

Deep Learning for Natural Language Processing

Develop Deep Learning Models for
Natural Language in Python

Jason Brownlee

MACHINE
LEARNING
MASTERY



Disclaimer

The information contained within this eBook is strictly for educational purposes. If you wish to apply ideas contained in this eBook, you are taking full responsibility for your actions.

The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause.

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic or mechanical, recording or by any information storage and retrieval system, without written permission from the author.

Acknowledgements

Special thanks to my copy editor Sarah Martin and my technical editors Arun Koshy and Andrei Cheremskoy.

Copyright

Deep Learning for Natural Language Processing

© Copyright 2017 Jason Brownlee. All Rights Reserved.

Edition: v1.1

Contents

Copyright	i
Contents	ii
Preface	iii
I Introductions	iv
Welcome	v
Who Is This Book For?	v
About Your Outcomes	vi
How to Read This Book	vi
About the Book Structure	vii
About Python Code Examples	viii
About Further Reading	viii
About Getting Help	ix
Summary	ix
II Foundations	1
1 Natural Language Processing	2
1.1 Natural Language	2
1.2 Challenge of Natural Language	3
1.3 From Linguistics to Natural Language Processing	3
1.4 Natural Language Processing	5
1.5 Further Reading	6
1.6 Summary	7
2 Deep Learning	8
2.1 Deep Learning is Large Neural Networks	8
2.2 Deep Learning is Hierarchical Feature Learning	11
2.3 Deep Learning as Scalable Learning Across Domains	12
2.4 Further Reading	13
2.5 Summary	14

3 Promise of Deep Learning for Natural Language	16
3.1 Promise of Deep Learning	16
3.2 Promise of Drop-in Replacement Models	17
3.3 Promise of New NLP Models	17
3.4 Promise of Feature Learning	18
3.5 Promise of Continued Improvement	18
3.6 Promise of End-to-End Models	19
3.7 Further Reading	20
3.8 Summary	20
4 How to Develop Deep Learning Models With Keras	21
4.1 Keras Model Life-Cycle	21
4.2 Keras Functional Models	26
4.3 Standard Network Models	27
4.4 Further Reading	32
4.5 Summary	33
III Data Preparation	34
5 How to Clean Text Manually and with NLTK	35
5.1 Tutorial Overview	35
5.2 Metamorphosis by Franz Kafka	36
5.3 Text Cleaning Is Task Specific	36
5.4 Manual Tokenization	37
5.5 Tokenization and Cleaning with NLTK	41
5.6 Additional Text Cleaning Considerations	46
5.7 Further Reading	46
5.8 Summary	47
6 How to Prepare Text Data with scikit-learn	48
6.1 The Bag-of-Words Model	48
6.2 Word Counts with CountVectorizer	49
6.3 Word Frequencies with TfidfVectorizer	50
6.4 Hashing with HashingVectorizer	51
6.5 Further Reading	52
6.6 Summary	53
7 How to Prepare Text Data With Keras	54
7.1 Tutorial Overview	54
7.2 Split Words with <code>text_to_word_sequence</code>	54
7.3 Encoding with <code>one_hot</code>	55
7.4 Hash Encoding with <code>hashing_trick</code>	56
7.5 <code>Tokenizer</code> API	57
7.6 Further Reading	59
7.7 Summary	59

IV Bag-of-Words	61
8 The Bag-of-Words Model	62
8.1 Tutorial Overview	62
8.2 The Problem with Text	62
8.3 What is a Bag-of-Words?	63
8.4 Example of the Bag-of-Words Model	63
8.5 Managing Vocabulary	65
8.6 Scoring Words	66
8.7 Limitations of Bag-of-Words	67
8.8 Further Reading	67
8.9 Summary	68
9 How to Prepare Movie Review Data for Sentiment Analysis	69
9.1 Tutorial Overview	69
9.2 Movie Review Dataset	70
9.3 Load Text Data	71
9.4 Clean Text Data	73
9.5 Develop Vocabulary	76
9.6 Save Prepared Data	80
9.7 Further Reading	83
9.8 Summary	84
10 Project: Develop a Neural Bag-of-Words Model for Sentiment Analysis	85
10.1 Tutorial Overview	85
10.2 Movie Review Dataset	86
10.3 Data Preparation	86
10.4 Bag-of-Words Representation	92
10.5 Sentiment Analysis Models	98
10.6 Comparing Word Scoring Methods	103
10.7 Predicting Sentiment for New Reviews	108
10.8 Extensions	112
10.9 Further Reading	112
10.10 Summary	113
V Word Embeddings	114
11 The Word Embedding Model	115
11.1 Overview	115
11.2 What Are Word Embeddings?	115
11.3 Word Embedding Algorithms	116
11.4 Using Word Embeddings	119
11.5 Further Reading	120
11.6 Summary	121

12 How to Develop Word Embeddings with Gensim	122
12.1 Tutorial Overview	122
12.2 Word Embeddings	123
12.3 Gensim Python Library	123
12.4 Develop Word2Vec Embedding	123
12.5 Visualize Word Embedding	126
12.6 Load Google's Word2Vec Embedding	128
12.7 Load Stanford's GloVe Embedding	129
12.8 Further Reading	131
12.9 Summary	132
13 How to Learn and Load Word Embeddings in Keras	133
13.1 Tutorial Overview	133
13.2 Word Embedding	134
13.3 Keras Embedding Layer	134
13.4 Example of Learning an Embedding	135
13.5 Example of Using Pre-Trained GloVe Embedding	138
13.6 Tips for Cleaning Text for Word Embedding	142
13.7 Further Reading	142
13.8 Summary	143
VI Text Classification	144
14 Neural Models for Document Classification	145
14.1 Overview	145
14.2 Word Embeddings + CNN = Text Classification	146
14.3 Use a Single Layer CNN Architecture	147
14.4 Dial in CNN Hyperparameters	148
14.5 Consider Character-Level CNNs	150
14.6 Consider Deeper CNNs for Classification	151
14.7 Further Reading	152
14.8 Summary	152
15 Project: Develop an Embedding + CNN Model for Sentiment Analysis	153
15.1 Tutorial Overview	153
15.2 Movie Review Dataset	153
15.3 Data Preparation	154
15.4 Train CNN With Embedding Layer	160
15.5 Evaluate Model	167
15.6 Extensions	171
15.7 Further Reading	172
15.8 Summary	173

16 Project: Develop an n-gram CNN Model for Sentiment Analysis	174
16.1 Tutorial Overview	174
16.2 Movie Review Dataset	174
16.3 Data Preparation	175
16.4 Develop Multichannel Model	179
16.5 Evaluate Model	185
16.6 Extensions	186
16.7 Further Reading	187
16.8 Summary	187
VII Language Modeling	189
17 Neural Language Modeling	190
17.1 Overview	190
17.2 Problem of Modeling Language	190
17.3 Statistical Language Modeling	191
17.4 Neural Language Models	192
17.5 Further Reading	194
17.6 Summary	196
18 How to Develop a Character-Based Neural Language Model	197
18.1 Tutorial Overview	197
18.2 Sing a Song of Sixpence	198
18.3 Data Preparation	198
18.4 Train Language Model	201
18.5 Generate Text	206
18.6 Further Reading	209
18.7 Summary	209
19 How to Develop a Word-Based Neural Language Model	211
19.1 Tutorial Overview	211
19.2 Framing Language Modeling	212
19.3 Jack and Jill Nursery Rhyme	212
19.4 Model 1: One-Word-In, One-Word-Out Sequences	212
19.5 Model 2: Line-by-Line Sequence	218
19.6 Model 3: Two-Words-In, One-Word-Out Sequence	222
19.7 Further Reading	224
19.8 Summary	225
20 Project: Develop a Neural Language Model for Text Generation	226
20.1 Tutorial Overview	226
20.2 The Republic by Plato	227
20.3 Data Preparation	227
20.4 Train Language Model	233
20.5 Use Language Model	239
20.6 Extensions	243

20.7 Further Reading	244
20.8 Summary	244
VIII Image Captioning	245
21 Neural Image Caption Generation	246
21.1 Overview	246
21.2 Describing an Image with Text	246
21.3 Neural Captioning Model	247
21.4 Encoder-Decoder Architecture	249
21.5 Further Reading	250
21.6 Summary	251
22 Neural Network Models for Caption Generation	252
22.1 Image Caption Generation	252
22.2 Inject Model	253
22.3 Merge Model	254
22.4 More on the Merge Model	255
22.5 Further Reading	256
22.6 Summary	256
23 How to Load and Use a Pre-Trained Object Recognition Model	257
23.1 Tutorial Overview	257
23.2 ImageNet	258
23.3 The Oxford VGG Models	259
23.4 Load the VGG Model in Keras	259
23.5 Develop a Simple Photo Classifier	263
23.6 Further Reading	266
23.7 Summary	266
24 How to Evaluate Generated Text With the BLEU Score	268
24.1 Tutorial Overview	268
24.2 Bilingual Evaluation Understudy Score	268
24.3 Calculate BLEU Scores	270
24.4 Cumulative and Individual BLEU Scores	271
24.5 Worked Examples	273
24.6 Further Reading	275
24.7 Summary	276
25 How to Prepare a Photo Caption Dataset For Modeling	277
25.1 Tutorial Overview	277
25.2 Download the Flickr8K Dataset	278
25.3 How to Load Photographs	279
25.4 Pre-Calculate Photo Features	280
25.5 How to Load Descriptions	282
25.6 Prepare Description Text	284

25.7 Whole Description Sequence Model	287
25.8 Word-By-Word Model	290
25.9 Progressive Loading	293
25.10 Further Reading	297
25.11 Summary	298
26 Project: Develop a Neural Image Caption Generation Model	299
26.1 Tutorial Overview	299
26.2 Photo and Caption Dataset	300
26.3 Prepare Photo Data	301
26.4 Prepare Text Data	303
26.5 Develop Deep Learning Model	307
26.6 Evaluate Model	318
26.7 Generate New Captions	324
26.8 Extensions	329
26.9 Further Reading	329
26.10 Summary	331
IX Machine Translation	332
27 Neural Machine Translation	333
27.1 What is Machine Translation?	333
27.2 What is Statistical Machine Translation?	334
27.3 What is Neural Machine Translation?	335
27.4 Further Reading	337
27.5 Summary	338
28 What are Encoder-Decoder Models for Neural Machine Translation	339
28.1 Encoder-Decoder Architecture for NMT	339
28.2 Sutskever NMT Model	340
28.3 Cho NMT Model	342
28.4 Further Reading	345
28.5 Summary	346
29 How to Configure Encoder-Decoder Models for Machine Translation	347
29.1 Encoder-Decoder Model for Neural Machine Translation	347
29.2 Baseline Model	348
29.3 Word Embedding Size	349
29.4 RNN Cell Type	349
29.5 Encoder-Decoder Depth	350
29.6 Direction of Encoder Input	350
29.7 Attention Mechanism	351
29.8 Inference	351
29.9 Final Model	352
29.10 Further Reading	352
29.11 Summary	353

30 Project: Develop a Neural Machine Translation Model	354
30.1 Tutorial Overview	354
30.2 German to English Translation Dataset	354
30.3 Preparing the Text Data	355
30.4 Train Neural Translation Model	359
30.5 Evaluate Neural Translation Model	366
30.6 Extensions	370
30.7 Further Reading	371
30.8 Summary	372
X Appendix	373
A Getting Help	374
A.1 Official Keras Destinations	374
A.2 Where to Get Help with Keras	374
A.3 Where to Get Help with Natural Language	375
A.4 How to Ask Questions	375
A.5 Contact the Author	375
B How to Setup a Workstation for Deep Learning	376
B.1 Overview	376
B.2 Download Anaconda	376
B.3 Install Anaconda	378
B.4 Start and Update Anaconda	380
B.5 Install Deep Learning Libraries	383
B.6 Further Reading	384
B.7 Summary	384
C How to Use Deep Learning in the Cloud	385
C.1 Overview	385
C.2 Setup Your AWS Account	386
C.3 Launch Your Server Instance	387
C.4 Login, Configure and Run	391
C.5 Build and Run Models on AWS	392
C.6 Close Your EC2 Instance	393
C.7 Tips and Tricks for Using Keras on AWS	395
C.8 Further Reading	395
C.9 Summary	395
XI Conclusions	396
How Far You Have Come	397

Preface

We are awash with text, from books, papers, blogs, tweets, news, and increasingly text from spoken utterances. Every day, I get questions asking how to develop machine learning models for text data. Working with text is hard as it requires drawing upon knowledge from diverse domains such as linguistics, machine learning, statistical natural language processing, and these days, deep learning.

I have done my best to write blog posts to answer frequently asked questions on the topic and decided to pull together my best knowledge on the matter into this book. I designed this book to teach you step-by-step how to bring modern deep learning methods to your natural language processing projects. I chose the programming language, programming libraries, and tutorial topics to give you the skills you need.

Python is the go-to language for applied machine learning and deep learning, both in terms of demand from employers and employees. This is not least because it could be a renaissance for machine learning tools. I have focused on showing you how to use the best of breed Python tools for natural language processing such as Gensim and NLTK, and even a little scikit-learn. Key to getting results is speed of development, and for this reason, we use the Keras deep learning library as you can define, train, and use complex deep learning models with just a few lines of Python code.

There are three key areas that you must know when working with text:

1. How to clean text. This includes loading, analyzing, filtering and cleaning tasks required prior to modeling.
2. How to represent text. This includes the classical bag-of-words model and the modern and powerful distributed representation in word embeddings.
3. How to generate text. This includes the range of most interesting problems, such as image captioning and translation.

These key topics provide the backbone for the book and the tutorials you will work through. I believe that after completing this book, you will have the skills that you need to both work through your own natural language processing projects and bring modern deep learning methods to bare.

Jason Brownlee
2017

Part I

Introductions

Welcome

Welcome to *Deep Learning for Natural Language Processing*. Natural language processing is the area of study dedicated to the automatic manipulation of speech and text by software. It is an old field of study, originally dominated by rule-based methods designed by linguists, then statistical methods, and, more recently, deep learning methods that show great promise in the field. So much so that the heart of the Google Translate service uses a deep learning method, a topic that you will learn more about in this book.

I designed this book to teach you step-by-step how to bring modern deep learning models to your own natural language processing projects.

Who Is This Book For?

Before we get started, let's make sure you are in the right place. This book is for developers that know some applied machine learning and some deep learning. Maybe you want or need to start using deep learning for text on your research project or on a project at work. This guide was written to help you do that quickly and efficiently by compressing years worth of knowledge and experience into a laser-focused course of hands-on tutorials. The lessons in this book assume a few things about you, such as:

- You know your way around basic Python for programming.
- You know your way around basic NumPy for array manipulation.
- You know your way around basic scikit-learn for machine learning.
- You know your way around basic Keras for deep learning.

For some bonus points, perhaps some of the below points apply to you. Don't panic if they don't.

- You may know how to work through a predictive modeling problem end-to-end.
- You may know a little bit of natural language processing.
- You may know a little bit of natural language libraries such as NLTK or Gensim.

This guide was written in the top-down and results-first machine learning style that you're used to from MachineLearningMastery.com.

About Your Outcomes

This book will teach you how to get results as a machine learning practitioner interested in using deep learning on your natural language processing project. After reading and working through this book, you will know:

- What natural language processing is and why it is challenging.
- What deep learning is and how it is different from other machine learning methods.
- The promise of deep learning methods for natural language processing problems.
- How to prepare text data for modeling using best-of-breed Python libraries.
- How to develop distributed representations of text using word embedding models.
- How to develop a bag-of-words model, a representation technique that can be used for machine learning and deep learning methods.
- How to develop a neural sentiment analysis model for automatically predicting the class label for a text document.
- How to develop a neural language model, required for any text generating neural network.
- How to develop a photo captioning system to automatically generate textual descriptions of photographs.
- How to develop a neural machine translation system for translating text from one language to another.

This book will NOT teach you how to be a research scientist and all the theory behind why specific methods work. For that, I would recommend good research papers and textbooks. See the *Further Reading* section at the end of each tutorial for a good starting point.

How to Read This Book

This book was written to be read linearly, from start to finish. That being said, if you know the basics and need help with a specific problem type or technique, then you can flip straight to that section and get started. This book was designed for you to read on your workstation, on the screen, not on an eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them.

This book is not intended to be read passively or be placed in a folder as a reference text. It is a playbook, a workbook, and a guidebook intended for you to learn by doing and then apply your new understanding to your own natural language projects. To get the most out of the book, I would recommend playing with the examples in each tutorial. Extend them, break them, then fix them. Try some of the extensions presented at the end of each lesson and let me know how you do.

About the Book Structure

This book was designed around major activities, techniques, and natural language processing problems. There are a lot of things you could learn about deep learning and natural language processing, from theory to applications to APIs. My goal is to take you straight to getting results with laser-focused tutorials. I designed the tutorials to focus on how to get things done. They give you the tools to both rapidly understand and apply each technique to your own natural language processing prediction problems.

Each of the tutorials are designed to take you about one hour to read through and complete, excluding the extensions and further reading. You can choose to work through the lessons one per day, one per week, or at your own pace. I think momentum is critically important, and this book is intended to be read and used, not to sit idle. I would recommend picking a schedule and sticking to it. The tutorials are divided into eight parts:

- **Part 1: Foundations.** Discover a gentle introduction to natural language processing, deep learning, and the promise of combining the two, as well as tutorials on how to get started with Keras.
- **Part 2: Data Preparation:** Discover tutorials that show how to clean, prepare and encode text ready for modeling with neural networks.
- **Part 3: Bag-of-Words.** Discover the bag-of-words model, a staple representation for machine learning and a good starting point for neural networks for sentiment analysis.
- **Part 4: Word Embeddings.** Discover a more powerful word representation in word embeddings, how to develop them as standalone models, and how to learn them as part of neural network models.
- **Part 5: Text Classification.** Discover how to leverage word embeddings and convolutional neural networks to learn spatial invariant models of text for sentiment analysis, a successor to the bag-of-words model.
- **Part 6: Language Modeling.** Discover how to develop character-based and word-based language models, a technique that is required as part of any modern text generating model.
- **Part 7: Image Captioning.** Discover how to combine a pre-trained object recognition model with a language model to automatically caption images.
- **Part 8: Machine Translation.** Discover how to combine two language models to automatically translate text from one language to another.

Each part targets a specific learning outcome, and so does each tutorial within each part. This acts as a filter to ensure you are only focused on the things you need to know to get to a specific result and do not get bogged down in the math or near-infinite number of configuration parameters. The tutorials were not designed to teach you everything there is to know about each of the techniques or natural language processing problems. They were designed to give you an understanding of how they work, how to use them on your projects the fastest way I know how: to learn by doing.

About Python Code Examples

The code examples were carefully designed to demonstrate the purpose of a given lesson. For this reason, the examples are highly targeted.

- Models were demonstrated on real-world datasets to give you the context and confidence to bring the techniques to your own natural language processing problems.
- Model configurations used were discovered through trial and error are skillful, but not optimized. This leaves the door open for you to explore new and possibly better configurations.
- Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third parties required beyond the installation of the required packages.

A complete working example is presented with each tutorial for you to inspect and copy-and-paste. All source code is also provided with the book and I would recommend running the provided files whenever possible to avoid any copy-paste issues. The provided code was developed in a text editor and intended to be run on the command line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead.

Neural network algorithms are stochastic. This means that they will make different predictions when the same model configuration is trained on the same training data. On top of that, each experimental problem in this book is based around generating stochastic predictions. As a result, this means you will not get exactly the same sample output presented in this book. This is by design. I want you to get used to the stochastic nature of the neural network algorithms. If this bothers you, please note:

- You can re-run a given example a few times and your results should be close to the values reported.
- You can make the output consistent by fixing the NumPy random number seed.
- You can develop a robust estimate of the skill of a model by fitting and evaluating it multiple times and taking the average of the final skill score (highly recommended).

All code examples were tested on a POSIX-compatiable machine with Python 3 and Keras 2. All code examples will run on modest and modern computer hardware and were executed on a CPU. No GPUs are required to run the presented examples, although a GPU would make the code run faster. I am only human and there may be a bug in the sample code. If you discover a bug, please let me know so I can fix it and update the book and send out a free update.

About Further Reading

Each lesson includes a list of further reading resources. This may include:

- Research papers.

- Books and book chapters.
- Webpages.
- API documentation.

Wherever possible, I try to list and link to the relevant API documentation for key objects and functions used in each lesson so you can learn more about them. When it comes to research papers, I try to list papers that are first to use a specific technique or first in a specific problem domain. These are not required reading, but can give you more technical details, theory, and configuration details if you're looking for it. Wherever possible, I have tried to link to the freely available version of the paper on arxiv.org. You can search for and download any of the papers listed on Google Scholar Search scholar.google.com. Wherever possible, I have tried to link to books on Amazon. I don't know everything, and if you discover a good resource related to a given lesson, please let me know so I can update the book.

About Getting Help

You might need help along the way. Don't worry, you are not alone.

- *Help with a Technique?* If you need help with the technical aspects of a specific model or method, see the *Further Reading* sections at the end of each lesson.
- *Help with Keras?* If you need help with using the Keras library, see the list of resources in Appendix A.
- *Help with your workstation?* If you need help setting up your environment, I would recommend using Anaconda and following my tutorial in Appendix B.
- *Help running large models?* I recommend renting time on Amazon Web Service (AWS) to run large models. If you need help getting started on AWS, see the tutorial in Appendix C.
- *Help in general?* You can shoot me an email. My details are in Appendix A.

Summary

Are you ready? Let's dive in!

Next up you will discover a concrete idea of what natural language processing actually means.

Part II

Foundations

Chapter 1

Natural Language Processing

Natural Language Processing, or NLP for short, is broadly defined as the automatic manipulation of natural language, like speech and text, by software. The study of natural language processing has been around for more than 50 years and grew out of the field of linguistics with the rise of computers. In this chapter, you will discover what natural language processing is and why it is so important. After reading this chapter, you will know:

- What natural language is and how it is different from other types of data.
- What makes working with natural language so challenging.
- Where the field of NLP came from and how it is defined by modern practitioners.

Let's get started.

1.1 Natural Language

Natural language refers to the way we, humans, communicate with each other. Namely, speech and text. We are surrounded by text. Think about how much text you see each day:

- Signs
- Menus
- Email
- SMS
- Web Pages
- and so much more...

The list is endless. Now think about speech. We may speak to each other, as a species, more than we write. It may even be easier to learn to speak than to write. Voice and text are how we communicate with each other. Given the importance of this type of data, we must have methods to understand and reason about natural language, just like we do for other types of data.

1.2 Challenge of Natural Language

Working with natural language data is not solved. It has been studied for half a century, and it is really hard.

It is hard from the standpoint of the child, who must spend many years acquiring a language ... it is hard for the adult language learner, it is hard for the scientist who attempts to model the relevant phenomena, and it is hard for the engineer who attempts to build systems that deal with natural language input or output. These tasks are so hard that Turing could rightly make fluent conversation in natural language the centerpiece of his test for intelligence.

— Page 248, *Mathematical Linguistics*, 2010.

Natural language is primarily hard because it is messy. There are few rules. And yet we can easily understand each other most of the time.

Human language is highly ambiguous ... It is also ever changing and evolving. People are great at producing language and understanding language, and are capable of expressing, perceiving, and interpreting very elaborate and nuanced meanings. At the same time, while we humans are great users of language, we are also very poor at formally understanding and describing the rules that govern language.

— Page 1, *Neural Network Methods in Natural Language Processing*, 2017.

1.3 From Linguistics to Natural Language Processing

1.3.1 Linguistics

Linguistics is the scientific study of language, including its grammar, semantics, and phonetics. Classical linguistics involved devising and evaluating rules of language. Great progress was made on formal methods for syntax and semantics, but for the most part, the interesting problems in natural language understanding resist clean mathematical formalisms.

Broadly, a linguist is anyone who studies language, but perhaps more colloquially, a self-defining linguist may be more focused on being out in the field. Mathematics is the tool of science. Mathematicians working on natural language may refer to their study as mathematical linguistics, focusing exclusively on the use of discrete mathematical formalisms and theory for natural language (e.g. formal languages and automata theory).

1.3.2 Computational Linguistics

Computational linguistics is the modern study of linguistics using the tools of computer science. Yesterday's linguistics may be today's computational linguist as the use of computational tools and thinking has overtaken most fields of study.

Computational linguistics is the study of computer systems for understanding and generating natural language. ... One natural function for computational linguistics would be the testing of grammars proposed by theoretical linguists.

— Pages 4-5, *Computational Linguistics: An Introduction*, 1986.

Large data and fast computers mean that new and different things can be discovered from large datasets of text by writing and running software. In the 1990s, statistical methods and statistical machine learning began to and eventually replaced the classical top-down rule-based approaches to language, primarily because of their better results, speed, and robustness. The statistical approach to studying natural language now dominates the field; it may define the field.

Data-Driven methods for natural language processing have now become so popular that they must be considered mainstream approaches to computational linguistics.

... A strong contributing factor to this development is undoubtedly the increase amount of available electronically stored data to which these methods can be applied; another factor might be a certain disenchantment with approaches relying exclusively on hand-crafted rules, due to their observed brittleness.

— Page 358, *The Oxford Handbook of Computational Linguistics*, 2005.

The statistical approach to natural language is not limited to statistics per-se, but also to advanced inference methods like those used in applied machine learning.

... understanding natural language require large amounts of knowledge about morphology, syntax, semantics and pragmatics as well as general knowledge about the world. Acquiring and encoding all of this knowledge is one of the fundamental impediments to developing effective and robust language systems. Like the statistical methods ... machine learning methods off the promise of te automatic acquisition of this knowledge from annotated or unannotated language corpora.

— Page 377, *The Oxford Handbook of Computational Linguistics*, 2005.

1.3.3 Statistical Natural Language Processing

Computational linguistics also became known by the name of natural language process, or NLP, to reflect the more engineer-based or empirical approach of the statistical methods. The statistical dominance of the field also often leads to NLP being described as Statistical Natural Language Processing, perhaps to distance it from the classical computational linguistics methods.

I view computational linguistics as having both a scientific and an engineering side. The engineering side of computational linguistics, often called natural language processing (NLP), is largely concerned with building computational tools that do useful things with language, e.g., machine translation, summarization, question-answering, etc. Like any engineering discipline, natural language processing draws on a variety of different scientific disciplines.

— *How the statistical revolution changes (computational) linguistics*, 2009.

Linguistics is a large topic of study, and, although the statistical approach to NLP has shown great success in some areas, there is still room and great benefit from the classical top-down methods.

Roughly speaking, statistical NLP associates probabilities with the alternatives encountered in the course of analyzing an utterance or a text and accepts the most probable outcome as the correct one. ... Not surprisingly, words that name phenomena that are closely related in the world, or our perception of it, frequently occur close to one another so that crisp facts about the world are reflected in somewhat fuzzier facts about texts. There is much room for debate in this view.

— Page xix, *The Oxford Handbook of Computational Linguistics*, 2005.

1.4 Natural Language Processing

As machine learning practitioners interested in working with text data, we are concerned with the tools and methods from the field of Natural Language Processing. We have seen the path from linguistics to NLP in the previous section. Now, let's take a look at how modern researchers and practitioners define what NLP is all about. In perhaps one of the more widely known textbooks written by top researchers in the field, they refer to the subject as *linguistic science*, permitting discussion of both classical linguistics and modern statistical methods.

The aim of a linguistic science is to be able to characterize and explain the multitude of linguistic observations circling around us, in conversations, writing, and other media. Part of that has to do with the cognitive size of how humans acquire, produce and understand language, part of it has to do with understanding the relationship between linguistic utterances and the world, and part of it has to do with understand the linguistic structures by which language communicates.

— Page 3, *Foundations of Statistical Natural Language Processing*, 1999.

They go on to focus on inference through the use of statistical methods in natural language processing.

Statistical NLP aims to do statistical inference for the field of natural language. Statistical inference in general consists of taking some data (generated in accordance with some unknown probability distribution) and then making some inference about this distribution.

— Page 191, *Foundations of Statistical Natural Language Processing*, 1999.

In their text on applied natural language processing, the authors and contributors to the popular NLTK Python library for NLP describe the field broadly as using computers to work with natural language data.

We will take Natural Language Processing - or NLP for short - in a wide sense to cover any kind of computer manipulation of natural language. At one extreme, it could be as simple as counting word frequencies to compare different writing styles. At the other extreme, NLP involves “understanding” complete human utterances, at least to the extent of being able to give useful responses to them.

— Page ix, *Natural Language Processing with Python*, 2009.

Statistical NLP has turned another corner and is now strongly focused on the use of deep learning neural networks to both perform inference on specific tasks and for developing robust end-to-end systems. In one of the first textbooks dedicated to this emerging topic, Yoav Goldberg succinctly defines NLP as automatic methods that take natural language as input or produce natural language as output.

Natural language processing (NLP) is a collective term referring to automatic computational processing of human languages. This includes both algorithms that take human-produced text as input, and algorithms that produce natural looking text as outputs.

— Page xvii, *Neural Network Methods in Natural Language Processing*, 2017.

1.5 Further Reading

This section provides more resources on the topic if you are looking go deeper.

1.5.1 Books

- *Mathematical Linguistics*, 2010.
<http://amzn.to/2t01c00>
- *Neural Network Methods in Natural Language Processing*, 2017.
<http://amzn.to/2u0JtPl>
- *Computational Linguistics: An Introduction*, 1986.
<http://amzn.to/2h6U4qY>
- *The Oxford Handbook of Computational Linguistics*, 2005.
<http://amzn.to/2uHeERE>
- *Foundations of Statistical Natural Language Processing*, 1999.
<http://amzn.to/2uzwxDE>
- *Natural Language Processing with Python*, 2009.
<http://amzn.to/2uZMF27>

1.5.2 Wikipedia

- Linguistics.
<https://en.wikipedia.org/wiki/Linguistics>
- Computational linguistics.
https://en.wikipedia.org/wiki/Computational_linguistics
- Natural language processing.
https://en.wikipedia.org/wiki/Natural_language_processing

- History of natural language processing.
https://en.wikipedia.org/wiki/History_of_natural_language_processing
- Outline of natural language processing.
https://en.wikipedia.org/wiki/Outline_of_natural_language_processing

1.6 Summary

In this chapter, you discovered what natural language processing is why it is so important. Specifically, you learned:

- What natural language is and how it is different from other types of data.
- What makes working with natural language so challenging.
- Where the field of NLP came from and how it is defined by modern practitioners.

1.6.1 Next

In the next chapter, you will discover what deep learning is and the motivation behind using deep artificial neural networks.

Chapter 2

Deep Learning

Deep Learning is a subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks. If you are just starting out in the field of deep learning or you had some experience with neural networks some time ago, you may be confused. I know I was confused initially and so were many of my colleagues and friends who learned and used neural networks in the 1990s and early 2000s.

The leaders and experts in the field have ideas of what deep learning is and these specific and nuanced perspectives shed a lot of light on what deep learning is all about. In this chapter, you will discover exactly what deep learning is by hearing from a range of experts and leaders in the field. After reading this chapter, you will know:

- The motivation for exploring and adopting large neural network models.
- The perspective on deep learning as hierarchical feature learning.
- The promise of scalability of deep learning with the size of data.

Let's dive in.

2.1 Deep Learning is Large Neural Networks

Andrew Ng from Coursera and formally Chief Scientist at Baidu Research and founder of Google Brain that eventually resulted in the productization of deep learning technologies across a large number of Google services. He has spoken and written a lot about what deep learning is and is a good place to start. In early talks on deep learning, Andrew described deep learning in the context of traditional artificial neural networks. In the 2013 talk titled *Deep Learning, Self-Taught Learning and Unsupervised Feature Learning* he described the idea of deep learning as:

Using brain simulations, hope to:

- Make learning algorithms much better and easier to use.
- Make revolutionary advances in machine learning and AI.

I believe this is our best shot at progress towards real AI

— *Deep Learning, Self-Taught Learning and Unsupervised Feature Learning*, 2013.

Later his comments became more nuanced. The core of deep learning according to Andrew is that we now have fast enough computers and enough data to actually train large neural networks. When discussing why now is the time that deep learning is taking off at ExtractConf 2015 in a talk titled *What data scientists should know about deep learning*, he commented:

... very large neural networks we can now have and ... huge amounts of data that we have access to

— *What data scientists should know about deep learning*, 2015.

He also commented on the important point that it is all about scale. That as we construct larger neural networks and train them with more and more data, their performance continues to increase. This is generally different to other machine learning techniques that reach a plateau in performance.

... for most flavors of the old generations of learning algorithms ... performance will plateau. ... deep learning ... is the first class of algorithms ... that is scalable. ... performance just keeps getting better as you feed them more data

— *What data scientists should know about deep learning*, 2015.

He provides a nice cartoon of this in his slides:

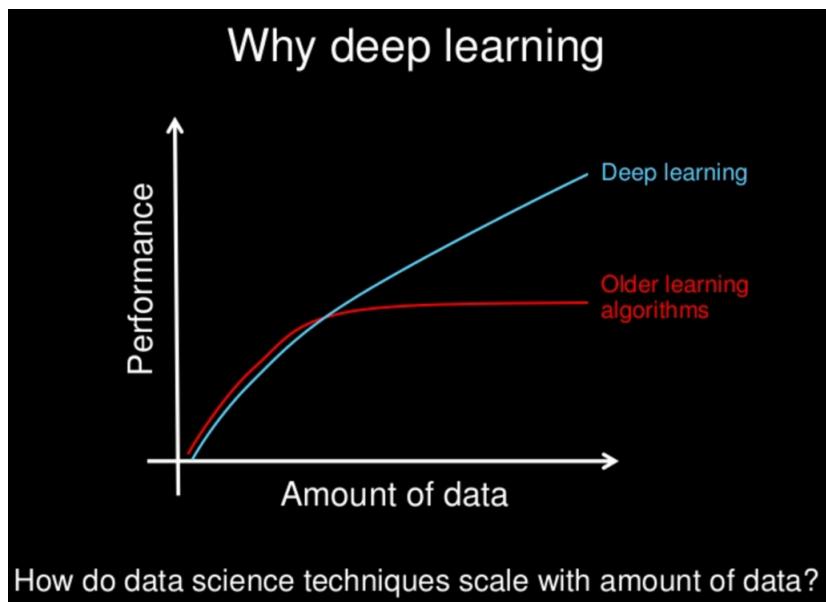


Figure 2.1: Why Deep Learning? Slide by Andrew Ng, taken from *What data scientists should know about deep learning*.

Finally, he is clear to point out that the benefits from deep learning that we are seeing in practice come from supervised learning. From the 2015 ExtractConf talk, he commented:

... almost all the value today of deep learning is through supervised learning or learning from labeled data

— *What data scientists should know about deep learning, 2015.*

Earlier at a talk to Stanford University titled *Deep Learning* in 2014 he made a similar comment:

... one reason that deep learning has taken off like crazy is because it is fantastic at supervised learning

— *Invited Talk: Andrew Ng (Stanford University): Deep Learning, 2014.*

Andrew often mentions that we should and will see more benefits coming from the unsupervised side of the tracks as the field matures to deal with the abundance of unlabeled data available. Jeff Dean is a Wizard and Google Senior Fellow in the Systems and Infrastructure Group at Google and has been involved and perhaps partially responsible for the scaling and adoption of deep learning within Google. Jeff was involved in the Google Brain project and the development of large-scale deep learning software DistBelief and later TensorFlow. In a 2016 talk titled *Deep Learning for Building Intelligent Computer Systems* he made a comment in the similar vein, that deep learning is really all about large neural networks.

When you hear the term deep learning, just think of a large deep neural net. Deep refers to the number of layers typically and so this kind of the popular term that's been adopted in the press. I think of them as deep neural networks generally.

— *Deep Learning for Building Intelligent Computer Systems, 2016.*

He has given this talk a few times, and in a modified set of slides for the same talk, he highlights the scalability of neural networks indicating that results get better with more data and larger models, that in turn require more computation to train.

Important Property of Neural Networks

Results get better with

**more data +
bigger models +
more computation**

**(Better algorithms, new insights and improved
techniques always help, too!)**



Figure 2.2: Results Get Better With More Data, Larger Models, More Compute. Taken from *Deep Learning for Building Intelligent Computer Systems*.

2.2 Deep Learning is Hierarchical Feature Learning

In addition to scalability, another often cited benefit of deep learning models is their ability to perform automatic feature extraction from raw data, also called feature learning. Yoshua Bengio is another leader in deep learning although began with a strong interest in the automatic feature learning that large neural networks are capable of achieving. He describes deep learning in terms of the algorithms ability to discover and learn good representations using feature learning. In his 2012 paper titled *Deep Learning of Representations for Unsupervised and Transfer Learning* he commented:

Deep learning algorithms seek to exploit the unknown structure in the input distribution in order to discover good representations, often at multiple levels, with higher-level learned features defined in terms of lower-level features

— *Deep Learning of Representations for Unsupervised and Transfer Learning*, 2012.

An elaborated perspective of deep learning along these lines is provided in his 2009 technical report titled *Learning deep architectures for AI* where he emphasizes the importance the hierarchy in feature learning.

Deep learning methods aim at learning feature hierarchies with features from higher levels of the hierarchy formed by the composition of lower level features. Automatically learning features at multiple levels of abstraction allow a system to learn complex functions mapping the input to the output directly from data, without depending completely on human-crafted features.

— *Learning deep architectures for AI*, 2009.

In the published book titled *Deep Learning* co-authored with Ian Goodfellow and Aaron Courville, they define deep learning in terms of the depth of the architecture of the models.

The hierarchy of concepts allows the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph showing how these concepts are built on top of each other, the graph is deep, with many layers. For this reason, we call this approach to AI deep learning.

— *Deep Learning*, 2016.

This is an important book and will likely may be the definitive resource for the field for some time. The book goes on to describe Multilayer Perceptrons as an algorithm used in the field of deep learning, giving the idea that deep learning has subsumed artificial neural networks.

The quintessential example of a deep learning model is the feedforward deep network or multilayer perceptron (MLP).

— *Deep Learning*, 2016.

Peter Norvig is the Director of Research at Google and famous for his textbook on AI titled *Artificial Intelligence: A Modern Approach*. In a 2016 talk he gave titled *Deep Learning and Understandability versus Software Engineering and Verification* he defined deep learning in a very similar way to Yoshua, focusing on the power of abstraction permitted by using a deeper network structure.

a kind of learning where the representation you form have several levels of abstraction, rather than a direct input to output

— *Deep Learning and Understandability versus Software Engineering and Verification*, 2016.

2.3 Deep Learning as Scalable Learning Across Domains

Deep learning excels on problem domains where the inputs (and even output) are analog. Meaning, they are not a few quantities in a tabular format but instead are images of pixel data, documents of text data or files of audio data. Yann LeCun is the director of Facebook Research and is the father of the network architecture that excels at object recognition in image data called the Convolutional Neural Network (CNN). This technique is seeing great success because like multilayer perceptron feedforward neural networks, the technique scales with data and model size and can be trained with backpropagation.

This biases his definition of deep learning as the development of very large CNNs, which have had great success on object recognition in photographs. In a 2016 talk at Lawrence Livermore National Laboratory titled *Accelerating Understanding: Deep Learning, Intelligent Applications, and GPUs* he described deep learning generally as learning hierarchical representations and defines it as a scalable approach to building object recognition systems:

deep learning [is] ... a pipeline of modules all of which are trainable. ... deep because [has] multiple stages in the process of recognizing an object and all of those stages are part of the training

— *Accelerating Understanding: Deep Learning, Intelligent Applications, and GPUs*, 2016.

Jurgen Schmidhuber is the father of another popular algorithm that like MLPs and CNNs also scales with model size and dataset size and can be trained with backpropagation, but is instead tailored to learning sequence data, called the Long Short-Term Memory Network (LSTM), a type of recurrent neural network. We do see some confusion in the phrasing of the field as *deep learning*. In his 2014 paper titled *Deep Learning in Neural Networks: An Overview* he does comment on the problematic naming of the field and the differentiation of deep from shallow learning. He also interestingly describes depth in terms of the complexity of the problem rather than the model used to solve the problem.

At which problem depth does Shallow Learning end, and Deep Learning begin? Discussions with DL experts have not yet yielded a conclusive response to this question. [...], let me just define for the purposes of this overview: problems of depth > 10 require Very Deep Learning.

— *Deep Learning in Neural Networks: An Overview*, 2014.

Demis Hassabis is the founder of DeepMind, later acquired by Google. DeepMind made the breakthrough of combining deep learning techniques with reinforcement learning to handle complex learning problems like game playing, famously demonstrated in playing Atari games and the game Go with Alpha Go. In keeping with the naming, they called their new technique a Deep Q-Network, combining Deep Learning with Q-Learning. They also name the broader field of study *Deep Reinforcement Learning*.

In their 2015 nature paper titled *Human-level control through deep reinforcement learning* they comment on the important role of deep neural networks in their breakthrough and highlight the need for hierarchical abstraction.

To achieve this, we developed a novel agent, a deep Q-network (DQN), which is able to combine reinforcement learning with a class of artificial neural network known as deep neural networks. Notably, recent advances in deep neural networks, in which several layers of nodes are used to build up progressively more abstract representations of the data, have made it possible for artificial neural networks to learn concepts such as object categories directly from raw sensory data.

— *Human-level control through deep reinforcement learning*, 2015.

Finally, in what may be considered a defining paper in the field, Yann LeCun, Yoshua Bengio and Geoffrey Hinton published a paper in Nature titled simply *Deep Learning*. In it, they open with a clean definition of deep learning highlighting the multilayered approach.

Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction.

— *Deep Learning*, 2015.

Later the multilayered approach is described in terms of representation learning and abstraction.

Deep-learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. [...] The key aspect of deep learning is that these layers of features are not designed by human engineers: they are learned from data using a general-purpose learning procedure.

— *Deep Learning*, 2015.

This is a nice and generic description, and could easily describe most artificial neural network algorithms. It is also a good note to end on.

2.4 Further Reading

This section provides more resources on the topic if you are looking go deeper.

2.4.1 Videos

- Deep Learning, Self-Taught Learning and Unsupervised Feature Learning, 2013.
<https://www.youtube.com/watch?v=n1ViNeWhC24>
- What data scientists should know about deep learning, 2015.
<https://www.youtube.com/watch?v=0OVN0pGgBZM>
- Invited Talk: Andrew Ng (Stanford University): Deep Learning 2014.
<https://www.youtube.com/watch?v=W15K9PegQt0>
- Deep Learning for Building Intelligent Computer Systems, 2016.
<https://www.youtube.com/watch?v=QSaZGT4-6EY>
- Deep Learning and Understandability versus Software Engineering and Verification, 2016.
<https://www.youtube.com/watch?v=X769cyzBNVw>
- Accelerating Understanding: Deep Learning, Intelligent Applications, and GPUs, 2016.
<https://www.youtube.com/watch?v=Qk4SqF9FT-M>

2.4.2 Books

- Deep Learning, 2016.
<http://amzn.to/2goLnbo>

2.4.3 Articles

- Deep Learning of Representations for Unsupervised and Transfer Learning, 2012.
<http://www.jmlr.org/proceedings/papers/v27/bengio12a/bengio12a.pdf>
- Learning deep architectures for AI, 2009.
<http://www.iro.umontreal.ca/~lisa/publications2/index.php/publications/show/239>
- Deep Learning in Neural Networks: An Overview, 2014.
<http://arxiv.org/pdf/1404.7828v4.pdf>
- Human-level control through deep reinforcement learning, 2015.
<http://www.nature.com/nature/journal/v518/n7540/full/nature14236.html>
- Deep Learning, 2015.
<http://www.nature.com/nature/journal/v521/n7553/full/nature14539.html>

2.5 Summary

In this chapter you discovered that deep learning is just very big neural networks on a lot more data, requiring bigger computers. Although early approaches published by Hinton and collaborators focus on greedy layer-wise training and unsupervised methods like autoencoders, modern state-of-the-art deep learning is focused on training deep (many layered) neural network

models using the backpropagation algorithm. The most popular techniques that we will focus on are:

- Multilayer Perceptron Networks (MLP).
- Convolutional Neural Networks (CNN).
- Long Short-Term Memory Recurrent Neural Networks (LSTM).

I hope this has cleared up what deep learning is and how leading definitions fit together under the one umbrella.

2.5.1 Next

In the next chapter, you will discover the promise of deep learning neural networks for the field of natural language processing.

Chapter 3

Promise of Deep Learning for Natural Language

The promise of deep learning in the field of natural language processing is the better performance by models that may require more data but less linguistic expertise to train and operate. There is a lot of hype and large claims around deep learning methods, but beyond the hype, deep learning methods are achieving state-of-the-art results on challenging problems. Notably in natural language processing. In this chapter, you will discover the specific promises that deep learning methods have for tackling natural language processing problems. After reading this chapter, you will know:

- The promises of deep learning for natural language processing.
- What practitioners and research scientists have to say about the promise of deep learning in NLP.
- Key deep learning methods and applications for natural language processing.

Let's get started.

3.1 Promise of Deep Learning

Deep learning methods are popular, primarily because they are delivering on their promise. That is not to say that there is no hype around the technology, but that the hype is based on very real results that are being demonstrated across a suite of very challenging artificial intelligence problems from computer vision and natural language processing. Some of the first large demonstrations of the power of deep learning were in natural language processing, specifically speech recognition. More recently in machine translation.

In this chapter, we will look at five specific promises of deep learning methods in the field of natural language processing. Promises highlighted recently by researchers and practitioners in the field, people who may be more tempered than the average reported in what the promises may be. In summary, they are:

- **The Promise of Drop-in Replacement Models.** That is, deep learning methods can be dropped into existing natural language systems as replacement models that can achieve commensurate or better performance.

- **The Promise of New NLP Models.** That is, deep learning methods offer the opportunity of new modeling approaches to challenging natural language problems like sequence-to-sequence prediction.
- **The Promise of Feature Learning.** That is, that deep learning methods can learn the features from natural language required by the model, rather than requiring that the features be specified and extracted by an expert.
- **The Promise of Continued Improvement.** That is, that the performance of deep learning in natural language processing is based on real results and that the improvements appear to be continuing and perhaps speeding up.
- **The Promise of End-to-End Models.** That is, that large end-to-end deep learning models can be fit on natural language problems offering a more general and better-performing approach.

We will now take a closer look at each. There are other promises of deep learning for natural language processing; these were just the 5 that I chose to highlight.

3.2 Promise of Drop-in Replacement Models

The first promise for deep learning in natural language processing is the ability to replace existing linear models with better performing models capable of learning and exploiting nonlinear relationships. Yoav Goldberg, in his primer on neural networks for NLP researchers, highlights both that deep learning methods are achieving impressive results.

More recently, neural network models started to be applied also to textual natural language signals, again with very promising results.

— *A Primer on Neural Network Models for Natural Language Processing*, 2015.

He goes on to highlight that the methods are easy to use and can sometimes be used to wholesale replace existing linear methods.

Recently, the field has seen some success in switching from such linear models over sparse inputs to non-linear neural-network models over dense inputs. While most of the neural network techniques are easy to apply, sometimes as almost drop-in replacements of the old linear classifiers, there is in many cases a strong barrier of entry.

— *A Primer on Neural Network Models for Natural Language Processing*, 2015.

3.3 Promise of New NLP Models

Another promise is that deep learning methods facilitate developing entirely new models. One strong example is the use of recurrent neural networks that are able learn and condition output over very long sequences. The approach is sufficiently different in that they allow the practitioner

to break free of traditional modeling assumptions and in turn achieve state-of-the-art results. In his book expanding on deep learning for NLP, Yoav Goldberg comments that sophisticated neural network models like recurrent neural networks allow for wholly new NLP modeling opportunities.

Around 2014, the field has started to see some success in switching from such linear models over sparse inputs to nonlinear neural network models over dense inputs. ... Others are more advanced, require a change of mindset, and provide new modeling opportunities. In particular, a family of approaches based on recurrent neural networks (RNNs) alleviates the reliance on the Markov Assumption that was prevalent in sequence models, allowing to condition on arbitrary long sequences and produce effective feature extractors. These advances lead to breakthroughs in language modeling, automatic machine translations and other applications.

— Page xvii, *Neural Network Methods in Natural Language Processing*, 2017.

3.4 Promise of Feature Learning

Deep learning methods have the ability to learn feature representations rather than requiring experts to manually specify and extract features from natural language. The NLP researcher Chris Manning, in the first lecture of his course on deep learning for natural language processing, highlights a different perspective. He describes the limitations of manually defined input features, where prior applications of machine learning in statistical NLP were really a testament to the humans defining the features and that the computers did very little learning.

Chris suggests that the promise of deep learning methods is the automatic feature learning. He highlights that feature learning is automatic rather than manual, easy to adapt rather than brittle, and can continually and automatically improve.

In general our manually designed features tend to be overspecified, incomplete, take a long time to design and validate, and only get you to a certain level of performance at the end of the day. Where the learned features are easy to adapt, fast to train and they can keep on learning so that they get to a better level of performance than we've been able to achieve previously.

— Chris Manning, Lecture 1 – Natural Language Processing with Deep Learning, 2017.

3.5 Promise of Continued Improvement

Another promise of deep learning for NLP is continued and rapid improvement on challenging problems. In the same initial lecture on deep learning for NLP, Chris Manning goes on to describe that deep learning methods are popular for natural language because they are working.

The real reason why deep learning is so exciting to most people is it has been working.

— Chris Manning, Lecture 1 – Natural Language Processing with Deep Learning, 2017.

He highlights that initial results were impressive and achieved results in speech better than any other methods in the last 30 years. Chris goes on to mention that it is not just the state-of-the-art results being achieved, but also the rate of improvement.

... what has just been totally stunning is over the last 6 or 7 years, there's just been this amazing ramp in which deep learning methods have been keeping on being improved and getting better at just an amazing speed. ... I'd actually just say it unprecedented, in terms of seeming a field that has been progressing quite so quickly in its ability to be sort of rolling out better methods of doing things month on month.

— Chris Manning, Lecture 1 – Natural Language Processing with Deep Learning, 2017.

3.6 Promise of End-to-End Models

A final promise of deep learning is the ability to develop and train end-to-end models for natural language problems instead of developing pipelines of specialized models. This is desirable both for the speed and simplicity of development in addition to the improved performance of these models.

Neural machine translation, or NMT for short, refers to large neural networks that attempt to learn to translate one language to another. This was a task traditionally handled by a pipeline of classical hand-tuned models, each of which required specialized expertise. This is described by Chris Manning in lecture 10 of his Stanford course on deep learning for NLP.

Neural machine translation is used to mean what we want to do is build one big neural network which we can train entire end-to-end machine translation process in and optimize end-to-end.

[...]

This move away from hand customized piecewise models towards end-to-end sequence-to-sequence prediction models has been the trend in speech recognition. Systems that do that are referred to as an NMT [neural machine translation] system.

— Chris Manning, Lecture 10: Neural Machine Translation and Models with Attention, 2017.

This trend towards end-to-end models rather than pipelines of specialized systems is also a trend in speech recognition. In his presentation of speech recognition in the Stanford NLP course, the NLP researcher Navdeep Jaitly, now at Nvidia, highlights that each component of a speech recognition can be replaced with a neural network. The large blocks of an automatic speech recognition pipeline are speech processing, acoustic models, pronunciation models, and language models. The problem is, the properties and importantly the errors of each sub-system are different. This motivates the need to develop one neural network to learn the whole problem end-to-end.

Over time people starting noticing that each of these components could be done better if we used a neural network. ... However, there's still a problem. There's neural networks in every component, but errors in each one are different, so they may not play well together. So that is the basic motivation for trying to go to a process where you train entire model as one big model itself.

- Navdeep Jaitly, Lecture 12: End-to-End Models for Speech Processing, Natural Language Processing with Deep Learning, 2017.

3.7 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- *A Primer on Neural Network Models for Natural Language Processing*, 2015.
<https://arxiv.org/abs/1510.00726>
- *Neural Network Methods in Natural Language Processing*, 2017.
<http://amzn.to/2eScGtY>
- *Stanford CS224n: Natural Language Processing with Deep Learning*, 2017.
<http://web.stanford.edu/class/cs224n/>

3.8 Summary

In this chapter, you discovered the promise of deep learning neural networks for natural language processing. Specifically, you learned:

- The promises of deep learning for natural language processing.
- What practitioners and research scientists have to say about the promise of deep learning in NLP.
- Key deep learning methods and applications for natural language processing.

3.8.1 Next

In the next chapter, you will discover how you can develop deep learning neural networks using the Keras Python library.

Chapter 4

How to Develop Deep Learning Models With Keras

Deep learning neural networks are very easy to create and evaluate in Python with Keras, but you must follow a strict model life-cycle. In this chapter you will discover the step-by-step life-cycle for creating, training and evaluating deep learning neural networks in Keras and how to make predictions with a trained model. You will also discover how to use the functional API that provides more flexibility when designing models. After reading this chapter you will know:

- How to define, compile, fit and evaluate a deep learning neural network in Keras.
- How to select standard defaults for regression and classification predictive modeling problems.
- How to use the functional API to develop standard Multilayer Perceptron, convolutional and recurrent neural networks.

Let's get started.

Note: It is assumed that you have a basic familiarity with deep learning and Keras, this chapter should provide a refresher for the Keras API, and perhaps an introduction to the Keras functional API. See the Appendix for installation instructions. Most code snippets in this tutorial are just for reference and are not complete examples.

4.1 Keras Model Life-Cycle

Below is an overview of the 5 steps in the neural network model life-cycle in Keras:

1. Define Network.
2. Compile Network.
3. Fit Network.
4. Evaluate Network.
5. Make Predictions.

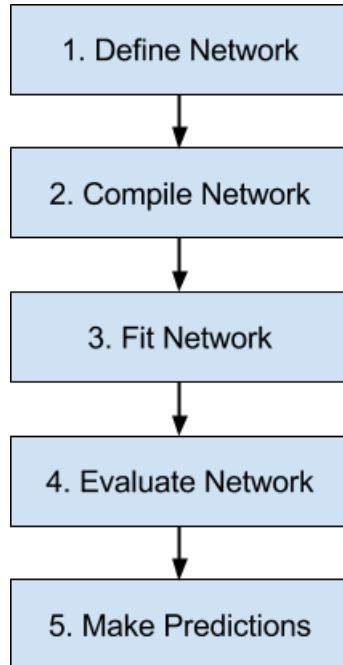


Figure 4.1: 5 Step Life-Cycle for Neural Network Models in Keras.

