prepare the dataset ready for modeling. Project Gutenberg adds a standard header and footer to each book and this is not part of the original text. Open the file in a text editor and delete the header and footer. The header is obvious and ends with the text:

```
*** START OF THIS PROJECT GUTENBERG EBOOK ALICE'S ADVENTURES IN WONDERLAND ***
```

Listing 28.1: Signal of the End of the File Header.

The footer is all of the text after the line of text that says:

```
THE END
```

Listing 28.2: Signal of the Start of the File Footer.

You should be left with a text file that has about 3,330 lines of text.

28.2 Develop a Small LSTM Recurrent Neural Network

In this section we will develop a simple LSTM network to learn sequences of characters from Alice in Wonderland. In the next section we will use this model to generate new sequences of characters. Let's start off by importing the classes and functions we intend to use to train our model.

```python
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
```

Listing 28.3: Import Classes and Functions.

Next we need to load the ASCII text for the book into memory and convert all of the characters to lowercase to reduce the vocabulary that the network must learn.

```python
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
```

Listing 28.4: Load the Dataset and Covert to Lowercase.

Now that the book is loaded, we must prepare the data for modeling by the neural network. We cannot model the characters directly, instead we must convert the characters to integers. We can do this easily by first creating a set of all of the distinct characters in the book, then creating a map of each character to a unique integer.

---

[3]http://www.gutenberg.org/cache/epub/11/pg11.txt

```
# create mapping of unique chars to integers, and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
```

Listing 28.5: Create Char-to-Integer Mapping.

For example, the list of unique sorted lowercase characters in the book is as follows:

```
['\n', '\r', ' ', '!', '"', "'", '(', ')', '*', ',', '-', '.', ':', ';', '?', '[', ']',
    '_', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
    'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '\xbb', '\xbf', '\xef']
```

Listing 28.6: List of Unique Characters in the Dataset.

You can see that there may be some characters that we could remove to further clean up the dataset that will reduce the vocabulary and may improve the modeling process. Now that the book has been loaded and the mapping prepared, we can summarize the dataset.

```
n_chars = len(raw_text)
n_vocab = len(chars)
print "Total Characters: ", n_chars
print "Total Vocab: ", n_vocab
```

Listing 28.7: Summarize the Loaded Dataset.

Running the code to this point produces the following output.

```
Total Characters: 147674
Total Vocab: 47
```

Listing 28.8: Output from Summarize the Dataset.

We can see that the book has just under 150,000 characters and that when converted to lowercase that there are only 47 distinct characters in the vocabulary for the network to learn. Much more than the 26 in the alphabet. We now need to define the training data for the network. There is a lot of flexibility in how you choose to break up the text and expose it to the network during training. In this tutorial we will split the book text up into subsequences with a fixed length of 100 characters, an arbitrary length. We could just as easily split the data up by sentences and pad the shorter sequences and truncate the longer ones.

Each training pattern of the network is comprised of 100 time steps of one character (X) followed by one character output (y). When creating these sequences, we slide this window along the whole book one character at a time, allowing each character a chance to be learned from the 100 characters that preceded it (except the first 100 characters of course). For example, if the sequence length is 5 (for simplicity) then the first two training patterns would be as follows:

```
CHAPT -> E
HAPTE -> R
```

Listing 28.9: Example of Sequence Construction.

As we split up the book into these sequences, we convert the characters to integers using our lookup table we prepared earlier.

```
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
```

```
dataY = []
for i in range(0, n_chars - seq_length, 1):
  seq_in = raw_text[i:i + seq_length]
  seq_out = raw_text[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print "Total Patterns: ", n_patterns
```

Listing 28.10: Create Input/Output Patterns from Raw Dataset.

Running the code to this point shows us that when we split up the dataset into training data for the network to learn that we have just under 150,000 training pattens. This makes sense as excluding the first 100 characters, we have one training pattern to predict each of the remaining characters.

```
Total Patterns: 147574
```

Listing 28.11: Output Summary of Created Patterns.

Now that we have prepared our training data we need to transform it so that it is suitable for use with Keras. First we must transform the list of input sequences into the form [samples, time steps, features] expected by an LSTM network. Next we need to rescale the integers to the range 0-to-1 to make the patterns easier to learn by the LSTM network that uses the sigmoid activation function by default.