Let's take a look at each step in turn using the easy-to-use Keras Sequential API.

4.1.1 Step 1. Define Network

The first step is to define your neural network. Neural networks are defined in Keras as a sequence of layers. The container for these layers is the `Sequential` class. The first step is to create an instance of the `Sequential` class. Then you can create your layers and add them in the order that they should be connected. For example, we can do this in two steps:

```
model = Sequential()
model.add(Dense(2))
```

Listing 4.1: Sequential model with one `Dense` layer with 2 neurons.

But we can also do this in one step by creating an array of layers and passing it to the constructor of the `Sequential` class.

```
layers = [Dense(2)]
model = Sequential(layers)
```

Listing 4.2: Layers for a Sequential model defined as an array.

The first layer in the network must define the number of inputs to expect. The way that this is specified can differ depending on the network type, but for a Multilayer Perceptron model this is specified by the `input_dim` attribute. For example, a small Multilayer Perceptron model with 2 inputs in the visible layer, 5 neurons in the hidden layer and one neuron in the output layer can be defined as:

```
model = Sequential()
model.add(Dense(5, input_dim=2))
model.add(Dense(1))
```

Listing 4.3: Sequential model with 2 inputs.

Think of a Sequential model as a pipeline with your raw data fed in at the bottom and predictions that come out at the top. This is a helpful conception in Keras as concerns that were traditionally associated with a layer can also be split out and added as separate layers, clearly showing their role in the transform of data from input to prediction. For example, activation functions that transform a summed signal from each neuron in a layer can be extracted and added to the Sequential as a layer-like object called the `Activation` class.

```
model = Sequential()
model.add(Dense(5, input_dim=2))
model.add(Activation('relu'))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Listing 4.4: Sequential model with Activation functions defined separately from layers.

The choice of activation function is most important for the output layer as it will define the format that predictions will take. For example, below are some common predictive modeling problem types and the structure and standard activation function that you can use in the output layer:

- **Regression:** Linear activation function, or `linear`, and the number of neurons matching the number of outputs.
- **Binary Classification (2 class):** Logistic activation function, or `sigmoid`, and one neuron the output layer.
- **Multiclass Classification (>2 class):** Softmax activation function, or `softmax`, and one output neuron per class value, assuming a one hot encoded output pattern.

4.1.2 Step 2. Compile Network

Once we have defined our network, we must compile it. Compilation is an efficiency step. It transforms the simple sequence of layers that we defined into a highly efficient series of matrix transforms in a format intended to be executed on your GPU or CPU, depending on how Keras is configured. Think of compilation as a precompute step for your network. It is always required after defining a model.

Compilation requires a number of parameters to be specified, specifically tailored to training your network. Specifically, the optimization algorithm to use to train the network and the loss function used to evaluate the network that is minimized by the optimization algorithm. For example, below is a case of compiling a defined model and specifying the stochastic gradient descent (`sgd`) optimization algorithm and the mean squared error (`mean_squared_error`) loss function, intended for a regression type problem.

```
model.compile(optimizer='sgd', loss='mean_squared_error')
```

Listing 4.5: Example of compiling a defined model.

Alternately, the optimizer can be created and configured before being provided as an argument to the compilation step.

```
algorithm = SGD(lr=0.1, momentum=0.3)
model.compile(optimizer=algorithm, loss='mean_squared_error')
```

Listing 4.6: Example of defining the optimization algorithm separately.

The type of predictive modeling problem imposes constraints on the type of loss function that can be used. For example, below are some standard loss functions for different predictive model types:

- **Regression:** Mean Squared Error or `mean_squared_error`.
- **Binary Classification (2 class):** Logarithmic Loss, also called cross entropy or `binary_crossentropy`.
- **Multiclass Classification (>2 class):** Multiclass Logarithmic Loss or `categorical_crossentropy`.

The most common optimization algorithm is stochastic gradient descent, but Keras also supports a suite of other state-of-the-art optimization algorithms that work well with little or no configuration. Perhaps the most commonly used optimization algorithms because of their generally better performance are:

- **Stochastic Gradient Descent**, or `sgd`, that requires the tuning of a learning rate and momentum.
- **Adam**, or `adam`, that requires the tuning of learning rate.
- **RMSprop**, or `rmsprop`, that requires the tuning of learning rate.

Finally, you can also specify metrics to collect while fitting your model in addition to the loss function. Generally, the most useful additional metric to collect is accuracy for classification problems. The metrics to collect are specified by name in an array. For example:

```
model.compile(optimizer='sgd', loss='mean_squared_error', metrics=['accuracy'])
```

Listing 4.7: Example of defining metrics when compiling the model.

4.1.3 Step 3. Fit Network

Once the network is compiled, it can be fit, which means adapt the weights on a training dataset. Fitting the network requires the training data to be specified, both a matrix of input patterns, X , and an array of matching output patterns, y . The network is trained using the backpropagation algorithm and optimized according to the optimization algorithm and loss function specified when compiling the model.

The backpropagation algorithm requires that the network be trained for a specified number of epochs or exposures to the training dataset. Each epoch can be partitioned into groups of input-output pattern pairs called batches. This defines the number of patterns that the network is exposed to before the weights are updated within an epoch. It is also an efficiency optimization, ensuring that not too many input patterns are loaded into memory at a time. A minimal example of fitting a network is as follows:

```
history = model.fit(X, y, batch_size=10, epochs=100)
```

Listing 4.8: Example of fitting a compiled model.

Once fit, a history object is returned that provides a summary of the performance of the model during training. This includes both the loss and any additional metrics specified when compiling the model, recorded each epoch. Training can take a long time, from seconds to hours to days depending on the size of the network and the size of the training data.

By default, a progress bar is displayed on the command line for each epoch. This may create too much noise for you, or may cause problems for your environment, such as if you are in an interactive notebook or IDE. You can reduce the amount of information displayed to just the loss each epoch by setting the verbose argument to 2. You can turn off all output by setting verbose to 0. For example:

```
history = model.fit(X, y, batch_size=10, epochs=100, verbose=0)
```

Listing 4.9: Example of turning off verbose output when fitting the model.

4.1.4 Step 4. Evaluate Network

Once the network is trained, it can be evaluated. The network can be evaluated on the training data, but this will not provide a useful indication of the performance of the network as a predictive model, as it has seen all of this data before. We can evaluate the performance of the network on a separate dataset, unseen during testing. This will provide an estimate of the performance of the network at making predictions for unseen data in the future.

The model evaluates the loss across all of the test patterns, as well as any other metrics specified when the model was compiled, like classification accuracy. A list of evaluation metrics is returned. For example, for a model compiled with the accuracy metric, we could evaluate it on a new dataset as follows:

```
loss, accuracy = model.evaluate(X, y)
```

Listing 4.10: Example of evaluating a fit model.

As with fitting the network, verbose output is provided to give an idea of the progress of evaluating the model. We can turn this off by setting the verbose argument to 0.

```
loss, accuracy = model.evaluate(X, y, verbose=0)
```

Listing 4.11: Example of turning off verbose output when evaluating a fit model.

4.1.5 Step 5. Make Predictions

Once we are satisfied with the performance of our fit model, we can use it to make predictions on new data. This is as easy as calling the `predict()` function on the model with an array of new input patterns. For example:

```
predictions = model.predict(X)
```

Listing 4.12: Example of making a prediction with a fit model.

The predictions will be returned in the format provided by the output layer of the network. In the case of a regression problem, these predictions may be in the format of the problem directly, provided by a linear activation function. For a binary classification problem, the predictions may be an array of probabilities for the first class that can be converted to a 1 or 0 by rounding.

For a multiclass classification problem, the results may be in the form of an array of probabilities (assuming a one hot encoded output variable) that may need to be converted to a single class output prediction using the `argmax()` NumPy function. Alternately, for classification problems, we can use the `predict_classes()` function that will automatically convert uncrisp predictions to crisp integer class values.

```
predictions = model.predict_classes(X)
```

Listing 4.13: Example of predicting classes with a fit model.

As with fitting and evaluating the network, verbose output is provided to give an idea of the progress of the model making predictions. We can turn this off by setting the `verbose` argument to 0.

```
predictions = model.predict(X, verbose=0)
```

Listing 4.14: Example of disabling verbose output when making predictions.

4.2 Keras Functional Models

The sequential API allows you to create models layer-by-layer for most problems. It is limited in that it does not allow you to create models that share layers or have multiple inputs or outputs. The functional API in Keras is an alternate way of creating models that offers a lot more flexibility, including creating more complex models.

It specifically allows you to define multiple input or output models as well as models that share layers. More than that, it allows you to define ad hoc acyclic network graphs. Models are defined by creating instances of layers and connecting them directly to each other in pairs, then defining a Model that specifies the layers to act as the input and output to the model. Let's look at the three unique aspects of Keras functional API in turn:

4.2.1 Defining Input

Unlike the Sequential model, you must create and define a standalone `Input` layer that specifies the shape of input data. The input layer takes a `shape` argument that is a tuple that indicates the dimensionality of the input data. When input data is one-dimensional, such as for a Multilayer Perceptron, the shape must explicitly leave room for the shape of the mini-batch size used when splitting the data when training the network. Therefore, the shape tuple is always defined with a hanging last dimension `(2,)`, for example:

```
from keras.layers import Input
visible = Input(shape=(2,))
```

Listing 4.15: Example of defining input for a functional model.

4.2.2 Connecting Layers

The layers in the model are connected pairwise. This is done by specifying where the input comes from when defining each new layer. A bracket notation is used, such that after the layer is created, the layer from which the input to the current layer comes from is specified. Let's make this clear with a short example. We can create the input layer as above, then create a hidden layer as a `Dense` that receives input only from the input layer.

```
from keras.layers import Input
from keras.layers import Dense
visible = Input(shape=(2,))
hidden = Dense(2)(visible)
```

Listing 4.16: Example of connecting a hidden layer to the visible layer.

Note it is the `visible` after the creation of the `Dense` layer that connects the input layer's output as the input to the `Dense` hidden layer. It is this way of connecting layers piece by piece that gives the functional API its flexibility. For example, you can see how easy it would be to start defining ad hoc graphs of layers.

4.2.3 Creating the Model

After creating all of your model layers and connecting them together, you must define the model. As with the Sequential API, the model is the thing you can summarize, fit, evaluate, and use to make predictions. Keras provides a `Model` class that you can use to create a model from your created layers. It requires that you only specify the input and output layers. For example:

```
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
visible = Input(shape=(2,))
hidden = Dense(2)(visible)
model = Model(inputs=visible, outputs=hidden)
```

Listing 4.17: Example of creating a full model with the functional API.

Now that we know all of the key pieces of the Keras functional API, let's work through defining a suite of different models and build up some practice with it. Each example is executable and prints the structure and creates a diagram of the graph. I recommend doing this for your own models to make it clear what exactly you have defined. My hope is that these examples provide templates for you when you want to define your own models using the functional API in the future.

4.3 Standard Network Models

When getting started with the functional API, it is a good idea to see how some standard neural network models are defined. In this section, we will look at defining a simple Multilayer Perceptron, convolutional neural network, and recurrent neural network. These examples will provide a foundation for understanding the more elaborate examples later.

4.3.1 Multilayer Perceptron

In this section, we define a Multilayer Perceptron model for binary classification. The model has 10 inputs, 3 hidden layers with 10, 20, and 10 neurons, and an output layer with 1 output. Rectified linear activation functions are used in each hidden layer and a sigmoid activation function is used in the output layer, for binary classification.

```
# Multilayer Perceptron
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
visible = Input(shape=(10,))
hidden1 = Dense(10, activation='relu')(visible)
hidden2 = Dense(20, activation='relu')(hidden1)
hidden3 = Dense(10, activation='relu')(hidden2)
output = Dense(1, activation='sigmoid')(hidden3)
model = Model(inputs=visible, outputs=output)
# summarize layers
model.summary()
# plot graph
plot_model(model, to_file='multilayer_perceptron_graph.png')
```

Listing 4.18: Example of defining an MLP with the functional API.

Running the example prints the structure of the network.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 10)	0
dense_1 (Dense)	(None, 10)	110
dense_2 (Dense)	(None, 20)	220
dense_3 (Dense)	(None, 10)	210
dense_4 (Dense)	(None, 1)	11

Total params: 551
Trainable params: 551
Non-trainable params: 0

Listing 4.19: Summary of MLP model defined with the functional API.

A plot of the model graph is also created and saved to file.

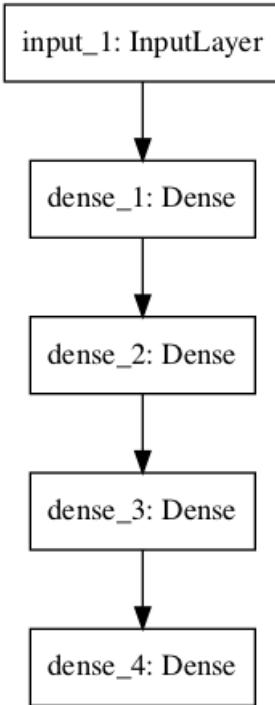


Figure 4.2: Plot of the MLP Model Graph.

Note, creating plots of Keras models requires that you install `pydot` and `pygraphviz` (the `graphviz` library and the python wrapper). Instructions for installing these libraries vary for different systems. If this is a challenge for you (e.g. you're on windows), consider commenting out the calls to `plot_model()` when you see them.

4.3.2 Convolutional Neural Network

In this section, we will define a convolutional neural network for image classification. The model receives black and white 64×64 images as input, then has a sequence of two convolutional and pooling layers as feature extractors, followed by a fully connected layer to interpret the features and an output layer with a sigmoid activation for two-class predictions.

```

# Convolutional Neural Network
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling2D
visible = Input(shape=(64,64,1))
conv1 = Conv2D(32, kernel_size=4, activation='relu')(visible)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
conv2 = Conv2D(16, kernel_size=4, activation='relu')(pool1)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
hidden1 = Dense(10, activation='relu')(pool2)
output = Dense(1, activation='sigmoid')(hidden1)
model = Model(inputs=visible, outputs=output)
# summarize layers
  
```

```
model.summary()  
# plot graph  
plot_model(model, to_file='convolutional_neural_network.png')
```

Listing 4.20: Example of defining an CNN with the functional API.

Running the example summarizes the model layers.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 64, 64, 1)	0
conv2d_1 (Conv2D)	(None, 61, 61, 32)	544
max_pooling2d_1 (MaxPooling2D)	(None, 30, 30, 32)	0
conv2d_2 (Conv2D)	(None, 27, 27, 16)	8208
max_pooling2d_2 (MaxPooling2D)	(None, 13, 13, 16)	0
dense_1 (Dense)	(None, 13, 13, 10)	170
dense_2 (Dense)	(None, 13, 13, 1)	11
Total params:	8,933	
Trainable params:	8,933	
Non-trainable params:	0	

Listing 4.21: Summary of CNN model defined with the functional API.

A plot of the model graph is also created and saved to file.

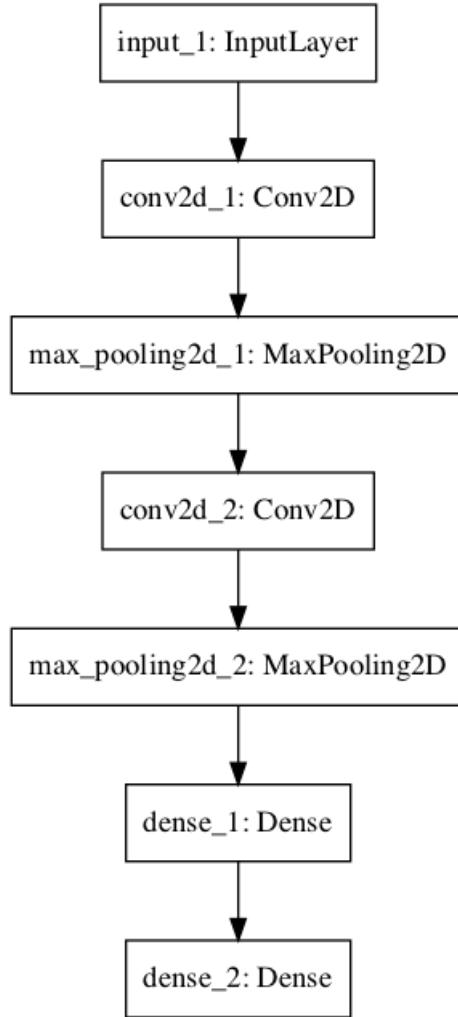


Figure 4.3: Plot of the CNN Model Graph.

4.3.3 Recurrent Neural Network

In this section, we will define a long short-term memory recurrent neural network for sequence classification. The model expects 100 time steps of one feature as input. The model has a single LSTM hidden layer to extract features from the sequence, followed by a fully connected layer to interpret the LSTM output, followed by an output layer for making binary predictions.

```

# Recurrent Neural Network
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers.recurrent import LSTM
visible = Input(shape=(100,1))
hidden1 = LSTM(10)(visible)
hidden2 = Dense(10, activation='relu')(hidden1)
output = Dense(1, activation='sigmoid')(hidden2)
model = Model(inputs=visible, outputs=output)
# summarize layers
  
```

```
model.summary()
# plot graph
plot_model(model, to_file='recurrent_neural_network.png')
```

Listing 4.22: Example of defining an RNN with the functional API.

Running the example summarizes the model layers.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100, 1)	0
lstm_1 (LSTM)	(None, 10)	480
dense_1 (Dense)	(None, 10)	110
dense_2 (Dense)	(None, 1)	11

Total params: 601
Trainable params: 601
Non-trainable params: 0

Listing 4.23: Summary of RNN model defined with the functional API.

A plot of the model graph is also created and saved to file.

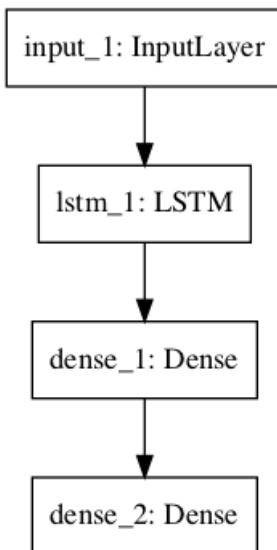


Figure 4.4: Plot of the RNN Model Graph.

4.4 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Keras documentation for Sequential Models.
<https://keras.io/models/sequential/>

- Keras documentation for Functional Models.
<https://keras.io/models/model/>
- Getting started with the Keras Sequential model.
<https://keras.io/models/model/>
- Getting started with the Keras functional API.
<https://keras.io/models/model/>
- Keras documentation for optimization algorithms.
<https://keras.io/optimizers/>
- Keras documentation for loss functions.
<https://keras.io/losses/>

4.5 Summary

In this tutorial, you discovered the step-by-step life-cycle for creating, training and evaluating deep learning neural networks in Keras and how to use the functional API that provides more flexibility when designing models. Specifically, you learned:

- How to define, compile, fit and evaluate a deep learning neural network in Keras.
- How to select standard defaults for regression and classification predictive modeling problems.
- How to use the functional API to develop standard Multilayer Perceptron, convolutional and recurrent neural networks.

4.5.1 Next

This is the last chapter in the foundations part. In the next part, you will discover how you can prepare text data ready for modeling.

Part III

Data Preparation

Chapter 5

How to Clean Text Manually and with NLTK

You cannot go straight from raw text to fitting a machine learning or deep learning model. You must clean your text first, which means splitting it into words and handling punctuation and case. In fact, there is a whole suite of text preparation methods that you may need to use, and the choice of methods really depends on your natural language processing task. In this tutorial, you will discover how you can clean and prepare your text ready for modeling with machine learning. After completing this tutorial, you will know:

- How to get started by developing your own very simple text cleaning tools.
- How to take a step up and use the more sophisticated methods in the NLTK library.
- Considerations when preparing text for natural language processing models.

Let's get started.

5.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Metamorphosis by Franz Kafka
2. Text Cleaning is Task Specific
3. Manual Tokenization
4. Tokenization and Cleaning with NLTK
5. Additional Text Cleaning Considerations

5.2 Metamorphosis by Franz Kafka

Let's start off by selecting a dataset. In this tutorial, we will use the text from the book Metamorphosis by Franz Kafka. No specific reason, other than it's short, I like it, and you may like it too. I expect it's one of those classics that most students have to read in school. The full text for Metamorphosis is available for free from Project Gutenberg. You can download the ASCII text version of the text here:

- Metamorphosis by Franz Kafka Plain Text UTF-8 (may need to load the page twice).
<http://www.gutenberg.org/cache/epub/5200/pg5200.txt>

Download the file and place it in your current working directory with the file name `metamorphosis.txt`. The file contains header and footer information that we are not interested in, specifically copyright and license information. Open the file and delete the header and footer information and save the file as `metamorphosis_clean.txt`. The start of the clean file should look like:

One morning, when Gregor Samsa woke from troubled dreams, he found himself transformed in his bed into a horrible vermin.

The file should end with:

And, as if in confirmation of their new dreams and good intentions, as soon as they reached their destination Grete was the first to get up and stretch out her young body.

Poor Gregor...

5.3 Text Cleaning Is Task Specific

After actually getting a hold of your text data, the first step in cleaning up text data is to have a strong idea about what you're trying to achieve, and in that context review your text to see what exactly might help. Take a moment to look at the text. What do you notice? Here's what I see:

- It's plain text so there is no markup to parse (yay!).
- The translation of the original German uses UK English (e.g. *travelling*).
- The lines are artificially wrapped with new lines at about 70 characters (meh).
- There are no obvious typos or spelling mistakes.
- There's punctuation like commas, apostrophes, quotes, question marks, and more.
- There's hyphenated descriptions like *armour-like*.
- There's a lot of use of the em dash (-) to continue sentences (maybe replace with commas?).
- There are names (e.g. *Mr. Samsa*)

- There does not appear to be numbers that require handling (e.g. 1999)
- There are section markers (e.g. *II* and *III*).

I'm sure there is a lot more going on to the trained eye. We are going to look at general text cleaning steps in this tutorial. Nevertheless, consider some possible objectives we may have when working with this text document. For example:

- If we were interested in developing a Kafkaesque language model, we may want to keep all of the case, quotes, and other punctuation in place.
- If we were interested in classifying documents as *Kafka* and *Not Kafka*, maybe we would want to strip case, punctuation, and even trim words back to their stem.

Use your task as the lens by which to choose how to ready your text data.

5.4 Manual Tokenization

Text cleaning is hard, but the text we have chosen to work with is pretty clean already. We could just write some Python code to clean it up manually, and this is a good exercise for those simple problems that you encounter. Tools like regular expressions and splitting strings can get you a long way.

5.4.1 Load Data

Let's load the text data so that we can work with it. The text is small and will load quickly and easily fit into memory. This will not always be the case and you may need to write code to memory map the file. Tools like NLTK (covered in the next section) will make working with large files much easier. We can load the entire `metamorphosis_clean.txt` into memory as follows:

```
# load text
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
```

Listing 5.1: Manually load the file.

5.4.2 Split by Whitespace

Clean text often means a list of words or tokens that we can work with in our machine learning models. This means converting the raw text into a list of words and saving it again. A very simple way to do this would be to split the document by white space, including “ ” (space), new lines, tabs and more. We can do this in Python with the `split()` function on the loaded string.

```
# load text
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
```

```
file.close()
# split into words by white space
words = text.split()
print(words[:100])
```

Listing 5.2: Manually split words by white space.

Running the example splits the document into a long list of words and prints the first 100 for us to review. We can see that punctuation is preserved (e.g. *wasn't* and *armour-like*), which is nice. We can also see that end of sentence punctuation is kept with the last word (e.g. *thought.*), which is not great.

```
['One', 'morning', 'when', 'Gregor', 'Samsa', 'woke', 'from', 'troubled', 'dreams', 'he',
 'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'horrible',
 'vermin.', 'He', 'lay', 'on', 'his', 'armour-like', 'back,', 'and', 'if', 'he',
 'lifted', 'his', 'head', 'a', 'little', 'he', 'could', 'see', 'his', 'brown', 'belly',
 'slightly', 'domed', 'and', 'divided', 'by', 'arches', 'into', 'stiff', 'sections.',
 'The', 'bedding', 'was', 'hardly', 'able', 'to', 'cover', 'it', 'and', 'seemed',
 'ready', 'to', 'slide', 'off', 'any', 'moment.', 'His', 'many', 'legs', 'pitifully',
 'thin', 'compared', 'with', 'the', 'size', 'of', 'the', 'rest', 'of', 'him', 'waved',
 'about', 'helplessly', 'as', 'he', 'looked.', '"What\'s', 'happened', 'to', 'me?"',
 'he', 'thought.', 'It', "wasn't", 'a', 'dream.', 'His', 'room,', 'a', 'proper', 'human']
```

Listing 5.3: Example output of splitting words by white space.

5.4.3 Select Words

Another approach might be to use the regex model (`re`) and split the document into words by selecting for strings of alphanumeric characters (a-z, A-Z, 0-9 and '_'). For example:

```
import re
# load text
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split based on words only
words = re.split(r'\W+', text)
print(words[:100])
```

Listing 5.4: Manually select words with regex.

Again, running the example we can see that we get our list of words. This time, we can see that *armour-like* is now two words *armour* and *like* (fine) but contractions like *What's* is also two words *What* and *s* (not great).

```
['One', 'morning', 'when', 'Gregor', 'Samsa', 'woke', 'from', 'troubled', 'dreams', 'he',
 'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'horrible',
 'vermin', 'He', 'lay', 'on', 'his', 'armour', 'like', 'back', 'and', 'if', 'he',
 'lifted', 'his', 'head', 'a', 'little', 'he', 'could', 'see', 'his', 'brown', 'belly',
 'slightly', 'domed', 'and', 'divided', 'by', 'arches', 'into', 'stiff', 'sections',
 'The', 'bedding', 'was', 'hardly', 'able', 'to', 'cover', 'it', 'and', 'seemed',
 'ready', 'to', 'slide', 'off', 'any', 'moment', 'His', 'many', 'legs', 'pitifully',
 'thin', 'compared', 'with', 'the', 'size', 'of', 'the', 'rest', 'of', 'him', 'waved',
 'about', 'helplessly', 'as', 'he', 'looked', 'What', 's', 'happened', 'to', 'me', 'he',
 'thought', 'It', 'wasn', 't', 'a', 'dream', 'His', 'room']
```

Listing 5.5: Example output of selecting words with regex.

5.4.4 Split by Whitespace and Remove Punctuation

We may want the words, but without the punctuation like commas and quotes. We also want to keep contractions together. One way would be to split the document into words by white space (as in the section *Split by Whitespace*), then use string translation to replace all punctuation with nothing (e.g. remove it). Python provides a constant called `string.punctuation` that provides a great list of punctuation characters. For example:

```
print(string.punctuation)
```

Listing 5.6: Print the known punctuation characters.

Results in:

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

Listing 5.7: Example output of printing the known punctuation characters.

We can use regular expressions to select for the punctuation characters and use the `sub()` function to replace them with nothing. For example:

```
re_punc = re.compile(' [s]' % re.escape(string.punctuation))
# remove punctuation from each word
stripped = [re_punc.sub('', w) for w in words]
```

Listing 5.8: Example of constructing a translation table that will remove punctuation.

We can put all of this together, load the text file, split it into words by white space, then translate each word to remove the punctuation.

```
import string
import re
# load text
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words by white space
words = text.split()
# prepare regex for char filtering
re_punc = re.compile(' [s]' % re.escape(string.punctuation))
# remove punctuation from each word
stripped = [re_punc.sub('', w) for w in words]
print(stripped[:100])
```

Listing 5.9: Manually remove punctuation.

We can see that this has had the desired effect, mostly. Contractions like *What's* have become *Whats* but *armour-like* has become *armourlike*.

```
[ 'One', 'morning', 'when', 'Gregor', 'Samsa', 'woke', 'from', 'troubled', 'dreams', 'he',
  'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'horrible',
  'vermin', 'He', 'lay', 'on', 'his', 'armourlike', 'back', 'and', 'if', 'he', 'lifted',
  'his', 'head', 'a', 'little', 'he', 'could', 'see', 'his', 'brown', 'belly',
  'slightly', 'domed', 'and', 'divided', 'by', 'arches', 'into', 'stiff', 'sections',
  'The', 'bedding', 'was', 'hardly', 'able', 'to', 'cover', 'it', 'and', 'seemed',
  'ready', 'to', 'slide', 'off', 'any', 'moment', 'His', 'many', 'legs', 'pitifully',
  'thin', 'compared', 'with', 'the', 'size', 'of', 'the', 'rest', 'of', 'him', 'waved',
  'about', 'helplessly', 'as', 'he', 'looked', 'Whats', 'happened', 'to', 'me', 'he',
  'thought', 'It', 'wasnt', 'a', 'dream', 'His', 'room', 'a', 'proper', 'human']
```

Listing 5.10: Example output of removing punctuation with translation tables.

Sometimes text data may contain non-printable characters. We can use a similar approach to filter out all non-printable characters by selecting the inverse of the `string.printable` constant. For example:

```
...
re_print = re.compile('[^%s]' % re.escape(string.printable))
result = [re_print.sub('', w) for w in words]
```

Listing 5.11: Example of removing non-printable characters.

5.4.5 Normalizing Case

It is common to convert all words to one case. This means that the vocabulary will shrink in size, but some distinctions are lost (e.g. *Apple* the company vs *apple* the fruit is a commonly used example). We can convert all words to lowercase by calling the `lower()` function on each word. For example:

```
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words by white space
words = text.split()
# convert to lower case
words = [word.lower() for word in words]
print(words[:100])
```

Listing 5.12: Manually normalize case.

Running the example, we can see that all words are now lowercase.

```
[ 'one', 'morning', 'when', 'gregor', 'samsa', 'woke', 'from', 'troubled', 'dreams', 'he',
  'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'horrible',
  'vermin.', 'he', 'lay', 'on', 'his', 'armour-like', 'back,', 'and', 'if', 'he',
  'lifted', 'his', 'head', 'a', 'little', 'he', 'could', 'see', 'his', 'brown', 'belly,',
  'slightly', 'domed', 'and', 'divided', 'by', 'arches', 'into', 'stiff', 'sections.',
  'the', 'bedding', 'was', 'hardly', 'able', 'to', 'cover', 'it', 'and', 'seemed',
  'ready', 'to', 'slide', 'off', 'any', 'moment.', 'his', 'many', 'legs,', 'pitifully',
  'thin', 'compared', 'with', 'the', 'size', 'of', 'the', 'rest', 'of', 'him,', 'waved',
  'about', 'helplessly', 'as', 'he', 'looked.', '"what\'s', 'happened', 'to', 'me?"',
  'he', 'thought.', 'it', "wasn't", 'a', 'dream.', 'his', 'room,', 'a', 'proper', 'human']
```

Listing 5.13: Example output of removing punctuation.

5.4.6 Note on Cleaning Text

Cleaning text is really hard, problem specific, and full of tradeoffs. Remember, simple is better. Simpler text data, simpler models, smaller vocabularies. You can always make things more complex later to see if it results in better model skill. Next, we'll look at some of the tools in the NLTK library that offer more than simple string splitting.

5.5 Tokenization and Cleaning with NLTK

The Natural Language Toolkit, or NLTK for short, is a Python library written for working and modeling text. It provides good tools for loading and cleaning text that we can use to get our data ready for working with machine learning and deep learning algorithms.

5.5.1 Install NLTK

You can install NLTK using your favorite package manager, such as pip. On a POSIX-compatable machine, this would be:

```
sudo pip install -U nltk
```

Listing 5.14: Command to install the NLTK library.

After installation, you will need to install the data used with the library, including a great set of documents that you can use later for testing other tools in NLTK. There are few ways to do this, such as from within a script:

```
import nltk
nltk.download()
```

Listing 5.15: NLTK script to download required text data.

Or from the command line:

```
python -m nltk.downloader all
```

Listing 5.16: Command to download NLTK required text data.

5.5.2 Split into Sentences

A good useful first step is to split the text into sentences. Some modeling tasks prefer input to be in the form of paragraphs or sentences, such as Word2Vec. You could first split your text into sentences, split each sentence into words, then save each sentence to file, one per line. NLTK provides the `sent_tokenize()` function to split text into sentences. The example below loads the `metamorphosis_clean.txt` file into memory, splits it into sentences, and prints the first sentence.

```
from nltk import sent_tokenize
# load data
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
```

```
# split into sentences
sentences = sent_tokenize(text)
print(sentences[0])
```

Listing 5.17: NLTK script to split text into sentences.

Running the example, we can see that although the document is split into sentences, that each sentence still preserves the new line from the artificial wrap of the lines in the original document.

```
One morning, when Gregor Samsa woke from troubled dreams, he found
himself transformed in his bed into a horrible vermin.
```

Listing 5.18: Example output of splitting text into sentences.

5.5.3 Split into Words

NLTK provides a function called `word_tokenize()` for splitting strings into tokens (nominally words). It splits tokens based on white space and punctuation. For example, commas and periods are taken as separate tokens. Contractions are split apart (e.g. *What's* becomes *What* and *'s*). Quotes are kept, and so on. For example:

```
from nltk.tokenize import word_tokenize
# load data
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words
tokens = word_tokenize(text)
print(tokens[:100])
```

Listing 5.19: NLTK script to split text into words.

Running the code, we can see that punctuation are now tokens that we could then decide to specifically filter out.

```
['One', 'morning', ',', 'when', 'Gregor', 'Samsa', 'woke', 'from', 'troubled', 'dreams',
',', 'he', 'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a',
'horrible', 'vermin', '.', 'He', 'lay', 'on', 'his', 'armour-like', 'back', ',', 'and',
'if', 'he', 'lifted', 'his', 'head', 'a', 'little', 'he', 'could', 'see', 'his',
'brown', 'belly', ',', 'slightly', 'domed', 'and', 'divided', 'by', 'arches', 'into',
'stiff', 'sections', '.', 'The', 'bedding', 'was', 'hardly', 'able', 'to', 'cover',
'it', 'and', 'seemed', 'ready', 'to', 'slide', 'off', 'any', 'moment', '.', 'His',
'many', 'legs', ',', 'pitifully', 'thin', 'compared', 'with', 'the', 'size', 'of',
'the', 'rest', 'of', 'him', ',', 'waved', 'about', 'helplessly', 'as', 'he', 'looked',
'.', '``', 'What', "'s", 'happened', 'to']
```

Listing 5.20: Example output of splitting text into words.

5.5.4 Filter Out Punctuation

We can filter out all tokens that we are not interested in, such as all standalone punctuation. This can be done by iterating over all tokens and only keeping those tokens that are all alphabetic. Python has the function `isalpha()` that can be used. For example:

```
from nltk.tokenize import word_tokenize
# load data
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words
tokens = word_tokenize(text)
# remove all tokens that are not alphabetic
words = [word for word in tokens if word.isalpha()]
print(words[:100])
```

Listing 5.21: NLTK script to remove punctuation.

Running the example, you can see that not only punctuation tokens, but examples like *armour-like* and *'s* were also filtered out.

```
['One', 'morning', 'when', 'Gregor', 'Samsa', 'woke', 'from', 'troubled', 'dreams', 'he',
'found', 'himself', 'transformed', 'in', 'his', 'bed', 'into', 'a', 'horrible',
'vermin', 'He', 'lay', 'on', 'his', 'back', 'and', 'if', 'he', 'lifted', 'his', 'head',
'a', 'little', 'he', 'could', 'see', 'his', 'brown', 'belly', 'slightly', 'domed',
'and', 'divided', 'by', 'arches', 'into', 'stiff', 'sections', 'The', 'bedding', 'was',
'hardly', 'able', 'to', 'cover', 'it', 'and', 'seemed', 'ready', 'to', 'slide', 'off',
'any', 'moment', 'His', 'many', 'legs', 'pitifully', 'thin', 'compared', 'with', 'the',
'size', 'of', 'the', 'rest', 'of', 'him', 'waved', 'about', 'helplessly', 'as', 'he',
'looked', 'What', 'happened', 'to', 'me', 'he', 'thought', 'It', 'was', 'a', 'dream',
'His', 'room', 'a', 'proper', 'human', 'room']
```

Listing 5.22: Example output of removing punctuation.

5.5.5 Filter out Stop Words (and Pipeline)

Stop words are those words that do not contribute to the deeper meaning of the phrase. They are the most common words such as: *the*, *a*, and *is*. For some applications like documentation classification, it may make sense to remove stop words. NLTK provides a list of commonly agreed upon stop words for a variety of languages, such as English. They can be loaded as follows:

```
from nltk.corpus import stopwords
stop_words = stopwords.words('english')
print(stop_words)
```

Listing 5.23: NLTK script print stop words.

You can see the full list as follows:

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your', 'yours',
'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', 'her', 'hers',
'herself', 'it', 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves',
'what', 'which', 'who', 'whom', 'this', 'that', 'these', 'those', 'am', 'is', 'are',
'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does',
'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until',
'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into',
'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down',
'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here',
```

```
'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more',
'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so',
'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', 'should', 'now', 'd',
'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', 'couldn', 'didn', 'doesn', 'hadn',
'hasn', 'haven', 'isn', 'ma', 'mightn', 'mustn', 'needn', 'shan', 'shouldn', 'wasn',
'weren', 'won', 'wouldn']
```

Listing 5.24: Example output of printing stop words.

You can see that they are all lower case and have punctuation removed. You could compare your tokens to the stop words and filter them out, but you must ensure that your text is prepared the same way. Let's demonstrate this with a small pipeline of text preparation including:

- Load the raw text.
- Split into tokens.
- Convert to lowercase.
- Remove punctuation from each token.
- Filter out remaining tokens that are not alphabetic.
- Filter out tokens that are stop words.

```
import string
import re
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
# load data
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words
tokens = word_tokenize(text)
# convert to lower case
tokens = [w.lower() for w in tokens]
# prepare regex for char filtering
re_punc = re.compile('[\s]' % re.escape(string.punctuation))
# remove punctuation from each word
stripped = [re_punc.sub('', w) for w in tokens]
# remove remaining tokens that are not alphabetic
words = [word for word in stripped if word.isalpha()]
# filter out stop words
stop_words = set(stopwords.words('english'))
words = [w for w in words if not w in stop_words]
print(words[:100])
```

Listing 5.25: NLTK script filter out stop words.

Running this example, we can see that in addition to all of the other transforms, stop words like *a* and *to* have been removed. I note that we are still left with tokens like *nt*. The rabbit hole is deep; there's always more we can do.

```
[ 'one', 'morning', 'gregor', 'samsa', 'woke', 'troubled', 'dreams', 'found', 'transformed',
  'bed', 'horrible', 'vermin', 'lay', 'armourlike', 'back', 'lifted', 'head', 'little',
  'could', 'see', 'brown', 'belly', 'slightly', 'domed', 'divided', 'arches', 'stiff',
  'sections', 'bedding', 'hardly', 'able', 'cover', 'seemed', 'ready', 'slide', 'moment',
  'many', 'legs', 'pitifully', 'thin', 'compared', 'size', 'rest', 'waved', 'helplessly',
  'looked', 'happened', 'thought', 'nt', 'dream', 'room', 'proper', 'human', 'room',
  'although', 'little', 'small', 'lay', 'peacefully', 'four', 'familiar', 'walls',
  'collection', 'textile', 'samples', 'lay', 'spread', 'table', 'samsa', 'travelling',
  'salesman', 'hung', 'picture', 'recently', 'cut', 'illustrated', 'magazine', 'housed',
  'nice', 'gilded', 'frame', 'showed', 'lady', 'fitted', 'fur', 'hat', 'fur', 'boa',
  'sat', 'upright', 'raising', 'heavy', 'fur', 'muff', 'covered', 'whole', 'lower',
  'arm', 'towards', 'viewer']
```

Listing 5.26: Example output of filtering out stop words.

5.5.6 Stem Words

Stemming refers to the process of reducing each word to its root or base. For example *fishing*, *fished*, *fisher* all reduce to the stem *fish*. Some applications, like document classification, may benefit from stemming in order to both reduce the vocabulary and to focus on the sense or sentiment of a document rather than deeper meaning. There are many stemming algorithms, although a popular and long-standing method is the Porter Stemming algorithm. This method is available in NLTK via the `PorterStemmer` class. For example:

```
from nltk.tokenize import word_tokenize
from nltk.stem.porter import PorterStemmer
# load data
filename = 'metamorphosis_clean.txt'
file = open(filename, 'rt')
text = file.read()
file.close()
# split into words
tokens = word_tokenize(text)
# stemming of words
porter = PorterStemmer()
stemmed = [porter.stem(word) for word in tokens]
print(stemmed[:100])
```

Listing 5.27: NLTK script stem words.

Running the example, you can see that words have been reduced to their stems, such as *trouble* has become *troubl*. You can also see that the stemming implementation has also reduced the tokens to lowercase, likely for internal look-ups in word tables.

```
[ 'one', 'morn', ',', 'when', 'gregor', 'samsa', 'woke', 'from', 'troubl', 'dream', ',', ,
  'he', 'found', 'himself', 'transform', 'in', 'hi', 'bed', 'into', 'a', 'horribl',
  'vermin', '.', 'He', 'lay', 'on', 'hi', 'armour-lik', 'back', ',', 'and', 'if', 'he',
  'lift', 'hi', 'head', 'a', 'littl', 'he', 'could', 'see', 'hi', 'brown', 'belli', ',',
  'slightli', 'dome', 'and', 'divid', 'by', 'arch', 'into', 'stiff', 'section', '',
  'the', 'bed', 'wa', 'hardli', 'abl', 'to', 'cover', 'it', 'and', 'seem', 'readi', 'to',
  'slide', 'off', 'ani', 'moment', '.', 'hi', 'mani', 'leg', ',', 'piti', 'thin',
  'compar', 'with', 'the', 'size', 'of', 'the', 'rest', 'of', 'him', ',', 'wave',
  'about', 'helplessli', 'as', 'he', 'look', '.', '``', 'what', "'s", 'happen', 'to'
```

Listing 5.28: Example output of stemming words.

There is a nice suite of stemming and lemmatization algorithms to choose from in NLTK, if reducing words to their root is something you need for your project.

5.6 Additional Text Cleaning Considerations

We are only getting started. Because the source text for this tutorial was reasonably clean to begin with, we skipped many concerns of text cleaning that you may need to deal with in your own project. Here is a shortlist of additional considerations when cleaning text:

- Handling large documents and large collections of text documents that do not fit into memory.
- Extracting text from markup like HTML, PDF, or other structured document formats.
- Transliteration of characters from other languages into English.
- Decoding Unicode characters into a normalized form, such as UTF8.
- Handling of domain specific words, phrases, and acronyms.
- Handling or removing numbers, such as dates and amounts.
- Locating and correcting common typos and misspellings.
- And much more...

The list could go on. Hopefully, you can see that getting truly clean text is impossible, that we are really doing the best we can based on the time, resources, and knowledge we have. The idea of *clean* is really defined by the specific task or concern of your project.

A pro tip is to continually review your tokens after every transform. I have tried to show that in this tutorial and I hope you take that to heart. Ideally, you would save a new file after each transform so that you can spend time with all of the data in the new form. Things always jump out at you when to take the time to review your data.

5.7 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Metamorphosis by Franz Kafka on Project Gutenberg.
<http://www.gutenberg.org/ebooks/5200>
- Installing NLTK.
<http://www.nltk.org/install.html>
- Installing NLTK Data.
<http://www.nltk.org/data.html>

- Python `isalpha()` function.
<https://docs.python.org/3/library/stdtypes.html#str.isalpha>
- Stop Words on Wikipedia.
https://en.wikipedia.org/wiki/Stop_words
- Stemming on Wikipedia.
<https://en.wikipedia.org/wiki/Stemming>
- `nltk.tokenize` package API.
<http://www.nltk.org/api/nltk.tokenize.html>
- Porter Stemming algorithm.
<https://tartarus.org/martin/PorterStemmer/>
- `nltk.stem` package API.
<http://www.nltk.org/api/nltk.stem.html>
- *Processing Raw Text, Natural Language Processing with Python.*
<http://www.nltk.org/book/ch03.html>

5.8 Summary

In this tutorial, you discovered how to clean text or machine learning in Python. Specifically, you learned:

- How to get started by developing your own very simple text cleaning tools.
- How to take a step up and use the more sophisticated methods in the NLTK library.
- Considerations when preparing text for natural language processing models.

5.8.1 Next

In the next chapter, you will discover how you can encode text data using the scikit-learn Python library.

Chapter 6

How to Prepare Text Data with scikit-learn

Text data requires special preparation before you can start using it for predictive modeling. The text must be parsed to remove words, called tokenization. Then the words need to be encoded as integers or floating point values for use as input to a machine learning algorithm, called feature extraction (or vectorization). The scikit-learn library offers easy-to-use tools to perform both tokenization and feature extraction of your text data. In this tutorial, you will discover exactly how you can prepare your text data for predictive modeling in Python with scikit-learn. After completing this tutorial, you will know:

- How to convert text to word count vectors with `CountVectorizer`.
- How to convert text to word frequency vectors with `TfidfVectorizer`.
- How to convert text to unique integers with `HashingVectorizer`.

Let's get started.

6.1 The Bag-of-Words Model

We cannot work with text directly when using machine learning algorithms. Instead, we need to convert the text to numbers. We may want to perform classification of documents, so each document is an *input* and a class label is the *output* for our predictive algorithm. Algorithms take vectors of numbers as input, therefore we need to convert documents to fixed-length vectors of numbers.

A simple and effective model for thinking about text documents in machine learning is called the Bag-of-Words Model, or BoW. Note, that we cover the BoW model in great detail in the next part, starting with Chapter 8. The model is simple in that it throws away all of the order information in the words and focuses on the occurrence of words in a document. This can be done by assigning each word a unique number. Then any document we see can be encoded as a fixed-length vector with the length of the vocabulary of known words. The value in each position in the vector could be filled with a count or frequency of each word in the encoded document.

This is the bag-of-words model, where we are only concerned with encoding schemes that represent what words are present or the degree to which they are present in encoded documents without any information about order. There are many ways to extend this simple method, both by better clarifying what a *word* is and in defining what to encode about each word in the vector. The scikit-learn library provides 3 different schemes that we can use, and we will briefly look at each.

6.2 Word Counts with CountVectorizer

The `CountVectorizer` provides a simple way to both tokenize a collection of text documents and build a vocabulary of known words, but also to encode new documents using that vocabulary. You can use it as follows:

- Create an instance of the `CountVectorizer` class.
- Call the `fit()` function in order to learn a vocabulary from one or more documents.
- Call the `transform()` function on one or more documents as needed to encode each as a vector.

An encoded vector is returned with a length of the entire vocabulary and an integer count for the number of times each word appeared in the document. Because these vectors will contain a lot of zeros, we call them sparse. Python provides an efficient way of handling sparse vectors in the `scipy.sparse` package. The vectors returned from a call to `transform()` will be sparse vectors, and you can transform them back to NumPy arrays to look and better understand what is going on by calling the `toarray()` function. Below is an example of using the `CountVectorizer` to tokenize, build a vocabulary, and then encode a document.

```
from sklearn.feature_extraction.text import CountVectorizer
# list of text documents
text = ["The quick brown fox jumped over the lazy dog."]
# create the transform
vectorizer = CountVectorizer()
# tokenize and build vocab
vectorizer.fit(text)
# summarize
print(vectorizer.vocabulary_)
# encode document
vector = vectorizer.transform(text)
# summarize encoded vector
print(vector.shape)
print(type(vector))
print(vector.toarray())
```

Listing 6.1: Example of training a `CountVectorizer`.

Above, you can see that we access the vocabulary to see what exactly was tokenized by calling:

```
print(vectorizer.vocabulary_)
```

Listing 6.2: Print the learned vocabulary.

We can see that all words were made lowercase by default and that the punctuation was ignored. These and other aspects of tokenizing can be configured and I encourage you to review all of the options in the API documentation. Running the example first prints the vocabulary, then the shape of the encoded document. We can see that there are 8 words in the vocab, and therefore encoded vectors have a length of 8. We can then see that the encoded vector is a sparse matrix. Finally, we can see an array version of the encoded vector showing a count of 1 occurrence for each word except the (index and id 7) that has an occurrence of 2.

```
{'dog': 1, 'fox': 2, 'over': 5, 'brown': 0, 'quick': 6, 'the': 7, 'lazy': 4, 'jumped': 3}
(1, 8)
<class 'scipy.sparse.csr.csr_matrix'>
[[1 1 1 1 1 1 1 2]]
```

Listing 6.3: Example output of training a `CountVectorizer`.

Importantly, the same vectorizer can be used on documents that contain words not included in the vocabulary. These words are ignored and no count is given in the resulting vector. For example, below is an example of using the vectorizer above to encode a document with one word in the vocab and one word that is not.

```
# encode another document
text2 = ["the puppy"]
vector = vectorizer.transform(text2)
print(vector.toarray())
```

Listing 6.4: Example of encoding another document with the fit `CountVectorizer`.

Running this example prints the array version of the encoded sparse vector showing one occurrence of the one word in the vocab and the other word not in the vocab completely ignored.

```
[[0 0 0 0 0 0 0 1]]
```

Listing 6.5: Example output of encoding another document.

The encoded vectors can then be used directly with a machine learning algorithm.

6.3 Word Frequencies with `TfidfVectorizer`

Word counts are a good starting point, but are very basic. One issue with simple counts is that some words like *the* will appear many times and their large counts will not be very meaningful in the encoded vectors. An alternative is to calculate word frequencies, and by far the most popular method is called TF-IDF. This is an acronym that stands for *Term Frequency - Inverse Document Frequency* which are the components of the resulting scores assigned to each word.

- **Term Frequency:** This summarizes how often a given word appears within a document.
- **Inverse Document Frequency:** This downscals words that appear a lot across documents.

Without going into the math, TF-IDF are word frequency scores that try to highlight words that are more interesting, e.g. frequent in a document but not across documents. The `TfidfVectorizer` will tokenize documents, learn the vocabulary and inverse document

frequency weightings, and allow you to encode new documents. Alternately, if you already have a learned `CountVectorizer`, you can use it with a `TfidfTransformer` to just calculate the inverse document frequencies and start encoding documents. The same create, fit, and transform process is used as with the `CountVectorizer`. Below is an example of using the `TfidfVectorizer` to learn vocabulary and inverse document frequencies across 3 small documents and then encode one of those documents.

```
from sklearn.feature_extraction.text import TfidfVectorizer
# list of text documents
text = ["The quick brown fox jumped over the lazy dog.",
        "The dog.",
        "The fox"]
# create the transform
vectorizer = TfidfVectorizer()
# tokenize and build vocab
vectorizer.fit(text)
# summarize
print(vectorizer.vocabulary_)
print(vectorizer.idf_)
# encode document
vector = vectorizer.transform([text[0]])
# summarize encoded vector
print(vector.shape)
print(vector.toarray())
```

Listing 6.6: Example of training a `TfidfVectorizer`.

A vocabulary of 8 words is learned from the documents and each word is assigned a unique integer index in the output vector. The inverse document frequencies are calculated for each word in the vocabulary, assigning the lowest score of 1.0 to the most frequently observed word: *the* at index 7. Finally, the first document is encoded as an 8-element sparse array and we can review the final scorings of each word with different values for *the*, *fox*, and *dog* from the other words in the vocabulary.

```
{'fox': 2, 'lazy': 4, 'dog': 1, 'quick': 6, 'the': 7, 'over': 5, 'brown': 0, 'jumped': 3}
[ 1.69314718 1.28768207 1.28768207 1.69314718 1.69314718 1.69314718
1.69314718 1. ]
(1, 8)
[[ 0.36388646 0.27674503 0.27674503 0.36388646 0.36388646 0.36388646
0.36388646 0.42983441]]
```

Listing 6.7: Example output of training a `TfidfVectorizer`.

The scores are normalized to values between 0 and 1 and the encoded document vectors can then be used directly with most machine learning algorithms.

6.4 Hashing with *HashingVectorizer*

Counts and frequencies can be very useful, but one limitation of these methods is that the vocabulary can become very large. This, in turn, will require large vectors for encoding documents and impose large requirements on memory and slow down algorithms. A clever work around is to use a one way hash of words to convert them to integers. The clever part is that no vocabulary is required and you can choose an arbitrary-long fixed length vector. A downside

is that the hash is a one-way function so there is no way to convert the encoding back to a word (which may not matter for many supervised learning tasks).

The `HashingVectorizer` class implements this approach that can be used to consistently hash words, then tokenize and encode documents as needed. The example below demonstrates the `HashingVectorizer` for encoding a single document. An arbitrary fixed-length vector size of 20 was chosen. This corresponds to the range of the hash function, where small values (like 20) may result in hash collisions. Remembering back to Computer Science classes, I believe there are heuristics that you can use to pick the hash length and probability of collision based on estimated vocabulary size (e.g. a load factor of 75%). See any good textbook on the topic. Note that this vectorizer does not require a call to fit on the training data documents. Instead, after instantiation, it can be used directly to start encoding documents.

```
from sklearn.feature_extraction.text import HashingVectorizer
# list of text documents
text = ["The quick brown fox jumped over the lazy dog."]
# create the transform
vectorizer = HashingVectorizer(n_features=20)
# encode document
vector = vectorizer.transform(text)
# summarize encoded vector
print(vector.shape)
print(vector.toarray())
```

Listing 6.8: Example of training a `HashingVectorizer`.

Running the example encodes the sample document as a 20-element sparse array. The values of the encoded document correspond to normalized word counts by default in the range of -1 to 1, but could be made simple integer counts by changing the default configuration.

(1, 20)
[[0. 0. 0. 0. 0. 0.33333333
0. -0.33333333 0.33333333 0. 0. 0.33333333
0. 0. 0. -0.33333333 0. 0.]]
-0.66666667 0.]]

Listing 6.9: Example output of training a `HashingVectorizer`.

6.5 Further Reading

This section provides more resources on the topic if you are looking go deeper.

6.5.1 Natural Language Processing

- Bag-of-words model on Wikipedia.
https://en.wikipedia.org/wiki/Bag-of-words_model
- Tokenization on Wikipedia.
https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization
- TF-IDF on Wikipedia.
<https://en.wikipedia.org/wiki/Tf%E2%80%93idf>

6.5.2 sciki-learn

- Section 4.2. Feature extraction, scikit-learn User Guide.
http://scikit-learn.org/stable/modules/feature_extraction.html
- scikit-learn Feature Extraction API.
http://scikit-learn.org/stable/modules/classes.html#module-sklearn.feature_extraction
- Working With Text Data, scikit-learn Tutorial.
http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html

6.5.3 Class APIs

- CountVectorizer scikit-learn API.
http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
- TfidfVectorizer scikit-learn API.
http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
- TfidfTransformer scikit-learn API.
http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html
- HashingVectorizer scikit-learn API.
http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html

6.6 Summary

In this tutorial, you discovered how to prepare text documents for machine learning with scikit-learn for bag-of-words models. Specifically, you learned:

- How to convert text to word count vectors with `CountVectorizer`.
- How to convert text to word frequency vectors with `TfidfVectorizer`.
- How to convert text to unique integers with `HashingVectorizer`.

We have only scratched the surface in these examples and I want to highlight that there are many configuration details for these classes to influence the tokenizing of documents that are worth exploring.

6.6.1 Next

In the next chapter, you will discover how you can prepare text data using the Keras deep learning library.

Chapter 7

How to Prepare Text Data With Keras

You cannot feed raw text directly into deep learning models. Text data must be encoded as numbers to be used as input or output for machine learning and deep learning models, such as word embeddings. The Keras deep learning library provides some basic tools to help you prepare your text data. In this tutorial, you will discover how you can use Keras to prepare your text data. After completing this tutorial, you will know:

- About the convenience methods that you can use to quickly prepare text data.
- The `Tokenizer` API that can be fit on training data and used to encode training, validation, and test documents.
- The range of 4 different document encoding schemes offered by the `Tokenizer` API.

Let's get started.

7.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Split words with `text_to_word_sequence`.
2. Encoding with `one_hot`.
3. Hash Encoding with `hashing_trick`.
4. `Tokenizer` API

7.2 Split Words with `text_to_word_sequence`

A good first step when working with text is to split it into words. Words are called tokens and the process of splitting text into tokens is called tokenization. Keras provides the `text_to_word_sequence()` function that you can use to split text into a list of words. By default, this function automatically does 3 things:

- Splits words by space.

- Filters out punctuation.
- Converts text to lowercase (`lower=True`).

You can change any of these defaults by passing arguments to the function. Below is an example of using the `text_to_word_sequence()` function to split a document (in this case a simple string) into a list of words.

```
from keras.preprocessing.text import text_to_word_sequence
# define the document
text = 'The quick brown fox jumped over the lazy dog.'
# tokenize the document
result = text_to_word_sequence(text)
print(result)
```

Listing 7.1: Example splitting words with the `Tokenizer`.

Running the example creates an array containing all of the words in the document. The list of words is printed for review.

```
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
```

Listing 7.2: Example output for splitting words with the `Tokenizer`.

This is a good first step, but further pre-processing is required before you can work with the text.

7.3 Encoding with `one_hot`

It is popular to represent a document as a sequence of integer values, where each word in the document is represented as a unique integer. Keras provides the `one_hot()` function that you can use to tokenize and integer encode a text document in one step. The name suggests that it will create a one hot encoding of the document, which is not the case. Instead, the function is a wrapper for the `hashing_trick()` function described in the next section. The function returns an integer encoded version of the document. The use of a hash function means that there may be collisions and not all words will be assigned unique integer values. As with the `text_to_word_sequence()` function in the previous section, the `one_hot()` function will make the text lower case, filter out punctuation, and split words based on white space.

In addition to the text, the vocabulary size (total words) must be specified. This could be the total number of words in the document or more if you intend to encode additional documents that contains additional words. The size of the vocabulary defines the hashing space from which words are hashed. By default, the `hash` function is used, although as we will see in the next section, alternate hash functions can be specified when calling the `hashing_trick()` function directly.

We can use the `text_to_word_sequence()` function from the previous section to split the document into words and then use a set to represent only the unique words in the document. The size of this set can be used to estimate the size of the vocabulary for one document. For example:

```
from keras.preprocessing.text import text_to_word_sequence
# define the document
text = 'The quick brown fox jumped over the lazy dog.'
```

```
# estimate the size of the vocabulary
words = set(text_to_word_sequence(text))
vocab_size = len(words)
print(vocab_size)
```

Listing 7.3: Example of preparing a vocabulary.

We can put this together with the `one_hot()` function and encode the words in the document. The complete example is listed below. The vocabulary size is increased by one-third to minimize collisions when hashing words.

```
from keras.preprocessing.text import one_hot
from keras.preprocessing.text import text_to_word_sequence
# define the document
text = 'The quick brown fox jumped over the lazy dog.'
# estimate the size of the vocabulary
words = set(text_to_word_sequence(text))
vocab_size = len(words)
print(vocab_size)
# integer encode the document
result = one_hot(text, round(vocab_size*1.3))
print(result)
```

Listing 7.4: Example of one hot encoding.

Running the example first prints the size of the vocabulary as 8. The encoded document is then printed as an array of integer encoded words.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
8
[5, 9, 8, 7, 9, 1, 5, 3, 8]
```

Listing 7.5: Example output for one hot encoding with the `Tokenizer`.

7.4 Hash Encoding with `hashing_trick`

A limitation of integer and count base encodings is that they must maintain a vocabulary of words and their mapping to integers. An alternative to this approach is to use a one-way hash function to convert words to integers. This avoids the need to keep track of a vocabulary, which is faster and requires less memory.

Keras provides the `hashing_trick()` function that tokenizes and then integer encodes the document, just like the `one_hot()` function. It provides more flexibility, allowing you to specify the hash function as either `hash` (the default) or other hash functions such as the built in `md5` function or your own function. Below is an example of integer encoding a document using the `md5` hash function.

```
from keras.preprocessing.text import hashing_trick
from keras.preprocessing.text import text_to_word_sequence
# define the document
text = 'The quick brown fox jumped over the lazy dog.'
```

```
# estimate the size of the vocabulary
words = set(text_to_word_sequence(text))
vocab_size = len(words)
print(vocab_size)
# integer encode the document
result = hashing_trick(text, round(vocab_size*1.3), hash_function='md5')
print(result)
```

Listing 7.6: Example of hash encoding.

Running the example prints the size of the vocabulary and the integer encoded document. We can see that the use of a different hash function results in consistent, but different integers for words as the `one_hot()` function in the previous section.

```
8
[6, 4, 1, 2, 7, 5, 6, 2, 6]
```

Listing 7.7: Example output for hash encoding with the `Tokenizer`.

7.5 Tokenizer API

So far we have looked at one-off convenience methods for preparing text with Keras. Keras provides a more sophisticated API for preparing text that can be fit and reused to prepare multiple text documents. This may be the preferred approach for large projects. Keras provides the `Tokenizer` class for preparing text documents for deep learning. The `Tokenizer` must be constructed and then fit on either raw text documents or integer encoded text documents. For example:

```
from keras.preprocessing.text import Tokenizer
# define 5 documents
docs = ['Well done!',
        'Good work',
        'Great effort',
        'nice work',
        'Excellent!']
# create the tokenizer
t = Tokenizer()
# fit the tokenizer on the documents
t.fit_on_texts(docs)
```

Listing 7.8: Example of fitting a `Tokenizer`.

Once fit, the `Tokenizer` provides 4 attributes that you can use to query what has been learned about your documents:

- `word_counts`: A dictionary of words and their counts.
- `word_docs`: An integer count of the total number of documents that were used to fit the `Tokenizer`.
- `word_index`: A dictionary of words and their uniquely assigned integers.
- `document_count`: A dictionary of words and how many documents each appeared in.

For example:

```
# summarize what was learned
print(t.word_counts)
print(t.document_count)
print(t.word_index)
print(t.word_docs)
```

Listing 7.9: Summarize the output of the fit Tokenizer.

Once the `Tokenizer` has been fit on training data, it can be used to encode documents in the train or test datasets. The `texts_to_matrix()` function on the `Tokenizer` can be used to create one vector per document provided per input. The length of the vectors is the total size of the vocabulary. This function provides a suite of standard bag-of-words model text encoding schemes that can be provided via a mode argument to the function. The modes available include:

- `binary`: Whether or not each word is present in the document. This is the default.
- `count`: The count of each word in the document.
- `tfidf`: The Text Frequency-Inverse DocumentFrequency (TF-IDF) scoring for each word in the document.
- `freq`: The frequency of each word as a ratio of words within each document.

We can put all of this together with a worked example.

```
from keras.preprocessing.text import Tokenizer
# define 5 documents
docs = ['Well done!',
        'Good work',
        'Great effort',
        'nice work',
        'Excellent!']
# create the tokenizer
t = Tokenizer()
# fit the tokenizer on the documents
t.fit_on_texts(docs)
# summarize what was learned
print(t.word_counts)
print(t.document_count)
print(t.word_index)
print(t.word_docs)
# integer encode documents
encoded_docs = t.texts_to_matrix(docs, mode='count')
print(encoded_docs)
```

Listing 7.10: Example of fitting and encoding with the `Tokenizer`.

Running the example fits the `Tokenizer` with 5 small documents. The details of the fit `Tokenizer` are printed. Then the 5 documents are encoded using a word count. Each document is encoded as a 9-element vector with one position for each word and the chosen encoding scheme value for each word position. In this case, a simple word count mode is used.

```
OrderedDict([('well', 1), ('done', 1), ('good', 1), ('work', 2), ('great', 1), ('effort', 1), ('nice', 1), ('excellent', 1)])
5
{'work': 1, 'effort': 6, 'done': 3, 'great': 5, 'good': 4, 'excellent': 8, 'well': 2, 'nice': 7}
{'work': 2, 'effort': 1, 'done': 1, 'well': 1, 'good': 1, 'great': 1, 'excellent': 1, 'nice': 1}
[[ 0.  0.  1.  1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  1.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.]]
```

Listing 7.11: Example output from fitting and encoding with the `Tokenizer`.

The `Tokenizer` will be the key way we will prepare text for word embeddings throughout this book.

7.6 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Text Preprocessing Keras API.
<https://keras.io/preprocessing/text/>
- `text_to_word_sequence` Keras API.
https://keras.io/preprocessing/text/#text_to_word_sequence
- `one_hot` Keras API.
https://keras.io/preprocessing/text/#one_hot
- `hashing_trick` Keras API.
https://keras.io/preprocessing/text/#hashing_trick
- `Tokenizer` Keras API.
<https://keras.io/preprocessing/text/#tokenizer>

7.7 Summary

In this tutorial, you discovered how you can use the Keras API to prepare your text data for deep learning. Specifically, you learned:

- About the convenience methods that you can use to quickly prepare text data.
- The `Tokenizer` API that can be fit on training data and used to encode training, validation, and test documents.
- The range of 4 different document encoding schemes offered by the `Tokenizer` API.

7.7.1 Next

This is the last chapter in the data preparation part. In the next part, you will discover how to develop bag-of-words models.

Part IV

Bag-of-Words

Chapter 8

The Bag-of-Words Model

The bag-of-words model is a way of representing text data when modeling text with machine learning algorithms. The bag-of-words model is simple to understand and implement and has seen great success in problems such as language modeling and document classification. In this tutorial, you will discover the bag-of-words model for feature extraction in natural language processing. After completing this tutorial, you will know:

- What the bag-of-words model is and why it is needed to represent text.
- How to develop a bag-of-words model for a collection of documents.
- How to use different techniques to prepare a vocabulary and score words.

Let's get started.

8.1 Tutorial Overview

This tutorial is divided into the following parts:

1. The Problem with Text
2. What is a Bag-of-Words?
3. Example of the Bag-of-Words Model
4. Managing Vocabulary
5. Scoring Words
6. Limitations of Bag-of-Words

8.2 The Problem with Text

A problem with modeling text is that it is messy, and techniques like machine learning algorithms prefer well defined fixed-length inputs and outputs. Machine learning algorithms cannot work with raw text directly; the text must be converted into numbers. Specifically, vectors of numbers.

In language processing, the vectors x are derived from textual data, in order to reflect various linguistic properties of the text.

— Page 65, *Neural Network Methods in Natural Language Processing*, 2017.

This is called feature extraction or feature encoding. A popular and simple method of feature extraction with text data is called the bag-of-words model of text.

8.3 What is a Bag-of-Words?

A bag-of-words model, or BoW for short, is a way of extracting features from text for use in modeling, such as with machine learning algorithms. The approach is very simple and flexible, and can be used in a myriad of ways for extracting features from documents. A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:

- A vocabulary of known words.
- A measure of the presence of known words.

It is called a *bag-of-words* , because any information about the order or structure of words in the document is discarded. The model is only concerned with whether known words occur in the document, not where in the document.

A very common feature extraction procedures for sentences and documents is the bag-of-words approach (BOW). In this approach, we look at the histogram of the words within the text, i.e. considering each word count as a feature.

— Page 69, *Neural Network Methods in Natural Language Processing*, 2017.

The intuition is that documents are similar if they have similar content. Further, that from the content alone we can learn something about the meaning of the document. The bag-of-words can be as simple or complex as you like. The complexity comes both in deciding how to design the vocabulary of known words (or tokens) and how to score the presence of known words. We will take a closer look at both of these concerns.

8.4 Example of the Bag-of-Words Model

Let's make the bag-of-words model concrete with a worked example.

8.4.1 Step 1: Collect Data

Below is a snippet of the first few lines of text from the book *A Tale of Two Cities* by Charles Dickens, taken from Project Gutenberg.

```
It was the best of times,
it was the worst of times,
it was the age of wisdom,
it was the age of foolishness,
```

Listing 8.1: Sample of text from *A Tale of Two Cities* by Charles Dickens.

For this small example, let's treat each line as a separate *document* and the 4 lines as our entire corpus of documents.

8.4.2 Step 2: Design the Vocabulary

Now we can make a list of all of the words in our model vocabulary. The unique words here (ignoring case and punctuation) are:

```
it
was
the
best
of
times
worst
age
wisdom
foolishness
```

Listing 8.2: List of unique words.

That is a vocabulary of 10 words from a corpus containing 24 words.

8.4.3 Step 3: Create Document Vectors

The next step is to score the words in each document. The objective is to turn each document of free text into a vector that we can use as input or output for a machine learning model. Because we know the vocabulary has 10 words, we can use a fixed-length document representation of 10, with one position in the vector to score each word. The simplest scoring method is to mark the presence of words as a boolean value, 0 for absent, 1 for present. Using the arbitrary ordering of words listed above in our vocabulary, we can step through the first document (*It was the best of times*) and convert it into a binary vector. The scoring of the document would look as follows:

```
it = 1
was = 1
the = 1
best = 1
of = 1
times = 1
worst = 0
age = 0
wisdom = 0
foolishness = 0
```

Listing 8.3: List of unique words and their occurrence in a document.

As a binary vector, this would look as follows:

[1, 1, 1, 1, 1, 1, 0, 0, 0, 0]

Listing 8.4: First line of text as a binary vector.

The other three documents would look as follows:

"it was the worst of times" = [1, 1, 1, 0, 1, 1, 1, 0, 0, 0]
"it was the age of wisdom" = [1, 1, 1, 0, 1, 0, 0, 1, 1, 0]
"it was the age of foolishness" = [1, 1, 1, 0, 1, 0, 0, 1, 0, 1]

Listing 8.5: Remaining three lines of text as binary vectors.

All ordering of the words is nominally discarded and we have a consistent way of extracting features from any document in our corpus, ready for use in modeling. New documents that overlap with the vocabulary of known words, but may contain words outside of the vocabulary, can still be encoded, where only the occurrence of known words are scored and unknown words are ignored. You can see how this might naturally scale to large vocabularies and larger documents.

8.5 Managing Vocabulary

As the vocabulary size increases, so does the vector representation of documents. In the previous example, the length of the document vector is equal to the number of known words. You can imagine that for a very large corpus, such as thousands of books, that the length of the vector might be thousands or millions of positions. Further, each document may contain very few of the known words in the vocabulary.

This results in a vector with lots of zero scores, called a sparse vector or sparse representation. Sparse vectors require more memory and computational resources when modeling and the vast number of positions or dimensions can make the modeling process very challenging for traditional algorithms. As such, there is pressure to decrease the size of the vocabulary when using a bag-of-words model.

There are simple text cleaning techniques that can be used as a first step, such as:

- Ignoring case.
- Ignoring punctuation.
- Ignoring frequent words that don't contain much information, called stop words, like *a*, *of*, etc.
- Fixing misspelled words.
- Reducing words to their stem (e.g. *play* from *playing*) using stemming algorithms.

A more sophisticated approach is to create a vocabulary of grouped words. This both changes the scope of the vocabulary and allows the bag-of-words to capture a little bit more meaning from the document. In this approach, each word or token is called a *gram*. Creating a vocabulary of two-word pairs is, in turn, called a bigram model. Again, only the bigrams that appear in the corpus are modeled, not all possible bigrams.

An n-gram is an n-token sequence of words: a 2-gram (more commonly called a bigram) is a two-word sequence of words like “please turn”, “turn your”, or “your homework”, and a 3-gram (more commonly called a trigram) is a three-word sequence of words like “please turn your”, or “turn your homework”.

— Page 85, *Speech and Language Processing*, 2009.

For example, the bigrams in the first line of text in the previous section: *It was the best of times* are as follows:

```
it was
was the
the best
best of
of times
```

Listing 8.6: List of bi-grams for a document.

A vocabulary that tracks triplets of words is called a trigram model and the general approach is called the n-gram model, where n refers to the number of grouped words. Often a simple bigram approach is better than a 1-gram bag-of-words model for tasks like documentation classification.

a bag-of-bigrams representation is much more powerful than bag-of-words, and in many cases proves very hard to beat.

— Page 75, *Neural Network Methods in Natural Language Processing*, 2017.

8.6 Scoring Words

Once a vocabulary has been chosen, the occurrence of words in example documents needs to be scored. In the worked example, we have already seen one very simple approach to scoring: a binary scoring of the presence or absence of words. Some additional simple scoring methods include:

- **Counts.** Count the number of times each word appears in a document.
- **Frequencies.** Calculate the frequency that each word appears in a document out of all the words in the document.

8.6.1 Word Hashing

You may remember from computer science that a hash function is a bit of math that maps data to a fixed size set of numbers. For example, we use them in hash tables when programming where perhaps names are converted to numbers for fast lookup. We can use a hash representation of known words in our vocabulary. This addresses the problem of having a very large vocabulary for a large text corpus because we can choose the size of the hash space, which is in turn the size of the vector representation of the document.

Words are hashed deterministically to the same integer index in the target hash space. A binary score or count can then be used to score the word. This is called the *hash trick* or *feature hashing*. The challenge is to choose a hash space to accommodate the chosen vocabulary size to minimize the probability of collisions and trade-off sparsity.

8.6.2 TF-IDF

A problem with scoring word frequency is that highly frequent words start to dominate in the document (e.g. larger score), but may not contain as much *informational content* to the model as rarer but perhaps domain specific words. One approach is to rescale the frequency of words by how often they appear in all documents, so that the scores for frequent words like *the* that are also frequent across all documents are penalized. This approach to scoring is called Term Frequency - Inverse Document Frequency, or TF-IDF for short, where:

- **Term Frequency:** is a scoring of the frequency of the word in the current document.
- **Inverse Document Frequency:** is a scoring of how rare the word is across documents.

The scores are a weighting where not all words are equally as important or interesting. The scores have the effect of highlighting words that are distinct (contain useful information) in a given document.

Thus the idf of a rare term is high, whereas the idf of a frequent term is likely to be low.

— Page 118, *An Introduction to Information Retrieval*, 2008.

8.7 Limitations of Bag-of-Words

The bag-of-words model is very simple to understand and implement and offers a lot of flexibility for customization on your specific text data. It has been used with great success on prediction problems like language modeling and documentation classification. Nevertheless, it suffers from some shortcomings, such as:

- **Vocabulary:** The vocabulary requires careful design, most specifically in order to manage the size, which impacts the sparsity of the document representations.
- **Sparsity:** Sparse representations are harder to model both for computational reasons (space and time complexity) and also for information reasons, where the challenge is for the models to harness so little information in such a large representational space.
- **Meaning:** Discarding word order ignores the context, and in turn meaning of words in the document (semantics). Context and meaning can offer a lot to the model, that if modeled could tell the difference between the same words differently arranged (*this is interesting* vs *is this interesting*), synonyms (*old bike* vs *used bike*), and much more.

8.8 Further Reading

This section provides more resources on the topic if you are looking go deeper.

8.8.1 Books

- *Neural Network Methods in Natural Language Processing*, 2017.
<http://amzn.to/2wycQKA>
- *Speech and Language Processing*, 2009.
<http://amzn.to/2vaEb7T>
- *An Introduction to Information Retrieval*, 2008.
<http://amzn.to/2vvnPHP>
- *Foundations of Statistical Natural Language Processing*, 1999.
<http://amzn.to/2vvnPHP>

8.8.2 Wikipedia

- Bag-of-words model.
<https://en.wikipedia.org/wiki/N-gram>
- n-gram.
<https://en.wikipedia.org/wiki/N-gram>
- Feature hashing.
https://en.wikipedia.org/wiki/Feature_hashing
- tf-idf.
<https://en.wikipedia.org/wiki/Tf-idf>

8.9 Summary

In this tutorial, you discovered the bag-of-words model for feature extraction with text data. Specifically, you learned:

- What the bag-of-words model is and why we need it.
- How to work through the application of a bag-of-words model to a collection of documents.
- What techniques can be used for preparing a vocabulary and scoring words.

8.9.1 Next

In the next chapter, you will can prepare movie review data for the bag-of-words model.

Chapter 9

How to Prepare Movie Review Data for Sentiment Analysis

Text data preparation is different for each problem. Preparation starts with simple steps, like loading data, but quickly gets difficult with cleaning tasks that are very specific to the data you are working with. You need help as to where to begin and what order to work through the steps from raw data to data ready for modeling. In this tutorial, you will discover how to prepare movie review text data for sentiment analysis, step-by-step. After completing this tutorial, you will know:

- How to load text data and clean it to remove punctuation and other non-words.
- How to develop a vocabulary, tailor it, and save it to file.
- How to prepare movie reviews using cleaning and a pre-defined vocabulary and save them to new files ready for modeling.

Let's get started.

9.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Movie Review Dataset
2. Load Text Data
3. Clean Text Data
4. Develop Vocabulary
5. Save Prepared Data

9.2 Movie Review Dataset

The Movie Review Data is a collection of movie reviews retrieved from the `imdb.com` website in the early 2000s by Bo Pang and Lillian Lee. The reviews were collected and made available as part of their research on natural language processing. The reviews were originally released in 2002, but an updated and cleaned up version was released in 2004, referred to as *v2.0*. The dataset is comprised of 1,000 positive and 1,000 negative movie reviews drawn from an archive of the `rec.arts.movies.reviews` newsgroup hosted at IMDB. The authors refer to this dataset as the *polarity dataset*.

Our data contains 1000 positive and 1000 negative reviews all written before 2002, with a cap of 20 reviews per author (312 authors total) per category. We refer to this corpus as the polarity dataset.

- A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts, 2004.

The data has been cleaned up somewhat, for example:

- The dataset is comprised of only English reviews.
- All text has been converted to lowercase.
- There is white space around punctuation like periods, commas, and brackets.
- Text has been split into one sentence per line.

The data has been used for a few related natural language processing tasks. For classification, the performance of classical models (such as Support Vector Machines) on the data is in the range of high 70% to low 80% (e.g. 78%-to-82%). More sophisticated data preparation may see results as high as 86% with 10-fold cross-validation. This gives us a ballpark of low-to-mid 80s if we were looking to use this dataset in experiments on modern methods.

... depending on choice of downstream polarity classifier, we can achieve highly statistically significant improvement (from 82.8% to 86.4%)

- A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts, 2004.

You can download the dataset from here:

- Movie Review Polarity Dataset (`review_polarity.tar.gz`, 3MB).
http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

After unzipping the file, you will have a directory called `txt_sentoken` with two sub-directories containing the text *neg* and *pos* for negative and positive reviews. Reviews are stored one per file with a naming convention from *cv000* to *cv999* for each of *neg* and *pos*. Next, let's look at loading the text data.

9.3 Load Text Data

In this section, we will look at loading individual text files, then processing the directories of files. We will assume that the review data is downloaded and available in the current working directory in the folder `txt_sentoken`. We can load an individual text file by opening it, reading in the ASCII text, and closing the file. This is standard file handling stuff. For example, we can load the first negative review file `cv000_29416.txt` as follows:

```
# load one file
filename = 'txt_sentoken/neg/cv000_29416.txt'
# open the file as read only
file = open(filename, 'r')
# read all text
text = file.read()
# close the file
file.close()
```

Listing 9.1: Example of loading a single movie review.

This loads the document as ASCII and preserves any white space, like new lines. We can turn this into a function called `load_doc()` that takes a filename of the document to load and returns the text.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text
```

Listing 9.2: Function to load a document into memory.

We have two directories each with 1,000 documents each. We can process each directory in turn by first getting a list of files in the directory using the `listdir()` function, then loading each file in turn. For example, we can load each document in the negative directory using the `load_doc()` function to do the actual loading.

```
from os import listdir

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# specify directory to load
directory = 'txt_sentoken/neg'
# walk through all files in the folder
for filename in listdir(directory):
```

```
# skip files that do not have the right extension
if not filename.endswith(".txt"):
    next
# create the full path of the file to open
path = directory + '/' + filename
# load document
doc = load_doc(path)
print('Loaded %s' % filename)
```

Listing 9.3: Example of loading a all movie reviews.

Running this example prints the filename of each review after it is loaded.

```
...
Loaded cv995_23113.txt
Loaded cv996_12447.txt
Loaded cv997_5152.txt
Loaded cv998_15691.txt
Loaded cv999_14636.txt
```

Listing 9.4: Example output of loading all movie reviews.

We can turn the processing of the documents into a function as well and use it as a template later for developing a function to clean all documents in a folder. For example, below we define a `process_docs()` function to do the same thing.

```
from os import listdir

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load all docs in a directory
def process_docs(directory):
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip files that do not have the right extension
        if not filename.endswith(".txt"):
            next
        # create the full path of the file to open
        path = directory + '/' + filename
        # load document
        doc = load_doc(path)
        print('Loaded %s' % filename)

# specify directory to load
directory = 'txt_sentoken/neg'
process_docs(directory)
```

Listing 9.5: Example of loading a all movie reviews with functions.

Now that we know how to load the movie review text data, let's look at cleaning it.

9.4 Clean Text Data

In this section, we will look at what data cleaning we might want to do to the movie review data. We will assume that we will be using a bag-of-words model or perhaps a word embedding that does not require too much preparation.

9.4.1 Split into Tokens

First, let's load one document and look at the raw tokens split by white space. We will use the `load_doc()` function developed in the previous section. We can use the `split()` function to split the loaded document into tokens separated by white space.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load the document
filename = 'txt_sentoken/neg/cv000_29416.txt'
text = load_doc(filename)
# split into tokens by white space
tokens = text.split()
print(tokens)
```

Listing 9.6: Load a movie review and split by white space.

Running the example gives a nice long list of raw tokens from the document.

```
...
'years', 'ago', 'and', 'has', 'been', 'sitting', 'on', 'the', 'shelves', 'ever', 'since',
'..', 'whatever', '.', '.', '.', 'skip', 'it', '!', "where's", 'joblo', 'coming',
'from', '?', 'a', 'nightmare', 'of', 'elm', 'street', '3', '(', '7/10', ')', '-',
'blair', 'witch', '2', '(', '7/10', ')', '-', 'the', 'crow', '(', '9/10', ')', '-',
'the', 'crow', ':', 'salvation', '(', '4/10', ')', '-', 'lost', 'highway', '(', '10/10', ')',
'-' , 'memento', '(', '10/10', ')', ' ', 'the', 'others', '(', '9/10', ')',
' ', 'stir', 'of', 'echoes', '(', '8/10', ')']
```

Listing 9.7: Example output of splitting a review by white space.

Just looking at the raw tokens can give us a lot of ideas of things to try, such as:

- Remove punctuation from words (e.g. ‘what’s’).
- Removing tokens that are just punctuation (e.g. ‘-’).
- Removing tokens that contain numbers (e.g. ‘10/10’).
- Remove tokens that have one character (e.g. ‘a’).
- Remove tokens that don’t have much meaning (e.g. ‘and’).

Some ideas:

- We can filter out punctuation from tokens using regular expressions.
- We can remove tokens that are just punctuation or contain numbers by using an `isalpha()` check on each token.
- We can remove English stop words using the list loaded using NLTK.
- We can filter out short tokens by checking their length.

Below is an updated version of cleaning this review.

```
from nltk.corpus import stopwords
import string
import re

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load the document
filename = 'txt_sentoken/neg/cv000_29416.txt'
text = load_doc(filename)
# split into tokens by white space
tokens = text.split()
# prepare regex for char filtering
re_punc = re.compile('[\s]' % re.escape(string.punctuation))
# remove punctuation from each word
tokens = [re_punc.sub('', w) for w in tokens]
# remove remaining tokens that are not alphabetic
tokens = [word for word in tokens if word.isalpha()]
# filter out stop words
stop_words = set(stopwords.words('english'))
tokens = [w for w in tokens if not w in stop_words]
# filter out short tokens
tokens = [word for word in tokens if len(word) > 1]
print(tokens)
```

Listing 9.8: Load and clean one movie review.

Running the example gives a much cleaner looking list of tokens.

```
...
'explanation', 'craziness', 'came', 'oh', 'way', 'horror', 'teen', 'slasher', 'flick',
'packaged', 'look', 'way', 'someone', 'apparently', 'assuming', 'genre', 'still',
'hot', 'kids', 'also', 'wrapped', 'production', 'two', 'years', 'ago', 'sitting',
'shelves', 'ever', 'since', 'whatever', 'skip', 'wheres', 'joblo', 'coming',
'nightmare', 'elm', 'street', 'blair', 'witch', 'crow', 'crow', 'salvation', 'lost',
'highway', 'memento', 'others', 'stir', 'echoes']
```

Listing 9.9: Example output of cleaning one movie review.

We can put this into a function called `clean_doc()` and test it on another review, this time a positive review.

```
from nltk.corpus import stopwords
import string
import re

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile(' [^s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub(' ', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load the document
filename = 'txt_sentoken/pos/cv000_29590.txt'
text = load_doc(filename)
tokens = clean_doc(text)
print(tokens)
```

Listing 9.10: Function to clean movie reviews.

Again, the cleaning procedure seems to produce a good set of tokens, at least as a first cut.

```
...
'comic', 'oscar', 'winner', 'martin', 'child', 'shakespeare', 'love', 'production',
'design', 'turns', 'original', 'prague', 'surroundings', 'one', 'creepy', 'place',
'even', 'acting', 'hell', 'solid', 'dreamy', 'depp', 'turning', 'typically', 'strong',
'performance', 'deftly', 'handling', 'british', 'accent', 'ians', 'holm', 'joe',
'goulds', 'secret', 'richardson', 'dalmatians', 'log', 'great', 'supporting', 'roles',
'big', 'surprise', 'graham', 'cringed', 'first', 'time', 'opened', 'mouth',
'imagining', 'attempt', 'irish', 'accent', 'actually', 'wasnt', 'half', 'bad', 'film',
'however', 'good', 'strong', 'violencegore', 'sexuality', 'language', 'drug', 'content']
```

Listing 9.11: Example output of a function to clean movie reviews.

There are many more cleaning steps we could take and I leave them to your imagination. Next, let's look at how we can manage a preferred vocabulary of tokens.

9.5 Develop Vocabulary

When working with predictive models of text, like a bag-of-words model, there is a pressure to reduce the size of the vocabulary. The larger the vocabulary, the more sparse the representation of each word or document. A part of preparing text for sentiment analysis involves defining and tailoring the vocabulary of words supported by the model. We can do this by loading all of the documents in the dataset and building a set of words. We may decide to support all of these words, or perhaps discard some. The final chosen vocabulary can then be saved to file for later use, such as filtering words in new documents in the future.

We can keep track of the vocabulary in a `Counter`, which is a dictionary of words and their count with some additional convenience functions. We need to develop a new function to process a document and add it to the vocabulary. The function needs to load a document by calling the previously developed `load_doc()` function. It needs to clean the loaded document using the previously developed `clean_doc()` function, then it needs to add all the tokens to the `Counter`, and update counts. We can do this last step by calling the `update()` function on the counter object. Below is a function called `add_doc_to_vocab()` that takes as arguments a document filename and a `Counter` vocabulary.

```
# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
    # load doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # update counts
    vocab.update(tokens)
```

Listing 9.12: Function add a movie review to a vocabulary.

Finally, we can use our template above for processing all documents in a directory called `process_docs()` and update it to call `add_doc_to_vocab()`.

```
# load all docs in a directory
def process_docs(directory, vocab):
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip files that do not have the right extension
        if not filename.endswith(".txt"):
            next
        # create the full path of the file to open
        path = directory + '/' + filename
        # add doc to vocab
        add_doc_to_vocab(path, vocab)
```

Listing 9.13: Updated process documents function.

We can put all of this together and develop a full vocabulary from all documents in the dataset.

```
import string
import re
from os import listdir
from collections import Counter
from nltk.corpus import stopwords
```

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
    # load doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # update counts
    vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip files that do not have the right extension
        if not filename.endswith(".txt"):
            next
        # create the full path of the file to open
        path = directory + '/' + filename
        # add doc to vocab
        add_doc_to_vocab(path, vocab)

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/neg', vocab)
process_docs('txt_sentoken/pos', vocab)
# print the size of the vocab
print(len(vocab))
```

```
# print the top words in the vocab
print(vocab.most_common(50))
```

Listing 9.14: Example of cleaning all reviews and building a vocabulary.

Running the example creates a vocabulary with all documents in the dataset, including positive and negative reviews. We can see that there are a little over 46,000 unique words across all reviews and the top 3 words are *film*, *one*, and *movie*.

```
46557
[('film', 8860), ('one', 5521), ('movie', 5440), ('like', 3553), ('even', 2555), ('good', 2320), ('time', 2283), ('story', 2118), ('films', 2102), ('would', 2042), ('much', 2024), ('also', 1965), ('characters', 1947), ('get', 1921), ('character', 1906), ('two', 1825), ('first', 1768), ('see', 1730), ('well', 1694), ('way', 1668), ('make', 1590), ('really', 1563), ('little', 1491), ('life', 1472), ('plot', 1451), ('people', 1420), ('movies', 1416), ('could', 1395), ('bad', 1374), ('scene', 1373), ('never', 1364), ('best', 1301), ('new', 1277), ('many', 1268), ('doesnt', 1267), ('man', 1266), ('scenes', 1265), ('dont', 1210), ('know', 1207), ('hes', 1150), ('great', 1141), ('another', 1111), ('love', 1089), ('action', 1078), ('go', 1075), ('us', 1065), ('director', 1056), ('something', 1048), ('end', 1047), ('still', 1038)]
```

Listing 9.15: Example output of building a vocabulary.

Perhaps the least common words, those that only appear once across all reviews, are not predictive. Perhaps some of the most common words are not useful too. These are good questions and really should be tested with a specific predictive model. Generally, words that only appear once or a few times across 2,000 reviews are probably not predictive and can be removed from the vocabulary, greatly cutting down on the tokens we need to model. We can do this by stepping through words and their counts and only keeping those with a count above a chosen threshold. Here we will use 5 occurrences.

```
# keep tokens with > 5 occurrence
min_occurane = 5
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
```

Listing 9.16: Example of filtering the vocabulary by an occurrence count.

This reduces the vocabulary from 46,557 to 14,803 words, a huge drop. Perhaps a minimum of 5 occurrences is too aggressive; you can experiment with different values. We can then save the chosen vocabulary of words to a new file. I like to save the vocabulary as ASCII with one word per line. Below defines a function called `save_list()` to save a list of items, in this case, tokens to file, one per line.

```
def save_list(lines, filename):
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()
```

Listing 9.17: Function to save the vocabulary to file.

The complete example for defining and saving the vocabulary is listed below.

```
import string
import re
from os import listdir
```

```
from collections import Counter
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub(' ', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
    # load doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # update counts
    vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip files that do not have the right extension
        if not filename.endswith(".txt"):
            next
        # create the full path of the file to open
        path = directory + '/' + filename
        # add doc to vocab
        add_doc_to_vocab(path, vocab)

# save list to file
def save_list(lines, filename):
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
```

```

file.close()

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/neg', vocab)
process_docs('txt_sentoken/pos', vocab)
# print the size of the vocab
print(len(vocab))
# print the top words in the vocab
print(vocab.most_common(50))
# keep tokens with > 5 occurrence
min_occurane = 5
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
# save tokens to a vocabulary file
save_list(tokens, 'vocab.txt')

```

Listing 9.18: Example building and saving a final vocabulary.

Running this final snippet after creating the vocabulary will save the chosen words to file. It is a good idea to take a look at, and even study, your chosen vocabulary in order to get ideas for better preparing this data, or text data in the future.

```

hasnt
updating
figuratively
symphony
civilians
might
fisherman
hokum
witch
buffoons
...

```

Listing 9.19: Sample of the saved vocabulary file.

Next, we can look at using the vocabulary to create a prepared version of the movie review dataset.

9.6 Save Prepared Data

We can use the data cleaning and chosen vocabulary to prepare each movie review and save the prepared versions of the reviews ready for modeling. This is a good practice as it decouples the data preparation from modeling, allowing you to focus on modeling and circle back to data prep if you have new ideas. We can start off by loading the vocabulary from `vocab.txt`.

```

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file

```

```

file.close()
return text

# load vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)

```

Listing 9.20: Load the saved vocabulary.

Next, we can clean the reviews, use the loaded vocab to filter out unwanted tokens, and save the clean reviews in a new file. One approach could be to save all the positive reviews in one file and all the negative reviews in another file, with the filtered tokens separated by white space for each review on separate lines. First, we can define a function to process a document, clean it, filter it, and return it as a single line that could be saved in a file. Below defines the `doc_to_line()` function to do just that, taking a filename and vocabulary (as a set) as arguments. It calls the previously defined `load_doc()` function to load the document and `clean_doc()` to tokenize the document.

```

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
    # load the doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # filter by vocab
    tokens = [w for w in tokens if w in vocab]
    return ' '.join(tokens)

```

Listing 9.21: Function to filter a review by the vocabulary

Next, we can define a new version of `process_docs()` to step through all reviews in a folder and convert them to lines by calling `doc_to_line()` for each document. A list of lines is then returned.

```

# load all docs in a directory
def process_docs(directory, vocab):
    lines = list()
    # walk through all files in the folder
    for filename in.listdir(directory):
        # skip files that do not have the right extension
        if not filename.endswith(".txt"):
            next
        # create the full path of the file to open
        path = directory + '/' + filename
        # load and clean the doc
        line = doc_to_line(path, vocab)
        # add to list
        lines.append(line)
    return lines

```

Listing 9.22: Updated process docs function to filter all documents by the vocabulary.

We can then call `process_docs()` for both the directories of positive and negative reviews, then call `save_list()` from the previous section to save each list of processed reviews to a file.

The complete code listing is provided below.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub(' ', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# save list to file
def save_list(lines, filename):
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
    # load the doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # filter by vocab
    tokens = [w for w in tokens if w in vocab]
    return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
    lines = list()
    # walk through all files in the folder
    for filename in listdir(directory):
```

```

# skip files that do not have the right extension
if not filename.endswith(".txt"):
    next
# create the full path of the file to open
path = directory + '/' + filename
# load and clean the doc
line = doc_to_line(path, vocab)
# add to list
lines.append(line)
return lines

# load vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)
# prepare negative reviews
negative_lines = process_docs('txt_sentoken/neg', vocab)
save_list(negative_lines, 'negative.txt')
# prepare positive reviews
positive_lines = process_docs('txt_sentoken/pos', vocab)
save_list(positive_lines, 'positive.txt')

```

Listing 9.23: Example of cleaning and filtering all reviews by the vocab and saving the results to file.

Running the example saves two new files, `negative.txt` and `positive.txt`, that contain the prepared negative and positive reviews respectively. The data is ready for use in a bag-of-words or even word embedding model.

9.7 Further Reading

This section provides more resources on the topic if you are looking go deeper.

9.7.1 Dataset

- Movie Review Data.
<http://www.cs.cornell.edu/people/pabo/movie-review-data/>
- *A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts*, 2004.
<http://xxx.lanl.gov/abs/cs/0409058>
- Movie Review Polarity Dataset.
http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz
- Dataset Readme v2.0 and v1.1.
<http://www.cs.cornell.edu/people/pabo/movie-review-data/poldata README.2.0.txt>
<http://www.cs.cornell.edu/people/pabo/movie-review-data/README.1.1>

9.7.2 APIs

- `nltk.tokenize` package API.
<http://www.nltk.org/api/nltk.tokenize.html>
- Chapter 2, *Accessing Text Corpora and Lexical Resources*.
<http://www.nltk.org/book/ch02.html>
- `os` API Miscellaneous operating system interfaces.
<https://docs.python.org/3/library/os.html>
- `collections` API - Container datatypes.
<https://docs.python.org/3/library/collections.html>

9.8 Summary

In this tutorial, you discovered how to prepare movie review text data for sentiment analysis, step-by-step. Specifically, you learned:

- How to load text data and clean it to remove punctuation and other non-words.
- How to develop a vocabulary, tailor it, and save it to file.
- How to prepare movie reviews using cleaning and a predefined vocabulary and save them to new files ready for modeling.

9.8.1 Next

In the next chapter, you will discover how you can develop a neural bag-of-words model for movie review sentiment analysis.

Chapter 10

Project: Develop a Neural Bag-of-Words Model for Sentiment Analysis

Movie reviews can be classified as either favorable or not. The evaluation of movie review text is a classification problem often called sentiment analysis. A popular technique for developing sentiment analysis models is to use a bag-of-words model that transforms documents into vectors where each word in the document is assigned a score. In this tutorial, you will discover how you can develop a deep learning predictive model using the bag-of-words representation for movie review sentiment classification. After completing this tutorial, you will know:

- How to prepare the review text data for modeling with a restricted vocabulary.
- How to use the bag-of-words model to prepare train and test data.
- How to develop a Multilayer Perceptron bag-of-words model and use it to make predictions on new review text data.

Let's get started.

10.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Movie Review Dataset
2. Data Preparation
3. Bag-of-Words Representation
4. Sentiment Analysis Models
5. Comparing Word Scoring Methods
6. Predicting Sentiment for New Reviews

10.2 Movie Review Dataset

In this tutorial, we will use the Movie Review Dataset. This dataset designed for sentiment analysis was described previously in Chapter 9. You can download the dataset from here:

- Movie Review Polarity Dataset (`review_polarity.tar.gz`, 3MB).
http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

After unzipping the file, you will have a directory called `txt_sentoken` with two sub-directories containing the text `neg` and `pos` for negative and positive reviews. Reviews are stored one per file with a naming convention `cv000` to `cv999` for each of `neg` and `pos`. Next, let's look at loading the text data.

10.3 Data Preparation

Note: The preparation of the movie review dataset was first described in Chapter 9. In this section, we will look at 3 things:

1. Separation of data into training and test sets.
2. Loading and cleaning the data to remove punctuation and numbers.
3. Defining a vocabulary of preferred words.

10.3.1 Split into Train and Test Sets

We are pretending that we are developing a system that can predict the sentiment of a textual movie review as either positive or negative. This means that after the model is developed, we will need to make predictions on new textual reviews. This will require all of the same data preparation to be performed on those new reviews as is performed on the training data for the model.

We will ensure that this constraint is built into the evaluation of our models by splitting the training and test datasets prior to any data preparation. This means that any knowledge in the test set that could help us better prepare the data (e.g. the words used) is unavailable during the preparation of data and the training of the model. That being said, we will use the last 100 positive reviews and the last 100 negative reviews as a test set (100 reviews) and the remaining 1,800 reviews as the training dataset. This is a 90% train, 10% split of the data. The split can be imposed easily by using the filenames of the reviews where reviews named 000 to 899 are for training data and reviews named 900 onwards are for testing the model.

10.3.2 Loading and Cleaning Reviews

The text data is already pretty clean, so not much preparation is required. Without getting too much into the details, we will prepare the data using the following method:

- Split tokens on white space.

- Remove all punctuation from words.
- Remove all words that are not purely comprised of alphabetical characters.
- Remove all words that are known stop words.
- Remove all words that have a length ≤ 1 character.

We can put all of these steps into a function called `clean_doc()` that takes as an argument the raw text loaded from a file and returns a list of cleaned tokens. We can also define a function `load_doc()` that loads a document from file ready for use with the `clean_doc()` function. An example of cleaning the first positive review is listed below.

```
from nltk.corpus import stopwords
import string
import re

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[\%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load the document
filename = 'txt_sentoken/pos/cv000_29590.txt'
text = load_doc(filename)
tokens = clean_doc(text)
print(tokens)
```

Listing 10.1: Example of cleaning a movie review.

Running the example prints a long list of clean tokens. There are many more cleaning steps we may want to explore, and I leave them as further exercises. I'd love to see what you can come up with.

```
...
'creepy', 'place', 'even', 'acting', 'hell', 'solid', 'dreamy', 'depp', 'turning',
'typically', 'strong', 'performance', 'deftly', 'handling', 'british', 'accent',
'ians', 'holm', 'joe', 'oulds', 'secret', 'richardson', 'dalmatians', 'log', 'great',
'supporting', 'roles', 'big', 'surprise', 'graham', 'cringed', 'first', 'time',
'opened', 'mouth', 'imagining', 'attempt', 'irish', 'accent', 'actually', 'wasnt',
'half', 'bad', 'film', 'however', 'good', 'strong', 'violencegore', 'sexuality',
'language', 'drug', 'content']
```

Listing 10.2: Example output of cleaning a movie review.

10.3.3 Define a Vocabulary

It is important to define a vocabulary of known words when using a bag-of-words model. The more words, the larger the representation of documents, therefore it is important to constrain the words to only those believed to be predictive. This is difficult to know beforehand and often it is important to test different hypotheses about how to construct a useful vocabulary. We have already seen how we can remove punctuation and numbers from the vocabulary in the previous section. We can repeat this for all documents and build a set of all known words.

We can develop a vocabulary as a `Counter`, which is a dictionary mapping of words and their count that allows us to easily update and query. Each document can be added to the counter (a new function called `add_doc_to_vocab()`) and we can step over all of the reviews in the negative directory and then the positive directory (a new function called `process_docs()`). The complete example is listed below.

```
import string
import re
from os import listdir
from collections import Counter
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
```

```

stop_words = set(stopwords.words('english'))
tokens = [w for w in tokens if not w in stop_words]
# filter out short tokens
tokens = [word for word in tokens if len(word) > 1]
return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
    # load doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # update counts
    vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # add doc to vocab
        add_doc_to_vocab(path, vocab)

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/pos', vocab)
process_docs('txt_sentoken/neg', vocab)
# print the size of the vocab
print(len(vocab))
# print the top words in the vocab
print(vocab.most_common(50))

```

Listing 10.3: Example of selecting a vocabulary for the dataset.

Running the example shows that we have a vocabulary of 44,276 words. We also can see a sample of the top 50 most used words in the movie reviews. Note that this vocabulary was constructed based on only those reviews in the training dataset.

```

44276
[('film', 7983), ('one', 4946), ('movie', 4826), ('like', 3201), ('even', 2262), ('good', 2080), ('time', 2041), ('story', 1907), ('films', 1873), ('would', 1844), ('much', 1824), ('also', 1757), ('characters', 1735), ('get', 1724), ('character', 1703), ('two', 1643), ('first', 1588), ('see', 1557), ('way', 1515), ('well', 1511), ('make', 1418), ('really', 1407), ('little', 1351), ('life', 1334), ('plot', 1288), ('people', 1269), ('could', 1248), ('bad', 1248), ('scene', 1241), ('movies', 1238), ('never', 1201), ('best', 1179), ('new', 1140), ('scenes', 1135), ('man', 1131), ('many', 1130), ('doesnt', 1118), ('know', 1092), ('dont', 1086), ('hes', 1024), ('great', 1014), ('another', 992), ('action', 985), ('love', 977), ('us', 967), ('go', 952), ('director', 948), ('end', 946), ('something', 945), ('still', 936)]

```

Listing 10.4: Example output of selecting a vocabulary for the dataset.

We can step through the vocabulary and remove all words that have a low occurrence, such as only being used once or twice in all reviews. For example, the following snippet will retrieve only the tokens that appear 2 or more times in all reviews.

```
# keep tokens with a min occurrence
min_occurane = 2
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
```

Listing 10.5: Example of filtering the vocabulary by occurrence.

Finally, the vocabulary can be saved to a new file called `vocab.txt` that we can later load and use to filter movie reviews prior to encoding them for modeling. We define a new function called `save_list()` that saves the vocabulary to file, with one word per file. For example:

```
# save list to file
def save_list(lines, filename):
    # convert lines to a single blob of text
    data = '\n'.join(lines)
    # open file
    file = open(filename, 'w')
    # write text
    file.write(data)
    # close file
    file.close()

# save tokens to a vocabulary file
save_list(tokens, 'vocab.txt')
```

Listing 10.6: Example of saving the filtered vocabulary.

Pulling all of this together, the complete example is listed below.

```
import string
import re
from os import listdir
from collections import Counter
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[\s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub(' ', w) for w in tokens]
```

```

# remove remaining tokens that are not alphabetic
tokens = [word for word in tokens if word.isalpha()]
# filter out stop words
stop_words = set(stopwords.words('english'))
tokens = [w for w in tokens if not w in stop_words]
# filter out short tokens
tokens = [word for word in tokens if len(word) > 1]
return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
    # load doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # update counts
    vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # add doc to vocab
        add_doc_to_vocab(path, vocab)

# save list to file
def save_list(lines, filename):
    # convert lines to a single blob of text
    data = '\n'.join(lines)
    # open file
    file = open(filename, 'w')
    # write text
    file.write(data)
    # close file
    file.close()

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/pos', vocab)
process_docs('txt_sentoken/neg', vocab)
# print the size of the vocab
print(len(vocab))
# keep tokens with a min occurrence
min_occurane = 2
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
# save tokens to a vocabulary file
save_list(tokens, 'vocab.txt')

```

Listing 10.7: Example of filtering the vocabulary for the dataset.

Running the above example with this addition shows that the vocabulary size drops by a little more than half its size, from about 44,000 to about 25,000 words.

```
25767
```

Listing 10.8: Example output of filtering the vocabulary by min occurrence.

Running the min occurrence filter on the vocabulary and saving it to file, you should now have a new file called `vocab.txt` with only the words we are interested in.

The order of words in your file will differ, but should look something like the following:

```
aberdeen
dupe
burt
libido
hamlet
arlene
available
corners
web
columbia
...
```

Listing 10.9: Sample of the vocabulary file `vocab.txt`.

We are now ready to look at extracting features from the reviews ready for modeling.

10.4 Bag-of-Words Representation

In this section, we will look at how we can convert each review into a representation that we can provide to a Multilayer Perceptron model. A bag-of-words model is a way of extracting features from text so the text input can be used with machine learning algorithms like neural networks. Each document, in this case a review, is converted into a vector representation. The number of items in the vector representing a document corresponds to the number of words in the vocabulary. The larger the vocabulary, the longer the vector representation, hence the preference for smaller vocabularies in the previous section. The bag-of-words model was introduced previously in Chapter 8.

Words in a document are scored and the scores are placed in the corresponding location in the representation. We will look at different word scoring methods in the next section. In this section, we are concerned with converting reviews into vectors ready for training a first neural network model. This section is divided into 2 steps:

1. Converting reviews to lines of tokens.
2. Encoding reviews with a bag-of-words model representation.

10.4.1 Reviews to Lines of Tokens

Before we can convert reviews to vectors for modeling, we must first clean them up. This involves loading them, performing the cleaning operation developed above, filtering out words not in the chosen vocabulary, and converting the remaining tokens into a single string or line

ready for encoding. First, we need a function to prepare one document. Below lists the function `doc_to_line()` that will load a document, clean it, filter out tokens not in the vocabulary, then return the document as a string of white space separated tokens.

```
# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
    # load the doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # filter by vocab
    tokens = [w for w in tokens if w in vocab]
    return ' '.join(tokens)
```

Listing 10.10: Function to filter a review by pre-defined vocabulary.

Next, we need a function to work through all documents in a directory (such as `pos` and `neg`) to convert the documents into lines. Below lists the `process_docs()` function that does just this, expecting a directory name and a vocabulary set as input arguments and returning a list of processed documents.

```
# load all docs in a directory
def process_docs(directory, vocab):
    lines = list()
    # walk through all files in the folder
    for filename in.listdir(directory):
        # skip any reviews in the test set
        if filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # load and clean the doc
        line = doc_to_line(path, vocab)
        # add to list
        lines.append(line)
    return lines
```

Listing 10.11: Function to filter all movie reviews by vocabulary.

We can call the `process_docs()` consistently for positive and negative reviews to construct a dataset of review text and their associated output labels, 0 for negative and 1 for positive. The `load_clean_dataset()` function below implements this behavior.

```
# load and clean a dataset
def load_clean_dataset(vocab):
    # load documents
    neg = process_docs('txt_sentoken/neg', vocab)
    pos = process_docs('txt_sentoken/pos', vocab)
    docs = neg + pos
    # prepare labels
    labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
    return docs, labels
```

Listing 10.12: Function to load movie reviews and prepare output labels.

Finally, we need to load the vocabulary and turn it into a set for use in cleaning reviews.