Finally, we need to convert the output patterns (single characters converted to integers) into a one hot encoding. This is so that we can configure the network to predict the probability of each of the 47 different characters in the vocabulary (an easier representation) rather than trying to force it to predict precisely the next character. Each y value is converted into a sparse vector with a length of 47, full of zeros except with a 1 in the column for the letter (integer) that the pattern represents. For example, when `n` (integer value 31) is one hot encoded it looks as follows:

```
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.]
```

Listing 28.12: Sample of One Hot Encoded Output Integer.

We can implement these steps as below.

```
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
```

Listing 28.13: Prepare Data Ready For Modeling.

We can now define our LSTM model. Here we define a single hidden LSTM layer with 256 memory units. The network uses dropout with a probability of 20%. The output layer is a Dense layer using the softmax activation function to output a probability prediction for each of the 47 characters between 0 and 1. The problem is really a single character classification problem with 47 classes and as such is defined as optimizing the log loss (cross entropy), here using the ADAM optimization algorithm for speed.

```python
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Listing 28.14: Create LSTM Network to Model the Dataset.

There is no test dataset. We are modeling the entire training dataset to learn the probability of each character in a sequence. We are not interested in the most accurate (classification accuracy) model of the training dataset. This would be a model that predicts each character in the training dataset perfectly. Instead we are interested in a generalization of the dataset that minimizes the chosen loss function. We are seeking a balance between generalization and overfitting but short of memorization.

The network is slow to train (about 300 seconds per epoch on an Nvidia K520 GPU). Because of the slowness and because of our optimization requirements, we will use model checkpointing to record all of the network weights to file each time an improvement in loss is observed at the end of the epoch. We will use the best set of weights (lowest loss) to instantiate our generative model in the next section.

```python
# define the checkpoint
filepath="weights-improvement-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True,
    mode='min')
callbacks_list = [checkpoint]
```

Listing 28.15: Create Checkpoints For Best Seen Model.

We can now fit our model to the data. Here we use a modest number of 20 epochs and a large batch size of 128 patterns.

```python
model.fit(X, y, nb_epoch=20, batch_size=128, callbacks=callbacks_list)
```

Listing 28.16: Fit the Model.

The full code listing is provided below for completeness.

```python
# Small LSTM Network to Generate Text for Alice in Wonderland
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers, and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
```

```python
n_vocab = len(chars)
print "Total Characters: ", n_chars
print "Total Vocab: ", n_vocab
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
  seq_in = raw_text[i:i + seq_length]
  seq_out = raw_text[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print "Total Patterns: ", n_patterns
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
# define the checkpoint
filepath="weights-improvement-{epoch:02d}-{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True,
    mode='min')
callbacks_list = [checkpoint]
# fit the model
model.fit(X, y, nb_epoch=20, batch_size=128, callbacks=callbacks_list)
```

Listing 28.17: Complete Code Listing for LSTM to Model Dataset.

You will see different results because of the stochastic nature of the model, and because it is hard to fix the random seed for LSTM models to get 100% reproducible results. This is not a concern for this generative model. After running the example, you should have a number of weight checkpoint files in the local directory. You can delete them all except the one with the smallest loss value. For example, when I ran this example, below was the checkpoint with the smallest loss that I achieved.

```
weights-improvement-19-1.9435.hdf5
```

Listing 28.18: Sample of Checkpoint Weights for Well Performing Model.

he network loss decreased almost every epoch and I expect the network could benefit from training for many more epochs. In the next section we will look at using this model to generate new text sequences.

28.3   Generating Text with an LSTM Network

Generating text using the trained LSTM network is relatively straightforward. Firstly, we load the data and define the network in exactly the same way, except the network weights are loaded from a checkpoint file and the network does not need to be trained.

```python
# load the network weights
filename = "weights-improvement-19-1.9435.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Listing 28.19: Load Checkpoint Network Weights.

Also, when preparing the mapping of unique characters to integers, we must also create a reverse mapping that we can use to convert the integers back to characters so that we can understand the predictions.

```python
int_to_char = dict((i, c) for i, c in enumerate(chars))
```

Listing 28.20: Mapping from Integers to Characters.

Finally, we need to actually make predictions. The simplest way to use the Keras LSTM model to make predictions is to first start off with a seed sequence as input, generate the next character then update the seed sequence to add the generated character on the end and trim off the first character. This process is repeated for as long as we want to predict new characters (e.g. a sequence of 1,000 characters in length). We can pick a random input pattern as our seed sequence, then print generated characters as we generate them.

```python
# pick a random seed
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print "Seed:"
print "\"", ''.join([int_to_char[value] for value in pattern]), "\""
# generate characters
for i in range(1000):
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(n_vocab)
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
  sys.stdout.write(result)
  pattern.append(index)
  pattern = pattern[1:len(pattern)]
print "\nDone."
```

Listing 28.21: Seed Network and Generate Text.

The full code example for generating text using the loaded LSTM model is listed below for completeness.

```python
# Load LSTM network and generate text
import sys
import numpy
from keras.models import Sequential
from keras.layers import Dense
```

```python
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers, and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
int_to_char = dict((i, c) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
n_vocab = len(chars)
print "Total Characters: ", n_chars
print "Total Vocab: ", n_vocab
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
  seq_in = raw_text[i:i + seq_length]
  seq_out = raw_text[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print "Total Patterns: ", n_patterns
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
# load the network weights
filename = "weights-improvement-19-1.9435.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')
# pick a random seed
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print "Seed:"
print "\"", ''.join([int_to_char[value] for value in pattern]), "\""
# generate characters
for i in range(1000):
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(n_vocab)
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
```

```
  sys.stdout.write(result)
  pattern.append(index)
  pattern = pattern[1:len(pattern)]
print "\nDone."
```

Listing 28.22: Complete Code Listing for Generating Text for the Small LSTM Network.

Running this example first outputs the selected random seed, then each character as it is generated. For example, below are the results from one run of this text generator. The random seed was:

```
be no mistake about it: it was neither more nor less than a pig, and she
felt that it would be quit
```

Listing 28.23: Randomly Selected Sequence Used to Seed the Network.

The generated text with the random seed (cleaned up for presentation) was:

```
be no mistake about it: it was neither more nor less than a pig, and she
felt that it would be quit e aelin that she was a little want oe toiet
ano a grtpersent to the tas a little war th tee the tase oa teettee
the had been tinhgtt a little toiee at the cadl in a long tuiee aedun
thet sheer was a little tare gereen to be a gentle of the tabdit soenee
the gad ouw ie the tay a tirt of toiet at the was a little
anonersen, and thiu had been woite io a lott of tueh a tiie and taede
bot her aeain she cere thth the bene tith the tere bane to tee
toaete to tee the harter was a little tire the same oare cade an anl ano
the garee and the was so seat the was a little gareen and the sabdit,
and the white rabbit wese tilel an the caoe and the sabbit se teeteer,
and the white rabbit wese tilel an the cade in a lonk tfne the sabdi
ano aroing to tea the was sf teet whitg the was a little tane oo thete
the sabeit she was a little tartig to the tar tf tee the tame of the
cagd, and the white rabbit was a little toiee to be anle tite thete ofs
and the tabdit was the wiite rabbit, and
```

Listing 28.24: Generated Text With Random Seed Text.

We can note some observations about the generated text.

- It generally conforms to the line format observed in the original text of less than 80 characters before a new line.

- The characters are separated into word-like groups and most groups are actual English words (e.g. *the*, *little* and *was*), but many do not (e.g. *lott*, *tiie* and *taede*).

- Some of the words in sequence make sense(e.g. *and the white rabbit*), but many do not (e.g. *wese tilel*).

The fact that this character based model of the book produces output like this is very impressive. It gives you a sense of the learning capabilities of LSTM networks. The results are not perfect. In the next section we look at improving the quality of results by developing a much larger LSTM network.

28.4 Larger LSTM Recurrent Neural Network

We got results, but not excellent results in the previous section. Now, we can try to improve the quality of the generated text by creating a much larger network. We will keep the number of memory units the same at 256, but add a second layer.

```
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Listing 28.25: Define a Stacked LSTM Network.

We will also change the filename of the checkpointed weights so that we can tell the difference between weights for this network and the previous (by appending the word `bigger` in the filename).

```
filepath="weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
```

Listing 28.26: Filename for Checkpointing Network Weights for Larger Model.

Finally, we will increase the number of training epochs from 20 to 50 and decrease the batch size from 128 to 64 to give the network more of an opportunity to be updated and learn. The full code listing is presented below for completeness.