```
# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
```

Listing 10.13: Load the pre-defined vocabulary of words.

We can put all of this together, reusing the loading and cleaning functions developed in previous sections. The complete example is listed below, demonstrating how to prepare the positive and negative reviews from the training dataset.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub(' ', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
    # load the doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # filter by vocab
    tokens = [w for w in tokens if w in vocab]
    return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
    lines = list()
```

```

# walk through all files in the folder
for filename in listdir(directory):
    # skip any reviews in the test set
    if filename.startswith('cv9'):
        continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load and clean the doc
    line = doc_to_line(path, vocab)
    # add to list
    lines.append(line)
return lines

# load and clean a dataset
def load_clean_dataset(vocab):
    # load documents
    neg = process_docs('txt_sentoken/neg', vocab)
    pos = process_docs('txt_sentoken/pos', vocab)
    docs = neg + pos
    # prepare labels
    labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
    return docs, labels

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = vocab.split()
vocab = set(vocab)
# load all training reviews
docs, labels = load_clean_dataset(vocab)
# summarize what we have
print(len(docs), len(labels))

```

Listing 10.14: Filter all movie reviews by the pre-defined vocabulary.

Running this example loads and cleans the review text and returns the labels.

```
1800 1800
```

Listing 10.15: Example output from loading, cleaning and filtering movie review data by a constrained vocabulary.

10.4.2 Movie Reviews to Bag-of-Words Vectors

We will use the Keras API to convert reviews to encoded document vectors. Keras provides the `Tokenizer` class that can do some of the cleaning and vocab definition tasks that we took care of in the previous section. It is better to do this ourselves to know exactly what was done and why. Nevertheless, the `Tokenizer` class is convenient and will easily transform documents into encoded vectors. First, the `Tokenizer` must be created, then fit on the text documents in the training dataset. In this case, these are the aggregation of the `positive_lines` and `negative_lines` arrays developed in the previous section.

```

# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()

```

```
tokenizer.fit_on_texts(lines)
return tokenizer
```

Listing 10.16: Function to fit a `Tokenizer` on the clean and filtered movie reviews.

This process determines a consistent way to convert the vocabulary to a fixed-length vector with 25,768 elements, which is the total number of words in the vocabulary file `vocab.txt`. Next, documents can then be encoded using the `Tokenizer` by calling `texts_to_matrix()`. The function takes both a list of documents to encode and an encoding mode, which is the method used to score words in the document. Here we specify `freq` to score words based on their frequency in the document. This can be used to encode the loaded training and test data, for example:

```
# encode data
Xtrain = tokenizer.texts_to_matrix(train_docs, mode='freq')
Xtest = tokenizer.texts_to_matrix(test_docs, mode='freq')
```

Listing 10.17: Encode training data.

This encodes all of the positive and negative reviews in the training dataset.

Next, the `process_docs()` function from the previous section needs to be modified to selectively process reviews in the test or train dataset. We support the loading of both the training and test datasets by adding an `is_train` argument and using that to decide what review file names to skip.

```
# load all docs in a directory
def process_docs(directory, vocab, is_train):
    lines = list()
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if is_train and filename.startswith('cv9'):
            continue
        if not is_train and not filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # load and clean the doc
        line = doc_to_line(path, vocab)
        # add to list
        lines.append(line)
    return lines
```

Listing 10.18: Updated process documents function to load all reviews.

Similarly, the `load_clean_dataset()` dataset must be updated to load either train or test data.

```
# load and clean a dataset
def load_clean_dataset(vocab, is_train):
    # load documents
    neg = process_docs('txt_sentoken/neg', vocab, is_train)
    pos = process_docs('txt_sentoken/pos', vocab, is_train)
    docs = neg + pos
    # prepare labels
    labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
```

```
    return docs, labels
```

Listing 10.19: Function to load text data and labels.

We can put all of this together in a single example.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords
from keras.preprocessing.text import Tokenizer

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub(' ', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
    # load the doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # filter by vocab
    tokens = [w for w in tokens if w in vocab]
    return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab, is_train):
    lines = list()
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if is_train and filename.startswith('cv9'):
            continue
```

```

if not is_train and not filename.startswith('cv9'):
    continue
# create the full path of the file to open
path = directory + '/' + filename
# load and clean the doc
line = doc_to_line(path, vocab)
# add to list
lines.append(line)
return lines

# load and clean a dataset
def load_clean_dataset(vocab, is_train):
    # load documents
    neg = process_docs('txt_sentoken/neg', vocab, is_train)
    pos = process_docs('txt_sentoken/pos', vocab, is_train)
    docs = neg + pos
    # prepare labels
    labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
    return docs, labels

# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
test_docs, ytest = load_clean_dataset(vocab, False)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# encode data
Xtrain = tokenizer.texts_to_matrix(train_docs, mode='freq')
Xtest = tokenizer.texts_to_matrix(test_docs, mode='freq')
print(Xtrain.shape, Xtest.shape)

```

Listing 10.20: Complete example of preparing train and test data.

Running the example prints both the shape of the encoded training dataset and test dataset with 1,800 and 200 documents respectively, each with the same sized encoding vocabulary (vector length).

(1800, 25768) (200, 25768)

Listing 10.21: Example output of loading and preparing the datasets.

10.5 Sentiment Analysis Models

In this section, we will develop Multilayer Perceptron (MLP) models to classify encoded documents as either positive or negative. The models will be simple feedforward network models

with fully connected layers called `Dense` in the Keras deep learning library. This section is divided into 3 sections:

1. First sentiment analysis model
2. Comparing word scoring modes
3. Making a prediction for new reviews

10.5.1 First Sentiment Analysis Model

We can develop a simple MLP model to predict the sentiment of encoded reviews. The model will have an input layer that equals the number of words in the vocabulary, and in turn the length of the input documents. We can store this in a new variable called `n_words`, as follows:

```
n_words = Xtest.shape[1]
```

Listing 10.22: Example of calculating the number of words.

We can now define the network. All model configuration was found with very little trial and error and should not be considered tuned for this problem. We will use a single hidden layer with 50 neurons and a rectified linear activation function. The output layer is a single neuron with a sigmoid activation function for predicting 0 for negative and 1 for positive reviews. The network will be trained using the efficient Adam implementation of gradient descent and the binary cross entropy loss function, suited to binary classification problems. We will keep track of accuracy when training and evaluating the model.

```
# define the model
def define_model(n_words):
    # define network
    model = Sequential()
    model.add(Dense(50, input_shape=(n_words,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # compile network
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model
```

Listing 10.23: Example of defining an MLP for the bag-of-words model.

Next, we can fit the model on the training data; in this case, the model is small and is easily fit in 10 epochs.

```
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
```

Listing 10.24: Example of fitting the defined model.

Finally, once the model is trained, we can evaluate its performance by making predictions in the test dataset and printing the accuracy.

```
# evaluate
loss, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %f' % (acc*100))
```

Listing 10.25: Example of evaluating the fit model.

The complete example is listed below.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords
from keras.preprocessing.text import Tokenizer
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
    # load the doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # filter by vocab
    tokens = [w for w in tokens if w in vocab]
    return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab, is_train):
```

```
lines = list()
# walk through all files in the folder
for filename in listdir(directory):
    # skip any reviews in the test set
    if is_train and filename.startswith('cv9'):
        continue
    if not is_train and not filename.startswith('cv9'):
        continue
    # create the full path of the file to open
    path = directory + '/' + filename
    # load and clean the doc
    line = doc_to_line(path, vocab)
    # add to list
    lines.append(line)
return lines

# load and clean a dataset
def load_clean_dataset(vocab, is_train):
    # load documents
    neg = process_docs('txt_sentoken/neg', vocab, is_train)
    pos = process_docs('txt_sentoken/pos', vocab, is_train)
    docs = neg + pos
    # prepare labels
    labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
    return docs, labels

# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# define the model
def define_model(n_words):
    # define network
    model = Sequential()
    model.add(Dense(50, input_shape=(n_words,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # compile network
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
test_docs, ytest = load_clean_dataset(vocab, False)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# encode data
Xtrain = tokenizer.texts_to_matrix(train_docs, mode='freq')
```

```
Xtest = tokenizer.texts_to_matrix(test_docs, mode='freq')
# define the model
n_words = Xtest.shape[1]
model = define_model(n_words)
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
# evaluate
loss, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %f' % (acc*100))
```

Listing 10.26: Complete example of training and evaluating an MLP bag-of-words model.

Running the example first prints a summary of the defined model.

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 50)	1288450
dense_2 (Dense)	(None, 1)	51
<hr/>		
Total params: 1,288,501		
Trainable params: 1,288,501		
Non-trainable params: 0		
<hr/>		

Listing 10.27: Summary of the defined model.

A plot the defined model is then saved to file with the name `model.png`.

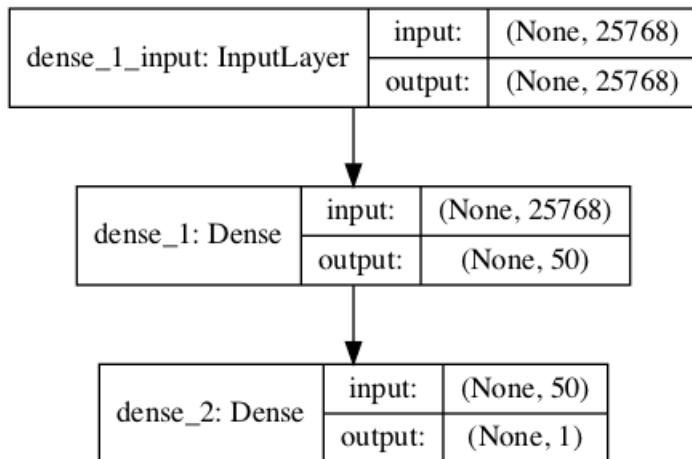


Figure 10.1: Plot of the defined bag-of-words model.

We can see that the model easily fits the training data within the 10 epochs, achieving close to 100% accuracy. Evaluating the model on the test dataset, we can see that model does well, achieving an accuracy of above 87%, well within the ballpark of low-to-mid 80s seen in the original paper. Although, it is important to note that this is not an apples-to-apples comparison, as the original paper used 10-fold cross-validation to estimate model skill instead of a single train/test split.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
...
Epoch 6/10
0s - loss: 0.5319 - acc: 0.9428
Epoch 7/10
0s - loss: 0.4839 - acc: 0.9506
Epoch 8/10
0s - loss: 0.4368 - acc: 0.9567
Epoch 9/10
0s - loss: 0.3927 - acc: 0.9611
Epoch 10/10
0s - loss: 0.3516 - acc: 0.9689

Test Accuracy: 87.000000
```

Listing 10.28: Example output of training and evaluating the MLP model.

Next, let's look at testing different word scoring methods for the bag-of-words model.

10.6 Comparing Word Scoring Methods

The `texts_to_matrix()` function for the `Tokenizer` in the Keras API provides 4 different methods for scoring words; they are:

- `binary` Where words are marked as present (1) or absent (0).
- `count` Where the occurrence count for each word is marked as an integer.
- `tfidf` Where each word is scored based on their frequency, where words that are common across all documents are penalized.
- `freq` Where words are scored based on their frequency of occurrence within the document.

We can evaluate the skill of the model developed in the previous section fit using each of the 4 supported word scoring modes. This first involves the development of a function to create an encoding of the loaded documents based on a chosen scoring model. The function creates the tokenizer, fits it on the training documents, then creates the train and test encodings using the chosen model. The function `prepare_data()` implements this behavior given lists of train and test documents.

```
# prepare bag-of-words encoding of docs
def prepare_data(train_docs, test_docs, mode):
    # create the tokenizer
    tokenizer = Tokenizer()
    # fit the tokenizer on the documents
    tokenizer.fit_on_texts(train_docs)
    # encode training data set
    Xtrain = tokenizer.texts_to_matrix(train_docs, mode=mode)
    # encode training data set
    Xtest = tokenizer.texts_to_matrix(test_docs, mode=mode)
```

```
return Xtrain, Xtest
```

Listing 10.29: Updated data preparation to take encoding mode as a parameter.

We also need a function to evaluate the MLP given a specific encoding of the data. Because neural networks are stochastic, they can produce different results when the same model is fit on the same data. This is mainly because of the random initial weights and the shuffling of patterns during mini-batch gradient descent. This means that any one scoring of a model is unreliable and we should estimate model skill based on an average of multiple runs. The function below, named `evaluate_mode()`, takes encoded documents and evaluates the MLP by training it on the train set and estimating skill on the test set 10 times and returns a list of the accuracy scores across all of these runs.

```
# evaluate a neural network model
def evaluate_mode(Xtrain, ytrain, Xtest, ytest):
    scores = list()
    n_repeats = 30
    n_words = Xtest.shape[1]
    for i in range(n_repeats):
        # define network
        model = Sequential()
        model.add(Dense(50, input_shape=(n_words,), activation='relu'))
        model.add(Dense(1, activation='sigmoid'))
        # compile network
        model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
        # fit network
        model.fit(Xtrain, ytrain, epochs=10, verbose=2)
        # evaluate
        loss, acc = model.evaluate(Xtest, ytest, verbose=0)
        scores.append(acc)
        print('%d accuracy: %s' % ((i+1), acc))
    return scores
```

Listing 10.30: Function to create, fit and evaluate a model multiple times.

We are now ready to evaluate the performance of the 4 different word scoring methods. Pulling all of this together, the complete example is listed below.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords
from keras.preprocessing.text import Tokenizer
from keras.models import Sequential
from keras.layers import Dense
from pandas import DataFrame
from matplotlib import pyplot

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
```

```
return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
    # load the doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # filter by vocab
    tokens = [w for w in tokens if w in vocab]
    return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab, is_train):
    lines = list()
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if is_train and filename.startswith('cv9'):
            continue
        if not is_train and not filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # load and clean the doc
        line = doc_to_line(path, vocab)
        # add to list
        lines.append(line)
    return lines

# load and clean a dataset
def load_clean_dataset(vocab, is_train):
    # load documents
    neg = process_docs('txt_sentoken/neg', vocab, is_train)
    pos = process_docs('txt_sentoken/pos', vocab, is_train)
    docs = neg + pos
    # prepare labels
    labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
    return docs, labels
```

```
# define the model
def define_model(n_words):
    # define network
    model = Sequential()
    model.add(Dense(50, input_shape=(n_words,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # compile network
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# evaluate a neural network model
def evaluate_mode(Xtrain, ytrain, Xtest, ytest):
    scores = list()
    n_repeats = 10
    n_words = Xtest.shape[1]
    for i in range(n_repeats):
        # define network
        model = define_model(n_words)
        # fit network
        model.fit(Xtrain, ytrain, epochs=10, verbose=0)
        # evaluate
        _, acc = model.evaluate(Xtest, ytest, verbose=0)
        scores.append(acc)
        print('%d accuracy: %s' % ((i+1), acc))
    return scores

# prepare bag of words encoding of docs
def prepare_data(train_docs, test_docs, mode):
    # create the tokenizer
    tokenizer = Tokenizer()
    # fit the tokenizer on the documents
    tokenizer.fit_on_texts(train_docs)
    # encode training data set
    Xtrain = tokenizer.texts_to_matrix(train_docs, mode=mode)
    # encode training data set
    Xtest = tokenizer.texts_to_matrix(test_docs, mode=mode)
    return Xtrain, Xtest

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
test_docs, ytest = load_clean_dataset(vocab, False)
# run experiment
modes = ['binary', 'count', 'tfidf', 'freq']
results = DataFrame()
for mode in modes:
    # prepare data for mode
    Xtrain, Xtest = prepare_data(train_docs, test_docs, mode)
    # evaluate model on data for mode
    results[mode] = evaluate_mode(Xtrain, ytrain, Xtest, ytest)
# summarize results
print(results.describe())
```

```
# plot results
results.boxplot()
pyplot.show()
```

Listing 10.31: Complete example of comparing document encoding schemes.

At the end of the run, summary statistics for each word scoring method are provided, summarizing the distribution of model skill scores across each of the 10 runs per mode. We can see that the mean score of both the `count` and `binary` methods appear to be better than `freq` and `tfidf`.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

	binary	count	tfidf	freq
count	10.000000	10.000000	10.000000	10.000000
mean	0.927000	0.903500	0.876500	0.871000
std	0.011595	0.009144	0.017958	0.005164
min	0.910000	0.885000	0.855000	0.865000
25%	0.921250	0.900000	0.861250	0.866250
50%	0.927500	0.905000	0.875000	0.870000
75%	0.933750	0.908750	0.888750	0.875000
max	0.945000	0.915000	0.910000	0.880000

Listing 10.32: Example output of comparing document encoding schemes.

A box and whisker plot of the results is also presented, summarizing the accuracy distributions per configuration. We can see that `binary` achieved the best results with a modest spread and might be the preferred approach for this dataset.

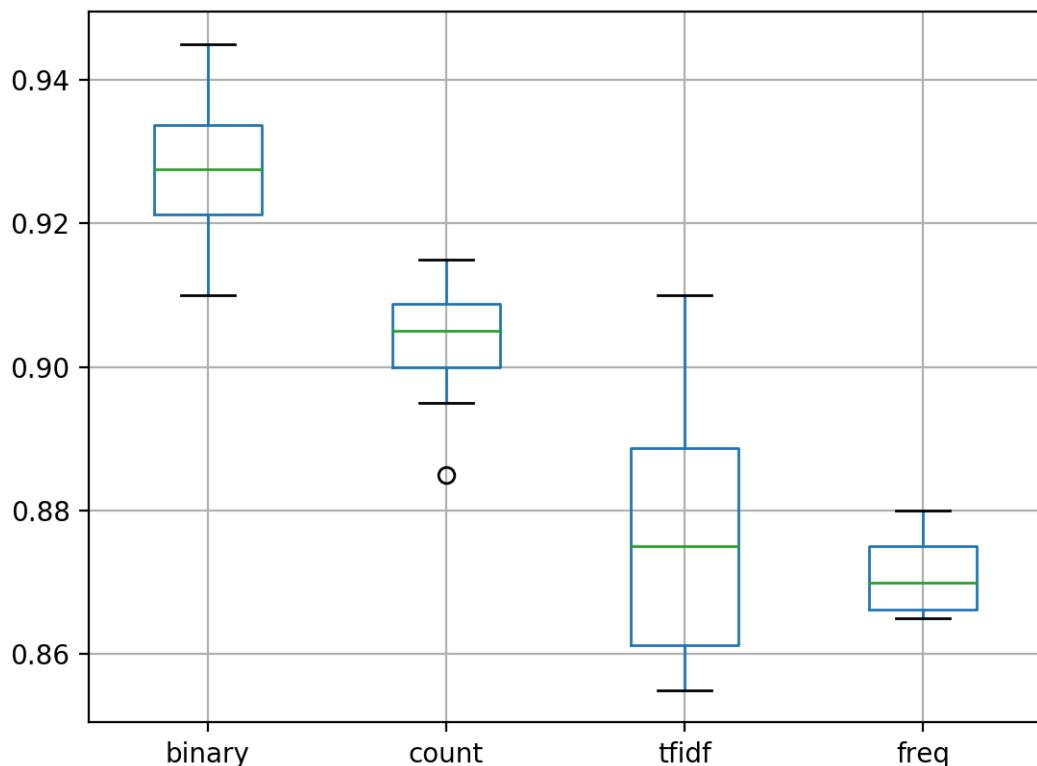


Figure 10.2: Box and Whisker Plot for Model Accuracy with Different Word Scoring Methods.

10.7 Predicting Sentiment for New Reviews

Finally, we can develop and use a final model to make predictions for new textual reviews. This is why we wanted the model in the first place. First we will train a final model on all of the available data. We will use the `binary` mode for scoring the bag-of-words model that was shown to give the best results in the previous section.

Predicting the sentiment of new reviews involves following the same steps used to prepare the test data. Specifically, loading the text, cleaning the document, filtering tokens by the chosen vocabulary, converting the remaining tokens to a line, encoding it using the `Tokenizer`, and making a prediction. We can make a prediction of a class value directly with the fit model by calling `predict()` that will return an integer of 0 for a negative review and 1 for a positive review. All of these steps can be put into a new function called `predict_sentiment()` that requires the review text, the vocabulary, the tokenizer, and the fit model and returns the predicted sentiment and an associated percentage or confidence-like output.

```
# classify a review as negative or positive
def predict_sentiment(review, vocab, tokenizer, model):
    # clean
    tokens = clean_doc(review)
    # filter by vocab
```

```

tokens = [w for w in tokens if w in vocab]
# convert to line
line = ' '.join(tokens)
# encode
encoded = tokenizer.texts_to_matrix([line], mode='binary')
# predict sentiment
yhat = model.predict(encoded, verbose=0)
# retrieve predicted percentage and label
percent_pos = yhat[0,0]
if round(percent_pos) == 0:
    return (1-percent_pos), 'NEGATIVE'
return percent_pos, 'POSITIVE'

```

Listing 10.33: Function for making predictions for new reviews.

We can now make predictions for new review texts. Below is an example with both a clearly positive and a clearly negative review using the simple MLP developed above with the frequency word scoring mode.

```

# test positive text
text = 'Best movie ever! It was great, I recommend it.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))
# test negative text
text = 'This is a bad movie.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))

```

Listing 10.34: Exampling of making predictions for new reviews.

Pulling this all together, the complete example for making predictions for new reviews is listed below.

```

import string
import re
from os import listdir
from nltk.corpus import stopwords
from keras.preprocessing.text import Tokenizer
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering

```

```
re_punc = re.compile(' [%s]' % re.escape(string.punctuation))
# remove punctuation from each word
tokens = [re_punc.sub('', w) for w in tokens]
# remove remaining tokens that are not alphabetic
tokens = [word for word in tokens if word.isalpha()]
# filter out stop words
stop_words = set(stopwords.words('english'))
tokens = [w for w in tokens if not w in stop_words]
# filter out short tokens
tokens = [word for word in tokens if len(word) > 1]
return tokens

# load doc, clean and return line of tokens
def doc_to_line(filename, vocab):
    # load the doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # filter by vocab
    tokens = [w for w in tokens if w in vocab]
    return ' '.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
    lines = list()
    # walk through all files in the folder
    for filename in listdir(directory):
        # create the full path of the file to open
        path = directory + '/' + filename
        # load and clean the doc
        line = doc_to_line(path, vocab)
        # add to list
        lines.append(line)
    return lines

# load and clean a dataset
def load_clean_dataset(vocab):
    # load documents
    neg = process_docs('txt_sentoken/neg', vocab)
    pos = process_docs('txt_sentoken/pos', vocab)
    docs = neg + pos
    # prepare labels
    labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
    return docs, labels

# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# define the model
def define_model(n_words):
    # define network
    model = Sequential()
    model.add(Dense(50, input_shape=(n_words,), activation='relu'))
```

```

model.add(Dense(1, activation='sigmoid'))
# compile network
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# summarize defined model
model.summary()
plot_model(model, to_file='model.png', show_shapes=True)
return model

# classify a review as negative or positive
def predict_sentiment(review, vocab, tokenizer, model):
    # clean
    tokens = clean_doc(review)
    # filter by vocab
    tokens = [w for w in tokens if w in vocab]
    # convert to line
    line = ' '.join(tokens)
    # encode
    encoded = tokenizer.texts_to_matrix([line], mode='binary')
    # predict sentiment
    yhat = model.predict(encoded, verbose=0)
    # retrieve predicted percentage and label
    percent_pos = yhat[0,0]
    if round(percent_pos) == 0:
        return (1-percent_pos), 'NEGATIVE'
    return percent_pos, 'POSITIVE'

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab)
test_docs, ytest = load_clean_dataset(vocab)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# encode data
Xtrain = tokenizer.texts_to_matrix(train_docs, mode='binary')
Xtest = tokenizer.texts_to_matrix(test_docs, mode='binary')
# define network
n_words = Xtrain.shape[1]
model = define_model(n_words)
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
# test positive text
text = 'Best movie ever! It was great, I recommend it.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))
# test negative text
text = 'This is a bad movie.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))

```

Listing 10.35: Complete example of making predictions for new review data.

Running the example correctly classifies these reviews.

Review: [Best movie ever! It was great, I recommend it.]
--

```
Sentiment: POSITIVE (57.124%)
Review: [This is a bad movie.]
Sentiment: NEGATIVE (64.404%)
```

Listing 10.36: Example output of making predictions for new reviews.

Ideally, we would fit the model on all available data (train and test) to create a final model and save the model and tokenizer to file so that they can be loaded and used in new software.

10.8 Extensions

This section lists some extensions if you are looking to get more out of this tutorial.

- **Manage Vocabulary.** Explore using a larger or smaller vocabulary. Perhaps you can get better performance with a smaller set of words.
- **Tune the Network Topology.** Explore alternate network topologies such as deeper or wider networks. Perhaps you can get better performance with a more suited network.
- **Use Regularization.** Explore the use of regularization techniques, such as dropout. Perhaps you can delay the convergence of the model and achieve better test set performance.
- **More Data Cleaning.** Explore more or less cleaning of the review text and see how it impacts the model skill.
- **Training Diagnostics.** Use the test dataset as a validation dataset during training and create plots of train and test loss. Use these diagnostics to tune the batch size and number of training epochs.
- **Trigger Words.** Explore whether there are specific words in reviews that are highly predictive of the sentiment.
- **Use Bigrams.** Prepare the model to score bigrams of words and evaluate the performance under different scoring schemes.
- **Truncated Reviews.** Explore how using a truncated version of the movie reviews results impacts model skill, try truncating the start, end and middle of reviews.
- **Ensemble Models.** Create models with different word scoring schemes and see if using ensembles of the models results in improves to model skill.
- **Real Reviews.** Train a final model on all data and evaluate the model on real movie reviews taken from the internet.

If you explore any of these extensions, I'd love to know.

10.9 Further Reading

This section provides more resources on the topic if you are looking go deeper.

10.9.1 Dataset

- Movie Review Data.
<http://www.cs.cornell.edu/people/pabo/movie-review-data/>
- *A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts*, 2004.
<http://xxx.lanl.gov/abs/cs/0409058>
- Movie Review Polarity Dataset.
http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

10.9.2 APIs

- `nltk.tokenize` package API.
<http://www.nltk.org/api/nltk.tokenize.html>
- Chapter 2, *Accessing Text Corpora and Lexical Resources*.
<http://www.nltk.org/book/ch02.html>
- `os` API Miscellaneous operating system interfaces.
<https://docs.python.org/3/library/os.html>
- `collections` API - Container datatypes.
<https://docs.python.org/3/library/collections.html>
- Tokenizer Keras API.
<https://keras.io/preprocessing/text/#tokenizer>

10.10 Summary

In this tutorial, you discovered how to develop a bag-of-words model for predicting the sentiment of movie reviews. Specifically, you learned:

- How to prepare the review text data for modeling with a restricted vocabulary.
- How to use the bag-of-words model to prepare train and test data.
- How to develop a Multilayer Perceptron bag-of-words model and use it to make predictions on new review text data.

10.10.1 Next

This is the final chapter in the bag-of-words part. In the next part, you will discover how to develop word embedding models.

Part V

Word Embeddings

Chapter 11

The Word Embedding Model

Word embeddings are a type of word representation that allows words with similar meaning to have a similar representation. They are a distributed representation for text that is perhaps one of the key breakthroughs for the impressive performance of deep learning methods on challenging natural language processing problems. In this chapter, you will discover the word embedding approach for representing text data. After completing this chapter, you will know:

- What the word embedding approach for representing text is and how it differs from other feature extraction methods.
- That there are 3 main algorithms for learning a word embedding from text data.
- That you can either train a new embedding or use a pre-trained embedding on your natural language processing task.

Let's get started.

11.1 Overview

This tutorial is divided into the following parts:

1. What Are Word Embeddings?
2. Word Embedding Algorithms
3. Using Word Embeddings

11.2 What Are Word Embeddings?

A word embedding is a learned representation for text where words that have the same meaning have a similar representation. It is this approach to representing words and documents that may be considered one of the key breakthroughs of deep learning on challenging natural language processing problems.

One of the benefits of using dense and low-dimensional vectors is computational: the majority of neural network toolkits do not play well with very high-dimensional, sparse vectors. ... The main benefit of the dense representations is generalization power: if we believe some features may provide similar clues, it is worthwhile to provide a representation that is able to capture these similarities.

— Page 92, *Neural Network Methods in Natural Language Processing*, 2017.

Word embeddings are in fact a class of techniques where individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network, and hence the technique is often lumped into the field of deep learning.

Key to the approach is the idea of using a dense distributed representation for each word. Each word is represented by a real-valued vector, often tens or hundreds of dimensions. This is contrasted to the thousands or millions of dimensions required for sparse word representations, such as a one hot encoding.

associate with each word in the vocabulary a distributed word feature vector ... The feature vector represents different aspects of the word: each word is associated with a point in a vector space. The number of features ... is much smaller than the size of the vocabulary

— *A Neural Probabilistic Language Model*, 2003.

The distributed representation is learned based on the usage of words. This allows words that are used in similar ways to result in having similar representations, naturally capturing their meaning. This can be contrasted with the crisp but fragile representation in a bag-of-words model where, unless explicitly managed, different words have different representations, regardless of how they are used.

There is deeper linguistic theory behind the approach, namely the *distributional hypothesis* by Zellig Harris that could be summarized as: words that have similar context will have similar meanings. For more depth see Harris' 1956 paper *Distributional structure*. This notion of letting the usage of the word define its meaning can be summarized by an oft repeated quip by John Firth:

You shall know a word by the company it keeps!

— Page 11, *A synopsis of linguistic theory 1930-1955*, in Studies in Linguistic Analysis
1930-1955, 1962.

11.3 Word Embedding Algorithms

Word embedding methods learn a real-valued vector representation for a predefined fixed sized vocabulary from a corpus of text. The learning process is either joint with the neural network model on some task, such as document classification, or is an unsupervised process, using document statistics. This section reviews three techniques that can be used to learn a word embedding from text data.

11.3.1 Embedding Layer

An embedding layer, for lack of a better name, is a word embedding that is learned jointly with a neural network model on a specific natural language processing task, such as language modeling or document classification. It requires that document text be cleaned and prepared such that each word is one hot encoded. The size of the vector space is specified as part of the model, such as 50, 100, or 300 dimensions. The vectors are initialized with small random numbers. The embedding layer is used on the front end of a neural network and is fit in a supervised way using the Backpropagation algorithm.

... when the input to a neural network contains symbolic categorical features (e.g. features that take one of k distinct symbols, such as words from a closed vocabulary), it is common to associate each possible feature value (i.e., each word in the vocabulary) with a d -dimensional vector for some d . These vectors are then considered parameters of the model, and are trained jointly with the other parameters.

— Page 49, *Neural Network Methods in Natural Language Processing*, 2017.

The one hot encoded words are mapped to the word vectors. If a Multilayer Perceptron model is used, then the word vectors are concatenated before being fed as input to the model. If a recurrent neural network is used, then each word may be taken as one input in a sequence. This approach of learning an embedding layer requires a lot of training data and can be slow, but will learn an embedding both targeted to the specific text data and the NLP task.

11.3.2 Word2Vec

Word2Vec is a statistical method for efficiently learning a standalone word embedding from a text corpus. It was developed by Tomas Mikolov, et al. at Google in 2013 as a response to make the neural-network-based training of the embedding more efficient and since then has become the de facto standard for developing pre-trained word embedding.

Additionally, the work involved analysis of the learned vectors and the exploration of vector math on the representations of words. For example, that subtracting the *man-ness* from *King* and adding *women-ness* results in the word *Queen*, capturing the analogy *king is to queen as man is to woman*.

We find that these representations are surprisingly good at capturing syntactic and semantic regularities in language, and that each relationship is characterized by a relation-specific vector offset. This allows vector-oriented reasoning based on the offsets between words. For example, the male/female relationship is automatically learned, and with the induced vector representations, *King - Man + Woman* results in a vector very close to *Queen*.

— *Linguistic Regularities in Continuous Space Word Representations*, 2013.

Two different learning models were introduced that can be used as part of the Word2Vec approach to learn the word embedding; they are:

- Continuous Bag-of-Words, or CBOW model.
- Continuous Skip-Gram Model.

The CBOW model learns the embedding by predicting the current word based on its context. The continuous skip-gram model learns by predicting the surrounding words given a current word.

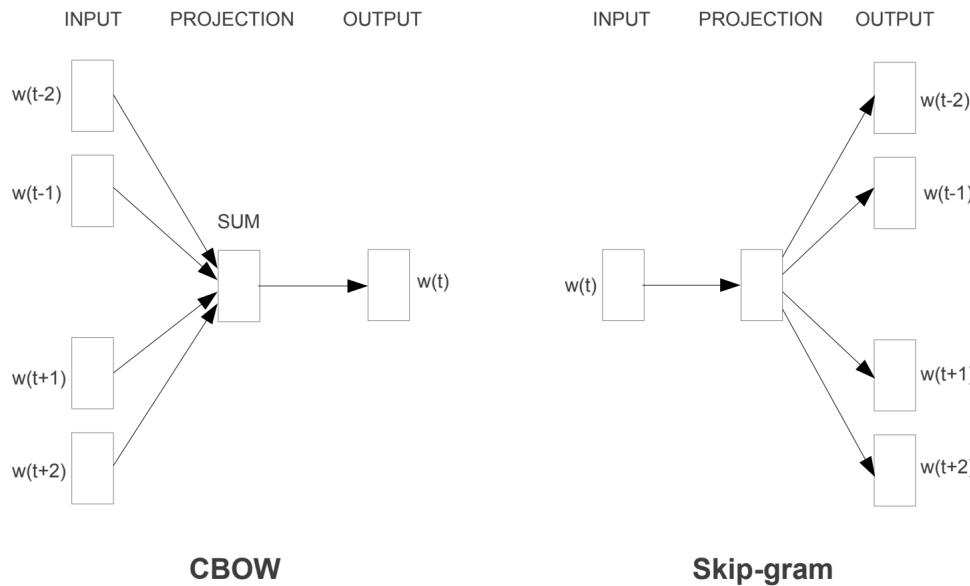


Figure 11.1: Word2Vec Training Models. Taken from *Efficient Estimation of Word Representations in Vector Space*, 2013

Both models are focused on learning about words given their local usage context, where the context is defined by a window of neighboring words. This window is a configurable parameter of the model.

The size of the sliding window has a strong effect on the resulting vector similarities. Large windows tend to produce more topical similarities [...], while smaller windows tend to produce more functional and syntactic similarities.

— Page 128, *Neural Network Methods in Natural Language Processing*, 2017.

The key benefit of the approach is that high-quality word embeddings can be learned efficiently (low space and time complexity), allowing larger embeddings to be learned (more dimensions) from much larger corpora of text (billions of words).

11.3.3 GloVe

The Global Vectors for Word Representation, or GloVe, algorithm is an extension to the Word2Vec method for efficiently learning word vectors, developed by Pennington, et al. at Stanford. Classical vector space model representations of words were developed using matrix

factorization techniques such as Latent Semantic Analysis (LSA) that do a good job of using global text statistics but are not as good as the learned methods like Word2Vec at capturing meaning and demonstrating it on tasks like calculating analogies (e.g. the King and Queen example above).

GloVe is an approach to marry both the global statistics of matrix factorization techniques like LSA with the local context-based learning in Word2Vec. Rather than using a window to define local context, GloVe constructs an explicit word-context or word co-occurrence matrix using statistics across the whole text corpus. The result is a learning model that may result in generally better word embeddings.

GloVe, is a new global log-bilinear regression model for the unsupervised learning of word representations that outperforms other models on word analogy, word similarity, and named entity recognition tasks.

— *GloVe: Global Vectors for Word Representation*, 2014.

11.4 Using Word Embeddings

You have some options when it comes time to using word embeddings on your natural language processing project. This section outlines those options.

11.4.1 Learn an Embedding

You may choose to learn a word embedding for your problem. This will require a large amount of text data to ensure that useful embeddings are learned, such as millions or billions of words. You have two main options when training your word embedding:

- **Learn it Standalone**, where a model is trained to learn the embedding, which is saved and used as a part of another model for your task later. This is a good approach if you would like to use the same embedding in multiple models.
- **Learn Jointly**, where the embedding is learned as part of a large task-specific model. This is a good approach if you only intend to use the embedding on one task.

11.4.2 Reuse an Embedding

It is common for researchers to make pre-trained word embeddings available for free, often under a permissive license so that you can use them on your own academic or commercial projects. For example, both Word2Vec and GloVe word embeddings are available for free download. These can be used on your project instead of training your own embeddings from scratch. You have two main options when it comes to using pre-trained embeddings:

- **Static**, where the embedding is kept static and is used as a component of your model. This is a suitable approach if the embedding is a good fit for your problem and gives good results.
- **Updated**, where the pre-trained embedding is used to seed the model, but the embedding is updated jointly during the training of the model. This may be a good option if you are looking to get the most out of the model and embedding on your task.

11.4.3 Which Option Should You Use?

Explore the different options, and if possible, test to see which gives the best results on your problem. Perhaps start with fast methods, like using a pre-trained embedding, and only use a new embedding if it results in better performance on your problem.

11.5 Further Reading

This section provides more resources on the topic if you are looking go deeper.

11.5.1 Books

- Neural Network Methods in Natural Language Processing, 2017.
<http://amzn.to/2wycQKA>

11.5.2 Articles

- Word embedding on Wikipedia.
https://en.wikipedia.org/wiki/Word_embedding
- Word2Vec on Wikipedia.
<https://en.wikipedia.org/wiki/Word2vec>
- GloVe on Wikipedia
[https://en.wikipedia.org/wiki/GloVe_\(machine_learning\)](https://en.wikipedia.org/wiki/GloVe_(machine_learning))
- *An overview of word embeddings and their connection to distributional semantic models*, 2016.
<http://blog.aylien.com/overview-word-embeddings-history-word2vec-cbow-glove/>
- *Deep Learning, NLP, and Representations*, 2014.
<http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/>

11.5.3 Papers

- *Distributional structure*, 1956.
<http://www.tandfonline.com/doi/pdf/10.1080/00437956.1954.11659520>
- *A Neural Probabilistic Language Model*, 2003.
<http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf>
- *A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning*, 2008.
https://ronan.collobert.com/pub/matos/2008_nlp_icml.pdf
- *Continuous space language models*, 2007.
<https://pdfs.semanticscholar.org/0fcc/184b3b90405ec3ceaf6a4007c749df7c363.pdf>

- *Efficient Estimation of Word Representations in Vector Space*, 2013.
<https://arxiv.org/pdf/1301.3781.pdf>
- *Distributed Representations of Words and Phrases and their Compositionality*, 2013.
<https://arxiv.org/pdf/1310.4546.pdf>
- *GloVe: Global Vectors for Word Representation*, 2014.
<https://nlp.stanford.edu/pubs/glove.pdf>

11.5.4 Projects

- Word2Vec on Google Code.
<https://code.google.com/archive/p/word2vec/>
- GloVe: Global Vectors for Word Representation.
<https://nlp.stanford.edu/projects/glove/>

11.6 Summary

In this chapter, you discovered Word Embeddings as a representation method for text in deep learning applications. Specifically, you learned:

- What the word embedding approach for representation text is and how it differs from other feature extraction methods.
- That there are 3 main algorithms for learning a word embedding from text data.
- That you can either train a new embedding or use a pre-trained embedding on your natural language processing task.

11.6.1 Next

In the next chapter, you will discover how you can train and manipulate word embeddings using the Gensim Python library.

Chapter 12

How to Develop Word Embeddings with Gensim

Word embeddings are a modern approach for representing text in natural language processing. Embedding algorithms like Word2Vec and GloVe are key to the state-of-the-art results achieved by neural network models on natural language processing problems like machine translation. In this tutorial, you will discover how to train and load word embedding models for natural language processing applications in Python using Gensim. After completing this tutorial, you will know:

- How to train your own Word2Vec word embedding model on text data.
- How to visualize a trained word embedding model using Principal Component Analysis.
- How to load pre-trained Word2Vec and GloVe word embedding models from Google and Stanford.

Let's get started.

12.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Word Embeddings
2. Gensim Library
3. Develop Word2Vec Embedding
4. Visualize Word Embedding
5. Load Google's Word2Vec Embedding
6. Load Stanford's GloVe Embedding

12.2 Word Embeddings

A word embedding is an approach to provide a dense vector representation of words that capture something about their meaning. Word embeddings are an improvement over simpler bag-of-word model word encoding schemes like word counts and frequencies that result in large and sparse vectors (mostly 0 values) that describe documents but not the meaning of the words.

Word embeddings work by using an algorithm to train a set of fixed-length dense and continuous-valued vectors based on a large corpus of text. Each word is represented by a point in the embedding space and these points are learned and moved around based on the words that surround the target word. It is defining a word by the company that it keeps that allows the word embedding to learn something about the meaning of words. The vector space representation of the words provides a projection where words with similar meanings are locally clustered within the space.

The use of word embeddings over other text representations is one of the key methods that has led to breakthrough performance with deep neural networks on problems like machine translation. In this tutorial, we are going to look at how to use two different word embedding methods called Word2Vec by researchers at Google and GloVe by researchers at Stanford.

12.3 Gensim Python Library

Gensim is an open source Python library for natural language processing, with a focus on topic modeling. It is billed as “*topic modeling for humans*”. Gensim was developed and is maintained by the Czech natural language processing researcher Radim Rehurek and his company RaRe Technologies. It is not an everything-including-the-kitchen-sink NLP research library (like NLTK); instead, Gensim is a mature, focused, and efficient suite of NLP tools for topic modeling. Most notably for this tutorial, it supports an implementation of the Word2Vec word embedding for learning new word vectors from text.

It also provides tools for loading pre-trained word embeddings in a few formats and for making use and querying a loaded embedding. We will use the Gensim library in this tutorial. Gensim can be installed easily using pip or easy_install. For example, you can install Gensim with pip by typing the following on your command line:

```
sudo pip install -U gensim
```

Listing 12.1: Install the Gensim library with pip.

If you need help installing Gensim on your system, you can see the Gensim Installation Instructions (linked at the end of the chapter).

12.4 Develop Word2Vec Embedding

Word2Vec is one algorithm for learning a word embedding from a text corpus. There are two main training algorithms that can be used to learn the embedding from text; they are Continuous Bag-of-Words (CBOW) and skip grams. We will not get into the algorithms other than to say that they generally look at a window of words for each target word to provide context and in turn meaning for words. The approach was developed by Tomas Mikolov, formerly at Google and currently at Facebook.

Word2Vec models require a lot of text, e.g. the entire Wikipedia corpus. Nevertheless, we will demonstrate the principles using a small in-memory example of text. Gensim provides the `Word2Vec` class for working with a Word2Vec model. Learning a word embedding from text involves loading and organizing the text into sentences and providing them to the constructor of a new `Word2Vec()` instance. For example:

```
sentences = ...
model = Word2Vec(sentences)
```

Listing 12.2: Example of creating a Word2Vec model.

Specifically, each sentence must be tokenized, meaning divided into words and prepared (e.g. perhaps pre-filtered and perhaps converted to a preferred case). The sentences could be text loaded into memory, or an iterator that progressively loads text, required for very large text corpora. There are many parameters on this constructor; a few noteworthy arguments you may wish to configure are:

- `size`: (default 100) The number of dimensions of the embedding, e.g. the length of the dense vector to represent each token (word).
- `window`: (default 5) The maximum distance between a target word and words around the target word.
- `min_count`: (default 5) The minimum count of words to consider when training the model; words with an occurrence less than this count will be ignored.
- `workers`: (default 3) The number of threads to use while training.
- `sg`: (default 0 or CBOW) The training algorithm, either CBOW (0) or skip gram (1).

The defaults are often good enough when just getting started. If you have a lot of cores, as most modern computers do, I strongly encourage you to increase workers to match the number of cores (e.g. 8). After the model is trained, it is accessible via the `wv` attribute. This is the actual word vector model in which queries can be made. For example, you can print the learned vocabulary of tokens (words) as follows:

```
words = list(model.wv.vocab)
print(words)
```

Listing 12.3: Example of summarizing the words in the model vocabulary.

You can review the embedded vector for a specific token as follows:

```
print(model['word'])
```

Listing 12.4: Example of printing the embedding for a specific word.

Finally, a trained model can then be saved to file by calling the `save_word2vec_format()` function on the word vector model. By default, the model is saved in a binary format to save space. For example:

```
model.wv.save_word2vec_format('model.bin')
```

Listing 12.5: Example of saving a word embedding.

When getting started, you can save the learned model in ASCII format and review the contents. You can do this by setting `binary=False` when calling the `save_word2vec_format()` function, for example:

```
model.wv.save_word2vec_format('model.txt', binary=False)
```

Listing 12.6: Example of saving a word embedding in ASCII format.

The saved model can then be loaded again by calling the `Word2Vec.load()` function. For example:

```
model = Word2Vec.load('model.bin')
```

Listing 12.7: Example of loading a saved word embedding.

We can tie all of this together with a worked example. Rather than loading a large text document or corpus from file, we will work with a small, in-memory list of pre-tokenized sentences. The model is trained and the minimum count for words is set to 1 so that no words are ignored. After the model is learned, we summarize, print the vocabulary, then print a single vector for the word “*sentence*”. Finally, the model is saved to a file in binary format, loaded, and then summarized.

```
from gensim.models import Word2Vec
# define training data
sentences = [['this', 'is', 'the', 'first', 'sentence', 'for', 'word2vec'],
             ['this', 'is', 'the', 'second', 'sentence'],
             ['yet', 'another', 'sentence'],
             ['one', 'more', 'sentence'],
             ['and', 'the', 'final', 'sentence']]
# train model
model = Word2Vec(sentences, min_count=1)
# summarize the loaded model
print(model)
# summarize vocabulary
words = list(model.wv.vocab)
print(words)
# access vector for one word
print(model['sentence'])
# save model
model.save('model.bin')
# load model
new_model = Word2Vec.load('model.bin')
print(new_model)
```

Listing 12.8: Example demonstrating the Word2Vec model in Gensim.

Running the example prints the following output.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
Word2Vec(vocab=14, size=100, alpha=0.025)
['second', 'sentence', 'and', 'this', 'final', 'word2vec', 'for', 'another', 'one',
 'first', 'more', 'the', 'yet', 'is']
[ -4.61881841e-03 -4.88735968e-03 -3.19508743e-03 4.08568839e-03
 -3.38211656e-03 1.93076557e-03 3.90265253e-03 -1.04349572e-03
```

```

4.14286414e-03 1.55219622e-03 3.85653134e-03 2.22428422e-03
-3.52565176e-03 2.82056746e-03 -2.11121864e-03 -1.38054823e-03
-1.12888147e-03 -2.87318649e-03 -7.99703528e-04 3.67874932e-03
2.68940022e-03 6.31021452e-04 -4.36326629e-03 2.38655557e-04
-1.94210222e-03 4.87691024e-03 -4.04118607e-03 -3.17813386e-03
4.94802603e-03 3.43150692e-03 -1.44031656e-03 4.25637932e-03
-1.15106850e-04 -3.73274647e-03 2.50349124e-03 4.28692997e-03
-3.57313151e-03 -7.24728088e-05 -3.46099050e-03 -3.39612062e-03
3.54845310e-03 1.56780297e-03 4.58260969e-04 2.52689526e-04
3.06256465e-03 2.37558200e-03 4.06933809e-03 2.94650183e-03
-2.96231941e-03 -4.47433954e-03 2.89590308e-03 -2.16034567e-03
-2.58548348e-03 -2.06163677e-04 1.72605237e-03 -2.27384618e-04
-3.70194600e-03 2.11557443e-03 2.03793868e-03 3.09839356e-03
-4.71800892e-03 2.32995977e-03 -6.70911541e-05 1.39375112e-03
-3.84263694e-03 -1.03898917e-03 4.13251948e-03 1.06330717e-03
1.38514000e-03 -1.18144893e-03 -2.60811858e-03 1.54952740e-03
2.49916781e-03 -1.95435272e-03 8.86975031e-05 1.89820060e-03
-3.41996481e-03 -4.08187555e-03 5.88635216e-04 4.13103355e-03
-3.25899688e-03 1.02130906e-03 -3.61028523e-03 4.17646067e-03
4.65870230e-03 3.64110398e-04 4.95479070e-03 -1.29743712e-03
-5.03367570e-04 -2.52546836e-03 3.31060472e-03 -3.12870182e-03
-1.14580349e-03 -4.34387522e-03 -4.62882593e-03 3.19007039e-03
2.88707414e-03 1.62976081e-04 -6.05802808e-04 -1.06368808e-03]
Word2Vec(vocab=14, size=100, alpha=0.025)

```

Listing 12.9: Example output of the Word2Vec model in Gensim.

You can see that with a little work to prepare your text document, you can create your own word embedding very easily with Gensim.

12.5 Visualize Word Embedding

After you learn word embedding for your text data, it can be nice to explore it with visualization. You can use classical projection methods to reduce the high-dimensional word vectors to two-dimensional plots and plot them on a graph. The visualizations can provide a qualitative diagnostic for your learned model. We can retrieve all of the vectors from a trained model as follows:

```
X = model[model.wv.vocab]
```

Listing 12.10: Access the model vocabulary.

We can then train a projection method on the vectors, such as those methods offered in scikit-learn, then use Matplotlib to plot the projection as a scatter plot. Let's look at an example with Principal Component Analysis or PCA.

12.5.1 Plot Word Vectors Using PCA

We can create a 2-dimensional PCA model of the word vectors using the scikit-learn PCA class as follows.

```
pca = PCA(n_components=2)
result = pca.fit_transform(X)
```

Listing 12.11: Example of fitting a 2D PCA model to the word vectors.

The resulting projection can be plotted using Matplotlib as follows, pulling out the two dimensions as x and y coordinates.

```
pyplot.scatter(result[:, 0], result[:, 1])
```

Listing 12.12: Example of plotting a scatter plot of the PCA vectors.

We can go one step further and annotate the points on the graph with the words themselves. A crude version without any nice offsets looks as follows.

```
words = list(model.wv.vocab)
for i, word in enumerate(words):
    pyplot.annotate(word, xy=(result[i, 0], result[i, 1]))
```

Listing 12.13: Example of plotting words on the scatter plot.

Putting this all together with the model from the previous section, the complete example is listed below.

```
from gensim.models import Word2Vec
from sklearn.decomposition import PCA
from matplotlib import pyplot
# define training data
sentences = [['this', 'is', 'the', 'first', 'sentence', 'for', 'word2vec'],
             ['this', 'is', 'the', 'second', 'sentence'],
             ['yet', 'another', 'sentence'],
             ['one', 'more', 'sentence'],
             ['and', 'the', 'final', 'sentence']]
# train model
model = Word2Vec(sentences, min_count=1)
# fit a 2d PCA model to the vectors
X = model[model.wv.vocab]
pca = PCA(n_components=2)
result = pca.fit_transform(X)
# create a scatter plot of the projection
pyplot.scatter(result[:, 0], result[:, 1])
words = list(model.wv.vocab)
for i, word in enumerate(words):
    pyplot.annotate(word, xy=(result[i, 0], result[i, 1]))
pyplot.show()
```

Listing 12.14: Example demonstrating how to plot word vectors.

Running the example creates a scatter plot with the dots annotated with the words. It is hard to pull much meaning out of the graph given such a tiny corpus was used to fit the model.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

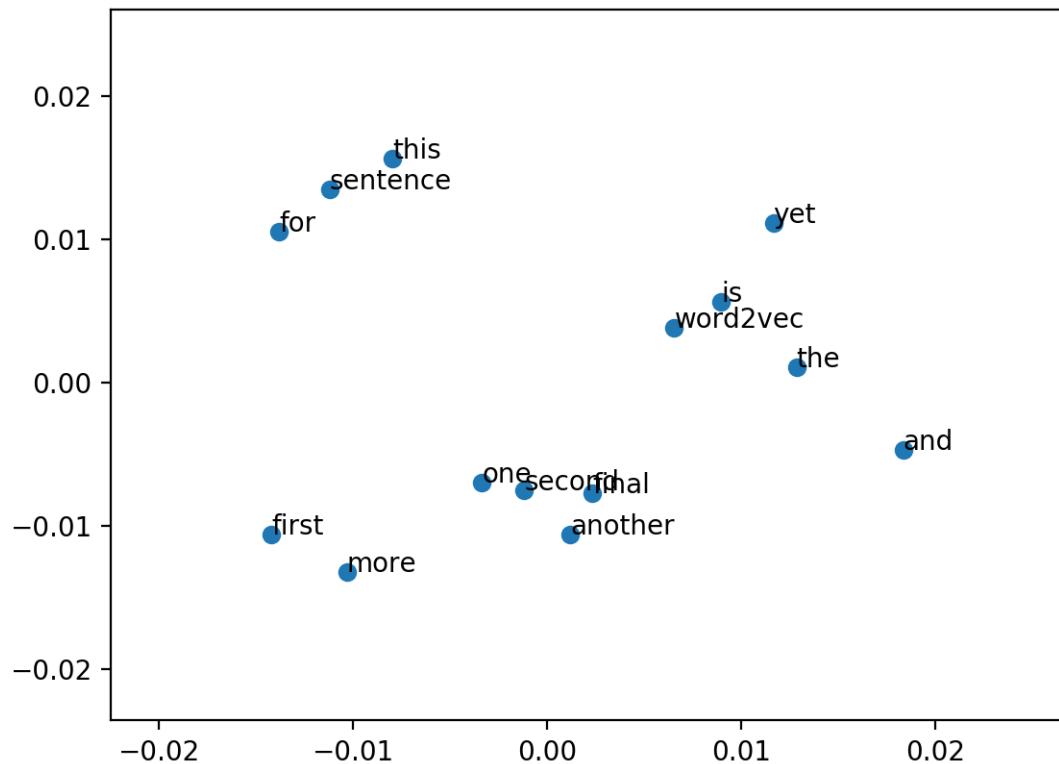


Figure 12.1: Scatter Plot of PCA Projection of Word2Vec Model

12.6 Load Google's Word2Vec Embedding

Training your own word vectors may be the best approach for a given NLP problem. But it can take a long time, a fast computer with a lot of RAM and disk space, and perhaps some expertise in finessing the input data and training algorithm. An alternative is to simply use an existing pre-trained word embedding. Along with the paper and code for Word2Vec, Google also published a pre-trained Word2Vec model on the Word2Vec Google Code Project.

A pre-trained model is nothing more than a file containing tokens and their associated word vectors. The pre-trained Google Word2Vec model was trained on Google news data (about 100 billion words); it contains 3 million words and phrases and was fit using 300-dimensional word vectors. It is a 1.53 Gigabyte file. You can download it from here:

- GoogleNews-vectors-negative300.bin.gz.
<https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTT1SS21pQmM/edit?usp=sharing>

Unzipped, the binary file (GoogleNews-vectors-negative300.bin) is 3.4 Gigabytes. The Gensim library provides tools to load this file. Specifically, you can call the `KeyedVectors.load_word2vec_format()` function to load this model into memory, for example:

```
from gensim.models import KeyedVectors
```

```
filename = 'GoogleNews-vectors-negative300.bin'
model = KeyedVectors.load_word2vec_format(filename, binary=True)
```

Listing 12.15: Example of loading the Google word vectors in Gensim.

Note, this example may require a workstation with 8 or more Gigabytes of RAM to execute. On my modern workstation, it takes about 43 seconds to load. Another interesting thing that you can do is do a little linear algebra arithmetic with words. For example, a popular example described in lectures and introduction papers is:

```
queen = (king - man) + woman
```

Listing 12.16: Example of arithmetic of word vectors.

That is the word queen is the closest word given the subtraction of the notion of man from king and adding the word woman. The *man-ness* in king is replaced with *woman-ness* to give us queen. A very cool concept. Gensim provides an interface for performing these types of operations in the `most_similar()` function on the trained or loaded model. For example:

```
result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print(result)
```

Listing 12.17: Example of arithmetic of word vectors in Gensim.

We can put all of this together as follows.

```
from gensim.models import KeyedVectors
# load the google word2vec model
filename = 'GoogleNews-vectors-negative300.bin'
model = KeyedVectors.load_word2vec_format(filename, binary=True)
# calculate: (king - man) + woman = ?
result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print(result)
```

Listing 12.18: Example demonstrating arithmetic with Google word vectors.

Running the example loads the Google pre-trained Word2Vec model and then calculates the $(\text{king} - \text{man}) + \text{woman} = ?$ operation on the word vectors for those words. The answer, as we would expect, is queen.

```
[('queen', 0.7118192315101624)]
```

Listing 12.19: Output of arithmetic with Google word vectors.

See some of the articles in the further reading section for more interesting arithmetic examples that you can explore.

12.7 Load Stanford's GloVe Embedding

Stanford researchers also have their own word embedding algorithm like Word2Vec called Global Vectors for Word Representation, or GloVe for short. I won't get into the details of the differences between Word2Vec and GloVe here, but generally, NLP practitioners seem to prefer GloVe at the moment based on results.

Like Word2Vec, the GloVe researchers also provide pre-trained word vectors, in this case, a great selection to choose from. You can download the GloVe pre-trained word vectors and load

them easily with Gensim. The first step is to convert the GloVe file format to the Word2Vec file format. The only difference is the addition of a small header line. This can be done by calling the `glove2word2vec()` function. For example (note, this example is just a demonstration with a mock input filename):

```
from gensim.scripts.glove2word2vec import glove2word2vec
glove_input_file = 'glove.txt'
word2vec_output_file = 'word2vec.txt'
glove2word2vec(glove_input_file, word2vec_output_file)
```

Listing 12.20: Example of converting a file from GloVe to Word2Vec format.

Once converted, the file can be loaded just like Word2Vec file above. Let's make this concrete with an example. You can download the smallest GloVe pre-trained model from the GloVe website. It is an 822 Megabyte zip file with 4 different models (50, 100, 200 and 300-dimensional vectors) trained on Wikipedia data with 6 billion tokens and a 400,000 word vocabulary. The direct download link is here:

- `glove.6B.zip`.
<http://nlp.stanford.edu/data/glove.6B.zip>

Working with the 100-dimensional version of the model, we can convert the file to Word2Vec format as follows:

```
from gensim.scripts.glove2word2vec import glove2word2vec
glove_input_file = 'glove.6B.100d.txt'
word2vec_output_file = 'glove.6B.100d.txt.word2vec'
glove2word2vec(glove_input_file, word2vec_output_file)
```

Listing 12.21: Example of converting a specific GloVe file to Word2Vec format.

You now have a copy of the GloVe model in Word2Vec format with the filename `glove.6B.100d.txt.word2vec`. Now we can load it and perform the same `(king - man) + woman = ?` test as in the previous section. The complete code listing is provided below. Note that the converted file is ASCII format, not binary, so we set `binary=False` when loading.

```
from gensim.models import KeyedVectors
# load the Stanford GloVe model
filename = 'glove.6B.100d.txt.word2vec'
model = KeyedVectors.load_word2vec_format(filename, binary=False)
# calculate: (king - man) + woman = ?
result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print(result)
```

Listing 12.22: Example of arithmetic with converted GloVe word vectors.

Pulling all of this together, the complete example is listed below.

```
from gensim.models import KeyedVectors
from gensim.scripts.glove2word2vec import glove2word2vec

# convert glove to word2vec format
glove_input_file = 'glove.6B.100d.txt'
word2vec_output_file = 'glove.6B.100d.txt.word2vec'
glove2word2vec(glove_input_file, word2vec_output_file)
```

```
# load the converted model
filename = 'glove.6B.100d.txt.word2vec'
model = KeyedVectors.load_word2vec_format(filename, binary=False)
# calculate: (king - man) + woman = ?
result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print(result)
```

Listing 12.23: Example demonstrating how to load and use GloVe word embeddings.

Running the example prints the same result of `queen`.

```
[('queen', 0.7698540687561035)]
```

Listing 12.24: Example output of arithmetic with converted GloVe word vectors.

12.8 Further Reading

This section provides more resources on the topic if you are looking go deeper.

12.8.1 Word Embeddings

- Word Embedding on Wikipedia.
<https://en.wikipedia.org/wiki/Word2vec>
- Word2Vec on Wikipedia.
<https://en.wikipedia.org/wiki/Word2vec>
- Google Word2Vec project.
<https://code.google.com/archive/p/word2vec/>
- Stanford GloVe project.
<https://nlp.stanford.edu/projects/glove/>

12.8.2 Gensim

- Gensim Python Library.
<https://radimrehurek.com/gensim/index.html>
- Gensim Installation Instructions.
<https://radimrehurek.com/gensim/install.html>
- `models.word2vec` Gensim API.
<https://radimrehurek.com/gensim/models/keyedvectors.html>
- `models.keyedvectors` Gensim API.
<https://radimrehurek.com/gensim/models/keyedvectors.html>
- `scripts.glove2word2vec` Gensim API.
<https://radimrehurek.com/gensim/scripts/glove2word2vec.html>

12.8.3 Articles

- *Messing Around With Word2Vec*, 2016.
<https://quomodocumque.wordpress.com/2016/01/15/messing-around-with-word2vec/>
- *Vector Space Models for the Digital Humanities*, 2015.
<http://bookworm.benschmidt.org/posts/2015-10-25-Word-Embeddings.html>
- *Gensim Word2Vec Tutorial*, 2014.
<https://rare-technologies.com/word2vec-tutorial/>

12.9 Summary

In this tutorial, you discovered how to develop and load word embedding layers in Python using Gensim. Specifically, you learned:

- How to train your own Word2Vec word embedding model on text data.
- How to visualize a trained word embedding model using Principal Component Analysis.
- How to load pre-trained Word2Vec and GloVe word embedding models from Google and Stanford.

12.9.1 Next

In the next chapter, you will discover how you can develop a neural network model with a learned word embedding.

Chapter 13

How to Learn and Load Word Embeddings in Keras

Word embeddings provide a dense representation of words and their relative meanings. They are an improvement over sparse representations used in simpler bag of word model representations. Word embeddings can be learned from text data and reused among projects. They can also be learned as part of fitting a neural network on text data. In this tutorial, you will discover how to use word embeddings for deep learning in Python with Keras. After completing this tutorial, you will know:

- About word embeddings and that Keras supports word embeddings via the `Embedding` layer.
- How to learn a word embedding while fitting a neural network.
- How to use a pre-trained word embedding in a neural network.

Let's get started.

13.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Word Embedding
2. Keras Embedding Layer
3. Example of Learning an Embedding
4. Example of Using Pre-Trained GloVe Embedding
5. Tips for Cleaning Text for Word Embedding

13.2 Word Embedding

A word embedding is a class of approaches for representing words and documents using a dense vector representation. It is an improvement over more the traditional bag-of-word model encoding schemes where large sparse vectors were used to represent each word or to score each word within a vector to represent an entire vocabulary. These representations were sparse because the vocabularies were vast and a given word or document would be represented by a large vector comprised mostly of zero values.

Instead, in an embedding, words are represented by dense vectors where a vector represents the projection of the word into a continuous vector space. The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used. The position of a word in the learned vector space is referred to as its embedding. Two popular examples of methods of learning word embeddings from text include:

- Word2Vec.
- GloVe.

In addition to these carefully designed methods, a word embedding can be learned as part of a deep learning model. This can be a slower approach, but tailors the model to a specific training dataset.

13.3 Keras Embedding Layer

Keras offers an `Embedding` layer that can be used for neural networks on text data. It requires that the input data be integer encoded, so that each word is represented by a unique integer. This data preparation step can be performed using the `Tokenizer` API also provided with Keras.

The `Embedding` layer is initialized with random weights and will learn an embedding for all of the words in the training dataset. It is a flexible layer that can be used in a variety of ways, such as:

- It can be used alone to learn a word embedding that can be saved and used in another model later.
- It can be used as part of a deep learning model where the embedding is learned along with the model itself.
- It can be used to load a pre-trained word embedding model, a type of transfer learning.

The `Embedding` layer is defined as the first hidden layer of a network. It must specify 3 arguments:

- `input_dim`: This is the size of the vocabulary in the text data. For example, if your data is integer encoded to values between 0-10, then the size of the vocabulary would be 11 words.

- **output_dim**: This is the size of the vector space in which words will be embedded. It defines the size of the output vectors from this layer for each word. For example, it could be 32 or 100 or even larger. Test different values for your problem.
- **input_length**: This is the length of input sequences, as you would define for any input layer of a Keras model. For example, if all of your input documents are comprised of 1000 words, this would be 1000.

For example, below we define an `Embedding` layer with a vocabulary of 200 (e.g. integer encoded words from 0 to 199, inclusive), a vector space of 32 dimensions in which words will be embedded, and input documents that have 50 words each.

```
e = Embedding(200, 32, input_length=50)
```

Listing 13.1: Example of creating a word embedding layer.

The `Embedding` layer has weights that are learned. If you save your model to file, this will include weights for the `Embedding` layer. The output of the `Embedding` layer is a 2D vector with one embedding for each word in the input sequence of words (input document). If you wish to connect a `Dense` layer directly to an `Embedding` layer, you must first flatten the 2D output matrix to a 1D vector using the `Flatten` layer. Now, let's see how we can use an `Embedding` layer in practice.

13.4 Example of Learning an Embedding

In this section, we will look at how we can learn a word embedding while fitting a neural network on a text classification problem. We will define a small problem where we have 10 text documents, each with a comment about a piece of work a student submitted. Each text document is classified as positive 1 or negative 0. This is a simple sentiment analysis problem. First, we will define the documents and their class labels.

```
# define documents
docs = ['Well done!',
        'Good work',
        'Great effort',
        'nice work',
        'Excellent!',
        'Weak',
        'Poor effort!',
        'not good',
        'poor work',
        'Could have done better.']
# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
```

Listing 13.2: Example of a small contrived classification problem.

Next, we can integer encode each document. This means that as input the `Embedding` layer will have sequences of integers. We could experiment with other more sophisticated bag of word model encoding like counts or TF-IDF. Keras provides the `one_hot()` function that creates a hash of each word as an efficient integer encoding. We will estimate the vocabulary size of 50, which is much larger than needed to reduce the probability of collisions from the hash function.

```
# integer encode the documents
vocab_size = 50
encoded_docs = [one_hot(d, vocab_size) for d in docs]
print(encoded_docs)
```

Listing 13.3: Integer encode the text.

The sequences have different lengths and Keras prefers inputs to be vectorized and all inputs to have the same length. We will pad all input sequences to have the length of 4. Again, we can do this with a built in Keras function, in this case the `pad_sequences()` function.

```
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
```

Listing 13.4: Pad the encoded text.

We are now ready to define our `Embedding` layer as part of our neural network model. The `Embedding` layer has a vocabulary of 50 and an input length of 4. We will choose a small embedding space of 8 dimensions. The model is a simple binary classification model. Importantly, the output from the `Embedding` layer will be 4 vectors of 8 dimensions each, one for each word. We flatten this to a one 32-element vector to pass on to the `Dense` output layer.

```
# define the model
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# summarize the model
model.summary()
```

Listing 13.5: Define a simple model with an `Embedding` input.

Finally, we can fit and evaluate the classification model.

```
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

Listing 13.6: Fit the defined model and print model accuracy.

The complete code listing is provided below.

```
from keras.preprocessing.text import one_hot
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
# define documents
docs = ['Well done!',
        'Good work',
```

```

'Great effort',
'nice work',
'Excellent!',
'Weak',
'Poor effort!',
'not good',
'poor work',
'Could have done better.']

# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
# integer encode the documents
vocab_size = 50
encoded_docs = [one_hot(d, vocab_size) for d in docs]
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
# define the model
model = Sequential()
model.add(Embedding(vocab_size, 8, input_length=max_length))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# summarize the model
model.summary()
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))

```

Listing 13.7: Example of fitting and evaluating a Keras model with an Embedding input layer.

Running the example first prints the integer encoded documents.

```
[[6, 16], [42, 24], [2, 17], [42, 24], [18], [17], [22, 17], [27, 42], [22, 24], [49, 46, 16, 34]]
```

Listing 13.8: Example output of the encoded documents.

Then the padded versions of each document are printed, making them all uniform length.

```
[[ 6 16  0  0]
 [42 24  0  0]
 [ 2 17  0  0]
 [42 24  0  0]
 [18  0  0  0]
 [17  0  0  0]
 [22 17  0  0]
 [27 42  0  0]
 [22 24  0  0]
 [49 46 16 34]]
```

Listing 13.9: Example output of the padded encoded documents.

After the network is defined, a summary of the layers is printed. We can see that as expected, the output of the `Embedding` layer is a 4 x 8 matrix and this is squashed to a 32-element vector by the `Flatten` layer.

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 4, 8)	400
flatten_1 (Flatten)	(None, 32)	0
dense_1 (Dense)	(None, 1)	33
Total params:	433	
Trainable params:	433	
Non-trainable params:	0	

Listing 13.10: Example output of the model summary.

Finally, the accuracy of the trained model is printed, showing that it learned the training dataset perfectly (which is not surprising).

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

Accuracy: 100.000000

Listing 13.11: Example output of the model accuracy.

You could save the learned weights from the `Embedding` layer to file for later use in other models. You could also use this model generally to classify other documents that have the same kind vocabulary seen in the test dataset. Next, let's look at loading a pre-trained word embedding in Keras.

13.5 Example of Using Pre-Trained GloVe Embedding

The Keras `Embedding` layer can also use a word embedding learned elsewhere. It is common in the field of Natural Language Processing to learn, save, and make freely available word embeddings. For example, the researchers behind GloVe method provide a suite of pre-trained word embeddings on their website released under a public domain license.

The smallest package of embeddings is 822 Megabytes, called `glove.6B.zip`. It was trained on a dataset of one billion tokens (words) with a vocabulary of 400 thousand words. There are a few different embedding vector sizes, including 50, 100, 200 and 300 dimensions. You can download this collection of embeddings and we can seed the Keras `Embedding` layer with weights from the pre-trained embedding for the words in your training dataset.

This example is inspired by an example in the Keras project: `pretrained_word_embeddings.py`. After downloading and unzipping, you will see a few files, one of which is `glove.6B.100d.txt`, which contains a 100-dimensional version of the embedding. If you peek inside the file, you will see a token (word) followed by the weights (100 numbers) on each line. For example, below are the first line of the embedding ASCII text file showing the embedding for *the*.

```
the -0.038194 -0.24487 0.72812 -0.39961 0.083172 0.043953 -0.39141 0.3344 -0.57545 0.087459
0.28787 -0.06731 0.30906 -0.26384 -0.13231 -0.20757 0.33395 -0.33848 -0.31743 -0.48336
0.1464 -0.37304 0.34577 0.052041 0.44946 -0.46971 0.02628 -0.54155 -0.15518 -0.14107
-0.039722 0.28277 0.14393 0.23464 -0.31021 0.086173 0.20397 0.52624 0.17164 -0.082378
-0.71787 -0.41531 0.20335 -0.12763 0.41367 0.55187 0.57908 -0.33477 -0.36559 -0.54857
-0.062892 0.26584 0.30205 0.99775 -0.80481 -3.0243 0.01254 -0.36942 2.2167 0.72201
-0.24978 0.92136 0.034514 0.46745 1.1079 -0.19358 -0.074575 0.23353 -0.052062 -0.22044
0.057162 -0.15806 -0.30798 -0.41625 0.37972 0.15006 -0.53212 -0.2055 -1.2526 0.071624
0.70565 0.49744 -0.42063 0.26148 -1.538 -0.30223 -0.073438 -0.28312 0.37104 -0.25217
0.016215 -0.017099 -0.38984 0.87424 -0.72569 -0.51058 -0.52028 -0.1459 0.8278 0.27062
```

Listing 13.12: Example GloVe word vector for the word 'the'.

As in the previous section, the first step is to define the examples, encode them as integers, then pad the sequences to be the same length. In this case, we need to be able to map words to integers as well as integers to words. Keras provides a `Tokenizer` class that can be fit on the training data, can convert text to sequences consistently by calling the `texts_to_sequences()` method on the `Tokenizer` class, and provides access to the dictionary mapping of words to integers in a `word_index` attribute.

```
# define documents
docs = ['Well done!',
        'Good work',
        'Great effort',
        'nice work',
        'Excellent!',
        'Weak',
        'Poor effort!',
        'not good',
        'poor work',
        'Could have done better.']

# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
# prepare tokenizer
t = Tokenizer()
t.fit_on_texts(docs)
vocab_size = len(t.word_index) + 1
# integer encode the documents
encoded_docs = t.texts_to_sequences(docs)
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
```

Listing 13.13: Define encode and pad sample documents.

Next, we need to load the entire GloVe word embedding file into memory as a dictionary of word to embedding array.

```
# load the whole embedding into memory
embeddings_index = dict()
f = open('glove.6B.100d.txt')
for line in f:
    values = line.split()
    word = values[0]
```

```

coefs = asarray(values[1:], dtype='float32')
embeddings_index[word] = coefs
f.close()
print('Loaded %s word vectors.' % len(embeddings_index))

```

Listing 13.14: Load the GloVe word embedding into memory.

This is pretty slow. It might be better to filter the embedding for the unique words in your training data. Next, we need to create a matrix of one embedding for each word in the training dataset. We can do that by enumerating all unique words in the `Tokenizer.word_index` and locating the embedding weight vector from the loaded GloVe embedding. The result is a matrix of weights only for words we will see during training.

```

# create a weight matrix for words in training docs
embedding_matrix = zeros((vocab_size, 100))
for word, i in t.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector

```

Listing 13.15: Covert the word embedding into a weight matrix.

Now we can define our model, fit, and evaluate it as before. The key difference is that the `Embedding` layer can be seeded with the GloVe word embedding weights. We chose the 100-dimensional version, therefore the `Embedding` layer must be defined with `output_dim` set to 100. Finally, we do not want to update the learned word weights in this model, therefore we will set the `trainable` attribute for the model to be `False`.

```
e = Embedding(vocab_size, 100, weights=[embedding_matrix], input_length=4, trainable=False)
```

Listing 13.16: Create an `Embedding` layer with the pre-loaded weights.

The complete worked example is listed below.

```

from numpy import asarray
from numpy import zeros
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Embedding
# define documents
docs = ['Well done!',
        'Good work',
        'Great effort',
        'nice work',
        'Excellent!!',
        'Weak',
        'Poor effort!!',
        'not good',
        'poor work',
        'Could have done better.']
# define class labels
labels = [1,1,1,1,1,0,0,0,0,0]
# prepare tokenizer
t = Tokenizer()

```

```
t.fit_on_texts(docs)
vocab_size = len(t.word_index) + 1
# integer encode the documents
encoded_docs = t.texts_to_sequences(docs)
print(encoded_docs)
# pad documents to a max length of 4 words
max_length = 4
padded_docs = pad_sequences(encoded_docs, maxlen=max_length, padding='post')
print(padded_docs)
# load the whole embedding into memory
embeddings_index = dict()
f = open('glove.6B.100d.txt', mode='rt', encoding='utf-8')
for line in f:
    values = line.split()
    word = values[0]
    coefs = asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('Loaded %s word vectors.' % len(embeddings_index))
# create a weight matrix for words in training docs
embedding_matrix = zeros((vocab_size, 100))
for word, i in t.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
# define model
model = Sequential()
e = Embedding(vocab_size, 100, weights=[embedding_matrix], input_length=4, trainable=False)
model.add(e)
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
# compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# summarize the model
model.summary()
# fit the model
model.fit(padded_docs, labels, epochs=50, verbose=0)
# evaluate the model
loss, accuracy = model.evaluate(padded_docs, labels, verbose=0)
print('Accuracy: %f' % (accuracy*100))
```

Listing 13.17: Example loading pre-trained GloVe weights into an `Embedding` input layer.

Running the example may take a bit longer, but then demonstrates that it is just as capable of fitting this simple problem.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
...
Accuracy: 100.000000
```

Listing 13.18: Example output of loading pre-trained GloVe weights into an `Embedding` input layer.

In practice, I would encourage you to experiment with learning a word embedding using a pre-trained embedding that is fixed and trying to perform learning on top of a pre-trained embedding. See what works best for your specific problem.

13.6 Tips for Cleaning Text for Word Embedding

Recently, the field of natural language processing has been moving away from bag-of-word models and word encoding toward word embeddings. The benefit of word embeddings is that they encode each word into a dense vector that captures something about its relative meaning within the training text. This means that variations of words like case, spelling, punctuation, and so on will automatically be learned to be similar in the embedding space. In turn, this can mean that the amount of cleaning required from your text may be less and perhaps quite different to classical text cleaning. For example, it may no-longer make sense to stem words or remove punctuation for contractions.

Tomas Mikolov is one of the developers of Word2Vec, a popular word embedding method. He suggests only very minimal text cleaning is required when learning a word embedding model. Below is his response when pressed with the question about how to best prepare text data for Word2Vec.

There is no universal answer. It all depends on what you plan to use the vectors for. In my experience, it is usually good to disconnect (or remove) punctuation from words, and sometimes also convert all characters to lowercase. One can also replace all numbers (possibly greater than some constant) with some single token such as .

All these pre-processing steps aim to reduce the vocabulary size without removing any important content (which in some cases may not be true when you lowercase certain words, ie. ‘Bush’ is different than ‘bush’, while ‘Another’ has usually the same sense as ‘another’). The smaller the vocabulary is, the lower is the memory complexity, and the more robustly are the parameters for the words estimated. You also have to pre-process the test data in the same way.

[...]

In short, you will understand all this much better if you will run experiments.

— Tomas Mikolov, *word2vec-toolkit: google groups thread.*, 2015.
<https://goo.gl/KtDGst>

13.7 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Word Embedding on Wikipedia.
https://en.wikipedia.org/wiki/Word_embedding
- Keras Embedding Layer API.
<https://keras.io/layers/embeddings/#embedding>

- Using pre-trained word embeddings in a Keras model, 2016.
<https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html>
- Example of using a pre-trained GloVe Embedding in Keras.
https://github.com/fchollet/keras/blob/master/examples/pretrained_word_embeddings.py
- GloVe Embedding.
<https://nlp.stanford.edu/projects/glove/>
- An overview of word embeddings and their connection to distributional semantic models, 2016.
<http://blog.aylien.com/overview-word-embeddings-history-word2vec-cbow-glove/>
- Deep Learning, NLP, and Representations, 2014.
<http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/>

13.8 Summary

In this tutorial, you discovered how to use word embeddings for deep learning in Python with Keras. Specifically, you learned:

- About word embeddings and that Keras supports word embeddings via the `Embedding` layer.
- How to learn a word embedding while fitting a neural network.
- How to use a pre-trained word embedding in a neural network.

13.8.1 Next

This is the end of the part on word embeddings. In the next part you will discover neural text classification.

Part VI

Text Classification

Chapter 14

Neural Models for Document Classification

Text classification describes a general class of problems such as predicting the sentiment of tweets and movie reviews, as well as classifying email as spam or not. Deep learning methods are proving very good at text classification, achieving state-of-the-art results on a suite of standard academic benchmark problems. In this chapter, you will discover some best practices to consider when developing deep learning models for text classification. After reading this chapter, you will know:

- The general combination of deep learning methods to consider when starting your text classification problems.
- The first architecture to try with specific advice on how to configure hyperparameters.
- That deeper networks may be the future of the field in terms of flexibility and capability.

Let's get started.

14.1 Overview

This tutorial is divided into the following parts:

1. Word Embeddings + CNN = Text Classification
2. Use a Single Layer CNN Architecture
3. Dial in CNN Hyperparameters
4. Consider Character-Level CNNs
5. Consider Deeper CNNs for Classification

14.2 Word Embeddings + CNN = Text Classification

The modus operandi for text classification involves the use of a word embedding for representing words and a Convolutional Neural Network (CNN) for learning how to discriminate documents on classification problems. Yoav Goldberg, in his primer on deep learning for natural language processing, comments that neural networks in general offer better performance than classical linear classifiers, especially when used with pre-trained word embeddings.

The non-linearity of the network, as well as the ability to easily integrate pre-trained word embeddings, often lead to superior classification accuracy.

— *A Primer on Neural Network Models for Natural Language Processing*, 2015.

He also comments that convolutional neural networks are effective at document classification, namely because they are able to pick out salient features (e.g. tokens or sequences of tokens) in a way that is invariant to their position within the input sequences.

Networks with convolutional and pooling layers are useful for classification tasks in which we expect to find strong local clues regarding class membership, but these clues can appear in different places in the input. [...] We would like to learn that certain sequences of words are good indicators of the topic, and do not necessarily care where they appear in the document. Convolutional and pooling layers allow the model to learn to find such local indicators, regardless of their position.

— *A Primer on Neural Network Models for Natural Language Processing*, 2015.

The architecture is therefore comprised of three key pieces:

- **Word Embedding:** A distributed representation of words where different words that have a similar meaning (based on their usage) also have a similar representation.
- **Convolutional Model:** A feature extraction model that learns to extract salient features from documents represented using a word embedding.
- **Fully Connected Model:** The interpretation of extracted features in terms of a predictive output.

Yoav Goldberg highlights the CNNs role as a feature extractor model in his book:

... the CNN is in essence a feature-extracting architecture. It does not constitute a standalone, useful network on its own, but rather is meant to be integrated into a larger network, and to be trained to work in tandem with it in order to produce an end result. The CNNs layer's responsibility is to extract meaningful sub-structures that are useful for the overall prediction task at hand.

— Page 152, *Neural Network Methods for Natural Language Processing*, 2017.

The tying together of these three elements is demonstrated in perhaps one of the most widely cited examples of the combination, described in the next section.

14.3 Use a Single Layer CNN Architecture

You can get good results for document classification with a single layer CNN, perhaps with differently sized kernels across the filters to allow grouping of word representations at different scales. Yoon Kim in his study of the use of pre-trained word vectors for classification tasks with Convolutional Neural Networks found that using pre-trained static word vectors does very well. He suggests that pre-trained word embeddings that were trained on very large text corpora, such as the freely available Word2Vec vectors trained on 100 billion tokens from Google news may offer good universal features for use in natural language processing.

Despite little tuning of hyperparameters, a simple CNN with one layer of convolution performs remarkably well. Our results add to the well-established evidence that unsupervised pre-training of word vectors is an important ingredient in deep learning for NLP

— *Convolutional Neural Networks for Sentence Classification*, 2014.

He also discovered that further task-specific tuning of the word vectors offer a small additional improvement in performance. Kim describes the general approach of using CNN for natural language processing. Sentences are mapped to embedding vectors and are available as a matrix input to the model. Convolutions are performed across the input word-wise using differently sized kernels, such as 2 or 3 words at a time. The resulting feature maps are then processed using a max pooling layer to condense or summarize the extracted features.

The architecture is based on the approach used by Ronan Collobert, et al. in their paper *Natural Language Processing (almost) from Scratch*, 2011. In it, they develop a single end-to-end neural network model with convolutional and pooling layers for use across a range of fundamental natural language processing problems. Kim provides a diagram that helps to see the sampling of the filters using differently sized kernels as different colors (red and yellow).

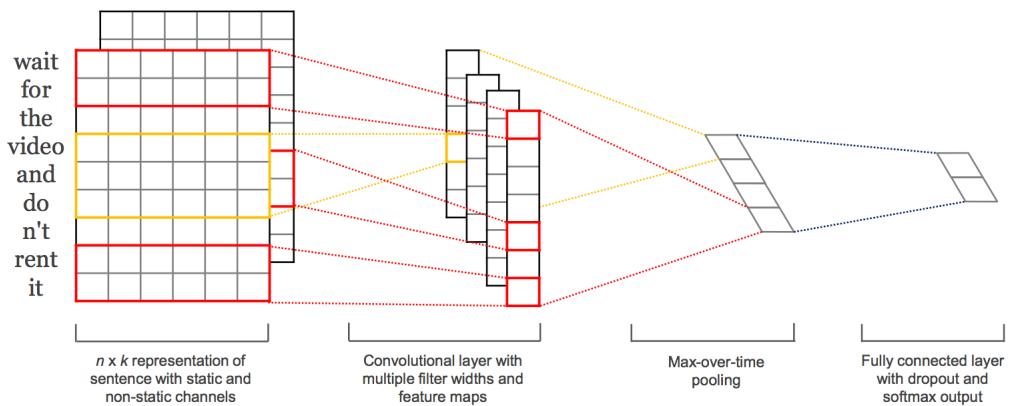


Figure 14.1: An example of a CNN Filter and Polling Architecture for Natural Language Processing. Taken from *Convolutional Neural Networks for Sentence Classification*.

Usefully, he reports his chosen model configuration, discovered via grid search and used across a suite of 7 text classification tasks, summarized as follows:

- Transfer function: rectified linear.
- Kernel sizes: 2, 4, 5.
- Number of filters: 100.
- Dropout rate: 0.5.
- Weight regularization (L2): 3.
- Batch Size: 50.
- Update Rule: Adadelta.

These configurations could be used to inspire a starting point for your own experiments.

14.4 Dial in CNN Hyperparameters

Some hyperparameters matter more than others when tuning a convolutional neural network on your document classification problem. Ye Zhang and Byron Wallace performed a sensitivity analysis into the hyperparameters needed to configure a single layer convolutional neural network for document classification. The study is motivated by their claim that the models are sensitive to their configuration.

Unfortunately, a downside to CNN-based models - even simple ones - is that they require practitioners to specify the exact model architecture to be used and to set the accompanying hyperparameters. To the uninitiated, making such decisions can seem like something of a black art because there are many free parameters in the model.

— *A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification*, 2015.

Their aim was to provide general configurations that can be used for configuring CNNs on new text classification tasks. They provide a nice depiction of the model architecture and the decision points for configuring the model, reproduced below.

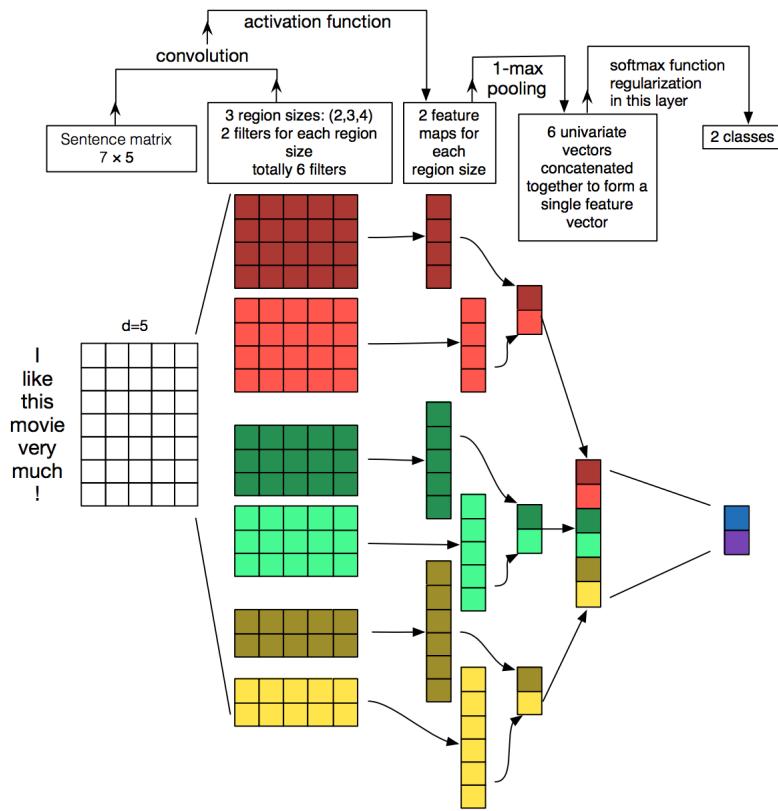


Figure 14.2: Convolutional Neural Network Architecture for Sentence Classification. Taken from *A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification*.

The study makes a number of useful findings that could be used as a starting point for configuring shallow CNN models for text classification. The general findings were as follows:

- The choice of pre-trained Word2Vec and GloVe embeddings differ from problem to problem, and both performed better than using one hot encoded word vectors.
- The size of the kernel is important and should be tuned for each problem.
- The number of feature maps is also important and should be tuned.
- The 1-max pooling generally outperformed other types of pooling.
- Dropout has little effect on the model performance.

They go on to provide more specific heuristics, as follows:

- Use Word2Vec or GloVe word embeddings as a starting point and tune them while fitting the model.
- Grid search across different kernel sizes to find the optimal configuration for your problem, in the range 1-10.

- Search the number of filters from 100-600 and explore a dropout of 0.0-0.5 as part of the same search.
- Explore using tanh, relu, and linear activation functions.

The key caveat is that the findings are based on empirical results on binary text classification problems using single sentences as input.

14.5 Consider Character-Level CNNs

Text documents can be modeled at the character level using convolutional neural networks that are capable of learning the relevant hierarchical structure of words, sentences, paragraphs, and more. Xiang Zhang, et al. use a character-based representation of text as input for a convolutional neural network. The promise of the approach is that all of the labor-intensive effort required to clean and prepare text could be overcome if a CNN can learn to abstract the salient details.

... deep ConvNets do not require the knowledge of words, in addition to the conclusion from previous research that ConvNets do not require the knowledge about the syntactic or semantic structure of a language. This simplification of engineering could be crucial for a single system that can work for different languages, since characters always constitute a necessary construct regardless of whether segmentation into words is possible. Working on only characters also has the advantage that abnormal character combinations such as misspellings and emoticons may be naturally learnt.

— *Character-level Convolutional Networks for Text Classification*, 2015.

The model reads in one hot encoded characters in a fixed-sized alphabet. Encoded characters are read in blocks or sequences of 1,024 characters. A stack of 6 convolutional layers with pooling follows, with 3 fully connected layers at the output end of the network in order to make a prediction.

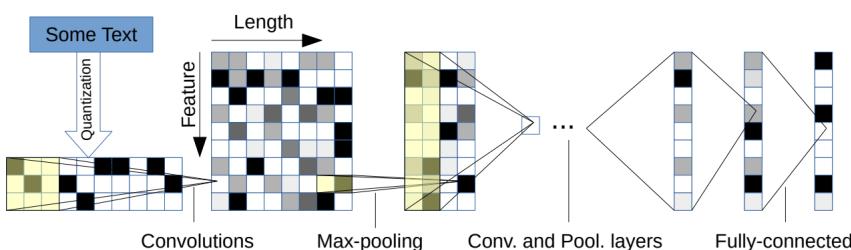


Figure 14.3: Character-based Convolutional Neural Network for Text Classification. Taken from *Character-level Convolutional Networks for Text Classification*.

The model achieves some success, performing better on problems that offer a larger corpus of text.

... analysis shows that character-level ConvNet is an effective method. [...] how well our model performs in comparisons depends on many factors, such as dataset size, whether the texts are curated and choice of alphabet.

— *Character-level Convolutional Networks for Text Classification*, 2015.

Results using an extended version of this approach were pushed to the state-of-the-art in a follow-up paper covered in the next section.

14.6 Consider Deeper CNNs for Classification

Better performance can be achieved with very deep convolutional neural networks, although standard and reusable architectures have not been adopted for classification tasks, yet. Alexis Conneau, et al. comment on the relatively shallow networks used for natural language processing and the success of much deeper networks used for computer vision applications. For example, Kim (above) restricted the model to a single convolutional layer.

Other architectures used for natural language reviewed in the paper are limited to 5 and 6 layers. These are contrasted with successful architectures used in computer vision with 19 or even up to 152 layers. They suggest and demonstrate that there are benefits for hierarchical feature learning with very deep convolutional neural network model, called VDCNN.

... we propose to use deep architectures of many convolutional layers to approach this goal, using up to 29 layers. The design of our architecture is inspired by recent progress in computer vision [...] The proposed deep convolutional network shows significantly better results than previous ConvNets approach.

— *Very Deep Convolutional Networks for Text Classification*, 2016.

Key to their approach is an embedding of individual characters, rather than a word embedding.

We present a new architecture (VDCNN) for text processing which operates directly at the character level and uses only small convolutions and pooling operations.

— *Very Deep Convolutional Networks for Text Classification*, 2016.

Results on a suite of 8 large text classification tasks show better performance than more shallow networks. Specifically, state-of-the-art results on all but two of the datasets tested, at the time of writing. Generally, they make some key findings from exploring the deeper architectural approach:

- The very deep architecture worked well on small and large datasets.
- Deeper networks decrease classification error.
- Max-pooling achieves better results than other, more sophisticated types of pooling.
- Generally going deeper degrades accuracy; the shortcut connections used in the architecture are important.

... this is the first time that the “benefit of depths” was shown for convolutional neural networks in NLP.

— *Very Deep Convolutional Networks for Text Classification*, 2016.

14.7 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- *A Primer on Neural Network Models for Natural Language Processing*, 2015.
<https://arxiv.org/abs/1510.00726>
- *Convolutional Neural Networks for Sentence Classification*, 2014.
<https://arxiv.org/abs/1103.0398>
- *Natural Language Processing (almost) from Scratch*, 2011.
<https://arxiv.org/abs/1103.0398>
- *Very Deep Convolutional Networks for Text Classification*, 2016.
<https://arxiv.org/abs/1606.01781>
- *Character-level Convolutional Networks for Text Classification*, 2015.
<https://arxiv.org/abs/1509.01626>
- *A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification*, 2015.
<https://arxiv.org/abs/1510.03820>

14.8 Summary

In this chapter, you discovered some best practices for developing deep learning models for document classification. Specifically, you learned:

- That a key approach is to use word embeddings and convolutional neural networks for text classification.
- That a single layer model can do well on moderate-sized problems, and ideas on how to configure it.
- That deeper models that operate directly on text may be the future of natural language processing.

14.8.1 Next

In the next chapter, you will discover how you can develop a neural text classification model with word embeddings and a convolutional neural network.

Chapter 15

Project: Develop an Embedding + CNN Model for Sentiment Analysis

Word embeddings are a technique for representing text where different words with similar meaning have a similar real-valued vector representation. They are a key breakthrough that has led to great performance of neural network models on a suite of challenging natural language processing problems. In this tutorial, you will discover how to develop word embedding models with convolutional neural networks to classify movie reviews. After completing this tutorial, you will know:

- How to prepare movie review text data for classification with deep learning methods.
- How to develop a neural classification model with word embedding and convolutional layers.
- How to evaluate the developed a neural classification model.

Let's get started.

15.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Movie Review Dataset
2. Data Preparation
3. Train CNN With Embedding Layer
4. Evaluate Model

15.2 Movie Review Dataset

In this tutorial, we will use the Movie Review Dataset. This dataset designed for sentiment analysis was described previously in Chapter 9. You can download the dataset from here:

- Movie Review Polarity Dataset (`review_polarity.tar.gz`, 3MB).
http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

After unzipping the file, you will have a directory called `txt_sentoken` with two sub-directories containing the text `neg` and `pos` for negative and positive reviews. Reviews are stored one per file with a naming convention `cv000` to `cv999` for each of `neg` and `pos`.

15.3 Data Preparation

Note: The preparation of the movie review dataset was first described in Chapter 9. In this section, we will look at 3 things:

1. Separation of data into training and test sets.
2. Loading and cleaning the data to remove punctuation and numbers.
3. Defining a vocabulary of preferred words.

15.3.1 Split into Train and Test Sets

We are pretending that we are developing a system that can predict the sentiment of a textual movie review as either positive or negative. This means that after the model is developed, we will need to make predictions on new textual reviews. This will require all of the same data preparation to be performed on those new reviews as is performed on the training data for the model. We will ensure that this constraint is built into the evaluation of our models by splitting the training and test datasets prior to any data preparation. This means that any knowledge in the data in the test set that could help us better prepare the data (e.g. the words used) are unavailable in the preparation of data used for training the model.

That being said, we will use the last 100 positive reviews and the last 100 negative reviews as a test set (100 reviews) and the remaining 1,800 reviews as the training dataset. This is a 90% train, 10% split of the data. The split can be imposed easily by using the filenames of the reviews where reviews named 000 to 899 are for training data and reviews named 900 onwards are for test.

15.3.2 Loading and Cleaning Reviews

The text data is already pretty clean; not much preparation is required. Without getting bogged down too much in the details, we will prepare the data using the following way:

- Split tokens on white space.
- Remove all punctuation from words.
- Remove all words that are not purely comprised of alphabetical characters.
- Remove all words that are known stop words.
- Remove all words that have a length ≤ 1 character.

We can put all of these steps into a function called `clean_doc()` that takes as an argument the raw text loaded from a file and returns a list of cleaned tokens. We can also define a function `load_doc()` that loads a document from file ready for use with the `clean_doc()` function. An example of cleaning the first positive review is listed below.

```
from nltk.corpus import stopwords
import string
import re

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[' + re.escape(string.punctuation) + ']')
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load the document
filename = 'txt_sentoken/pos/cv000_29590.txt'
text = load_doc(filename)
tokens = clean_doc(text)
print(tokens)
```

Listing 15.1: Example of cleaning a movie review.

Running the example prints a long list of clean tokens. There are many more cleaning steps we may want to explore and I leave them as further exercises.

```
...
'creepy', 'place', 'even', 'acting', 'hell', 'solid', 'dreamy', 'depp', 'turning',
'typically', 'strong', 'performance', 'deftly', 'handling', 'british', 'accent',
'ians', 'holm', 'joe', 'oulds', 'secret', 'richardson', 'dalmatians', 'log', 'great',
'supporting', 'roles', 'big', 'surprise', 'graham', 'cringed', 'first', 'time',
'opened', 'mouth', 'imagining', 'attempt', 'irish', 'accent', 'actually', 'wasnt',
'half', 'bad', 'film', 'however', 'good', 'strong', 'violencegore', 'sexuality',
'language', 'drug', 'content']
```

Listing 15.2: Example output of cleaning a movie review.

15.3.3 Define a Vocabulary

It is important to define a vocabulary of known words when using a text model. The more words, the larger the representation of documents, therefore it is important to constrain the words to only those believed to be predictive. This is difficult to know beforehand and often it is important to test different hypotheses about how to construct a useful vocabulary. We have already seen how we can remove punctuation and numbers from the vocabulary in the previous section. We can repeat this for all documents and build a set of all known words.

We can develop a vocabulary as a `Counter`, which is a dictionary mapping of words and their count that allows us to easily update and query. Each document can be added to the counter (a new function called `add_doc_to_vocab()`) and we can step over all of the reviews in the negative directory and then the positive directory (a new function called `process_docs()`). The complete example is listed below.

```
import string
import re
from os import listdir
from collections import Counter
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
    # load doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # update counts
```

```

vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # add doc to vocab
        add_doc_to_vocab(path, vocab)

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/pos', vocab)
process_docs('txt_sentoken/neg', vocab)
# print the size of the vocab
print(len(vocab))
# print the top words in the vocab
print(vocab.most_common(50))

```

Listing 15.3: Example of selecting a vocabulary for the dataset.

Running the example shows that we have a vocabulary of 44,276 words. We also can see a sample of the top 50 most used words in the movie reviews. Note that this vocabulary was constructed based on only those reviews in the training dataset.

```

44276
[('film', 7983), ('one', 4946), ('movie', 4826), ('like', 3201), ('even', 2262), ('good', 2080), ('time', 2041), ('story', 1907), ('films', 1873), ('would', 1844), ('much', 1824), ('also', 1757), ('characters', 1735), ('get', 1724), ('character', 1703), ('two', 1643), ('first', 1588), ('see', 1557), ('way', 1515), ('well', 1511), ('make', 1418), ('really', 1407), ('little', 1351), ('life', 1334), ('plot', 1288), ('people', 1269), ('could', 1248), ('bad', 1248), ('scene', 1241), ('movies', 1238), ('never', 1201), ('best', 1179), ('new', 1140), ('scenes', 1135), ('man', 1131), ('many', 1130), ('doesnt', 1118), ('know', 1092), ('dont', 1086), ('hes', 1024), ('great', 1014), ('another', 992), ('action', 985), ('love', 977), ('us', 967), ('go', 952), ('director', 948), ('end', 946), ('something', 945), ('still', 936)]

```

Listing 15.4: Example output of selecting a vocabulary for the dataset.

We can step through the vocabulary and remove all words that have a low occurrence, such as only being used once or twice in all reviews. For example, the following snippet will retrieve only the tokens that appear 2 or more times in all reviews.

```

# keep tokens with a min occurrence
min_occurane = 2
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))

```

Listing 15.5: Example of filtering the vocabulary by occurrence.

Finally, the vocabulary can be saved to a new file called `vocab.txt` that we can later load and use to filter movie reviews prior to encoding them for modeling. We define a new function

called `save_list()` that saves the vocabulary to file, with one word per line. For example:

```
# save list to file
def save_list(lines, filename):
    # convert lines to a single blob of text
    data = '\n'.join(lines)
    # open file
    file = open(filename, 'w')
    # write text
    file.write(data)
    # close file
    file.close()

# save tokens to a vocabulary file
save_list(tokens, 'vocab.txt')
```

Listing 15.6: Example of saving the filtered vocabulary.

Pulling all of this together, the complete example is listed below.

```
import string
import re
from os import listdir
from collections import Counter
from nltk.corpus import stopwords

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[' + re.escape(string.punctuation) + ']')
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load doc and add to vocab
def add_doc_to_vocab(filename, vocab):
    # load doc
    doc = load_doc(filename)
```

```

# clean doc
tokens = clean_doc(doc)
# update counts
vocab.update(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # add doc to vocab
        add_doc_to_vocab(path, vocab)

# save list to file
def save_list(lines, filename):
    # convert lines to a single blob of text
    data = '\n'.join(lines)
    # open file
    file = open(filename, 'w')
    # write text
    file.write(data)
    # close file
    file.close()

# define vocab
vocab = Counter()
# add all docs to vocab
process_docs('txt_sentoken/pos', vocab)
process_docs('txt_sentoken/neg', vocab)
# print the size of the vocab
print(len(vocab))
# keep tokens with a min occurrence
min_occurane = 2
tokens = [k for k,c in vocab.items() if c >= min_occurane]
print(len(tokens))
# save tokens to a vocabulary file
save_list(tokens, 'vocab.txt')

```

Listing 15.7: Example of filtering the vocabulary for the dataset.

Running the above example with this addition shows that the vocabulary size drops by a little more than half its size, from 44,276 to 25,767 words.

25767

Listing 15.8: Example output of filtering the vocabulary by min occurrence.

Running the min occurrence filter on the vocabulary and saving it to file, you should now have a new file called `vocab.txt` with only the words we are interested in. The order of words in your file will differ, but should look something like the following:

aberdeen
dupe

```
burt
libido
hamlet
arlene
available
corners
web
columbia
...
```

Listing 15.9: Sample of the vocabulary file `vocab.txt`.

We are now ready to look at extracting features from the reviews ready for modeling.

15.4 Train CNN With Embedding Layer

In this section, we will learn a word embedding while training a convolutional neural network on the classification problem. A word embedding is a way of representing text where each word in the vocabulary is represented by a real valued vector in a high-dimensional space. The vectors are learned in such a way that words that have similar meanings will have similar representation in the vector space (close in the vector space). This is a more expressive representation for text than more classical methods like bag-of-words, where relationships between words or tokens are ignored, or forced in bigram and trigram approaches.

The real valued vector representation for words can be learned while training the neural network. We can do this in the Keras deep learning library using the `Embedding` layer. The first step is to load the vocabulary. We will use it to filter out words from movie reviews that we are not interested in. If you have worked through the previous section, you should have a local file called `vocab.txt` with one word per line. We can load that file and build a vocabulary as a set for checking the validity of tokens.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
```

Listing 15.10: Load vocabulary.

Next, we need to load all of the training data movie reviews. For that we can adapt the `process_docs()` from the previous section to load the documents, clean them, and return them as a list of strings, with one document per string. We want each document to be a string for easy encoding as a sequence of integers later. Cleaning the document involves splitting each review based on white space, removing punctuation, and then filtering out all tokens not in the vocabulary. The updated `clean_doc()` function is listed below.

```
# turn a doc into clean tokens
def clean_doc(doc, vocab):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub(' ', w) for w in tokens]
    # filter out tokens not in vocab
    tokens = [w for w in tokens if w in vocab]
    tokens = ' '.join(tokens)
    return tokens
```

Listing 15.11: Function to load and filter a loaded review.

The updated `process_docs()` can then call the `clean_doc()` for each document in a given directory.

```
# load all docs in a directory
def process_docs(directory, vocab, is_train):
    documents = list()
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if is_train and filename.startswith('cv9'):
            continue
        if not is_train and not filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # load the doc
        doc = load_doc(path)
        # clean doc
        tokens = clean_doc(doc, vocab)
        # add to list
        documents.append(tokens)
    return documents
```

Listing 15.12: Example to clean all movie reviews.

We can call the `process_docs` function for both the `neg` and `pos` directories and combine the reviews into a single train or test dataset. We also can define the class labels for the dataset. The `load_clean_dataset()` function below will load all reviews and prepare class labels for the training or test dataset.

```
# load and clean a dataset
def load_clean_dataset(vocab, is_train):
    # load documents
    neg = process_docs('txt_sentoken/neg', vocab, is_train)
    pos = process_docs('txt_sentoken/pos', vocab, is_train)
    docs = neg + pos
    # prepare labels
    labels = array([0 for _ in range(len(neg))] + [1 for _ in range(len(pos))])
    return docs, labels
```

Listing 15.13: Function to load and clean all train or test movie reviews.

The next step is to encode each document as a sequence of integers. The Keras `Embedding` layer requires integer inputs where each integer maps to a single token that has a specific real-valued vector representation within the embedding. These vectors are random at the beginning of training, but during training become meaningful to the network. We can encode the training documents as sequences of integers using the `Tokenizer` class in the Keras API. First, we must construct an instance of the class then train it on all documents in the training dataset. In this case, it develops a vocabulary of all tokens in the training dataset and develops a consistent mapping from words in the vocabulary to unique integers. We could just as easily develop this mapping ourselves using our vocabulary file. The `create_tokenizer()` function below will prepare a `Tokenizer` from the training data.

```
# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer
```

Listing 15.14: Function to create a `Tokenizer` from training.

Now that the mapping of words to integers has been prepared, we can use it to encode the reviews in the training dataset. We can do that by calling the `texts_to_sequences()` function on the `Tokenizer`. We also need to ensure that all documents have the same length. This is a requirement of Keras for efficient computation. We could truncate reviews to the smallest size or zero-pad (pad with the value 0) reviews to the maximum length, or some hybrid. In this case, we will pad all reviews to the length of the longest review in the training dataset. First, we can find the longest review using the `max()` function on the training dataset and take its length. We can then call the Keras function `pad_sequences()` to pad the sequences to the maximum length by adding 0 values on the end.

```
max_length = max([len(s.split()) for s in train_docs])
print('Maximum length: %d' % max_length)
```

Listing 15.15: Calculate the maximum movie review length.

We can then use the maximum length as a parameter to a function to integer encode and pad the sequences.

```
# integer encode and pad documents
def encode_docs(tokenizer, max_length, docs):
    # integer encode
    encoded = tokenizer.texts_to_sequences(docs)
    # pad sequences
    padded = pad_sequences(encoded, maxlen=max_length, padding='post')
    return padded
```

Listing 15.16: Function to integer encode and pad movie reviews.

We are now ready to define our neural network model. The model will use an `Embedding` layer as the first hidden layer. The `Embedding` layer requires the specification of the vocabulary size, the size of the real-valued vector space, and the maximum length of input documents. The vocabulary size is the total number of words in our vocabulary, plus one for unknown words. This could be the vocab set length or the size of the vocab within the tokenizer used to integer encode the documents, for example:

```
# define vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary size: %d' % vocab_size)
```

Listing 15.17: Calculate the size of the vocabulary for the Embedding layer.

We will use a 100-dimensional vector space, but you could try other values, such as 50 or 150. Finally, the maximum document length was calculated above in the `max_length` variable used during padding. The complete model definition is listed below including the `Embedding` layer. We use a Convolutional Neural Network (CNN) as they have proven to be successful at document classification problems. A conservative CNN configuration is used with 32 filters (parallel fields for processing words) and a kernel size of 8 with a rectified linear (`relu`) activation function. This is followed by a pooling layer that reduces the output of the convolutional layer by half.