```
# Larger LSTM Network to Generate Text for Alice in Wonderland
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers, and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
# summarize the loaded data
n_chars = len(raw_text)
n_vocab = len(chars)
print "Total Characters: ", n_chars
print "Total Vocab: ", n_vocab
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
  seq_in = raw_text[i:i + seq_length]
  seq_out = raw_text[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
```

```
n_patterns = len(dataX)
print "Total Patterns: ", n_patterns
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
# define the checkpoint
filepath="weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True,
    mode='min')
callbacks_list = [checkpoint]
# fit the model
model.fit(X, y, nb_epoch=50, batch_size=64, callbacks=callbacks_list)
```

Listing 28.27: Complete Code Listing for the Larger LSTM Network.

Running this example takes some time, at least 700 seconds per epoch. After running this example you may achieved a loss of about 1.2. For example the best result I achieved from running this model was stored in a checkpoint file with the name:

```
weights-improvement-47-1.2219-bigger.hdf5
```

Listing 28.28: Sample of Checkpoint Weights for Well Performing Larger Model.

Achieving a loss of 1.2219 at epoch 47. As in the previous section, we can use this best model from the run to generate text. The only change we need to make to the text generation script from the previous section is in the specification of the network topology and from which file to seed the network weights. The full code listing is provided below for completeness.

```
# Load Larger LSTM network and generate text
import sys
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
# create mapping of unique chars to integers, and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
int_to_char = dict((i, c) for i, c in enumerate(chars))
# summarize the loaded data
```

```
n_chars = len(raw_text)
n_vocab = len(chars)
print "Total Characters: ", n_chars
print "Total Vocab: ", n_vocab
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
  seq_in = raw_text[i:i + seq_length]
  seq_out = raw_text[i + seq_length]
  dataX.append([char_to_int[char] for char in seq_in])
  dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print "Total Patterns: ", n_patterns
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = np_utils.to_categorical(dataY)
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation='softmax'))
# load the network weights
filename = "weights-improvement-47-1.2219-bigger.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy', optimizer='adam')
# pick a random seed
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print "Seed:"
print "\"", ''.join([int_to_char[value] for value in pattern]), "\""
# generate characters
for i in range(1000):
  x = numpy.reshape(pattern, (1, len(pattern), 1))
  x = x / float(n_vocab)
  prediction = model.predict(x, verbose=0)
  index = numpy.argmax(prediction)
  result = int_to_char[index]
  seq_in = [int_to_char[value] for value in pattern]
  sys.stdout.write(result)
  pattern.append(index)
  pattern = pattern[1:len(pattern)]
print "\nDone."
```

Listing 28.29: Complete Code Listing for Generating Text With the Larger LSTM Network.

One example of running this text generation script produces the output below. The randomly chosen seed text was:

```
d herself lying on the bank, with her
head in the lap of her sister, who was gently brushing away s
```

Listing 28.30: Randomly Selected Sequence Used to Seed the Network.

The generated text with the seed (cleaned up for presentation) was:

```
herself lying on the bank, with her
head in the lap of her sister, who was gently brushing away
so siee, and she sabbit said to herself and the sabbit said to herself and the sood
way of the was a little that she was a little lad good to the garden,
and the sood of the mock turtle said to herself, 'it was a little that
the mock turtle said to see it said to sea it said to sea it say it
the marge hard sat hn a little that she was so sereated to herself, and
she sabbit said to herself, 'it was a little little shated of the sooe
of the coomouse it was a little lad good to the little gooder head. and
said to herself, 'it was a little little shated of the mouse of the
good of the courte, and it was a little little shated in a little that
the was a little little shated of the thmee said to see it was a little
book of the was a little that she was so sereated to hare a little the
began sitee of the was of the was a little that she was so seally and
the sabbit was a little lad good to the little gooder head of the gad
seared to see it was a little lad good to the little good
```

Listing 28.31: Generated Text With Random Seed Text.

We can see that generally there are fewer spelling mistakes and the text looks more realistic, but is still quite nonsensical. For example the same phrases get repeated again and again like *said to herself* and *little*. Quotes are opened but not closed. These are better results but there is still a lot of room for improvement.

28.5 Extension Ideas to Improve the Model

Below are a sample of ideas that may that you may want to investigate to further improve the model:

- Predict fewer than 1,000 characters as output for a given seed.

- Remove all punctuation from the source text, and therefore from the models' vocabulary.

- Try a one hot encoded for the input sequences.

- Train the model on padded sentences rather than random sequences of characters.

- Increase the number of training epochs to 100 or many hundreds.

- Add dropout to the visible input layer and consider tuning the dropout percentage.

- Tune the batch size, try a batch size of 1 as a (very slow) baseline and larger sizes from there.

- Add more memory units to the layers and/or more layers.

- Experiment with scale factors (temperature) when interpreting the prediction probabilities.

- Change the LSTM layers to be *stateful* to maintain state across batches.

- Did you try any of these extensions? Share your results in the comments.

28.6   Summary

In this project you discovered how you can develop an LSTM recurrent neural network for text generation in Python with the Keras deep learning library. After completing this project you know:

- Where to download the ASCII text for classical books for free that you can use for training.

- How to train an LSTM network on text sequences and how to use the trained network to generate new sequences.

- How to develop stacked LSTM networks and lift the performance of the model.

28.6.1   Next

This tutorial concludes Part VI and your introduction to recurrent neural networks in Keras. Next in Part VII we will conclude this book and highlight additional resources that you can use to continue your studies.

Part VII

Conclusions

Chapter 29

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come.

- You started off with an interest in deep learning and a strong desire to be able to practice and apply deep learning using Python.

- You downloaded, installed and started using Keras, perhaps for the first time, and started to get familiar with how to develop neural network models with the library.

- Slowly and steadily over the course of a number of lessons you learned how to use the various different features of the Keras library on neural network and deeper models for classical tabular, image and textual data.

- Building upon the recipes for common machine learning tasks you worked through your first machine learning problems with using deep learning models end-to-end using Python.

- Using standard templates, the recipes and experience you have gathered, you are now capable of working through new and different predictive modeling machine learning problems with deep learning on your own.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable skill of being able to work through machine learning problems with deep learning end-to-end using Python. This is a platform that is used by a majority of working data scientist professionals. The sky is the limit.

I want to take a moment and sincerely thank you for letting me help you start your deep learning journey with in Python. I hope you keep learning and have fun as you continue to master machine learning.

Chapter 30

Getting More Help

This book has given you a foundation for applying deep learning in your own machine learning projects, but there is still a lot more to learn. In this chapter you will discover the places that you can go to get more help with deep learning, the Keras library as well as neural networks in general.

30.1 Artificial Neural Networks

The field of neural networks has been around for decades. As such there are a wealth of papers, books and websites on the topic. Below are a few books that I recommend if you are looking for a deeper background in the field.

- Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks
  http://amzn.to/1pZDFn0

- Neural Networks for Pattern Recognition
  http://amzn.to/1W7J8GQ

- An Introduction to Neural Networks
  http://amzn.to/1pZDFTP

30.2 Deep Learning

Deep learning is a new field. Unfortunately, resources on the topic are predominately academic focused rather than practical. Nevertheless, if you are looking to go deeper into the field of deep learning, below are some suggested resources.

- Deep Learning, a soon to be published textbook on deep learning by some of the pioneers in the field.
  http://www.deeplearningbook.org

- Learning Deep Architectures for AI (2009) provides a good but academic introduction paper to the field.
  http://goo.gl/MkUt6B

- Deep Learning in Neural Networks: An Overview (2014), another excellent but academic introduction paper to the field.
  http://arxiv.org/abs/1404.7828

30.3 Python Machine Learning

Python is a growing platform for applied machine learning. The strong attraction is because Python is a fully featured programming language (unlike R) and as such you can use the same code and libraries in developing your model as you use to deploy the model into operations. The premier machine learning library in Python is scikit-learn built on top of SciPy.

- Visit the scikit-learn home page to learn more about the library and it's capabilities.
  http://scikit-learn.org

- Visit the SciPy home page to learn more about the SciPy platform for scientific computing in Python.
  http://scipy.org

- Machine Learning Mastery with Python, the precursor to this book.
  https://machinelearningmastery.com/machine-learning-with-python

30.4 Keras Library

Keras is a fantastic but fast moving library. Large updates are still being made to the API and it is good to stay abreast of changes. You also need to know where you can ask questions to get more help with the platform.

- The Keras homepage is an excellent place to start, giving you full access to the user documentation.
  http://keras.io

- The Keras blog provides updates and tutorials on the platform.
  http://blog.keras.io

- The Keras project on GitHub hosts the code for the library. Importantly, you can use the issue tracker on the project to raise concerns and make feature requests after you have used the library.
  https://github.com/fchollet/keras

- The best place to get answers to your Keras questions is on the Google group email list.
  https://groups.google.com/forum/#!forum/keras-users

- A good secondary place to ask questions is Stack Overflow and use the Keras tag with your question.
  http://stackoverflow.com/questions/tagged/keras

I am always here to help if have any questions. You can email me directly via jason@MachineLearningMastery.com and put this book title in the subject of your email.