Next, the 2D output from the CNN part of the model is flattened to one long 2D vector to represent the *features* extracted by the CNN. The back-end of the model is a standard Multilayer Perceptron layers to interpret the CNN features. The output layer uses a sigmoid activation function to output a value between 0 and 1 for the negative and positive sentiment in the review.

```
# define the model
def define_model(vocab_size, max_length):
    model = Sequential()
    model.add(Embedding(vocab_size, 100, input_length=max_length))
    model.add(Conv1D(filters=32, kernel_size=8, activation='relu'))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # compile network
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model
```

Listing 15.18: Define a CNN model with the Embedding Layer.

Running just this piece provides a summary of the defined network. We can see that the `Embedding` layer expects documents with a length of 1,317 words as input and encodes each word in the document as a 100 element vector.

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 1317, 100)	2576800
conv1d_1 (Conv1D)	(None, 1310, 32)	25632
max_pooling1d_1 (MaxPooling1D)	(None, 655, 32)	0
flatten_1 (Flatten)	(None, 20960)	0
dense_1 (Dense)	(None, 10)	209610

```

dense_2 (Dense)           (None, 1)          11
=====
Total params: 2,812,053
Trainable params: 2,812,053
Non-trainable params: 0
=====
```

Listing 15.19: Summary of the defined model.

A plot the defined model is then saved to file with the name `model.png`.

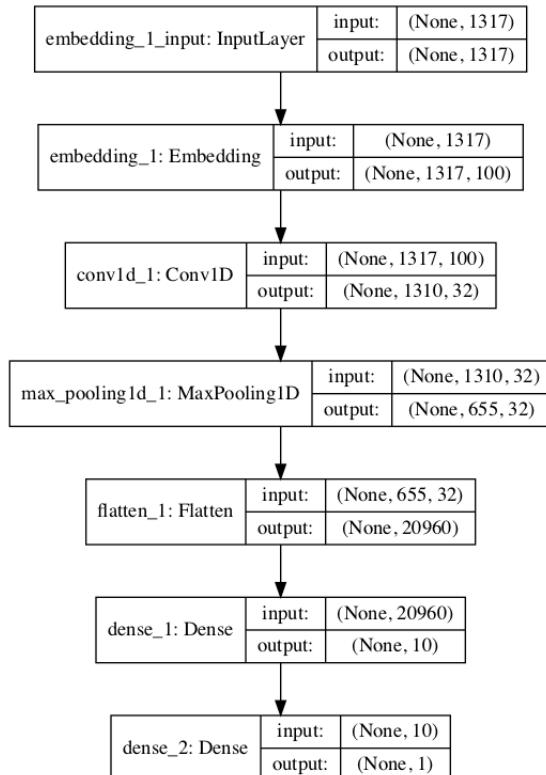


Figure 15.1: Plot of the defined CNN classification model.

Next, we fit the network on the training data. We use a binary cross entropy loss function because the problem we are learning is a binary classification problem. The efficient Adam implementation of stochastic gradient descent is used and we keep track of accuracy in addition to loss during training. The model is trained for 10 epochs, or 10 passes through the training data. The network configuration and training schedule were found with a little trial and error, but are by no means optimal for this problem. If you can get better results with a different configuration, let me know.

```
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
```

Listing 15.20: Train the defined classification model.

After the model is fit, it is saved to a file named `model.h5` for later evaluation.

```
# save the model
model.save('model.h5')
```

Listing 15.21: Save the fit model to file.

We can tie all of this together. The complete code listing is provided below.

```

import string
import re
from os import listdir
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc, vocab):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub(' ', w) for w in tokens]
    # filter out tokens not in vocab
    tokens = [w for w in tokens if w in vocab]
    tokens = ' '.join(tokens)
    return tokens

# load all docs in a directory
def process_docs(directory, vocab, is_train):
    documents = list()
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if is_train and filename.startswith('cv9'):
            continue
        if not is_train and not filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # load the doc
        doc = load_doc(path)

```

```
# clean doc
tokens = clean_doc(doc, vocab)
# add to list
documents.append(tokens)
return documents

# load and clean a dataset
def load_clean_dataset(vocab, is_train):
    # load documents
    neg = process_docs('txt_sentoken/neg', vocab, is_train)
    pos = process_docs('txt_sentoken/pos', vocab, is_train)
    docs = neg + pos
    # prepare labels
    labels = array([0 for _ in range(len(neg))] + [1 for _ in range(len(pos))])
    return docs, labels

# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# integer encode and pad documents
def encode_docs(tokenizer, max_length, docs):
    # integer encode
    encoded = tokenizer.texts_to_sequences(docs)
    # pad sequences
    padded = pad_sequences(encoded, maxlen=max_length, padding='post')
    return padded

# define the model
def define_model(vocab_size, max_length):
    model = Sequential()
    model.add(Embedding(vocab_size, 100, input_length=max_length))
    model.add(Conv1D(filters=32, kernel_size=8, activation='relu'))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # compile network
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load training data
train_docs, ytrain = load_clean_dataset(vocab, True)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# define vocabulary size
vocab_size = len(tokenizer.word_index) + 1
```

```

print('Vocabulary size: %d' % vocab_size)
# calculate the maximum sequence length
max_length = max([len(s.split()) for s in train_docs])
print('Maximum length: %d' % max_length)
# encode data
Xtrain = encode_docs(tokenizer, max_length, train_docs)
# define model
model = define_model(vocab_size, max_length)
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
# save the model
model.save('model.h5')

```

Listing 15.22: Complete example of fitting a CNN model with an Embedding input layer.

Running the example will first provide a summary of the training dataset vocabulary (25,768) and maximum input sequence length in words (1,317). The example should run in a few minutes and the fit model will be saved to file.

```

...
Vocabulary size: 25768
Maximum length: 1317
Epoch 1/10
7s - loss: 0.6927 - acc: 0.4800
Epoch 2/10
7s - loss: 0.6610 - acc: 0.5922
Epoch 3/10
7s - loss: 0.3461 - acc: 0.8844
Epoch 4/10
7s - loss: 0.0441 - acc: 0.9889
Epoch 5/10
7s - loss: 0.0058 - acc: 1.0000
Epoch 6/10
7s - loss: 0.0024 - acc: 1.0000
Epoch 7/10
7s - loss: 0.0015 - acc: 1.0000
Epoch 8/10
7s - loss: 0.0011 - acc: 1.0000
Epoch 9/10
7s - loss: 8.0111e-04 - acc: 1.0000
Epoch 10/10
7s - loss: 5.4109e-04 - acc: 1.0000

```

Listing 15.23: Example output from fitting the model.

15.5 Evaluate Model

In this section, we will evaluate the trained model and use it to make predictions on new data. First, we can use the built-in `evaluate()` function to estimate the skill of the model on both the training and test dataset. This requires that we load and encode both the training and test datasets.

```

# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)

```

```

test_docs, ytest = load_clean_dataset(vocab, False)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# define vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary size: %d' % vocab_size)
# calculate the maximum sequence length
max_length = max([len(s.split()) for s in train_docs])
print('Maximum length: %d' % max_length)
# encode data
Xtrain = encode_docs(tokenizer, max_length, train_docs)
Xtest = encode_docs(tokenizer, max_length, test_docs)

```

Listing 15.24: Load and encode both training and test datasets.

We can then load the model and evaluate it on both datasets and print the accuracy.

```

# load the model
model = load_model('model.h5')
# evaluate model on training dataset
_, acc = model.evaluate(Xtrain, ytrain, verbose=0)
print('Train Accuracy: %f' % (acc*100))
# evaluate model on test dataset
_, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %f' % (acc*100))

```

Listing 15.25: Load and evaluate model on both train and test datasets.

New data must then be prepared using the same text encoding and encoding schemes as was used on the training dataset. Once prepared, a prediction can be made by calling the `predict()` function on the model. The function below named `predict_sentiment()` will encode and pad a given movie review text and return a prediction in terms of both the percentage and a label.

```

# classify a review as negative or positive
def predict_sentiment(review, vocab, tokenizer, max_length, model):
    # clean review
    line = clean_doc(review, vocab)
    # encode and pad review
    padded = encode_docs(tokenizer, max_length, [line])
    # predict sentiment
    yhat = model.predict(padded, verbose=0)
    # retrieve predicted percentage and label
    percent_pos = yhat[0,0]
    if round(percent_pos) == 0:
        return (1-percent_pos), 'NEGATIVE'
    return percent_pos, 'POSITIVE'

```

Listing 15.26: Function to predict the sentiment for an ad hoc movie review.

We can test out this model with two ad hoc movie reviews. The complete example is listed below.

```

import string
import re
from os import listdir
from numpy import array
from keras.preprocessing.text import Tokenizer

```

```
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc, vocab):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub(' ', w) for w in tokens]
    # filter out tokens not in vocab
    tokens = [w for w in tokens if w in vocab]
    tokens = ' '.join(tokens)
    return tokens

# load all docs in a directory
def process_docs(directory, vocab, is_train):
    documents = list()
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if is_train and filename.startswith('cv9'):
            continue
        if not is_train and not filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # load the doc
        doc = load_doc(path)
        # clean doc
        tokens = clean_doc(doc, vocab)
        # add to list
        documents.append(tokens)
    return documents

# load and clean a dataset
def load_clean_dataset(vocab, is_train):
    # load documents
    neg = process_docs('txt_sentoken/neg', vocab, is_train)
    pos = process_docs('txt_sentoken/pos', vocab, is_train)
    docs = neg + pos
    # prepare labels
    labels = array([0 for _ in range(len(neg))] + [1 for _ in range(len(pos))])
    return docs, labels
```

```
# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# integer encode and pad documents
def encode_docs(tokenizer, max_length, docs):
    # integer encode
    encoded = tokenizer.texts_to_sequences(docs)
    # pad sequences
    padded = pad_sequences(encoded, maxlen=max_length, padding='post')
    return padded

# classify a review as negative or positive
def predict_sentiment(review, vocab, tokenizer, max_length, model):
    # clean review
    line = clean_doc(review, vocab)
    # encode and pad review
    padded = encode_docs(tokenizer, max_length, [line])
    # predict sentiment
    yhat = model.predict(padded, verbose=0)
    # retrieve predicted percentage and label
    percent_pos = yhat[0,0]
    if round(percent_pos) == 0:
        return (1-percent_pos), 'NEGATIVE'
    return percent_pos, 'POSITIVE'

# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
test_docs, ytest = load_clean_dataset(vocab, False)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# define vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary size: %d' % vocab_size)
# calculate the maximum sequence length
max_length = max([len(s.split()) for s in train_docs])
print('Maximum length: %d' % max_length)
# encode data
Xtrain = encode_docs(tokenizer, max_length, train_docs)
Xtest = encode_docs(tokenizer, max_length, test_docs)
# load the model
model = load_model('model.h5')
# evaluate model on training dataset
_, acc = model.evaluate(Xtrain, ytrain, verbose=0)
print('Train Accuracy: %.2f' % (acc*100))
# evaluate model on test dataset
_, acc = model.evaluate(Xtest, ytest, verbose=0)
print('Test Accuracy: %.2f' % (acc*100))

# test positive text
```

```

text = 'Everyone will enjoy this film. I love it, recommended!'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, max_length, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))
# test negative text
text = 'This is a bad movie. Do not watch it. It sucks.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, max_length, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)' % (text, sentiment, percent*100))

```

Listing 15.27: Complete example of making a prediction on new text data.

Running the example first prints the skill of the model on the training and test dataset. We can see that the model achieves 100% accuracy on the training dataset and 87.5% on the test dataset, an impressive score.

Next, we can see that the model makes the correct prediction on two contrived movie reviews. We can see that the percentage or confidence of the prediction is close to 50% for both, this may be because the two contrived reviews are very short and the model is expecting sequences of 1,000 or more words.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```

Train Accuracy: 100.00
Test Accuracy: 87.50
Review: [Everyone will enjoy this film. I love it, recommended!]
Sentiment: POSITIVE (55.431%)
Review: [This is a bad movie. Do not watch it. It sucks.]
Sentiment: NEGATIVE (54.746%)

```

Listing 15.28: Example output from making a prediction on new reviews.

15.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Data Cleaning.** Explore better data cleaning, perhaps leaving some punctuation in tact or normalizing contractions.
- **Truncated Sequences.** Padding all sequences to the length of the longest sequence might be extreme if the longest sequence is very different to all other reviews. Study the distribution of review lengths and truncate reviews to a mean length.
- **Truncated Vocabulary.** We removed infrequently occurring words, but still had a large vocabulary of more than 25,000 words. Explore further reducing the size of the vocabulary and the effect on model skill.
- **Filters and Kernel Size.** The number of filters and kernel size are important to model skill and were not tuned. Explore tuning these two CNN parameters.
- **Epochs and Batch Size.** The model appears to fit the training dataset quickly. Explore alternate configurations of the number of training epochs and batch size and use the test dataset as a validation set to pick a better stopping point for training the model.

- **Deeper Network.** Explore whether a deeper network results in better skill, either in terms of CNN layers, MLP layers and both.
- **Pre-Train an Embedding.** Explore pre-training a Word2Vec word embedding in the model and the impact on model skill with and without further fine tuning during training.
- **Use GloVe Embedding.** Explore loading the pre-trained GloVe embedding and the impact on model skill with and without further fine tuning during training.
- **Longer Test Reviews.** Explore whether the skill of model predictions is dependent on the length of movie reviews as suspected in the final section on evaluating the model.
- **Train Final Model.** Train a final model on all available data and use it make predictions on real ad hoc movie reviews from the internet.

If you explore any of these extensions, I'd love to know.

15.7 Further Reading

This section provides more resources on the topic if you are looking go deeper.

15.7.1 Dataset

- Movie Review Data.
<http://www.cs.cornell.edu/people/pabo/movie-review-data/>
- *A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts*, 2004.
<http://xxx.lanl.gov/abs/cs/0409058>
- Movie Review Polarity Dataset.
http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

15.7.2 APIs

- `collections` API - Container datatypes.
<https://docs.python.org/3/library/collections.html>
- `Tokenizer` Keras API.
<https://keras.io/preprocessing/text/#tokenizer>
- `Embedding` Keras API.
<https://keras.io/layers/embeddings/>

15.8 Summary

In this tutorial, you discovered how to develop word embeddings for the classification of movie reviews. Specifically, you learned:

- How to prepare movie review text data for classification with deep learning methods.
- How to develop a neural classification model with word embedding and convolutional layers.
- How to evaluate the developed a neural classification model.

15.8.1 Next

In the next chapter, you will discover how you can develop an n-gram multichannel convolutional neural network for text classification.

Chapter 16

Project: Develop an n-gram CNN Model for Sentiment Analysis

A standard deep learning model for text classification and sentiment analysis uses a word embedding layer and one-dimensional convolutional neural network. The model can be expanded by using multiple parallel convolutional neural networks that read the source document using different kernel sizes. This, in effect, creates a multichannel convolutional neural network for text that reads text with different n-gram sizes (groups of words). In this tutorial, you will discover how to develop a multichannel convolutional neural network for sentiment prediction on text movie review data. After completing this tutorial, you will know:

- How to prepare movie review text data for modeling.
- How to develop a multichannel convolutional neural network for text in Keras.
- How to evaluate a fit model on unseen movie review data.

Let's get started.

16.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Movie Review Dataset.
2. Data Preparation.
3. Develop Multichannel Model.
4. Evaluate Model.

16.2 Movie Review Dataset

In this tutorial, we will use the Movie Review Dataset. This dataset designed for sentiment analysis was described previously in Chapter 9. You can download the dataset from here:

- Movie Review Polarity Dataset (`review_polarity.tar.gz`, 3MB).
http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz

After unzipping the file, you will have a directory called `txt_sentoken` with two sub-directories containing the text `neg` and `pos` for negative and positive reviews. Reviews are stored one per file with a naming convention `cv000` to `cv999` for each of `neg` and `pos`.

16.3 Data Preparation

Note: The preparation of the movie review dataset was first described in Chapter 9. In this section, we will look at 3 things:

1. Separation of data into training and test sets.
2. Loading and cleaning the data to remove punctuation and numbers.
3. Clean All Reviews and Save.

16.3.1 Split into Train and Test Sets

We are pretending that we are developing a system that can predict the sentiment of a textual movie review as either positive or negative. This means that after the model is developed, we will need to make predictions on new textual reviews. This will require all of the same data preparation to be performed on those new reviews as is performed on the training data for the model. We will ensure that this constraint is built into the evaluation of our models by splitting the training and test datasets prior to any data preparation. This means that any knowledge in the data in the test set that could help us better prepare the data (e.g. the words used) are unavailable in the preparation of data used for training the model.

That being said, we will use the last 100 positive reviews and the last 100 negative reviews as a test set (100 reviews) and the remaining 1,800 reviews as the training dataset. This is a 90% train, 10% split of the data. The split can be imposed easily by using the filenames of the reviews where reviews named 000 to 899 are for training data and reviews named 900 onwards are for test.

16.3.2 Loading and Cleaning Reviews

The text data is already pretty clean; not much preparation is required. Without getting bogged down too much in the details, we will prepare the data using the following way:

- Split tokens on white space.
- Remove all punctuation from words.
- Remove all words that are not purely comprised of alphabetical characters.
- Remove all words that are known stop words.
- Remove all words that have a length ≤ 1 character.

We can put all of these steps into a function called `clean_doc()` that takes as an argument the raw text loaded from a file and returns a list of cleaned tokens. We can also define a function `load_doc()` that loads a document from file ready for use with the `clean_doc()` function. An example of cleaning the first positive review is listed below.

```
from nltk.corpus import stopwords
import string
import re

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[' + re.escape(string.punctuation) + ']')
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    return tokens

# load the document
filename = 'txt_sentoken/pos/cv000_29590.txt'
text = load_doc(filename)
tokens = clean_doc(text)
print(tokens)
```

Listing 16.1: Example of cleaning a movie review.

Running the example prints a long list of clean tokens. There are many more cleaning steps we may want to explore and I leave them as further exercises.

```
...
'creepy', 'place', 'even', 'acting', 'hell', 'solid', 'dreamy', 'depp', 'turning',
'typically', 'strong', 'performance', 'deftly', 'handling', 'british', 'accent',
'ians', 'holm', 'joe', 'oulds', 'secret', 'richardson', 'dalmatians', 'log', 'great',
'supporting', 'roles', 'big', 'surprise', 'graham', 'cringed', 'first', 'time',
'opened', 'mouth', 'imagining', 'attempt', 'irish', 'accent', 'actually', 'wasnt',
'half', 'bad', 'film', 'however', 'good', 'strong', 'violencegore', 'sexuality',
'language', 'drug', 'content']
```

Listing 16.2: Example output of cleaning a movie review.

16.3.3 Clean All Reviews and Save

We can now use the function to clean reviews and apply it to all reviews. To do this, we will develop a new function named `process_docs()` below that will walk through all reviews in a directory, clean them, and return them as a list. We will also add an argument to the function to indicate whether the function is processing train or test reviews, that way the filenames can be filtered (as described above) and only those train or test reviews requested will be cleaned and returned. The full function is listed below.

```
# load all docs in a directory
def process_docs(directory, is_train):
    documents = list()
    # walk through all files in the folder
    for filename in.listdir(directory):
        # skip any reviews in the test set
        if is_train and filename.startswith('cv9'):
            continue
        if not is_train and not filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
        # load the doc
        doc = load_doc(path)
        # clean doc
        tokens = clean_doc(doc)
        # add to list
        documents.append(tokens)
    return documents
```

Listing 16.3: Function for cleaning multiple review documents.

We can call this function with negative training reviews. We also need labels for the train and test documents. We know that we have 900 training documents and 100 test documents. We can use a Python list comprehension to create the labels for the negative (0) and positive (1) reviews for both train and test sets. The function below named `load_clean_dataset()` will load and clean the movie review text and also create the labels for the reviews.

```
# load and clean a dataset
def load_clean_dataset(is_train):
    # load documents
    neg = process_docs('txt_sentoken/neg', is_train)
    pos = process_docs('txt_sentoken/pos', is_train)
    docs = neg + pos
    # prepare labels
    labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
    return docs, labels
```

Listing 16.4: Function to prepare reviews and labels for a dataset.

Finally, we want to save the prepared train and test sets to file so that we can load them later for modeling and model evaluation. The function below-named `save_dataset()` will save a given prepared dataset (`X` and `y` elements) to a file using the pickle API (this is the standard API for saving objects in Python).

```
# save a dataset to file
def save_dataset(dataset, filename):
```

```
dump(dataset, open(filename, 'wb'))
print('Saved: %s' % filename)
```

Listing 16.5: Function for saving clean documents to file.

16.3.4 Complete Example

We can tie all of these data preparation steps together. The complete example is listed below.

```
import string
import re
from os import listdir
from nltk.corpus import stopwords
from pickle import dump

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # filter out stop words
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # filter out short tokens
    tokens = [word for word in tokens if len(word) > 1]
    tokens = ' '.join(tokens)
    return tokens

# load all docs in a directory
def process_docs(directory, is_train):
    documents = list()
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if is_train and filename.startswith('cv9'):
            continue
        if not is_train and not filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/' + filename
```

```

# load the doc
doc = load_doc(path)
# clean doc
tokens = clean_doc(doc)
# add to list
documents.append(tokens)
return documents

# load and clean a dataset
def load_clean_dataset(is_train):
    # load documents
    neg = process_docs('txt_sentoken/neg', is_train)
    pos = process_docs('txt_sentoken/pos', is_train)
    docs = neg + pos
    # prepare labels
    labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
    return docs, labels

# save a dataset to file
def save_dataset(dataset, filename):
    dump(dataset, open(filename, 'wb'))
    print('Saved: %s' % filename)

# load and clean all reviews
train_docs, ytrain = load_clean_dataset(True)
test_docs, ytest = load_clean_dataset(False)
# save training datasets
save_dataset([train_docs, ytrain], 'train.pkl')
save_dataset([test_docs, ytest], 'test.pkl')

```

Listing 16.6: Complete example of cleaning and saving all movie reviews.

Running the example cleans the text movie review documents, creates labels, and saves the prepared data for both train and test datasets in `train.pkl` and `test.pkl` respectively. Now we are ready to develop our model.

16.4 Develop Multichannel Model

In this section, we will develop a multichannel convolutional neural network for the sentiment analysis prediction problem. This section is divided into 3 parts:

1. Encode Data
2. Define Model.
3. Complete Example.

16.4.1 Encode Data

The first step is to load the cleaned training dataset. The function below-named `load_dataset()` can be called to load the pickled training dataset.

```
# load a clean dataset
def load_dataset(filename):
    return load(open(filename, 'rb'))

trainLines, trainLabels = load_dataset('train.pkl')
```

Listing 16.7: Example of loading the cleaned and saved reviews.

Next, we must fit a Keras `Tokenizer` on the training dataset. We will use this tokenizer to both define the vocabulary for the `Embedding` layer and encode the review documents as integers. The function `create_tokenizer()` below will create a `Tokenizer` given a list of documents.

```
# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer
```

Listing 16.8: Function for creating a `Tokenizer`.

We also need to know the maximum length of input sequences as input for the model and to pad all sequences to the fixed length. The function `max_length()` below will calculate the maximum length (number of words) for all reviews in the training dataset.

```
# calculate the maximum document length
def max_length(lines):
    return max([len(s.split()) for s in lines])
```

Listing 16.9: Function to calculate the maximum movie review length.

We also need to know the size of the vocabulary for the `Embedding` layer. This can be calculated from the prepared `Tokenizer`, as follows:

```
# calculate vocabulary size
vocab_size = len(tokenizer.word_index) + 1
```

Listing 16.10: Calculate the size of the vocabulary.

Finally, we can integer encode and pad the clean movie review text. The function below named `encode_text()` will both encode and pad text data to the maximum review length.

```
# encode a list of lines
def encode_text(tokenizer, lines, length):
    # integer encode
    encoded = tokenizer.texts_to_sequences(lines)
    # pad encoded sequences
    padded = pad_sequences(encoded, maxlen=length, padding='post')
    return padded
```

Listing 16.11: Function to encode and pad movie review text.

16.4.2 Define Model

A standard model for document classification is to use an `Embedding` layer as input, followed by a one-dimensional convolutional neural network, pooling layer, and then a prediction output layer. The kernel size in the convolutional layer defines the number of words to consider as

the convolution is passed across the input text document, providing a grouping parameter. A multi-channel convolutional neural network for document classification involves using multiple versions of the standard model with different sized kernels. This allows the document to be processed at different resolutions or different n-grams (groups of words) at a time, whilst the model learns how to best integrate these interpretations.

This approach was first described by Yoon Kim in his 2014 paper titled *Convolutional Neural Networks for Sentence Classification*. In the paper, Kim experimented with static and dynamic (updated) embedding layers, we can simplify the approach and instead focus only on the use of different kernel sizes. This approach is best understood with a diagram taken from Kim's paper, see Chapter 14.

In Keras, a multiple-input model can be defined using the functional API. We will define a model with three input channels for processing 4-grams, 6-grams, and 8-grams of movie review text. Each channel is comprised of the following elements:

- **Input** layer that defines the length of input sequences.
- **Embedding** layer set to the size of the vocabulary and 100-dimensional real-valued representations.
- **Conv1D** layer with 32 filters and a kernel size set to the number of words to read at once.
- **MaxPooling1D** layer to consolidate the output from the convolutional layer.
- **Flatten** layer to reduce the three-dimensional output to two dimensional for concatenation.

The output from the three channels are concatenated into a single vector and process by a **Dense** layer and an output layer. The function below defines and returns the model. As part of defining the model, a summary of the defined model is printed and a plot of the model graph is created and saved to file.

```
# define the model
def define_model(length, vocab_size):
    # channel 1
    inputs1 = Input(shape=(length,))
    embedding1 = Embedding(vocab_size, 100)(inputs1)
    conv1 = Conv1D(filters=32, kernel_size=4, activation='relu')(embedding1)
    drop1 = Dropout(0.5)(conv1)
    pool1 = MaxPooling1D(pool_size=2)(drop1)
    flat1 = Flatten()(pool1)

    # channel 2
    inputs2 = Input(shape=(length,))
    embedding2 = Embedding(vocab_size, 100)(inputs2)
    conv2 = Conv1D(filters=32, kernel_size=6, activation='relu')(embedding2)
    drop2 = Dropout(0.5)(conv2)
    pool2 = MaxPooling1D(pool_size=2)(drop2)
    flat2 = Flatten()(pool2)

    # channel 3
    inputs3 = Input(shape=(length,))
    embedding3 = Embedding(vocab_size, 100)(inputs3)
    conv3 = Conv1D(filters=32, kernel_size=8, activation='relu')(embedding3)
    drop3 = Dropout(0.5)(conv3)
    pool3 = MaxPooling1D(pool_size=2)(drop3)
    flat3 = Flatten()(pool3)
```

```

# merge
merged = concatenate([flat1, flat2, flat3])
# interpretation
dense1 = Dense(10, activation='relu')(merged)
outputs = Dense(1, activation='sigmoid')(dense1)
model = Model(inputs=[inputs1, inputs2, inputs3], outputs=outputs)
# compile
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
# summarize
model.summary()
plot_model(model, show_shapes=True, to_file='multichannel.png')
return model

```

Listing 16.12: Function for defining the classification model.

16.4.3 Complete Example

Pulling all of this together, the complete example is listed below.

```

from pickle import load
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils.vis_utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.merge import concatenate

# load a clean dataset
def load_dataset(filename):
    return load(open(filename, 'rb'))

# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# calculate the maximum document length
def max_length(lines):
    return max([len(s.split()) for s in lines])

# encode a list of lines
def encode_text(tokenizer, lines, length):
    # integer encode
    encoded = tokenizer.texts_to_sequences(lines)
    # pad encoded sequences
    padded = pad_sequences(encoded, maxlen=length, padding='post')
    return padded

```

```

# define the model
def define_model(length, vocab_size):
    # channel 1
    inputs1 = Input(shape=(length,))
    embedding1 = Embedding(vocab_size, 100)(inputs1)
    conv1 = Conv1D(filters=32, kernel_size=4, activation='relu')(embedding1)
    drop1 = Dropout(0.5)(conv1)
    pool1 = MaxPooling1D(pool_size=2)(drop1)
    flat1 = Flatten()(pool1)
    # channel 2
    inputs2 = Input(shape=(length,))
    embedding2 = Embedding(vocab_size, 100)(inputs2)
    conv2 = Conv1D(filters=32, kernel_size=6, activation='relu')(embedding2)
    drop2 = Dropout(0.5)(conv2)
    pool2 = MaxPooling1D(pool_size=2)(drop2)
    flat2 = Flatten()(pool2)
    # channel 3
    inputs3 = Input(shape=(length,))
    embedding3 = Embedding(vocab_size, 100)(inputs3)
    conv3 = Conv1D(filters=32, kernel_size=8, activation='relu')(embedding3)
    drop3 = Dropout(0.5)(conv3)
    pool3 = MaxPooling1D(pool_size=2)(drop3)
    flat3 = Flatten()(pool3)
    # merge
    merged = concatenate([flat1, flat2, flat3])
    # interpretation
    dense1 = Dense(10, activation='relu')(merged)
    outputs = Dense(1, activation='sigmoid')(dense1)
    model = Model(inputs=[inputs1, inputs2, inputs3], outputs=outputs)
    # compile
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize
    model.summary()
    plot_model(model, show_shapes=True, to_file='model.png')
    return model

# load training dataset
trainLines, trainLabels = load_dataset('train.pkl')
# create tokenizer
tokenizer = create_tokenizer(trainLines)
# calculate max document length
length = max_length(trainLines)
print('Max document length: %d' % length)
# calculate vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary size: %d' % vocab_size)
# encode data
trainX = encode_text(tokenizer, trainLines, length)
# define model
model = define_model(length, vocab_size)
# fit model
model.fit([trainX, trainX, trainX], trainLabels, epochs=7, batch_size=16)
# save the model
model.save('model.h5')

```

Listing 16.13: Complete example of fitting the n-gram CNN model.

Running the example first prints a summary of the prepared training dataset.

```
Max document length: 1380
Vocabulary size: 44277
```

Listing 16.14: Example output from preparing the training data.

The model is fit relatively quickly and appears to show good skill on the training dataset.

```
...
Epoch 3/7
1800/1800 [=====] - 29s - loss: 0.0460 - acc: 0.9894
Epoch 4/7
1800/1800 [=====] - 30s - loss: 0.0041 - acc: 1.0000
Epoch 5/7
1800/1800 [=====] - 31s - loss: 0.0010 - acc: 1.0000
Epoch 6/7
1800/1800 [=====] - 30s - loss: 3.0271e-04 - acc: 1.0000
Epoch 7/7
1800/1800 [=====] - 28s - loss: 1.3875e-04 - acc: 1.0000
```

Listing 16.15: Example output from fitting the model.

A plot of the defined model is saved to file, clearly showing the three input channels for the model.

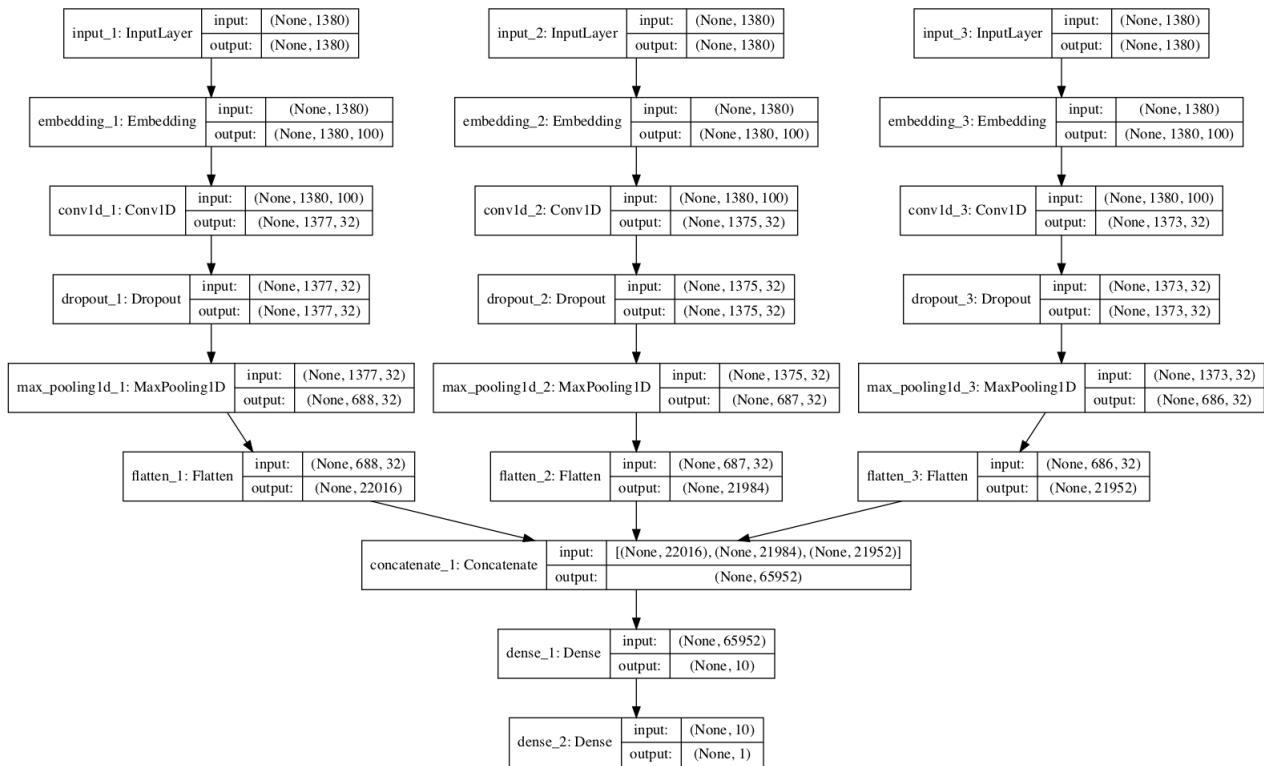


Figure 16.1: Plot of the Multichannel Convolutional Neural Network For Text.

The model is fit for a number of epochs and saved to the file `model.h5` for later evaluation.

16.5 Evaluate Model

In this section, we can evaluate the fit model by predicting the sentiment on all reviews in the unseen test dataset. Using the data loading functions developed in the previous section, we can load and encode both the training and test datasets.

```
# load datasets
trainLines, trainLabels = load_dataset('train.pkl')
testLines, testLabels = load_dataset('test.pkl')

# create tokenizer
tokenizer = create_tokenizer(trainLines)
# calculate max document length
length = max_length(trainLines)
# calculate vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Max document length: %d' % length)
print('Vocabulary size: %d' % vocab_size)
# encode data
trainX = encode_text(tokenizer, trainLines, length)
testX = encode_text(tokenizer, testLines, length)
print(trainX.shape, testX.shape)
```

Listing 16.16: Prepare train and test data for evaluating the model.

We can load the saved model and evaluate it on both the training and test datasets. The complete example is listed below.

```
from pickle import load
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model

# load a clean dataset
def load_dataset(filename):
    return load(open(filename, 'rb'))

# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# calculate the maximum document length
def max_length(lines):
    return max([len(s.split()) for s in lines])

# encode a list of lines
def encode_text(tokenizer, lines, length):
    # integer encode
    encoded = tokenizer.texts_to_sequences(lines)
    # pad encoded sequences
    padded = pad_sequences(encoded, maxlen=length, padding='post')
    return padded

# load datasets
```

```

trainLines, trainLabels = load_dataset('train.pkl')
testLines, testLabels = load_dataset('test.pkl')
# create tokenizer
tokenizer = create_tokenizer(trainLines)
# calculate max document length
length = max_length(trainLines)
print('Max document length: %d' % length)
# calculate vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary size: %d' % vocab_size)
# encode data
trainX = encode_text(tokenizer, trainLines, length)
testX = encode_text(tokenizer, testLines, length)
# load the model
model = load_model('model.h5')
# evaluate model on training dataset
_, acc = model.evaluate([trainX,trainX,trainX], trainLabels, verbose=0)
print('Train Accuracy: %.2f' % (acc*100))
# evaluate model on test dataset
_, acc = model.evaluate([testX,testX,testX], testLabels, verbose=0)
print('Test Accuracy: %.2f' % (acc*100))

```

Listing 16.17: Complete example of evaluating the fit model.

Running the example prints the skill of the model on both the training and test datasets. We can see that, as expected, the skill on the training dataset is excellent, here at 100% accuracy. We can also see that the skill of the model on the unseen test dataset is also very impressive, achieving 88.5%, which is above the skill of the model reported in the 2014 paper (although not a direct apples-to-apples comparison).

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

Train Accuracy: 100.00
Test Accuracy: 88.50

Listing 16.18: Example output from evaluating the fit model.

16.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Different n-grams.** Explore the model by changing the kernel size (number of n-grams) used by the channels in the model to see how it impacts model skill.
- **More or Fewer Channels.** Explore using more or fewer channels in the model and see how it impacts model skill.
- **Shared Embedding.** Explore configurations where each channel shares the same word embedding and report on the impact on model skill.

- **Deeper Network.** Convolutional neural networks perform better in computer vision when they are deeper. Explore using deeper models here and see how it impacts model skill.
- **Truncated Sequences.** Padding all sequences to the length of the longest sequence might be extreme if the longest sequence is very different to all other reviews. Study the distribution of review lengths and truncate reviews to a mean length.
- **Truncated Vocabulary.** We removed infrequently occurring words, but still had a large vocabulary of more than 25,000 words. Explore further reducing the size of the vocabulary and the effect on model skill.
- **Epochs and Batch Size.** The model appears to fit the training dataset quickly. Explore alternate configurations of the number of training epochs and batch size and use the test dataset as a validation set to pick a better stopping point for training the model.
- **Pre-Train an Embedding.** Explore pre-training a Word2Vec word embedding in the model and the impact on model skill with and without further fine tuning during training.
- **Use GloVe Embedding.** Explore loading the pre-trained GloVe embedding and the impact on model skill with and without further fine tuning during training.
- **Train Final Model.** Train a final model on all available data and use it make predictions on real ad hoc movie reviews from the internet.

If you explore any of these extensions, I'd love to know.

16.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- *Convolutional Neural Networks for Sentence Classification*, 2014.
<https://arxiv.org/abs/1408.5882>
- Convolutional Neural Networks for Sentence Classification (code).
https://github.com/yoonkim/CNN_sentence
- Keras Functional API.
<https://keras.io/getting-started/functional-api-guide/>

16.8 Summary

In this tutorial, you discovered how to develop a multichannel convolutional neural network for sentiment prediction on text movie review data. Specifically, you learned:

- How to prepare movie review text data for modeling.
- How to develop a multichannel convolutional neural network for text in Keras.
- How to evaluate a fit model on unseen movie review data.

16.8.1 Next

This chapter is the last in the text classification part. In the next part, you will discover how to develop neural language models.

Part VII

Language Modeling

Chapter 17

Neural Language Modeling

Language modeling is central to many important natural language processing tasks. Recently, neural-network-based language models have demonstrated better performance than classical methods both standalone and as part of more challenging natural language processing tasks. In this chapter, you will discover language modeling for natural language processing. After reading this chapter, you will know:

- Why language modeling is critical to addressing tasks in natural language processing.
- What a language model is and some examples of where they are used.
- How neural networks can be used for language modeling.

Let's get started.

17.1 Overview

This tutorial is divided into the following parts:

1. Problem of Modeling Language
2. Statistical Language Modeling
3. Neural Language Models

17.2 Problem of Modeling Language

Formal languages, like programming languages, can be fully specified. All the reserved words can be defined and the valid ways that they can be used can be precisely defined. We cannot do this with natural language. Natural languages are not designed; they emerge, and therefore there is no formal specification.

There may be formal rules and heuristics for parts of the language, but as soon as rules are defined, you will devise or encounter counter examples that contradict the rules. Natural languages involve vast numbers of terms that can be used in ways that introduce all kinds of ambiguities, yet can still be understood by other humans. Further, languages change, word

usages change: it is a moving target. Nevertheless, linguists try to specify the language with formal grammars and structures. It can be done, but it is very difficult and the results can be fragile. An alternative approach to specifying the model of the language is to learn it from examples.

17.3 Statistical Language Modeling

Statistical Language Modeling, or Language Modeling and LM for short, is the development of probabilistic models that are able to predict the next word in the sequence given the words that precede it.

Language modeling is the task of assigning a probability to sentences in a language. [...] Besides assigning a probability to each sequence of words, the language models also assigns a probability for the likelihood of a given word (or a sequence of words) to follow a sequence of words

— Page 105, *Neural Network Methods in Natural Language Processing*, 2017.

A language model learns the probability of word occurrence based on examples of text. Simpler models may look at a context of a short sequence of words, whereas larger models may work at the level of sentences or paragraphs. Most commonly, language models operate at the level of words.

The notion of a language model is inherently probabilistic. A language model is a function that puts a probability measure over strings drawn from some vocabulary.

— Page 238, *An Introduction to Information Retrieval*, 2008.

A language model can be developed and used standalone, such as to generate new sequences of text that appear to have come from the corpus. Language modeling is a root problem for a large range of natural language processing tasks. More practically, language models are used on the front-end or back-end of a more sophisticated model for a task that requires language understanding.

... language modeling is a crucial component in real-world applications such as machine-translation and automatic speech recognition, [...] For these reasons, language modeling plays a central role in natural-language processing, AI, and machine-learning research.

— Page 105, *Neural Network Methods in Natural Language Processing*, 2017.

A good example is speech recognition, where audio data is used as an input to the model and the output requires a language model that interprets the input signal and recognizes each new word within the context of the words already recognized.

Speech recognition is principally concerned with the problem of transcribing the speech signal as a sequence of words. [...] From this point of view, speech is assumed to be generated by a language model which provides estimates of $\Pr(w)$ for all word strings w independently of the observed signal [...] The goal of speech recognition is to find the most likely word sequence given the observed acoustic signal.

— Pages 205-206, *The Oxford Handbook of Computational Linguistics*, 2005

Similarly, language models are used to generate text in many similar natural language processing tasks, for example:

- Optical Character Recognition
- Handwriting Recognition.
- Machine Translation.
- Spelling Correction.
- Image Captioning.
- Text Summarization
- And much more.

Language modeling is the art of determining the probability of a sequence of words. This is useful in a large variety of areas including speech recognition, optical character recognition, handwriting recognition, machine translation, and spelling correction

— *A Bit of Progress in Language Modeling*, 2001.

Developing better language models often results in models that perform better on their intended natural language processing task. This is the motivation for developing better and more accurate language models.

[language models] have played a key role in traditional NLP tasks such as speech recognition, machine translation, or text summarization. Often (although not always), training better language models improves the underlying metrics of the downstream task (such as word error rate for speech recognition, or BLEU score for translation), which makes the task of training better LMs valuable by itself.

— *Exploring the Limits of Language Modeling*, 2016.

17.4 Neural Language Models

Recently, the use of neural networks in the development of language models has become very popular, to the point that it may now be the preferred approach. The use of neural networks in language modeling is often called Neural Language Modeling, or NLM for short. Neural network approaches are achieving better results than classical methods both on standalone language models and when models are incorporated into larger models on challenging tasks like speech recognition and machine translation. A key reason for the leaps in improved performance may be the method's ability to generalize.

Nonlinear neural network models solve some of the shortcomings of traditional language models: they allow conditioning on increasingly large context sizes with only a linear increase in the number of parameters, they alleviate the need for manually designing backoff orders, and they support generalization across different contexts.

— Page 109, *Neural Network Methods in Natural Language Processing*, 2017.

Specifically, a word embedding is adopted that uses a real-valued vector to represent each word in a projected vector space. This learned representation of words based on their usage allows words with a similar meaning to have a similar representation.

Neural Language Models (NLM) address the n-gram data sparsity issue through parameterization of words as vectors (word embeddings) and using them as inputs to a neural network. The parameters are learned as part of the training process. Word embeddings obtained through NLMs exhibit the property whereby semantically close words are likewise close in the induced vector space.

— *Character-Aware Neural Language Model*, 2015.

This generalization is something that the representation used in classical statistical language models cannot easily achieve.

“True generalization” is difficult to obtain in a discrete word indice space, since there is no obvious relation between the word indices.

— *Connectionist language modeling for large vocabulary continuous speech recognition*, 2002.

Further, the distributed representation approach allows the embedding representation to scale better with the size of the vocabulary. Classical methods that have one discrete representation per word fight the curse of dimensionality with larger and larger vocabularies of words that result in longer and more sparse representations. The neural network approach to language modeling can be described using the three following model properties, taken from *A Neural Probabilistic Language Model*, 2003.

1. Associate each word in the vocabulary with a distributed word feature vector.
2. Express the joint probability function of word sequences in terms of the feature vectors of these words in the sequence.
3. Learn simultaneously the word feature vector and the parameters of the probability function.

This represents a relatively simple model where both the representation and probabilistic model are learned together directly from raw text data. Recently, the neural based approaches have started to outperform the classical statistical approaches.

We provide ample empirical evidence to suggest that connectionist language models are superior to standard n-gram techniques, except their high computational (training) complexity.

— *Recurrent neural network based language model*, 2010.

Initially, feedforward neural network models were used to introduce the approach. More recently, recurrent neural networks and then networks with a long-term memory like the Long Short-Term Memory network, or LSTM, allow the models to learn the relevant context over much longer input sequences than the simpler feedforward networks.

[an RNN language model] provides further generalization: instead of considering just several preceding words, neurons with input from recurrent connections are assumed to represent short term memory. The model learns itself from the data how to represent memory. While shallow feedforward neural networks (those with just one hidden layer) can only cluster similar words, recurrent neural network (which can be considered as a deep architecture) can perform clustering of similar histories. This allows for instance efficient representation of patterns with variable length.

— *Extensions of recurrent neural network language model*, 2011.

Recently, researchers have been seeking the limits of these language models. In the paper *Exploring the Limits of Language Modeling*, evaluating language models over large datasets, such as the corpus of one million words, the authors find that LSTM-based neural language models out-perform the classical methods.

... we have shown that RNN LMs can be trained on large amounts of data, and outperform competing models including carefully tuned N-grams.

— *Exploring the Limits of Language Modeling*, 2016.

Further, they propose some heuristics for developing high-performing neural language models in general:

- **Size matters.** The best models were the largest models, specifically number of memory units.
- **Regularization matters.** Use of regularization like dropout on input connections improves results.
- **CNNs vs Embeddings.** Character-level Convolutional Neural Network (CNN) models can be used on the front-end instead of word embeddings, achieving similar and sometimes better results.
- **Ensembles matter.** Combining the prediction from multiple models can offer large improvements in model performance.

17.5 Further Reading

This section provides more resources on the topic if you are looking go deeper.

17.5.1 Books

- *Neural Network Methods in Natural Language Processing*, 2017.
<http://amzn.to/2vStiIS>
- *Natural Language Processing, Artificial Intelligence A Modern Approach*, 2009.
<http://amzn.to/2fDPfF3>
- *Language models for information retrieval, An Introduction to Information Retrieval*, 2008.
<http://amzn.to/2vAavQd>

17.5.2 Papers

- *A Neural Probabilistic Language Model*, NIPS, 2001.
<https://papers.nips.cc/paper/1839-a-neural-probabilistic-language-model.pdf>
- *A Neural Probabilistic Language Model*, JMLR, 2003.
<http://www.jmlr.org/papers/v3/bengio03a.html>
- *Connectionist language modeling for large vocabulary continuous speech recognition*, 2002.
<https://pdfs.semanticscholar.org/b4db/83366f925e9a1e1528ee9f6b41d7cd666f41.pdf>
- *Recurrent neural network based language model*, 2010.
http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf
- *Extensions of recurrent neural network language model*, 2011.
<http://ieeexplore.ieee.org/abstract/document/5947611/>
- *Character-Aware Neural Language Model*, 2015.
<https://arxiv.org/abs/1508.06615>
- *LSTM Neural Networks for Language Modeling*, 2012.
<https://pdfs.semanticscholar.org/f9a1/b3850dfd837793743565a8af95973d395a4e.pdf>
- *Exploring the Limits of Language Modeling*, 2016.
<https://arxiv.org/abs/1602.02410>

17.5.3 Articles

- Language Model, Wikipedia.
https://en.wikipedia.org/wiki/Language_model
- Neural net language models, Scholarpedia.
http://www.scholarpedia.org/article/Neural_net_language_models

17.6 Summary

In this chapter, you discovered language modeling for natural language processing tasks. Specifically, you learned:

- That natural language is not formally specified and requires the use of statistical models to learn from examples.
- That statistical language models are central to many challenging natural language processing tasks.
- That state-of-the-art results are achieved using neural language models, specifically those with word embeddings and recurrent neural network algorithms.

17.6.1 Next

In the next chapter, you will discover how you can develop a character-based neural language model.

Chapter 18

How to Develop a Character-Based Neural Language Model

A language model predicts the next word in the sequence based on the specific words that have come before it in the sequence. It is also possible to develop language models at the character level using neural networks. The benefit of character-based language models is their small vocabulary and flexibility in handling any words, punctuation, and other document structure. This comes at the cost of requiring larger models that are slower to train. Nevertheless, in the field of neural language models, character-based models offer a lot of promise for a general, flexible and powerful approach to language modeling. In this tutorial, you will discover how to develop a character-based neural language model. After completing this tutorial, you will know:

- How to prepare text for character-based language modeling.
- How to develop a character-based language model using LSTMs.
- How to use a trained character-based language model to generate text.

Let's get started.

18.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Sing a Song of Sixpence
2. Data Preparation
3. Train Language Model
4. Generate Text

18.2 Sing a Song of Sixpence

The nursery rhyme *Sing a Song of Sixpence* is well known in the west. The first verse is common, but there is also a 4 verse version that we will use to develop our character-based language model. It is short, so fitting the model will be fast, but not so short that we won't see anything interesting. The complete 4 verse version we will use as source text is listed below.

```
Sing a song of sixpence,  
A pocket full of rye.  
Four and twenty blackbirds,  
Baked in a pie.
```

```
When the pie was opened  
The birds began to sing;  
Wasn't that a dainty dish,  
To set before the king.
```

```
The king was in his counting house,  
Counting out his money;  
The queen was in the parlour,  
Eating bread and honey.
```

```
The maid was in the garden,  
Hanging out the clothes,  
When down came a blackbird  
And pecked off her nose.
```

Listing 18.1: *Sing a Song of Sixpence* nursery rhyme.

Copy the text and save it in a new file in your current working directory with the file name `rhyme.txt`.

18.3 Data Preparation

The first step is to prepare the text data. We will start by defining the type of language model.

18.3.1 Language Model Design

A language model must be trained on the text, and in the case of a character-based language model, the input and output sequences must be characters. The number of characters used as input will also define the number of characters that will need to be provided to the model in order to elicit the first predicted character. After the first character has been generated, it can be appended to the input sequence and used as input for the model to generate the next character.

Longer sequences offer more context for the model to learn what character to output next but take longer to train and impose more burden on seeding the model when generating text. We will use an arbitrary length of 10 characters for this model. There is not a lot of text, and 10 characters is a few words. We can now transform the raw text into a form that our model can learn; specifically, input and output sequences of characters.

18.3.2 Load Text

We must load the text into memory so that we can work with it. Below is a function named `load_doc()` that will load a text file given a filename and return the loaded text.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text
```

Listing 18.2: Function to load a document into memory.

We can call this function with the filename of the nursery rhyme `rhyme.txt` to load the text into memory. The contents of the file are then printed to screen as a sanity check.

```
# load text
raw_text = load_doc('rhyme.txt')
print(raw_text)
```

Listing 18.3: Load the document into memory.

18.3.3 Clean Text

Next, we need to clean the loaded text. We will not do much to it on this example. Specifically, we will strip all of the new line characters so that we have one long sequence of characters separated only by white space.

```
# clean
tokens = raw_text.split()
raw_text = ' '.join(tokens)
```

Listing 18.4: Tokenize the loaded document.

You may want to explore other methods for data cleaning, such as normalizing the case to lowercase or removing punctuation in an effort to reduce the final vocabulary size and develop a smaller and leaner model.

18.3.4 Create Sequences

Now that we have a long list of characters, we can create our input-output sequences used to train the model. Each input sequence will be 10 characters with one output character, making each sequence 11 characters long. We can create the sequences by enumerating the characters in the text, starting at the 11th character at index 10.

```
# organize into sequences of characters
length = 10
sequences = list()
for i in range(length, len(raw_text)):
    # select sequence of tokens
    seq = raw_text[i-length:i+1]
```

```
# store
sequences.append(seq)
print('Total Sequences: %d' % len(sequences))
```

Listing 18.5: Convert text into fixed-length sequences.

Running this snippet, we can see that we end up with just under 400 sequences of characters for training our language model.

```
Total Sequences: 399
```

Listing 18.6: Example output of converting text into fixed-length sequences.

18.3.5 Save Sequences

Finally, we can save the prepared data to file so that we can load it later when we develop our model. Below is a function `save_doc()` that, given a list of strings and a filename, will save the strings to file, one per line.

```
# save tokens to file, one dialog per line
def save_doc(lines, filename):
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()
```

Listing 18.7: Function to save sequences to file.

We can call this function and save our prepared sequences to the filename `char_sequences.txt` in our current working directory.

```
# save sequences to file
out_filename = 'char_sequences.txt'
save_doc(sequences, out_filename)
```

Listing 18.8: Call function to save sequences to file.

18.3.6 Complete Example

Tying all of this together, the complete code listing is provided below.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# save tokens to file, one dialog per line
def save_doc(lines, filename):
    data = '\n'.join(lines)
    file = open(filename, 'w')
```

```

file.write(data)
file.close()

# load text
raw_text = load_doc('rhyme.txt')
print(raw_text)
# clean
tokens = raw_text.split()
raw_text = ' '.join(tokens)
# organize into sequences of characters
length = 10
sequences = list()
for i in range(length, len(raw_text)):
    # select sequence of tokens
    seq = raw_text[i-length:i+1]
    # store
    sequences.append(seq)
print('Total Sequences: %d' % len(sequences))
# save sequences to file
out_filename = 'char_sequences.txt'
save_doc(sequences, out_filename)

```

Listing 18.9: Complete example of preparing the text data.

Run the example to create the `char_sequences.txt` file. Take a look inside you should see something like the following:

```

Sing a song
ing a song
ng a song o
g a song of
 a song of
a song of s
 song of si
song of six
ong of sixp
ng of sixpe
...

```

Listing 18.10: Sample of the output file.

We are now ready to train our character-based neural language model.

18.4 Train Language Model

In this section, we will develop a neural language model for the prepared sequence data. The model will read encoded characters and predict the next character in the sequence. A Long Short-Term Memory recurrent neural network hidden layer will be used to learn the context from the input sequence in order to make the predictions.

18.4.1 Load Data

The first step is to load the prepared character sequence data from `char_sequences.txt`. We can use the same `load_doc()` function developed in the previous section. Once loaded, we split

the text by new line to give a list of sequences ready to be encoded.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load
in_filename = 'char_sequences.txt'
raw_text = load_doc(in_filename)
lines = raw_text.split('\n')
```

Listing 18.11: Load the prepared text data.

18.4.2 Encode Sequences

The sequences of characters must be encoded as integers. This means that each unique character will be assigned a specific integer value and each sequence of characters will be encoded as a sequence of integers. We can create the mapping given a sorted set of unique characters in the raw input data. The mapping is a dictionary of character values to integer values.

```
chars = sorted(list(set(raw_text)))
mapping = dict((c, i) for i, c in enumerate(chars))
```

Listing 18.12: Create a mapping between chars and integers.

Next, we can process each sequence of characters one at a time and use the dictionary mapping to look up the integer value for each character.

```
sequences = list()
for line in lines:
    # integer encode line
    encoded_seq = [mapping[char] for char in line]
    # store
    sequences.append(encoded_seq)
```

Listing 18.13: Integer encode sequences of characters.

The result is a list of integer lists. We need to know the size of the vocabulary later. We can retrieve this as the size of the dictionary mapping.

```
# vocabulary size
vocab_size = len(mapping)
print('Vocabulary Size: %d' % vocab_size)
```

Listing 18.14: Summarize the size of the vocabulary.

Running this piece, we can see that there are 38 unique characters in the input sequence data.

```
Vocabulary Size: 38
```

Listing 18.15: Example output from summarizing the size of the vocabulary.

18.4.3 Split Inputs and Output

Now that the sequences have been integer encoded, we can separate the columns into input and output sequences of characters. We can do this using a simple array slice.

```
sequences = array(sequences)
X, y = sequences[:, :-1], sequences[:, -1]
```

Listing 18.16: Split sequences into input and output elements.

Next, we need to one hot encode each character. That is, each character becomes a vector as long as the vocabulary (38 elements) with a 1 marked for the specific character. This provides a more precise input representation for the network. It also provides a clear objective for the network to predict, where a probability distribution over characters can be output by the model and compared to the ideal case of all 0 values with a 1 for the actual next character. We can use the `to_categorical()` function in the Keras API to one hot encode the input and output sequences.

```
sequences = [to_categorical(x, num_classes=vocab_size) for x in X]
X = array(sequences)
y = to_categorical(y, num_classes=vocab_size)
```

Listing 18.17: Convert sequences into a format ready for training.

We are now ready to fit the model.

18.4.4 Fit Model

The model is defined with an input layer that takes sequences that have 10 time steps and 38 features for the one hot encoded input sequences. Rather than specify these numbers, we use the second and third dimensions on the X input data. This is so that if we change the length of the sequences or size of the vocabulary, we do not need to change the model definition. The model has a single LSTM hidden layer with 75 memory cells, chosen with a little trial and error. The model has a fully connected output layer that outputs one vector with a probability distribution across all characters in the vocabulary. A softmax activation function is used on the output layer to ensure the output has the properties of a probability distribution.

```
# define the model
def define_model(X):
    model = Sequential()
    model.add(LSTM(75, input_shape=(X.shape[1], X.shape[2])))
    model.add(Dense(vocab_size, activation='softmax'))
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model
```

Listing 18.18: Define the language model.

The model is learning a multiclass classification problem, therefore we use the categorical log loss intended for this type of problem. The efficient Adam implementation of gradient descent is used to optimize the model and accuracy is reported at the end of each batch update. The

model is fit for 100 training epochs, again found with a little trial and error. Running this prints a summary of the defined network as a sanity check.

```
-----  
Layer (type)          Output Shape         Param #  
=====-----  
lstm_1 (LSTM)        (None, 75)           34200  
-----  
dense_1 (Dense)      (None, 38)            2888  
-----  
Total params: 37,088  
Trainable params: 37,088  
Non-trainable params: 0  
-----
```

Listing 18.19: Example output from summarizing the defined model.

A plot the defined model is then saved to file with the name `model.png`.

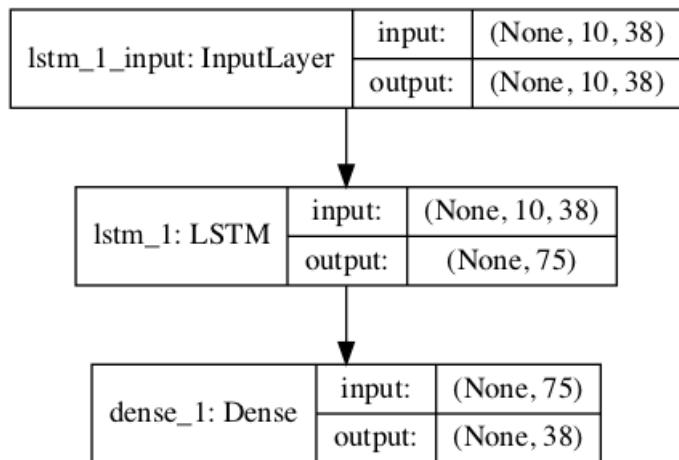


Figure 18.1: Plot of the defined character-based language model.

18.4.5 Save Model

After the model is fit, we save it to file for later use. The Keras model API provides the `save()` function that we can use to save the model to a single file, including weights and topology information.

```
# save the model to file
model.save('model.h5')
```

Listing 18.20: Save the fit model to file.

We also save the mapping from characters to integers that we will need to encode any input when using the model and decode any output from the model.

```
# save the mapping
dump(mapping, open('mapping.pkl', 'wb'))
```

Listing 18.21: Save the mapping of chars to integers to file.

18.4.6 Complete Example

Tying all of this together, the complete code listing for fitting the character-based neural language model is listed below.

```
from numpy import array
from pickle import dump
from keras.utils import to_categorical
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# define the model
def define_model(X):
    model = Sequential()
    model.add(LSTM(75, input_shape=(X.shape[1], X.shape[2])))
    model.add(Dense(vocab_size, activation='softmax'))
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model

# load
in_filename = 'char_sequences.txt'
raw_text = load_doc(in_filename)
lines = raw_text.split('\n')
# integer encode sequences of characters
chars = sorted(list(set(raw_text)))
mapping = dict((c, i) for i, c in enumerate(chars))
sequences = list()
for line in lines:
    # integer encode line
    encoded_seq = [mapping[char] for char in line]
    # store
    sequences.append(encoded_seq)
# vocabulary size
vocab_size = len(mapping)
print('Vocabulary Size: %d' % vocab_size)
# separate into input and output
sequences = array(sequences)
X, y = sequences[:, :-1], sequences[:, -1]
sequences = [to_categorical(x, num_classes=vocab_size) for x in X]
X = array(sequences)
```

```

y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(X)
# fit model
model.fit(X, y, epochs=100, verbose=2)
# save the model to file
model.save('model.h5')
# save the mapping
dump(mapping, open('mapping.pkl', 'wb'))

```

Listing 18.22: Complete example of training the language model.

Running the example might take one minute. You will see that the model learns the problem well, perhaps too well for generating surprising sequences of characters.

```

...
Epoch 96/100
0s - loss: 0.2193 - acc: 0.9950
Epoch 97/100
0s - loss: 0.2124 - acc: 0.9950
Epoch 98/100
0s - loss: 0.2054 - acc: 0.9950
Epoch 99/100
0s - loss: 0.1982 - acc: 0.9950
Epoch 100/100
0s - loss: 0.1910 - acc: 0.9950

```

Listing 18.23: Example output from training the language model.

At the end of the run, you will have two files saved to the current working directory, specifically `model.h5` and `mapping.pkl`. Next, we can look at using the learned model.

18.5 Generate Text

We will use the learned language model to generate new sequences of text that have the same statistical properties.

18.5.1 Load Model

The first step is to load the model saved to the file `model.h5`. We can use the `load_model()` function from the Keras API.

```

# load the model
model = load_model('model.h5')

```

Listing 18.24: Load the saved model.

We also need to load the pickled dictionary for mapping characters to integers from the file `mapping.pkl`. We will use the Pickle API to load the object.

```

# load the mapping
mapping = load(open('mapping.pkl', 'rb'))

```

Listing 18.25: Load the saved mapping from chars to integers.

We are now ready to use the loaded model.

18.5.2 Generate Characters

We must provide sequences of 10 characters as input to the model in order to start the generation process. We will pick these manually. A given input sequence will need to be prepared in the same way as preparing the training data for the model. First, the sequence of characters must be integer encoded using the loaded mapping.

```
# encode the characters as integers
encoded = [mapping[char] for char in in_text]
```

Listing 18.26: Encode input text to integers.

Next, the integers need to be one hot encoded using the `to_categorical()` Keras function. We also need to reshape the sequence to be 3-dimensional, as we only have one sequence and LSTMs require all input to be three dimensional (samples, time steps, features).

```
# one hot encode
encoded = to_categorical(encoded, num_classes=len(mapping))
encoded = encoded.reshape(1, encoded.shape[0], encoded.shape[1])
```

Listing 18.27: One hot encode the integer encoded text.

We can then use the model to predict the next character in the sequence. We use `predict_classes()` instead of `predict()` to directly select the integer for the character with the highest probability instead of getting the full probability distribution across the entire set of characters.

```
# predict character
yhat = model.predict_classes(encoded, verbose=0)
```

Listing 18.28: Predict the next character in the sequence.

We can then decode this integer by looking up the mapping to see the character to which it maps.

```
out_char = ''
for char, index in mapping.items():
    if index == yhat:
        out_char = char
        break
```

Listing 18.29: Map the predicted integer back to a character.

This character can then be added to the input sequence. We then need to make sure that the input sequence is 10 characters by truncating the first character from the input sequence text. We can use the `pad_sequences()` function from the Keras API that can perform this truncation operation. Putting all of this together, we can define a new function named `generate_seq()` for using the loaded model to generate new sequences of text.

```
# generate a sequence of characters with a language model
def generate_seq(model, mapping, seq_length, seed_text, n_chars):
    in_text = seed_text
    # generate a fixed number of characters
    for _ in range(n_chars):
        # encode the characters as integers
        encoded = [mapping[char] for char in in_text]
        # truncate sequences to a fixed length
```

```

encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
# one hot encode
encoded = to_categorical(encoded, num_classes=len(mapping))
encoded = encoded.reshape(1, encoded.shape[0], encoded.shape[1])
# predict character
yhat = model.predict_classes(encoded, verbose=0)
# reverse map integer to character
out_char = ''
for char, index in mapping.items():
    if index == yhat:
        out_char = char
        break
# append to input
in_text += char
return in_text

```

Listing 18.30: Function to predict a sequence of characters given seed text.

18.5.3 Complete Example

Tying all of this together, the complete example for generating text using the fit neural language model is listed below.

```

from pickle import load
from keras.models import load_model
from keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences

# generate a sequence of characters with a language model
def generate_seq(model, mapping, seq_length, seed_text, n_chars):
    in_text = seed_text
    # generate a fixed number of characters
    for _ in range(n_chars):
        # encode the characters as integers
        encoded = [mapping[char] for char in in_text]
        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
        # one hot encode
        encoded = to_categorical(encoded, num_classes=len(mapping))
        encoded = encoded.reshape(1, encoded.shape[0], encoded.shape[1])
        # predict character
        yhat = model.predict_classes(encoded, verbose=0)
        # reverse map integer to character
        out_char = ''
        for char, index in mapping.items():
            if index == yhat:
                out_char = char
                break
        # append to input
        in_text += out_char
    return in_text

# load the model
model = load_model('model.h5')
# load the mapping

```

```

mapping = load(open('mapping.pkl', 'rb'))
# test start of rhyme
print(generate_seq(model, mapping, 10, 'Sing a son', 20))
# test mid-line
print(generate_seq(model, mapping, 10, 'king was i', 20))
# test not in original
print(generate_seq(model, mapping, 10, 'hello worl', 20))

```

Listing 18.31: Complete example of generating characters with the fit model.

Running the example generates three sequences of text. The first is a test to see how the model does at starting from the beginning of the rhyme. The second is a test to see how well it does at beginning in the middle of a line. The final example is a test to see how well it does with a sequence of characters never seen before.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```

Sing a song of sixpence, A poc
king was in his counting house
hello worls e pake wofey. The

```

Listing 18.32: Example output from generating sequences of characters.

We can see that the model did very well with the first two examples, as we would expect. We can also see that the model still generated something for the new text, but it is nonsense.

18.6 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Sing a Song of Sixpence on Wikipedia.
https://en.wikipedia.org/wiki/Sing_a_Song_of_Sixpence
- Keras Utils API.
<https://keras.io/utils/>
- Keras Sequence Processing API.
<https://keras.io/preprocessing/sequence/>

18.7 Summary

In this tutorial, you discovered how to develop a character-based neural language model. Specifically, you learned:

- How to prepare text for character-based language modeling.
- How to develop a character-based language model using LSTMs.
- How to use a trained character-based language model to generate text.

18.7.1 Next

In the next chapter, you will discover how you can develop a word-based neural language model.

Chapter 19

How to Develop a Word-Based Neural Language Model

Language modeling involves predicting the next word in a sequence given the sequence of words already present. A language model is a key element in many natural language processing models such as machine translation and speech recognition. The choice of how the language model is framed must match how the language model is intended to be used. In this tutorial, you will discover how the framing of a language model affects the skill of the model when generating short sequences from a nursery rhyme. After completing this tutorial, you will know:

- The challenge of developing a good framing of a word-based language model for a given application.
- How to develop one-word, two-word, and line-based framings for word-based language models.
- How to generate sequences using a fit language model.

Let's get started.

19.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Framing Language Modeling
2. Jack and Jill Nursery Rhyme
3. Model 1: One-Word-In, One-Word-Out Sequences
4. Model 2: Line-by-Line Sequence
5. Model 3: Two-Words-In, One-Word-Out Sequence

19.2 Framing Language Modeling

A statistical language model is learned from raw text and predicts the probability of the next word in the sequence given the words already present in the sequence. Language models are a key component in larger models for challenging natural language processing problems, like machine translation and speech recognition. They can also be developed as standalone models and used for generating new sequences that have the same statistical properties as the source text.

Language models both learn and predict one word at a time. The training of the network involves providing sequences of words as input that are processed one at a time where a prediction can be made and learned for each input sequence. Similarly, when making predictions, the process can be seeded with one or a few words, then predicted words can be gathered and presented as input on subsequent predictions in order to build up a generated output sequence.

Therefore, each model will involve splitting the source text into input and output sequences, such that the model can learn to predict words. There are many ways to frame the sequences from a source text for language modeling. In this tutorial, we will explore 3 different ways of developing word-based language models in the Keras deep learning library. There is no single best approach, just different framings that may suit different applications.

19.3 Jack and Jill Nursery Rhyme

Jack and Jill is a simple nursery rhyme. It is comprised of 4 lines, as follows:

```
Jack and Jill went up the hill
To fetch a pail of water
Jack fell down and broke his crown
And Jill came tumbling after
```

Listing 19.1: *Jack and Jill* nursery rhyme.

We will use this as our source text for exploring different framings of a word-based language model. We can define this text in Python as follows:

```
# source text
data = """ Jack and Jill went up the hill\n
    To fetch a pail of water\n
    Jack fell down and broke his crown\n
    And Jill came tumbling after\n """
```

Listing 19.2: Sample text for this tutorial.

19.4 Model 1: One-Word-In, One-Word-Out Sequences

We can start with a very simple model. Given one word as input, the model will learn to predict the next word in the sequence. For example:

```
x,      y
Jack,   and
and,   Jill
Jill,  went
```

...

Listing 19.3: Example of input and output pairs.

The first step is to encode the text as integers. Each lowercase word in the source text is assigned a unique integer and we can convert the sequences of words to sequences of integers. Keras provides the `Tokenizer` class that can be used to perform this encoding. First, the `Tokenizer` is fit on the source text to develop the mapping from words to unique integers. Then sequences of text can be converted to sequences of integers by calling the `texts_to_sequences()` function.

```
# integer encode text
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
encoded = tokenizer.texts_to_sequences([data])[0]
```

Listing 19.4: Example of training a `Tokenizer` on the sample text.

We will need to know the size of the vocabulary later for both defining the word embedding layer in the model, and for encoding output words using a one hot encoding. The size of the vocabulary can be retrieved from the trained `Tokenizer` by accessing the `word_index` attribute.

```
# determine the vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
```

Listing 19.5: Summarize the size of the vocabulary.

Running this example, we can see that the size of the vocabulary is 21 words. We add one, because we will need to specify the integer for the largest encoded word as an array index, e.g. words encoded 1 to 21 with array indices 0 to 21 or 22 positions. Next, we need to create sequences of words to fit the model with one word as input and one word as output.

```
# create word -> word sequences
sequences = list()
for i in range(1, len(encoded)):
    sequence = encoded[i-1:i+1]
    sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
```

Listing 19.6: Example of encoding the source text.

Running this piece shows that we have a total of 24 input-output pairs to train the network.

```
Total Sequences: 24
```

Listing 19.7: Example of output of summarizing the encoded text.

We can then split the sequences into input (`X`) and output elements (`y`). This is straightforward as we only have two columns in the data.

```
# split into X and y elements
sequences = array(sequences)
X, y = sequences[:,0], sequences[:,1]
```

Listing 19.8: Split the encoded text into input and output pairs.

We will fit our model to predict a probability distribution across all words in the vocabulary. That means that we need to turn the output element from a single integer into a one hot encoding with a 0 for every word in the vocabulary and a 1 for the actual word that the value. This gives the network a ground truth to aim for from which we can calculate error and update the model. Keras provides the `to_categorical()` function that we can use to convert the integer to a one hot encoding while specifying the number of classes as the vocabulary size.

```
# one hot encode outputs
y = to_categorical(y, num_classes=vocab_size)
```

Listing 19.9: One hot encode the output words.

We are now ready to define the neural network model. The model uses a learned word embedding in the input layer. This has one real-valued vector for each word in the vocabulary, where each word vector has a specified length. In this case we will use a 10-dimensional projection. The input sequence contains a single word, therefore the `input_length=1`. The model has a single hidden LSTM layer with 50 units. This is far more than is needed. The output layer is comprised of one neuron for each word in the vocabulary and uses a softmax activation function to ensure the output is normalized to look like a probability.

```
# define the model
def define_model(vocab_size):
    model = Sequential()
    model.add(Embedding(vocab_size, 10, input_length=1))
    model.add(LSTM(50))
    model.add(Dense(vocab_size, activation='softmax'))
    # compile network
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model
```

Listing 19.10: Define and compile the language model.

The structure of the network can be summarized as follows:

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 1, 10)	220
lstm_1 (LSTM)	(None, 50)	12200
dense_1 (Dense)	(None, 22)	1122
Total params:	13,542	
Trainable params:	13,542	
Non-trainable params:	0	

Listing 19.11: Example output summarizing the defined model.

A plot the defined model is then saved to file with the name `model.png`.

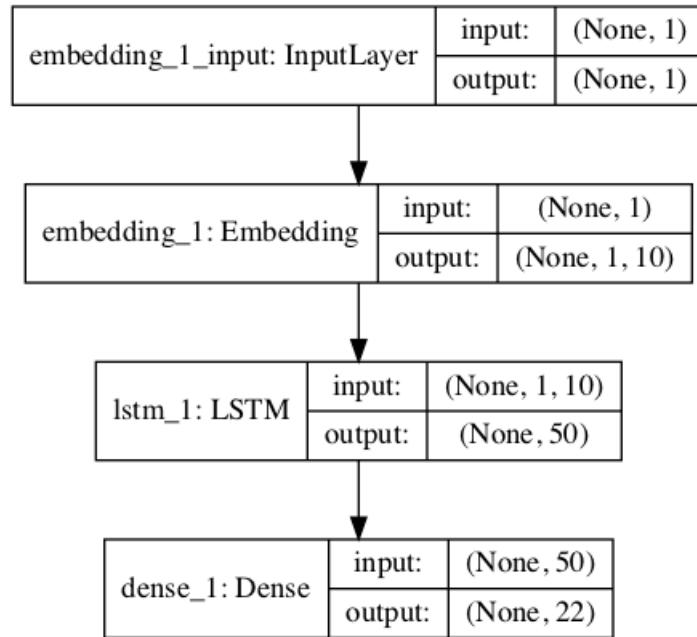


Figure 19.1: Plot of the defined word-based language model.

We will use this same general network structure for each example in this tutorial, with minor changes to the learned embedding layer. We can compile and fit the network on the encoded text data. Technically, we are modeling a multiclass classification problem (predict the word in the vocabulary), therefore using the categorical cross entropy loss function. We use the efficient Adam implementation of gradient descent and track accuracy at the end of each epoch. The model is fit for 500 training epochs, again, perhaps more than is needed. The network configuration was not tuned for this and later experiments; an over-prescribed configuration was chosen to ensure that we could focus on the framing of the language model.

After the model is fit, we test it by passing it a given word from the vocabulary and having the model predict the next word. Here we pass in ‘Jack’ by encoding it and calling `model.predict_classes()` to get the integer output for the predicted word. This is then looked up in the vocabulary mapping to give the associated word.

```
# evaluate
in_text = 'Jack'
print(in_text)
encoded = tokenizer.texts_to_sequences([in_text])[0]
encoded = array(encoded)
yhat = model.predict_classes(encoded, verbose=0)
for word, index in tokenizer.word_index.items():
    if index == yhat:
        print(word)
```

Listing 19.12: Evaluate the fit language model.

This process could then be repeated a few times to build up a generated sequence of words. To make this easier, we wrap up the behavior in a function that we can call by passing in our model and the seed word.

```
# generate a sequence from the model
def generate_seq(model, tokenizer, seed_text, n_words):
```

```

in_text, result = seed_text, seed_text
# generate a fixed number of words
for _ in range(n_words):
    # encode the text as integer
    encoded = tokenizer.texts_to_sequences([in_text])[0]
    encoded = array(encoded)
    # predict a word in the vocabulary
    yhat = model.predict_classes(encoded, verbose=0)
    # map predicted word index to word
    out_word = ''
    for word, index in tokenizer.word_index.items():
        if index == yhat:
            out_word = word
            break
    # append to input
    in_text, result = out_word, result + ' ' + out_word
return result

```

Listing 19.13: Function to generate output sequences given a fit model.

We can tie all of this together. The complete code listing is provided below.

```

from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.utils import to_categorical
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding

# generate a sequence from the model
def generate_seq(model, tokenizer, seed_text, n_words):
    in_text, result = seed_text, seed_text
    # generate a fixed number of words
    for _ in range(n_words):
        # encode the text as integer
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        encoded = array(encoded)
        # predict a word in the vocabulary
        yhat = model.predict_classes(encoded, verbose=0)
        # map predicted word index to word
        out_word = ''
        for word, index in tokenizer.word_index.items():
            if index == yhat:
                out_word = word
                break
        # append to input
        in_text, result = out_word, result + ' ' + out_word
    return result

# define the model
def define_model(vocab_size):
    model = Sequential()
    model.add(Embedding(vocab_size, 10, input_length=1))
    model.add(LSTM(50))
    model.add(Dense(vocab_size, activation='softmax'))

```

```

# compile network
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# summarize defined model
model.summary()
plot_model(model, to_file='model.png', show_shapes=True)
return model

# source text
data = """ Jack and Jill went up the hill\n
    To fetch a pail of water\n
    Jack fell down and broke his crown\n
    And Jill came tumbling after\n """
# integer encode text
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
encoded = tokenizer.texts_to_sequences([data])[0]
# determine the vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# create word -> word sequences
sequences = list()
for i in range(1, len(encoded)):
    sequence = encoded[i-1:i+1]
    sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
# split into X and y elements
sequences = array(sequences)
X, y = sequences[:,0], sequences[:,1]
# one hot encode outputs
y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(vocab_size)
# fit network
model.fit(X, y, epochs=500, verbose=2)
# evaluate
print(generate_seq(model, tokenizer, 'Jack', 6))

```

Listing 19.14: Complete example of model1.

Running the example prints the loss and accuracy each training epoch.

```

...
Epoch 496/500
0s - loss: 0.2358 - acc: 0.8750
Epoch 497/500
0s - loss: 0.2355 - acc: 0.8750
Epoch 498/500
0s - loss: 0.2352 - acc: 0.8750
Epoch 499/500
0s - loss: 0.2349 - acc: 0.8750
Epoch 500/500
0s - loss: 0.2346 - acc: 0.8750

```

Listing 19.15: Example output of fitting the language model.

We can see that the model does not memorize the source sequences, likely because there is some ambiguity in the input sequences, for example:

```
jack => and
jack => fell
```

Listing 19.16: Example output of predicting the next word.

And so on. At the end of the run, *Jack* is passed in and a prediction or new sequence is generated. We get a reasonable sequence as output that has some elements of the source.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
Jack and jill came tumbling after down
```

Listing 19.17: Example output of predicting a sequence of words.

This is a good first cut language model, but does not take full advantage of the LSTM's ability to handle sequences of input and disambiguate some of the ambiguous pairwise sequences by using a broader context.

19.5 Model 2: Line-by-Line Sequence

Another approach is to split up the source text line-by-line, then break each line down into a series of words that build up. For example:

x,	y
-, -, -, -, -, Jack,	and
-, -, -, -, Jack, and,	Jill
-, -, -, Jack, and, Jill,	went
-, -, Jack, and, Jill, went,	up
-, Jack, and, Jill, went, up,	the
Jack, and, Jill, went, up, the,	hill

Listing 19.18: Example framing of the problem as sequences of words.

This approach may allow the model to use the context of each line to help the model in those cases where a simple one-word-in-and-out model creates ambiguity. In this case, this comes at the cost of predicting words across lines, which might be fine for now if we are only interested in modeling and generating lines of text. Note that in this representation, we will require a padding of sequences to ensure they meet a fixed length input. This is a requirement when using Keras. First, we can create the sequences of integers, line-by-line by using the `Tokenizer` already fit on the source text.

```
# create line-based sequences
sequences = list()
for line in data.split('\n'):
    encoded = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(encoded)):
        sequence = encoded[:i+1]
        sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
```

Listing 19.19: Example of preparing sequences of words.

Next, we can pad the prepared sequences. We can do this using the `pad_sequences()` function provided in Keras. This first involves finding the longest sequence, then using that as the length by which to pad-out all other sequences.

```
# pad input sequences
max_length = max([len(seq) for seq in sequences])
sequences = pad_sequences(sequences, maxlen=max_length, padding='pre')
print('Max Sequence Length: %d' % max_length)
```

Listing 19.20: Example of padding sequences of words.

Next, we can split the sequences into input and output elements, much like before.

```
# split into input and output elements
sequences = array(sequences)
X, y = sequences[:, :-1], sequences[:, -1]
y = to_categorical(y, num_classes=vocab_size)
```

Listing 19.21: Example of preparing the input and output sequences.

The model can then be defined as before, except the input sequences are now longer than a single word. Specifically, they are `max_length-1` in length, -1 because when we calculated the maximum length of sequences, they included the input and output elements.

```
# define the model
def define_model(vocab_size, max_length):
    model = Sequential()
    model.add(Embedding(vocab_size, 10, input_length=max_length-1))
    model.add(LSTM(50))
    model.add(Dense(vocab_size, activation='softmax'))
    # compile network
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model
```

Listing 19.22: Define and compile the language model.

We can use the model to generate new sequences as before. The `generate_seq()` function can be updated to build up an input sequence by adding predictions to the list of input words each iteration.

```
# generate a sequence from a language model
def generate_seq(model, tokenizer, max_length, seed_text, n_words):
    in_text = seed_text
    # generate a fixed number of words
    for _ in range(n_words):
        # encode the text as integer
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        # pre-pad sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=max_length, padding='pre')
        # predict probabilities for each word
        yhat = model.predict_classes(encoded, verbose=0)
        # map predicted word index to word
        out_word = ''
        for word, index in tokenizer.word_index.items():
            if index == yhat:
```

```

        out_word = word
        break
    # append to input
    in_text += ' ' + out_word
return in_text

```

Listing 19.23: Function to generate sequences of words given input text.

Tying all of this together, the complete code example is provided below.

```

from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding

# generate a sequence from a language model
def generate_seq(model, tokenizer, max_length, seed_text, n_words):
    in_text = seed_text
    # generate a fixed number of words
    for _ in range(n_words):
        # encode the text as integer
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        # pre-pad sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=max_length, padding='pre')
        # predict probabilities for each word
        yhat = model.predict_classes(encoded, verbose=0)
        # map predicted word index to word
        out_word = ''
        for word, index in tokenizer.word_index.items():
            if index == yhat:
                out_word = word
                break
        # append to input
        in_text += ' ' + out_word
    return in_text

# define the model
def define_model(vocab_size, max_length):
    model = Sequential()
    model.add(Embedding(vocab_size, 10, input_length=max_length-1))
    model.add(LSTM(50))
    model.add(Dense(vocab_size, activation='softmax'))
    # compile network
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model

# source text
data = """ Jack and Jill went up the hill\n
To fetch a pail of water\n

```

```

Jack fell down and broke his crown\n
And Jill came tumbling after\n """
# prepare the tokenizer on the source text
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
# determine the vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# create line-based sequences
sequences = list()
for line in data.split('\n'):
    encoded = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(encoded)):
        sequence = encoded[:i+1]
        sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
# pad input sequences
max_length = max([len(seq) for seq in sequences])
sequences = pad_sequences(sequences, maxlen=max_length, padding='pre')
print('Max Sequence Length: %d' % max_length)
# split into input and output elements
sequences = array(sequences)
X, y = sequences[:, :-1], sequences[:, -1]
y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(vocab_size, max_length)
# fit network
model.fit(X, y, epochs=500, verbose=2)
# evaluate model
print(generate_seq(model, tokenizer, max_length-1, 'Jack', 4))
print(generate_seq(model, tokenizer, max_length-1, 'Jill', 4))

```

Listing 19.24: Complete example of model2.

Running the example achieves a better fit on the source data. The added context has allowed the model to disambiguate some of the examples. There are still two lines of text that start with “*Jack*” that may still be a problem for the network.

```

...
Epoch 496/500
0s - loss: 0.1039 - acc: 0.9524
Epoch 497/500
0s - loss: 0.1037 - acc: 0.9524
Epoch 498/500
0s - loss: 0.1035 - acc: 0.9524
Epoch 499/500
0s - loss: 0.1033 - acc: 0.9524
Epoch 500/500
0s - loss: 0.1032 - acc: 0.9524

```

Listing 19.25: Example output of fitting the language model.

At the end of the run, we generate two sequences with different seed words: *Jack* and *Jill*.

The first generated line looks good, directly matching the source text. The second is a bit strange. This makes sense, because the network only ever saw *Jill* within an input sequence, not at the beginning of the sequence, so it has forced an output to use the word *Jill*, i.e. the

last line of the rhyme.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
Jack fell down and broke
Jill jill came tumbling after
```

Listing 19.26: Example output of generating sequences of words.

This was a good example of how the framing may result in better new lines, but not good partial lines of input.

19.6 Model 3: Two-Words-In, One-Word-Out Sequence

We can use an intermediate between the one-word-in and the whole-sentence-in approaches and pass in a sub-sequences of words as input. This will provide a trade-off between the two framings allowing new lines to be generated and for generation to be picked up mid line. We will use 3 words as input to predict one word as output. The preparation of the sequences is much like the first example, except with different offsets in the source sequence arrays, as follows:

```
# encode 2 words -> 1 word
sequences = list()
for i in range(2, len(encoded)):
    sequence = encoded[i-2:i+1]
    sequences.append(sequence)
```

Listing 19.27: Example of preparing constrained sequence data.

The complete example is listed below

```
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding

# generate a sequence from a language model
def generate_seq(model, tokenizer, max_length, seed_text, n_words):
    in_text = seed_text
    # generate a fixed number of words
    for _ in range(n_words):
        # encode the text as integer
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        # pre-pad sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=max_length, padding='pre')
        # predict probabilities for each word
        yhat = model.predict_classes(encoded, verbose=0)
        # map predicted word index to word
        out_word = ''
```

```

for word, index in tokenizer.word_index.items():
    if index == yhat:
        out_word = word
        break
    # append to input
    in_text += ' ' + out_word
return in_text

# define the model
def define_model(vocab_size, max_length):
    model = Sequential()
    model.add(Embedding(vocab_size, 10, input_length=max_length-1))
    model.add(LSTM(50))
    model.add(Dense(vocab_size, activation='softmax'))
    # compile network
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model

# source text
data = """ Jack and Jill went up the hill\n
    To fetch a pail of water\n
    Jack fell down and broke his crown\n
    And Jill came tumbling after """
# integer encode sequences of words
tokenizer = Tokenizer()
tokenizer.fit_on_texts([data])
encoded = tokenizer.texts_to_sequences([data])[0]
# retrieve vocabulary size
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# encode 2 words -> 1 word
sequences = list()
for i in range(2, len(encoded)):
    sequence = encoded[i-2:i+1]
    sequences.append(sequence)
print('Total Sequences: %d' % len(sequences))
# pad sequences
max_length = max([len(seq) for seq in sequences])
sequences = pad_sequences(sequences, maxlen=max_length, padding='pre')
print('Max Sequence Length: %d' % max_length)
# split into input and output elements
sequences = array(sequences)
X, y = sequences[:, :-1], sequences[:, -1]
y = to_categorical(y, num_classes=vocab_size)
# define model
model = define_model(vocab_size, max_length)
# fit network
model.fit(X, y, epochs=500, verbose=2)
# evaluate model
print(generate_seq(model, tokenizer, max_length-1, 'Jack and', 5))
print(generate_seq(model, tokenizer, max_length-1, 'And Jill', 3))
print(generate_seq(model, tokenizer, max_length-1, 'fell down', 5))
print(generate_seq(model, tokenizer, max_length-1, 'pail of', 5))

```

Listing 19.28: Complete example of model3.

Running the example again gets a good fit on the source text at around 95% accuracy.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
...
Epoch 496/500
0s - loss: 0.0685 - acc: 0.9565
Epoch 497/500
0s - loss: 0.0685 - acc: 0.9565
Epoch 498/500
0s - loss: 0.0684 - acc: 0.9565
Epoch 499/500
0s - loss: 0.0684 - acc: 0.9565
Epoch 500/500
0s - loss: 0.0684 - acc: 0.9565
```

Listing 19.29: Example output of fitting the language model.

We look at 4 generation examples, two start of line cases and two starting mid line.

```
Jack and jill went up the hill
And Jill went up the
fell down and broke his crown and
pail of water jack fell down and
```

Listing 19.30: Example output of generating sequences of words.

The first start of line case generated correctly, but the second did not. The second case was an example from the 4th line, which is ambiguous with content from the first line. Perhaps a further expansion to 3 input words would be better. The two mid-line generation examples were generated correctly, matching the source text.

We can see that the choice of how the language model is framed and the requirements on how the model will be used must be compatible. That careful design is required when using language models in general, perhaps followed-up by spot testing with sequence generation to confirm model requirements have been met.

19.7 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Jack and Jill on Wikipedia.
[https://en.wikipedia.org/wiki/Jack_and_Jill_\(nursery_rhyme\)](https://en.wikipedia.org/wiki/Jack_and_Jill_(nursery_rhyme))
- Language Model on Wikipedia.
https://en.wikipedia.org/wiki/Language_model
- Keras Embedding Layer API.
<https://keras.io/layers/embeddings/#embedding>

- Keras Text Processing API.
<https://keras.io/preprocessing/text/>
- Keras Sequence Processing API.
<https://keras.io/preprocessing/sequence/>
- Keras Utils API.
<https://keras.io/utils/>

19.8 Summary

In this tutorial, you discovered how to develop different word-based language models for a simple nursery rhyme. Specifically, you learned:

- The challenge of developing a good framing of a word-based language model for a given application.
- How to develop one-word, two-word, and line-based framings for word-based language models.
- How to generate sequences using a fit language model.

19.8.1 Next

In the next chapter, you will discover how you can develop a word-based neural language model on a large corpus of text.

Chapter 20

Project: Develop a Neural Language Model for Text Generation

A language model can predict the probability of the next word in the sequence, based on the words already observed in the sequence. Neural network models are a preferred method for developing statistical language models because they can use a distributed representation where different words with similar meanings have similar representation and because they can use a large context of recently observed words when making predictions. In this tutorial, you will discover how to develop a statistical language model using deep learning in Python. After completing this tutorial, you will know:

- How to prepare text for developing a word-based language model.
- How to design and fit a neural language model with a learned embedding and an LSTM hidden layer.
- How to use the learned language model to generate new text with similar statistical properties as the source text.

Let's get started.

20.1 Tutorial Overview

This tutorial is divided into the following parts:

1. The Republic by Plato
2. Data Preparation
3. Train Language Model
4. Use Language Model

20.2 The Republic by Plato

The Republic is the classical Greek philosopher Plato's most famous work. It is structured as a dialog (e.g. conversation) on the topic of order and justice within a city state. The entire text is available for free in the public domain. It is available on the Project Gutenberg website in a number of formats. You can download the ASCII text version of the entire book (or books) here (you might need to open the URL twice):

- Download The Republic by Plato.
<http://www.gutenberg.org/cache/epub/1497/pg1497.txt>

Download the book text and place it in your current working directly with the filename `republic.txt`. Open the file in a text editor and delete the front and back matter. This includes details about the book at the beginning, a long analysis, and license information at the end. The text should begin with:

BOOK I.

I went down yesterday to the Piraeus with Glaucon the son of Ariston, ...

And end with:

... And it shall be well with us both in this life and in the pilgrimage of a thousand years which we have been describing.

Save the cleaned version as `republic_clean.txt` in your current working directory. The file should be about 15,802 lines of text. Now we can develop a language model from this text.

20.3 Data Preparation

We will start by preparing the data for modeling. The first step is to look at the data.

20.3.1 Review the Text

Open the text in an editor and just look at the text data. For example, here is the first piece of dialog:

BOOK I.

I went down yesterday to the Piraeus with Glaucon the son of Ariston, that I might offer up my prayers to the goddess (Bendis, the Thracian Artemis.); and also because I wanted to see in what manner they would celebrate the festival, which was a new thing. I was delighted with the procession of the inhabitants; but that of the Thracians was equally, if not more, beautiful. When we had finished our prayers and viewed the spectacle, we turned in the direction of the city; and at that instant Polemarchus the son of Cephalus chanced to catch sight of us from a distance as we were starting on our way home, and told his servant to run and bid us wait for him. The servant took hold of me by the cloak behind, and said: Polemarchus desires you to wait.

I turned round, and asked him where his master was.

There he is, said the youth, coming after you, if you will only wait.

Certainly we will, said Glaucon; and in a few minutes Polemarchus appeared, and with him Adeimantus, Glaucon's brother, Niceratus the son of Nicias, and several others who had been at the procession.

Polemarchus said to me: I perceive, Socrates, that you and your companion are already on your way to the city.

You are not far wrong, I said.

...

What do you see that we will need to handle in preparing the data? Here's what I see from a quick look:

- Book/Chapter headings (e.g. *BOOK I.*).
- Lots of punctuation (e.g. -, ;-, ?-, and more).
- Strange names (e.g. *Polemarchus*).
- Some long monologues that go on for hundreds of lines.
- Some quoted dialog (e.g. ‘...’).

These observations, and more, suggest at ways that we may wish to prepare the text data. The specific way we prepare the data really depends on how we intend to model it, which in turn depends on how we intend to use it.

20.3.2 Language Model Design

In this tutorial, we will develop a model of the text that we can then use to generate new sequences of text. The language model will be statistical and will predict the probability of each word given an input sequence of text. The predicted word will be fed in as input to in turn generate the next word. A key design decision is how long the input sequences should be. They need to be long enough to allow the model to learn the context for the words to predict. This input length will also define the length of seed text used to generate new sequences when we use the model.

There is no correct answer. With enough time and resources, we could explore the ability of the model to learn with differently sized input sequences. Instead, we will pick a length of 50 words for the length of the input sequences, somewhat arbitrarily. We could process the data so that the model only ever deals with self-contained sentences and pad or truncate the text to meet this requirement for each input sequence. You could explore this as an extension to this tutorial.

Instead, to keep the example brief, we will let all of the text flow together and train the model to predict the next word across sentences, paragraphs, and even books or chapters in the text. Now that we have a model design, we can look at transforming the raw text into sequences of 100 input words to 1 output word, ready to fit a model.

20.3.3 Load Text

The first step is to load the text into memory. We can develop a small function to load the entire text file into memory and return it. The function is called `load_doc()` and is listed below. Given a filename, it returns a sequence of loaded text.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text
```

Listing 20.1: Function to load text into memory.

Using this function, we can load the cleaner version of the document in the file `republic_clean.txt` as follows:

```
# load document
in_filename = 'republic_clean.txt'
doc = load_doc(in_filename)
print(doc[:200])
```

Listing 20.2: Example of loading the text into memory.

Running this snippet loads the document and prints the first 200 characters as a sanity check.

BOOK I.

I went down yesterday to the Piraeus with Glaucon the son of Ariston, that I might offer up my prayers to the goddess (Bendis, the Thracian Artemis.); and also because I wanted to see in what

Listing 20.3: Example output of loading the text into memory.

So far, so good. Next, let's clean the text.

20.3.4 Clean Text

We need to transform the raw text into a sequence of tokens or words that we can use as a source to train the model. Based on reviewing the raw text (above), below are some specific operations we will perform to clean the text. You may want to explore more cleaning operations yourself as an extension.

- Replace ‘-’ with a white space so we can split words better.
- Split words based on white space.
- Remove all punctuation from words to reduce the vocabulary size (e.g. ‘What?’ becomes ‘What’).
- Remove all words that are not alphabetic to remove standalone punctuation tokens.

- Normalize all words to lowercase to reduce the vocabulary size.

Vocabulary size is a big deal with language modeling. A smaller vocabulary results in a smaller model that trains faster. We can implement each of these cleaning operations in this order in a function. Below is the function `clean_doc()` that takes a loaded document as an argument and returns an array of clean tokens.

```
# turn a doc into clean tokens
def clean_doc(doc):
    # replace '--' with a space ''
    doc = doc.replace('--', ' ')
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[' + string.punctuation + ']')
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # make lower case
    tokens = [word.lower() for word in tokens]
    return tokens
```

Listing 20.4: Function to clean text.

We can run this cleaning operation on our loaded document and print out some of the tokens and statistics as a sanity check.

```
# clean document
tokens = clean_doc(doc)
print(tokens[:200])
print('Total Tokens: %d' % len(tokens))
print('Unique Tokens: %d' % len(set(tokens)))
```

Listing 20.5: Example of cleaning text.

First, we can see a nice list of tokens that look cleaner than the raw text. We could remove the 'Book I' chapter markers and more, but this is a good start.

```
['book', 'i', 'i', 'went', 'down', 'yesterday', 'to', 'the', 'piraeus', 'with', 'glaucon',
 'the', 'son', 'of', 'ariston', 'that', 'i', 'might', 'offer', 'up', 'my', 'prayers',
 'to', 'the', 'goddess', 'bendis', 'the', 'thracian', 'artemis', 'and', 'also',
 'because', 'i', 'wanted', 'to', 'see', 'in', 'what', 'manner', 'they', 'would',
 'celebrate', 'the', 'festival', 'which', 'was', 'a', 'new', 'thing', 'i', 'was',
 'delighted', 'with', 'the', 'procession', 'of', 'the', 'inhabitants', 'but', 'that',
 'of', 'the', 'thracians', 'was', 'equally', 'if', 'not', 'more', 'beautiful', 'when',
 'we', 'had', 'finished', 'our', 'prayers', 'and', 'viewed', 'the', 'spectacle', 'we',
 'turned', 'in', 'the', 'direction', 'of', 'the', 'city', 'and', 'at', 'that',
 'instant', 'polemarchus', 'the', 'son', 'of', 'cephalus', 'chanced', 'to', 'catch',
 'sight', 'of', 'us', 'from', 'a', 'distance', 'as', 'we', 'were', 'starting', 'on',
 'our', 'way', 'home', 'and', 'told', 'his', 'servant', 'to', 'run', 'and', 'bid', 'us',
 'wait', 'for', 'him', 'the', 'servant', 'took', 'hold', 'of', 'me', 'by', 'the',
 'cloak', 'behind', 'and', 'said', 'polemarchus', 'desires', 'you', 'to', 'wait', 'i',
 'turned', 'round', 'and', 'asked', 'him', 'where', 'his', 'master', 'was', 'there',
 'he', 'is', 'said', 'the', 'youth', 'coming', 'after', 'you', 'if', 'you', 'will',
 'only', 'wait', 'certainly', 'we', 'will', 'said', 'glaucon', 'and', 'in', 'a', 'few',
 'minutes', 'polemarchus', 'appeared', 'and', 'with', 'him', 'adeimantus', 'glaucons',
```

```
'brother', 'niceratus', 'the', 'son', 'of', 'nicias', 'and', 'several', 'others',
'who', 'had', 'been', 'at', 'the', 'procession', 'polemarchus', 'said']
```

Listing 20.6: Example output of tokenized and clean text.

We also get some statistics about the clean document. We can see that there are just under 120,000 words in the clean text and a vocabulary of just under 7,500 words. This is smallish and models fit on this data should be manageable on modest hardware.

```
Total Tokens: 118684
Unique Tokens: 7409
```

Listing 20.7: Example output summarizing properties of the clean text.

Next, we can look at shaping the tokens into sequences and saving them to file.

20.3.5 Save Clean Text

We can organize the long list of tokens into sequences of 50 input words and 1 output word. That is, sequences of 51 words. We can do this by iterating over the list of tokens from token 51 onwards and taking the prior 50 tokens as a sequence, then repeating this process to the end of the list of tokens. We will transform the tokens into space-separated strings for later storage in a file. The code to split the list of clean tokens into sequences with a length of 51 tokens is listed below.

```
# organize into sequences of tokens
length = 50 + 1
sequences = list()
for i in range(length, len(tokens)):
    # select sequence of tokens
    seq = tokens[i-length:i]
    # convert into a line
    line = ' '.join(seq)
    # store
    sequences.append(line)
print('Total Sequences: %d' % len(sequences))
```

Listing 20.8: Split document into sequences of text.

Running this piece creates a long list of lines. Printing statistics on the list, we can see that we will have exactly 118,633 training patterns to fit our model.

```
Total Sequences: 118633
```

Listing 20.9: Example output of splitting the document into sequences.

Next, we can save the sequences to a new file for later loading. We can define a new function for saving lines of text to a file. This new function is called `save_doc()` and is listed below. It takes as input a list of lines and a filename. The lines are written, one per line, in ASCII format.

```
# save tokens to file, one dialog per line
def save_doc(lines, filename):
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()
```

Listing 20.10: Function to save sequences of text to file.

We can call this function and save our training sequences to the file `republic_sequences.txt`.

```
# save sequences to file
out_filename = 'republic_sequences.txt'
save_doc(sequences, out_filename)
```

Listing 20.11: Example of saving sequences to file.

Take a look at the file with your text editor. You will see that each line is shifted along one word, with a new word at the end to be predicted; for example, here are the first 3 lines in truncated form:

```
book i i ... catch sight of
i i went ... sight of us
i went down ... of us from
...
```

Listing 20.12: Example contents of sequences saved to file.

20.3.6 Complete Example

Tying all of this together, the complete code listing is provided below.

```
import string
import re

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    # replace '--' with a space ' '
    doc = doc.replace('--', ' ')
    # split into tokens by white space
    tokens = doc.split()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub(' ', w) for w in tokens]
    # remove remaining tokens that are not alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # make lower case
    tokens = [word.lower() for word in tokens]
    return tokens

# save tokens to file, one dialog per line
```

```

def save_doc(lines, filename):
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()

# load document
in_filename = 'republic_clean.txt'
doc = load_doc(in_filename)
print(doc[:200])
# clean document
tokens = clean_doc(doc)
print(tokens[:200])
print('Total Tokens: %d' % len(tokens))
print('Unique Tokens: %d' % len(set(tokens)))
# organize into sequences of tokens
length = 50 + 1
sequences = list()
for i in range(length, len(tokens)):
    # select sequence of tokens
    seq = tokens[i-length:i]
    # convert into a line
    line = ' '.join(seq)
    # store
    sequences.append(line)
print('Total Sequences: %d' % len(sequences))
# save sequences to file
out_filename = 'republic_sequences.txt'
save_doc(sequences, out_filename)

```

Listing 20.13: Complete example preparing text data for modeling.

You should now have training data stored in the file `republic_sequences.txt` in your current working directory. Next, let's look at how to fit a language model to this data.

20.4 Train Language Model

We can now train a statistical language model from the prepared data. The model we will train is a neural language model. It has a few unique characteristics:

- It uses a distributed representation for words so that different words with similar meanings will have a similar representation.
- It learns the representation at the same time as learning the model.
- It learns to predict the probability for the next word using the context of the last 100 words.

Specifically, we will use an Embedding Layer to learn the representation of words, and a Long Short-Term Memory (LSTM) recurrent neural network to learn to predict words based on their context. Let's start by loading our training data.

20.4.1 Load Sequences

We can load our training data using the `load_doc()` function we developed in the previous section. Once loaded, we can split the data into separate training sequences by splitting based on new lines. The snippet below will load the `republic_sequences.txt` data file from the current working directory.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
```

Listing 20.14: Load the clean sequences from file.

Next, we can encode the training data.

20.4.2 Encode Sequences

The word embedding layer expects input sequences to be comprised of integers. We can map each word in our vocabulary to a unique integer and encode our input sequences. Later, when we make predictions, we can convert the prediction to numbers and look up their associated words in the same mapping. To do this encoding, we will use the `Tokenizer` class in the Keras API.

First, the `Tokenizer` must be trained on the entire training dataset, which means it finds all of the unique words in the data and assigns each a unique integer. We can then use the `fit` `Tokenizer` to encode all of the training sequences, converting each sequence from a list of words to a list of integers.

```
# integer encode sequences of words
tokenizer = Tokenizer()
tokenizer.fit_on_texts(lines)
sequences = tokenizer.texts_to_sequences(lines)
```

Listing 20.15: Train a tokenizer on the loaded sequences.

We can access the mapping of words to integers as a dictionary attribute called `word_index` on the `Tokenizer` object. We need to know the size of the vocabulary for defining the embedding layer later. We can determine the vocabulary by calculating the size of the mapping dictionary.

Words are assigned values from 1 to the total number of words (e.g. 7,409). The `Embedding` layer needs to allocate a vector representation for each word in this vocabulary from index 1 to the largest index and because indexing of arrays is zero-offset, the index of the word at the end of the vocabulary will be 7,409; that means the array must be $7,409 + 1$ in length. Therefore, when specifying the vocabulary size to the `Embedding` layer, we specify it as 1 larger than the actual vocabulary.

```
# vocabulary size
vocab_size = len(tokenizer.word_index) + 1
```

Listing 20.16: Calculate the size of the vocabulary.

20.4.3 Sequence Inputs and Output

Now that we have encoded the input sequences, we need to separate them into input (X) and output (y) elements. We can do this with array slicing. After separating, we need to one hot encode the output word. This means converting it from an integer to a vector of 0 values, one for each word in the vocabulary, with a 1 to indicate the specific word at the index of the words integer value.

This is so that the model learns to predict the probability distribution for the next word and the ground truth from which to learn from is 0 for all words except the actual word that comes next. Keras provides the `to_categorical()` that can be used to one hot encode the output words for each input-output sequence pair.

Finally, we need to specify to the `Embedding` layer how long input sequences are. We know that there are 50 words because we designed the model, but a good generic way to specify that is to use the second dimension (number of columns) of the input data's shape. That way, if you change the length of sequences when preparing data, you do not need to change this data loading code; it is generic.

```
# separate into input and output
sequences = array(sequences)
X, y = sequences[:, :-1], sequences[:, -1]
y = to_categorical(y, num_classes=vocab_size)
seq_length = X.shape[1]
```

Listing 20.17: Split text data into input and output sequences.

20.4.4 Fit Model

We can now define and fit our language model on the training data. The learned embedding needs to know the size of the vocabulary and the length of input sequences as previously discussed. It also has a parameter to specify how many dimensions will be used to represent each word. That is, the size of the embedding vector space.

Common values are 50, 100, and 300. We will use 50 here, but consider testing smaller or larger values. We will use a two LSTM hidden layers with 100 memory cells each. More memory cells and a deeper network may achieve better results.

A dense fully connected layer with 100 neurons connects to the LSTM hidden layers to interpret the features extracted from the sequence. The output layer predicts the next word as a single vector the size of the vocabulary with a probability for each word in the vocabulary. A softmax activation function is used to ensure the outputs have the characteristics of normalized probabilities.

```
# define the model
def define_model(vocab_size, seq_length):
    model = Sequential()
    model.add(Embedding(vocab_size, 50, input_length=seq_length))
```

```

model.add(LSTM(100, return_sequences=True))
model.add(LSTM(100))
model.add(Dense(100, activation='relu'))
model.add(Dense(vocab_size, activation='softmax'))
# compile network
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# summarize defined model
model.summary()
plot_model(model, to_file='model.png', show_shapes=True)
return model

```

Listing 20.18: Define the language model.

A summary of the defined network is printed as a sanity check to ensure we have constructed what we intended.

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 50, 50)	370500
lstm_1 (LSTM)	(None, 50, 100)	60400
lstm_2 (LSTM)	(None, 100)	80400
dense_1 (Dense)	(None, 100)	10100
dense_2 (Dense)	(None, 7410)	748410
<hr/>		
Total params:	1,269,810	
Trainable params:	1,269,810	
Non-trainable params:	0	

Listing 20.19: Example output from summarizing the defined model.

A plot the defined model is then saved to file with the name `model.png`.

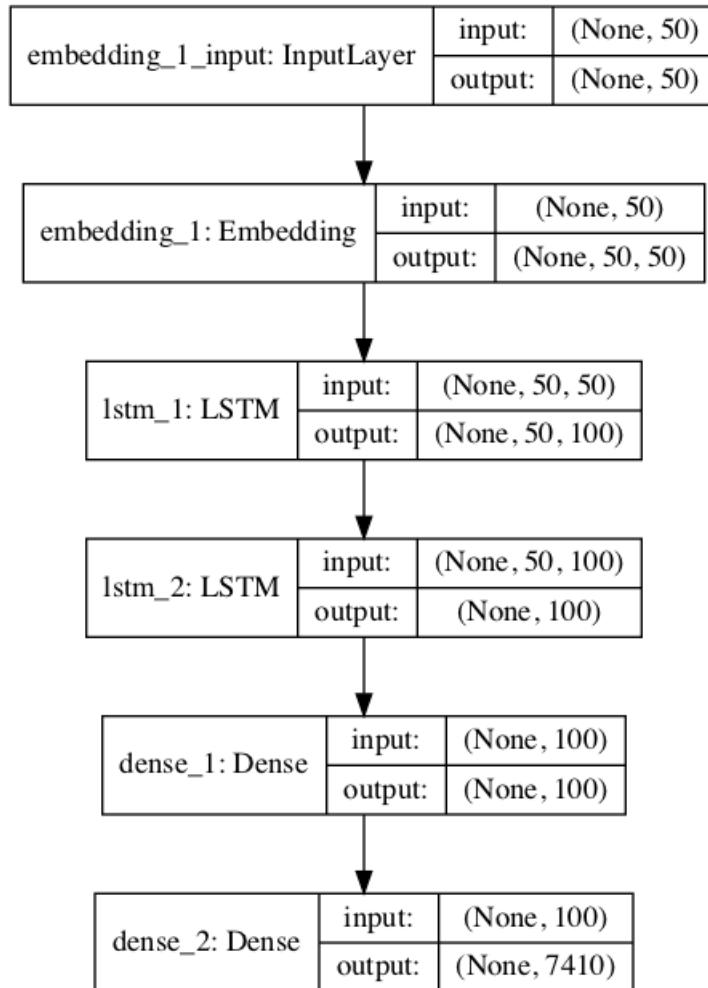


Figure 20.1: Plot of the defined word-based language model.

The model is compiled specifying the categorical cross entropy loss needed to fit the model. Technically, the model is learning a multiclass classification and this is the suitable loss function for this type of problem. The efficient Adam implementation to mini-batch gradient descent is used and accuracy is evaluated of the model. Finally, the model is fit on the data for 100 training epochs with a modest batch size of 128 to speed things up. Training may take a few hours on modern hardware without GPUs. You can speed it up with a larger batch size and/or fewer training epochs.

During training, you will see a summary of performance, including the loss and accuracy evaluated from the training data at the end of each batch update. You will get different results, but perhaps an accuracy of just over 50% of predicting the next word in the sequence, which is not bad. We are not aiming for 100% accuracy (e.g. a model that memorized the text), but rather a model that captures the essence of the text.

```

...
Epoch 96/100
118633/118633 [=====] - 265s - loss: 2.0324 - acc: 0.5187
Epoch 97/100
118633/118633 [=====] - 265s - loss: 2.0136 - acc: 0.5247
Epoch 98/100

```

```
118633/118633 [=====] - 267s - loss: 1.9956 - acc: 0.5262
Epoch 99/100
118633/118633 [=====] - 266s - loss: 1.9812 - acc: 0.5291
Epoch 100/100
118633/118633 [=====] - 270s - loss: 1.9709 - acc: 0.5315
```

Listing 20.20: Example output from training the language model.

20.4.5 Save Model

At the end of the run, the trained model is saved to file. Here, we use the Keras model API to save the model to the file `model.h5` in the current working directory. Later, when we load the model to make predictions, we will also need the mapping of words to integers. This is in the `Tokenizer` object, and we can save that too using Pickle.

```
# save the model to file
model.save('model.h5')
# save the tokenizer
dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

Listing 20.21: Save the fit model and `Tokenizer` to file.

20.4.6 Complete Example

We can put all of this together; the complete example for fitting the language model is listed below.

```
from numpy import array
from pickle import dump
from keras.preprocessing.text import Tokenizer
from keras.utils.vis_utils import plot_model
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# define the model
def define_model(vocab_size, seq_length):
    model = Sequential()
    model.add(Embedding(vocab_size, 50, input_length=seq_length))
    model.add(LSTM(100, return_sequences=True))
    model.add(LSTM(100))
    model.add(Dense(100, activation='relu'))
```

```

model.add(Dense(vocab_size, activation='softmax'))
# compile network
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# summarize defined model
model.summary()
plot_model(model, to_file='model.png', show_shapes=True)
return model

# load
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
# integer encode sequences of words
tokenizer = Tokenizer()
tokenizer.fit_on_texts(lines)
sequences = tokenizer.texts_to_sequences(lines)
# vocabulary size
vocab_size = len(tokenizer.word_index) + 1
# separate into input and output
sequences = array(sequences)
X, y = sequences[:, :-1], sequences[:, -1]
y = to_categorical(y, num_classes=vocab_size)
seq_length = X.shape[1]
# define model
model = define_model(vocab_size, seq_length)
# fit model
model.fit(X, y, batch_size=128, epochs=100)
# save the model to file
model.save('model.h5')
# save the tokenizer
dump(tokenizer, open('tokenizer.pkl', 'wb'))

```

Listing 20.22: Complete example training the language model.

20.5 Use Language Model

Now that we have a trained language model, we can use it. In this case, we can use it to generate new sequences of text that have the same statistical properties as the source text. This is not practical, at least not for this example, but it gives a concrete example of what the language model has learned. We will start by loading the training sequences again.

20.5.1 Load Data

We can use the same code from the previous section to load the training data sequences of text. Specifically, the `load_doc()` function.

```

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file

```

```

file.close()
return text

# load cleaned text sequences
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')

```

Listing 20.23: Load the clean sequences from file.

We need the text so that we can choose a source sequence as input to the model for generating a new sequence of text. The model will require 50 words as input. Later, we will need to specify the expected length of input. We can determine this from the input sequences by calculating the length of one line of the loaded data and subtracting 1 for the expected output word that is also on the same line.

```
seq_length = len(lines[0].split()) - 1
```

Listing 20.24: Calculate the expected input length.

20.5.2 Load Model

We can now load the model from file. Keras provides the `load_model()` function for loading the model, ready for use.

```
# load the model
model = load_model('model.h5')
```

Listing 20.25: Load the saved model from file.

We can also load the tokenizer from file using the Pickle API.

```
# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))
```

Listing 20.26: Load the saved Tokenizer from file.

We are ready to use the loaded model.

20.5.3 Generate Text

The first step in generating text is preparing a seed input. We will select a random line of text from the input text for this purpose. Once selected, we will print it so that we have some idea of what was used.

```
# select a seed text
seed_text = lines[randint(0, len(lines))]
print(seed_text + '\n')
```

Listing 20.27: Select random examples as seed text.

Next, we can generate new words, one at a time. First, the seed text must be encoded to integers using the same tokenizer that we used when training the model.

```
encoded = tokenizer.texts_to_sequences([seed_text])[0]
```

Listing 20.28: Encode the selected seed text.

The model can predict the next word directly by calling `model.predict_classes()` that will return the index of the word with the highest probability.

```
# predict probabilities for each word
yhat = model.predict_classes(encoded, verbose=0)
```

Listing 20.29: Predict the next word in the sequence.

We can then look up the index in the `Tokenizer`'s mapping to get the associated word.

```
out_word = ''
for word, index in tokenizer.word_index.items():
    if index == yhat:
        out_word = word
        break
```

Listing 20.30: Map the predicted integer to a word in the known vocabulary.

We can then append this word to the seed text and repeat the process. Importantly, the input sequence is going to get too long. We can truncate it to the desired length after the input sequence has been encoded to integers. Keras provides the `pad_sequences()` function that we can use to perform this truncation.

```
encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
```

Listing 20.31: Pad the encoded sequence.

We can wrap all of this into a function called `generate_seq()` that takes as input the model, the tokenizer, input sequence length, the seed text, and the number of words to generate. It then returns a sequence of words generated by the model.

```
# generate a sequence from a language model
def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
    result = list()
    in_text = seed_text
    # generate a fixed number of words
    for _ in range(n_words):
        # encode the text as integer
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
        # predict probabilities for each word
        yhat = model.predict_classes(encoded, verbose=0)
        # map predicted word index to word
        out_word = ''
        for word, index in tokenizer.word_index.items():
            if index == yhat:
                out_word = word
                break
        # append to input
        in_text += ' ' + out_word
        result.append(out_word)
    return ' '.join(result)
```

Listing 20.32: Function to generate a sequence of words given the model and seed text.

We are now ready to generate a sequence of new words given some seed text.

```
# generate new text
generated = generate_seq(model, tokenizer, seq_length, seed_text, 50)
print(generated)
```

Listing 20.33: Example of generating a sequence of text.

Putting this all together, the complete code listing for generating text from the learned-language model is listed below.

```
from random import randint
from pickle import load
from keras.models import load_model
from keras.preprocessing.sequence import pad_sequences

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# generate a sequence from a language model
def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
    result = list()
    in_text = seed_text
    # generate a fixed number of words
    for _ in range(n_words):
        # encode the text as integer
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
        # predict probabilities for each word
        yhat = model.predict_classes(encoded, verbose=0)
        # map predicted word index to word
        out_word = ''
        for word, index in tokenizer.word_index.items():
            if index == yhat:
                out_word = word
                break
        # append to input
        in_text += ' ' + out_word
        result.append(out_word)
    return ' '.join(result)

# load cleaned text sequences
in_filename = 'republic_sequences.txt'
doc = load_doc(in_filename)
lines = doc.split('\n')
seq_length = len(lines[0].split()) - 1
# load the model
model = load_model('model.h5')
# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))
```

```
# select a seed text
seed_text = lines[randint(0, len(lines))]
print(seed_text + '\n')
# generate new text
generated = generate_seq(model, tokenizer, seq_length, seed_text, 50)
print(generated)
```

Listing 20.34: Complete example of generating sequences of text.

Running the example first prints the seed text.

```
when he said that a man when he grows old may learn many things for he can no more learn
much than he can run much youth is the time for any extraordinary toil of course and
therefore calculation and geometry and all the other elements of instruction which are a
```

Listing 20.35: Example output from selecting seed text.

Then 50 words of generated text are printed.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
preparation for dialectic should be presented to the name of idle spendthrifts of whom the
other is the manifold and the unjust and is the best and the other which delighted to
be the opening of the soul of the soul and the embroiderer will have to be said at
```

Listing 20.36: Example output of generated text.

You can see that the text seems reasonable. In fact, the addition of concatenation would help in interpreting the seed and the generated text. Nevertheless, the generated text gets the right kind of words in the right kind of order. Try running the example a few times to see other examples of generated text.

20.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Contrived Seed Text.** Hand craft or select seed text and evaluate how the seed text impacts the generated text, specifically the initial words or sentences generated.
- **Simplify Vocabulary.** Explore a simpler vocabulary, perhaps with stemmed words or stop words removed.
- **Data Cleaning.** Consider using more or less cleaning of the text, perhaps leave in some punctuation or perhaps replacing all fancy names with one or a handful. Evaluate how these changes to the size of the vocabulary impact the generated text.
- **Tune Model.** Tune the model, such as the size of the embedding or number of memory cells in the hidden layer, to see if you can develop a better model.
- **Deeper Model.** Extend the model to have multiple LSTM hidden layers, perhaps with dropout to see if you can develop a better model.

- **Develop Pre-Trained Embedding.** Extend the model to use pre-trained Word2Vec vectors to see if it results in a better model.
- **Use GloVe Embedding.** Use the GloVe word embedding vectors with and without fine tuning by the network and evaluate how it impacts training and the generated words.
- **Sequence Length.** Explore training the model with different length input sequences, both shorter and longer, and evaluate how it impacts the quality of the generated text.
- **Reduce Scope.** Consider training the model on one book (chapter) or a subset of the original text and evaluate the impact on training, training speed and the resulting generated text.
- **Sentence-Wise Model.** Split the raw data based on sentences and pad each sentence to a fixed length (e.g. the longest sentence length).

If you explore any of these extensions, I'd love to know.

20.7 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- Project Gutenberg.
<https://www.gutenberg.org/>
- The Republic by Plato on Project Gutenberg.
<https://www.gutenberg.org/ebooks/1497>
- Republic (Plato) on Wikipedia.
[https://en.wikipedia.org/wiki/Republic_\(Plato\)](https://en.wikipedia.org/wiki/Republic_(Plato))
- Language model on Wikipedia.
https://en.wikipedia.org/wiki/Language_model

20.8 Summary

In this tutorial, you discovered how to develop a word-based language model using a word embedding and a recurrent neural network. Specifically, you learned:

- How to prepare text for developing a word-based language model.
- How to design and fit a neural language model with a learned embedding and an LSTM hidden layer.
- How to use the learned language model to generate new text with similar statistical properties as the source text.

20.8.1 Next

This is the final chapter in the language modeling part. In the next part you will discover how to develop automatic caption generation for photographs.

Part VIII

Image Captioning

Chapter 21

Neural Image Caption Generation

Captioning an image involves generating a human readable textual description given an image, such as a photograph. It is an easy problem for a human, but very challenging for a machine as it involves both understanding the content of an image and how to translate this understanding into natural language. Recently, deep learning methods have displaced classical methods and are achieving state-of-the-art results for the problem of automatically generating descriptions, called *captions*, for images. In this chapter, you will discover how deep neural network models can be used to automatically generate descriptions for images, such as photographs. After completing this chapter, you will know:

- About the challenge of generating textual descriptions for images and the need to combine breakthroughs from computer vision and natural language processing.
- About the elements that comprise a neural feature captioning model, namely the feature extractor and language model.
- How the elements of the model can be arranged into an Encoder-Decoder, possibly with the use of an attention mechanism.

Let's get started.

21.1 Overview

This tutorial is divided into the following parts:

1. Describing an Image with Text
2. Neural Captioning Model
3. Encoder-Decoder Architecture

21.2 Describing an Image with Text

Describing an image is the problem of generating a human-readable textual description of an image, such as a photograph of an object or scene. The problem is sometimes called *automatic image annotation* or *image tagging*. It is an easy problem for a human, but very challenging for a machine.

A quick glance at an image is sufficient for a human to point out and describe an immense amount of details about the visual scene. However, this remarkable ability has proven to be an elusive task for our visual recognition models

— *Deep Visual-Semantic Alignments for Generating Image Descriptions*, 2015.

A solution requires both that the content of the image be understood and translated to meaning in the terms of words, and that the words must string together to be comprehensible. It combines both computer vision and natural language processing and marks a true challenging problem in broader artificial intelligence.

Automatically describing the content of an image is a fundamental problem in artificial intelligence that connects computer vision and natural language processing.

— *Show and Tell: A Neural Image Caption Generator*, 2015.

Further, the problems can range in difficulty; let's look at three different variations on the problem with examples.

- **Classify Image.** Assign an image a class label from one of many known classes.
- **Describe Image.** Generate a textual description of the contents image.
- **Annotate Image.** Generate textual descriptions for specific regions on the image.

The general problem can also be extended to describe images over time in video. In this chapter, we will focus our attention on describing images, which we will describe as *image captioning*.

21.3 Neural Captioning Model

Neural network models have come to dominate the field of automatic caption generation; this is primarily because the methods are demonstrating state-of-the-art results. The two dominant methods prior to end-to-end neural network models for generating image captions were template-based methods and nearest-neighbor-based methods and modifying existing captions.

Prior to the use of neural networks for generating captions, two main approaches were dominant. The first involved generating caption templates which were filled in based on the results of object detections and attribute discovery. The second approach was based on first retrieving similar captioned images from a large database then modifying these retrieved captions to fit the query. [...] Both of these approaches have since fallen out of favour to the now dominant neural network methods.

— *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*, 2015.

Neural network models for captioning involve two main elements:

1. Feature Extraction.
2. Language Model.

21.3.1 Feature Extraction Model

The feature extraction model is a neural network that given an image is able to extract the salient features, often in the form of a fixed-length vector. The extracted features are an internal representation of the image, not something directly intelligible. A deep convolutional neural network, or CNN, is used as the feature extraction submodel. This network can be trained directly on the images in the image captioning dataset. Alternately, a pre-trained model, such as a state-of-the-art model used for image classification, can be used, or some hybrid where a pre-trained model is used and fine tuned on the problem. It is popular to use top performing models in the ImageNet dataset developed for the ILSVRC challenge, such as the Oxford Vision Geometry Group model, called VGG for short.

[...] we explored several techniques to deal with overfitting. The most obvious way to not overfit is to initialize the weights of the CNN component of our system to a pretrained model (e.g., on ImageNet)

— *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*, 2015.

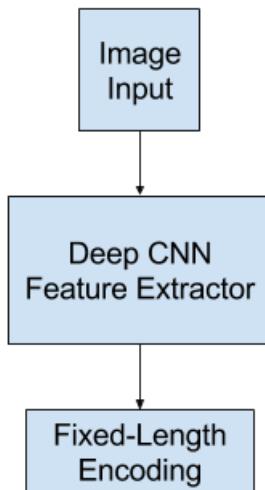


Figure 21.1: Feature Extractor

21.3.2 Language Model

Generally, a language model predicts the probability of the next word in the sequence given the words already present in the sequence. For image captioning, the language model is a neural network that given the extracted features from the network is capable of predicting the sequence of words in the description and build up the description conditional on the words that have already been generated. It is popular to use a recurrent neural network, such as a Long Short-Term Memory network, or LSTM, as the language model. Each output time step generates a new word in the sequence. Each word that is generated is then encoded using a word embedding (such as Word2Vec) and passed as input to the decoder for generating the subsequent word.

An improvement to the model involves gathering the probability distribution of words across the vocabulary for the output sequence and searching it to generate multiple possible descriptions. These descriptions can be scored and ranked by likelihood. It is common to use a Beam Search for this search. The language model can be trained standalone using pre-computed features extracted from the image dataset; it can be trained jointly with the feature extraction network, or some combination.

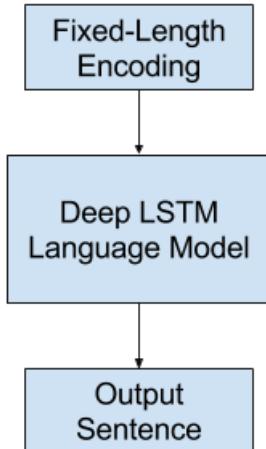


Figure 21.2: Language Model

21.4 Encoder-Decoder Architecture

A popular way to structure the sub-models is to use an Encoder-Decoder architecture where both models are trained jointly.

[the model] is based on a convolution neural network that encodes an image into a compact representation, followed by a recurrent neural network that generates a corresponding sentence. The model is trained to maximize the likelihood of the sentence given the image.

— *Show and Tell: A Neural Image Caption Generator*, 2015.

This is an architecture developed for machine translation where an input sequence, say in French, is encoded as a fixed-length vector by an encoder network. A separate decoder network then reads the encoding and generates an output sequence in the new language, say English. A benefit of this approach in addition to the impressive skill of the approach is that a single end-to-end model can be trained on the problem. When adapted for image captioning, the encoder network is a deep convolutional neural network, and the decoder network is a stack of LSTM layers.

[in machine translation] An “encoder” RNN reads the source sentence and transforms it into a rich fixed-length vector representation, which in turn is used as the initial hidden state of a “decoder” RNN that generates the target sentence. Here, we propose

to follow this elegant recipe, replacing the encoder RNN by a deep convolution neural network (CNN).

— *Show and Tell: A Neural Image Caption Generator*, 2015.

21.4.1 Captioning Model with Attention

A limitation of the Encoder-Decoder architecture is that a single fixed-length representation is used to hold the extracted features. This was addressed in machine translation through the development of attention across a richer encoding, allowing the decoder to learn where to place attention as each word in the translation is generated. The approach of attention has also been used to improve the performance of the Encoder-Decoder architecture for image captioning by allowing the decoder to learn where to put attention in the image when generating each word in the description.

Encouraged by recent advances in caption generation and inspired by recent success in employing attention in machine translation and object recognition we investigate models that can attend to salient part of an image while generating its caption.

— *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*, 2015.

A benefit of this approach is that it is possible to visualize exactly where attention is placed while generating each word in a description.

We also show through visualization how the model is able to automatically learn to fix its gaze on salient objects while generating the corresponding words in the output sequence.

— *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*, 2015.

21.5 Further Reading

This section provides more resources on the topic if you are looking go deeper.

21.5.1 Papers

- *Show and Tell: A Neural Image Caption Generator*, 2015.
<https://arxiv.org/abs/1411.4555>
- *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*, 2015.
<https://arxiv.org/abs/1502.03044>
- *Long-term recurrent convolutional networks for visual recognition and description*, 2015.
<https://arxiv.org/abs/1411.4389>
- *Deep Visual-Semantic Alignments for Generating Image Descriptions*, 2015.
<https://arxiv.org/abs/1412.2306>

21.5.2 Articles

- Automatic image annotation on Wikipedia.
https://en.wikipedia.org/wiki/Automatic_image_annotation
- *Show and Tell: image captioning open sourced in TensorFlow*, 2016.
<https://research.googleblog.com/2016/09/show-and-tell-image-captioning-open.html>
- Presentation: *Automated Image Captioning with ConvNets and Recurrent Nets*, Andrej Karpathy and Fei-Fei Li.
<https://www.youtube.com/watch?v=xKt21ucdBY0>

21.5.3 Projects

- *Deep Visual-Semantic Alignments for Generating Image Descriptions*, 2015.
<http://cs.stanford.edu/people/karpathy/deepimagesent/>
- *NeuralTalk2: Efficient Image Captioning code in Torch, runs on GPU*, Andrej Karpathy.
<https://github.com/karpathy/neuraltalk2>

21.6 Summary

In this chapter, you discovered how deep neural network models can be used to automatically generate descriptions for images, such as photographs. Specifically, you learned:

- About the challenge of generating textual descriptions for images and the need to combine breakthroughs from computer vision and natural language processing.
- About the elements that comprise a neural feature captioning model, namely the feature extractor and language model.
- How the elements of the model can be arranged into an Encoder-Decoder, possibly with the use of an attention mechanism.

21.6.1 Next

In the next chapter, you will discover how you can develop neural caption models.

Chapter 22

Neural Network Models for Caption Generation

Caption generation is a challenging artificial intelligence problem that draws on both computer vision and natural language processing. The encoder-decoder recurrent neural network architecture has been shown to be effective at this problem. The implementation of this architecture can be distilled into inject and merge based models, and both make different assumptions about the role of the recurrent neural network in addressing the problem. In this chapter, you will discover the inject and merge architectures for the encoder-decoder recurrent neural network models on caption generation. After reading this chapter, you will know:

- The challenge of caption generation and the use of the encoder-decoder architecture.
- The inject model that combines the encoded image with each word to generate the next word in the caption.
- The merge model that separately encodes the image and description which are decoded in order to generate the next word in the caption.

Let's get started.

22.1 Image Caption Generation

The problem of image caption generation involves outputting a readable and concise description of the contents of a photograph. It is a challenging artificial intelligence problem as it requires both techniques from computer vision to interpret the contents of the photograph and techniques from natural language processing to generate the textual description. Recently, deep learning methods have achieved state-of-the-art results on this challenging problem. The results are so impressive that this problem has become a standard demonstration problem for the capabilities of deep learning.

22.1.1 Encoder-Decoder Architecture

A standard encoder-decoder recurrent neural network architecture is used to address the image caption generation problem. This involves two elements:

- **Encoder:** A network model that reads the photograph input and encodes the content into a fixed-length vector using an internal representation.
- **Decoder:** A network model that reads the encoded photograph and generates the textual description output.

Generally, a convolutional neural network is used to encode the images and a recurrent neural network, such as a Long Short-Term Memory network, is used to either encode the text sequence generated so far, and/or generate the next word in the sequence. There are many ways to realize this architecture for the problem of caption generation. It is common to use a pre-trained convolutional neural network model trained on a challenging photograph classification problem to encode the photograph. The pre-trained model can be loaded, the output of the model removed, and the internal representation of the photograph used as the encoding or internal representation of the input image.

It is also common to frame the problem such that the model generates one word of the output textual description, given both the photograph and the description generated so far as input. In this framing, the model is called recursively until the entire output sequence is generated.

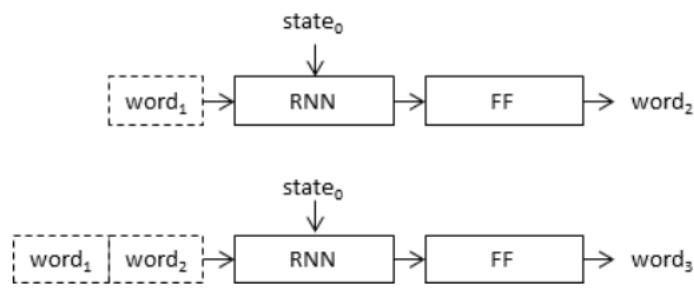


Figure 22.1: Recursive Framing of the Caption Generation Model. Taken from *Where to put the Image in an Image Caption Generator*.

This framing can be implemented using one of two architectures, called by Marc Tanti, et al. as the inject and the merge models.

22.2 Inject Model

The inject model combines the encoded form of the image with each word from the text description generated so-far. The approach uses the recurrent neural network as a text generation model that uses a sequence of both image and word information as input in order to generate the next word in the sequence.

In these 'inject' architectures, the image vector (usually derived from the activation values of a hidden layer in a convolutional neural network) is injected into the RNN, for example by treating the image vector on a par with a 'word' and including it as part of the caption prefix.

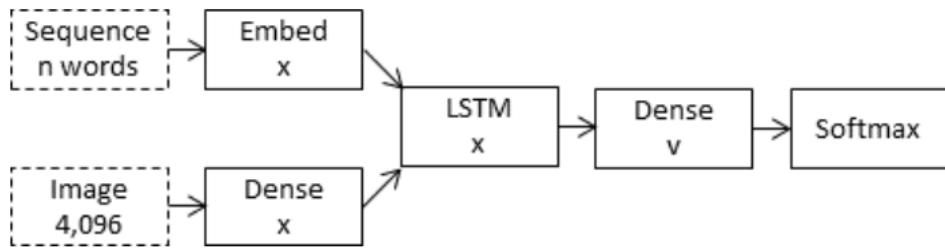


Figure 22.2: Inject Architecture for Encoder-Decoder Model. Taken from *What is the Role of Recurrent Neural Networks (RNNs) in an Image Caption Generator?*.

This model combines the concerns of the image with each input word, requiring the encoder to develop an encoding that incorporates both visual and linguistic information together.

In an inject model, the RNN is trained to predict sequences based on histories consisting of both linguistic and perceptual features. Hence, in this model, the RNN is primarily responsible for image-conditioned language generation.

— *What is the Role of Recurrent Neural Networks (RNNs) in an Image Caption Generator?, 2017.*

22.3 Merge Model

The merge model combines both the encoded form of the image input with the encoded form of the text description generated so far. The combination of these two encoded inputs is then used by a very simple decoder model to generate the next word in the sequence. The approach uses the recurrent neural network only to encode the text generated so far.

In the case of ‘merge’ architectures, the image is left out of the RNN subnetwork, such that the RNN handles only the caption prefix, that is, handles only purely linguistic information. After the prefix has been vectorised, the image vector is then merged with the prefix vector in a separate ‘multimodal layer’ which comes after the RNN subnetwork

— *Where to put the Image in an Image Caption Generator, 2017.*

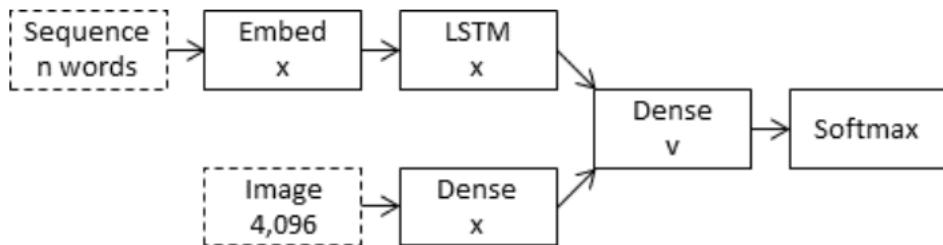


Figure 22.3: Merge Architecture for Encoder-Decoder Model. Taken from *What is the Role of Recurrent Neural Networks (RNNs) in an Image Caption Generator?*.

This separates the concern of modeling the image input, the text input and the combining and interpretation of the encoded inputs. As mentioned, it is common to use a pre-trained model for encoding the image, but similarly, this architecture also permits a pre-trained language model to be used to encode the caption text input.

... in the merge architecture, RNNs in effect encode linguistic representations, which themselves constitute the input to a later prediction stage that comes after a multimodal layer. It is only at this late stage that image features are used to condition predictions

- *What is the Role of Recurrent Neural Networks (RNNs) in an Image Caption Generator?*,
2017.

There are multiple ways to combine the two encoded inputs, such as concatenation, multiplication, and addition, although experiments by Marc Tanti, et al. have shown addition to work better. Generally, Marc Tanti, et al. found the merge architecture to be more effective compared to the inject approach.

Overall, the evidence suggests that delaying the merging of image features with linguistic encodings to a late stage in the architecture may be advantageous [...] results suggest that a merge architecture has a higher capacity than an inject architecture and can generate better quality captions with smaller layers.

- *What is the Role of Recurrent Neural Networks (RNNs) in an Image Caption Generator?*,
2017.

22.4 More on the Merge Model

The success of the merge model for the encoder-decoder architecture suggests that the role of the recurrent neural network is to encode input rather than generate output. This is a departure from the common understanding where it is believed that the contribution of the recurrent neural network is that of a generative model.

If the RNN had the primary role of generating captions, then it would need to have access to the image in order to know what to generate. This does not seem to be the case as including the image into the RNN is not generally beneficial to its performance as a caption generator.

- *What is the Role of Recurrent Neural Networks (RNNs) in an Image Caption Generator?*,
2017.

The explicit comparison of the inject and merge models, and the success of merge over inject for caption generation, raises the question of whether this approach translates to related sequence-to-sequence generation problems. Instead of pre-trained models used to encode images, pre-trained language models could be used to encode source text in problems such as text summarization, question answering, and machine translation.

We would like to investigate whether similar changes in architecture would work in sequence-to-sequence tasks such as machine translation, where instead of conditioning a language model on an image we are conditioning a target language model on sentences in a source language.

- *What is the Role of Recurrent Neural Networks (RNNs) in an Image Caption Generator?*, 2017.

22.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Marc Tanti's Blog.
<https://geekyisawesome.blogspot.com.au/>
- *Where to put the Image in an Image Caption Generator*, 2017.
<https://arxiv.org/abs/1703.09137>
- *What is the Role of Recurrent Neural Networks (RNNs) in an Image Caption Generator?*, 2017.
<https://arxiv.org/abs/1708.02043>

22.6 Summary

In this chapter, you discovered the inject and merge architectures for the encoder-decoder recurrent neural network model on caption generation. Specifically, you learned:

- The challenge of caption generation and the use of the encoder-decoder architecture.
- The inject model that combines the encoded image with each word to generate the next word in the caption.
- The merge model that separately encodes the image and description which are decoded in order to generate the next word in the caption.

22.6.1 Next

In the next chapter, you will discover how you can load and re-use a pre-trained deep computer vision model.

Chapter 23

How to Load and Use a Pre-Trained Object Recognition Model

Convolutional neural networks are now capable of outperforming humans on some computer vision tasks, such as classifying images. That is, given a photograph of an object, answer the question as to which of 1,000 specific objects the photograph shows. A competition-winning model for this task is the VGG model by researchers at Oxford. What is important about this model, besides its capability of classifying objects in photographs, is that the model weights are freely available and can be loaded and used in your own models and applications. In this tutorial, you will discover the VGG convolutional neural network models for image classification. After completing this tutorial, you will know:

- About the ImageNet dataset and competition and the VGG winning models.
- How to load the VGG model in Keras and summarize its structure.
- How to use the loaded VGG model to classifying objects in ad hoc photographs.

Let's get started.

23.1 Tutorial Overview

This tutorial is divided into the following parts:

1. ImageNet
2. The Oxford VGG Models
3. Load the VGG Model in Keras
4. Develop a Simple Photo Classifier

Note, Keras makes use of the Python Imaging Library or PIL library for manipulating images. Installation on your system may vary.

23.2 ImageNet

ImageNet is a research project to develop a large database of images with annotations, e.g. images and their descriptions. The images and their annotations have been the basis for an image classification challenge called the ImageNet Large Scale Visual Recognition Challenge or ILSVRC since 2010. The result is that research organizations battle it out on pre-defined datasets to see who has the best model for classifying the objects in images.

The ImageNet Large Scale Visual Recognition Challenge is a benchmark in object category classification and detection on hundreds of object categories and millions of images. The challenge has been run annually from 2010 to present, attracting participation from more than fifty institutions.

— *ImageNet Large Scale Visual Recognition Challenge*, 2015.

For the classification task, images must be classified into one of 1,000 different categories. For the last few years very deep convolutional neural network models have been used to win these challenges and results on the tasks have exceeded human performance.

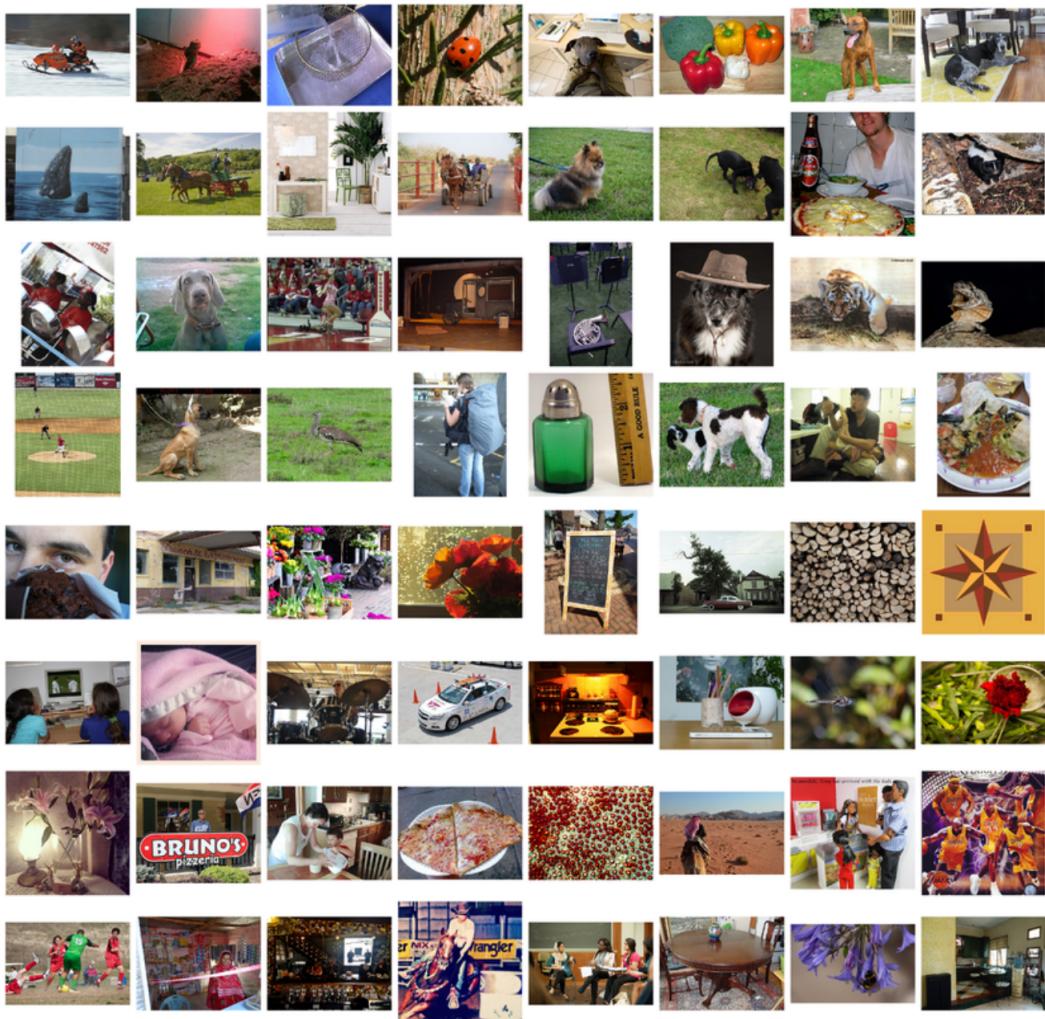


Figure 23.1: Sample of Images from the ImageNet Dataset used in the ILSVRC Challenge. Taken From *ImageNet Large Scale Visual Recognition Challenge*.

23.3 The Oxford VGG Models

Researchers from the Oxford Visual Geometry Group, or VGG for short, participate in the ILSVRC challenge. In 2014, convolutional neural network models (CNN) developed by the VGG won the image classification tasks.

Classification+localization

Task 2a: Classification+localization with provided training data

Classification+localization with provided training data: Ordered by localization error

Team name	Entry description	Localization error	Classification error
VGG	a combination of multiple ConvNets (by averaging)	0.253231	0.07405
VGG	a combination of multiple ConvNets (fusion weights learnt on the validation set)	0.253501	0.07407
VGG	a combination of multiple ConvNets, including a net trained on images of different size (fusion done by averaging); detected boxes were not updated	0.255431	0.07337
VGG	a combination of multiple ConvNets, including a net trained on images of different size (fusion weights learnt on the validation set); detected boxes were not updated	0.256167	0.07325

Figure 23.2: ILSVRC Results in 2014 for the Classification task.

After the competition, the participants wrote up their findings in the paper *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014. They also made their models and learned weights available online. This allowed other researchers and developers to use a state-of-the-art image classification model in their own work and programs. This helped to fuel a rash of transfer learning work where pre-trained models are used with minor modification on wholly new predictive modeling tasks, harnessing the state-of-the-art feature extraction capabilities of proven models.

... we come up with significantly more accurate ConvNet architectures, which not only achieve the state-of-the-art accuracy on ILSVRC classification and localisation tasks, but are also applicable to other image recognition datasets, where they achieve excellent performance even when used as a part of a relatively simple pipelines (e.g. deep features classified by a linear SVM without fine-tuning). We have released our two best-performing models to facilitate further research.

— *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014.

VGG released two different CNN models, specifically a 16-layer model and a 19-layer model. Refer to the paper for the full details of these models. The VGG models are not longer state-of-the-art by only a few percentage points. Nevertheless, they are very powerful models and useful both as image classifiers and as the basis for new models that use image inputs. In the next section, we will see how we can use the VGG model directly in Keras.

23.4 Load the VGG Model in Keras

The VGG model can be loaded and used in the Keras deep learning library. Keras provides an Applications interface for loading and using pre-trained models. Using this interface, you can

create a VGG model using the pre-trained weights provided by the Oxford group and use it as a starting point in your own model, or use it as a model directly for classifying images. In this tutorial, we will focus on the use case of classifying new images using the VGG model. Keras provides both the 16-layer and 19-layer version via the VGG16 and VGG19 classes. Let's focus on the VGG16 model. The model can be created as follows:

```
from keras.applications.vgg16 import VGG16
model = VGG16()
```

Listing 23.1: Create the VGG16 model in Keras.

That's it. The first time you run this example, Keras will download the weight files from the Internet and store them in the `~/.keras/models` directory. **Note** that the weights are about 528 megabytes, so the download may take a few minutes depending on the speed of your Internet connection.

The weights are only downloaded once. The next time you run the example, the weights are loaded locally and the model should be ready to use in seconds. We can use the standard Keras tools for inspecting the model structure. For example, you can print a summary of the network layers as follows:

```
from keras.applications.vgg16 import VGG16
model = VGG16()
model.summary()
```

Listing 23.2: Create and summarize the VGG16 model.

You can see that the model is huge. You can also see that, by default, the model expects images as input with the size 224 x 224 pixels with 3 channels (e.g. color).

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160

```
block4_conv2 (Conv2D)      (None, 28, 28, 512)    2359808
-----
block4_conv3 (Conv2D)      (None, 28, 28, 512)    2359808
-----
block4_pool (MaxPooling2D) (None, 14, 14, 512)    0
-----
block5_conv1 (Conv2D)      (None, 14, 14, 512)    2359808
-----
block5_conv2 (Conv2D)      (None, 14, 14, 512)    2359808
-----
block5_conv3 (Conv2D)      (None, 14, 14, 512)    2359808
-----
block5_pool (MaxPooling2D) (None, 7, 7, 512)      0
-----
flatten (Flatten)         (None, 25088)          0
-----
fc1 (Dense)               (None, 4096)           102764544
-----
fc2 (Dense)               (None, 4096)           16781312
-----
predictions (Dense)       (None, 1000)          4097000
=====
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
```

Listing 23.3: Output summary for the VGG16 model.

We can also create a plot of the layers in the VGG model, as follows:

```
from keras.applications.vgg16 import VGG16
from keras.utils.vis_utils import plot_model
model = VGG16()
plot_model(model, to_file='vgg.png')
```

Listing 23.4: Create and plot the graph of the VGG16 model.

Again, because the model is large, the plot is a little too large and perhaps unreadable. Nevertheless, it is provided below.

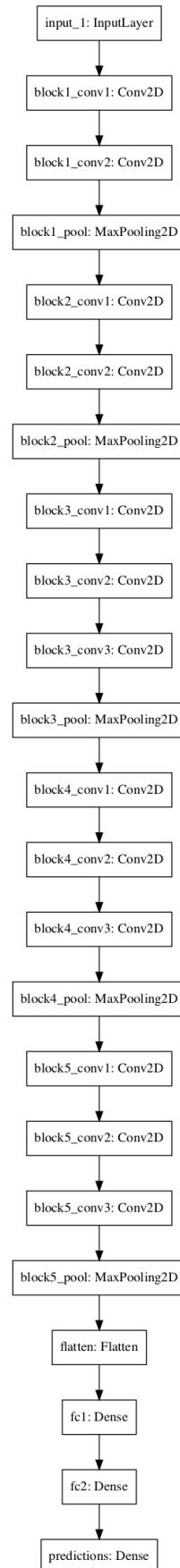


Figure 23.3: Plot of Layers in the VGG Model.

The `VGG()` class takes a few arguments that may only interest you if you are looking to use the model in your own project, e.g. for transfer learning. For example:

- `include_top (True)`: Whether or not to include the output layers for the model. You don't need these if you are fitting the model on your own problem.
- `weights ('imagenet')`: What weights to load. You can specify `None` to not load pre-trained weights if you are interested in training the model yourself from scratch.
- `input_tensor (None)`: A new input layer if you intend to fit the model on new data of a different size.
- `input_shape (None)`: The size of images that the model is expected to take if you change the input layer.
- `pooling (None)`: The type of pooling to use when you are training a new set of output layers.
- `classes (1000)`: The number of classes (e.g. size of output vector) for the model.

Next, let's look at using the loaded VGG model to classify ad hoc photographs.

23.5 Develop a Simple Photo Classifier

Let's develop a simple image classification script.

23.5.1 Get a Sample Image

First, we need an image we can classify. You can download a random photograph of a coffee mug from Flickr.



Figure 23.4: Coffee Mug. Photo by [jfanaian](#), some rights reserved.

Download the image and save it to your current working directory with the filename `mug.jpg`.

23.5.2 Load the VGG Model

Load the weights for the VGG-16 model, as we did in the previous section.

```
from keras.applications.vgg16 import VGG16
# load the model
model = VGG16()
```

Listing 23.5: Create the VGG16 model.

23.5.3 Load and Prepare Image

Next, we can load the image as pixel data and prepare it to be presented to the network. Keras provides some tools to help with this step. First, we can use the `load_img()` function to load the image and resize it to the required size of 224 x 224 pixels.

```
from keras.preprocessing.image import load_img
# load an image from file
image = load_img('mug.jpg', target_size=(224, 224))
```

Listing 23.6: Load and resize the image.

Next, we can convert the pixels to a NumPy array so that we can work with it in Keras. We can use the `img_to_array()` function for this.

```
from keras.preprocessing.image import img_to_array
# convert the image pixels to a NumPy array
image = img_to_array(image)
```

Listing 23.7: Convert the image pixels to a NumPy array.

The network expects one or more images as input; that means the input array will need to be 4-dimensional: samples, rows, columns, and channels. We only have one sample (one image). We can reshape the array by calling `reshape()` and adding the extra dimension.

```
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
```

Listing 23.8: Reshape the NumPy array of pixels.

Next, the image pixels need to be prepared in the same way as the ImageNet training data was prepared. Specifically, from the paper:

The only preprocessing we do is subtracting the mean RGB value, computed on the training set, from each pixel.

— *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014.

Keras provides a function called `preprocess_input()` to prepare new input for the network.

```
from keras.applications.vgg16 import preprocess_input
# prepare the image for the VGG model
image = preprocess_input(image)
```

Listing 23.9: Pre-process the pixel data for the model.

We are now ready to make a prediction for our loaded and prepared image.

23.5.4 Make a Prediction

We can call the `predict()` function on the model in order to get a prediction of the probability of the image belonging to each of the 1,000 known object types.

```
# predict the probability across all output classes
yhat = model.predict(image)
```

Listing 23.10: Classify the image with the VGG16 model.

Nearly there, now we need to interpret the probabilities.

23.5.5 Interpret Prediction

Keras provides a function to interpret the probabilities called `decode_predictions()`. It can return a list of classes and their probabilities in case you would like to present the top 3 objects that may be in the photo. We will just report the first most likely object.

```
from keras.applications.vgg16 import decode_predictions
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

Listing 23.11: Interpret the prediction probabilities.

And that's it.

23.5.6 Complete Example

Tying all of this together, the complete example is listed below:

```
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg16 import decode_predictions
from keras.applications.vgg16 import VGG16
# load the model
model = VGG16()
# load an image from file
image = load_img('mug.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
# predict the probability across all output classes
yhat = model.predict(image)
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
# print the classification
```

```
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

Listing 23.12: Complete example for classifying an image with the VGG model.

Running the example, we can see that the image is correctly classified as a *coffee mug* with a 75% likelihood.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
coffee_mug (75.27%)
```

Listing 23.13: Sample output of making a prediction for the image.

23.6 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- ImageNet.
<http://www.image-net.org/>
- ImageNet on Wikipedia.
<https://en.wikipedia.org/wiki/ImageNet>
- *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2015.
<https://arxiv.org/abs/1409.1556>
- *Very Deep Convolutional Networks for Large-Scale Visual Recognition*, at Oxford.
http://www.robots.ox.ac.uk/~vgg/research/very_deep/
- *Building powerful image classification models using very little data*, 2016.
<https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html>
- Keras Applications API.
<https://keras.io/applications/>
- Keras weight files files.
<https://github.com/fchollet/deep-learning-models/releases/>

23.7 Summary

In this tutorial, you discovered the VGG convolutional neural network models for image classification. Specifically, you learned:

- About the ImageNet dataset and competition and the VGG winning models.
- How to load the VGG model in Keras and summarize its structure.
- How to use the loaded VGG model to classifying objects in ad hoc photographs.

23.7.1 Next

In the next chapter, you will discover how you can evaluate generated text against a ground truth.

Chapter 24

How to Evaluate Generated Text With the BLEU Score

BLEU, or the Bilingual Evaluation Understudy, is a score for comparing a candidate translation of text to one or more reference translations. Although developed for translation, it can be used to evaluate text generated for a suite of natural language processing tasks. In this tutorial, you will discover the BLEU score for evaluating and scoring candidate text using the NLTK library in Python. After completing this tutorial, you will know:

- A gentle introduction to the BLEU score and an intuition for what is being calculated.
- How you can calculate BLEU scores in Python using the NLTK library for sentences and documents.
- How you can use a suite of small examples to develop an intuition for how differences between a candidate and reference text impact the final BLEU score.

Let's get started.

24.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Bilingual Evaluation Understudy Score
2. Calculate BLEU Scores
3. Cumulative and Individual BLEU Scores
4. Worked Examples

24.2 Bilingual Evaluation Understudy Score

The Bilingual Evaluation Understudy Score, or BLEU for short, is a metric for evaluating a generated sentence to a reference sentence. A perfect match results in a score of 1.0, whereas a perfect mismatch results in a score of 0.0. The score was developed for evaluating the predictions

made by automatic machine translation systems. It is not perfect, but does offer 5 compelling benefits:

- It is quick and inexpensive to calculate.
- It is easy to understand.
- It is language independent.
- It correlates highly with human evaluation.
- It has been widely adopted.

The BLEU score was proposed by Kishore Papineni, et al. in their 2002 paper *BLEU: a Method for Automatic Evaluation of Machine Translation*. The approach works by counting matching n-grams in the candidate translation to n-grams in the reference text, where 1-gram or unigram would be each token and a bigram comparison would be each word pair. The comparison is made regardless of word order.

The primary programming task for a BLEU implementor is to compare n-grams of the candidate with the n-grams of the reference translation and count the number of matches. These matches are position-independent. The more the matches, the better the candidate translation is.

— *BLEU: a Method for Automatic Evaluation of Machine Translation*, 2002.

The counting of matching n-grams is modified to ensure that it takes the occurrence of the words in the reference text into account, not rewarding a candidate translation that generates an abundance of reasonable words. This is referred to in the paper as modified n-gram precision.

Unfortunately, MT systems can overgenerate “reasonable” words, resulting in improbable, but high-precision, translations [...] Intuitively the problem is clear: a reference word should be considered exhausted after a matching candidate word is identified. We formalize this intuition as the modified unigram precision.

— *BLEU: a Method for Automatic Evaluation of Machine Translation*, 2002.

The score is for comparing sentences, but a modified version that normalizes n-grams by their occurrence is also proposed for better scoring blocks of multiple sentences.

We first compute the n-gram matches sentence by sentence. Next, we add the clipped n-gram counts for all the candidate sentences and divide by the number of candidate n-grams in the test corpus to compute a modified precision score, p_n , for the entire test corpus.

— *BLEU: a Method for Automatic Evaluation of Machine Translation*, 2002.

A perfect score is not possible in practice as a translation would have to match the reference exactly. This is not even possible by human translators. The number and quality of the references used to calculate the BLEU score means that comparing scores across datasets can be troublesome.

The BLEU metric ranges from 0 to 1. Few translations will attain a score of 1 unless they are identical to a reference translation. For this reason, even a human translator will not necessarily score 1. [...] on a test corpus of about 500 sentences (40 general news stories), a human translator scored 0.3468 against four references and scored 0.2571 against two references.

— *BLEU: a Method for Automatic Evaluation of Machine Translation*, 2002.

In addition to translation, we can use the BLEU score for other language generation problems with deep learning methods such as:

- Language generation.
- Image caption generation.
- Text summarization.
- Speech recognition.
- And much more.

24.3 Calculate BLEU Scores

The Python Natural Language Toolkit library, or NLTK, provides an implementation of the BLEU score that you can use to evaluate your generated text against a reference.

24.3.1 Sentence BLEU Score

NLTK provides the `sentence_bleu()` function for evaluating a candidate sentence against one or more reference sentences. The reference sentences must be provided as a list of sentences where each reference is a list of tokens. The candidate sentence is provided as a list of tokens. For example:

```
from nltk.translate.bleu_score import sentence_bleu
reference = [['this', 'is', 'a', 'test'], ['this', 'is' 'test']]
candidate = ['this', 'is', 'a', 'test']
score = sentence_bleu(reference, candidate)
print(score)
```

Listing 24.1: Example of calculating a sentence BLEU score.

Running this example prints a perfect score as the candidate matches one of the references exactly.

```
1.0
```

Listing 24.2: Sample output of calculating the sentence BLEU score.

24.3.2 Corpus BLEU Score

NLTK also provides a function called `corpus_bleu()` for calculating the BLEU score for multiple sentences such as a paragraph or a document. The references must be specified as a list of documents where each document is a list of references and each alternative reference is a list of tokens, e.g. a list of lists of tokens. The candidate documents must be specified as a list where each document is a list of tokens, e.g. a list of lists of tokens. This is a little confusing; here is an example of two references for one document.

```
# two references for one document
from nltk.translate.bleu_score import corpus_bleu
references = [[[['this', 'is', 'a', 'test'], ['this', 'is', 'a', 'test']]}}
candidates = [['this', 'is', 'a', 'test']]
score = corpus_bleu(references, candidates)
print(score)
```

Listing 24.3: Example of calculating a corpus BLEU score.

Running the example prints a perfect score as before.

```
1.0
```

Listing 24.4: Sample output of calculating the corpus BLEU score.

24.4 Cumulative and Individual BLEU Scores

The BLEU score calculations in NLTK allow you to specify the weighting of different n-grams in the calculation of the BLEU score. This gives you the flexibility to calculate different types of BLEU score, such as individual and cumulative n-gram scores. Let's take a look.

24.4.1 Individual n-gram Scores

An individual n-gram score is the evaluation of just matching grams of a specific order, such as single words (1-gram) or word pairs (2-gram or bigram). The weights are specified as a tuple where each index refers to the gram order. To calculate the BLEU score only for 1-gram matches, you can specify a weight of 1 for 1-gram and 0 for 2, 3 and 4 (1, 0, 0, 0). For example:

```
# 1-gram individual BLEU
from nltk.translate.bleu_score import sentence_bleu
reference = [['this', 'is', 'small', 'test']]
candidate = ['this', 'is', 'a', 'test']
score = sentence_bleu(reference, candidate, weights=(1, 0, 0, 0))
print(score)
```

Listing 24.5: Example of calculating an individual 1-gram BLEU score.

Running this example prints a score of 0.5.

```
0.75
```

Listing 24.6: Sample output of calculating an individual 1-gram BLEU score.

We can repeat this example for individual n-grams from 1 to 4 as follows:

```
# n-gram individual BLEU
from nltk.translate.bleu_score import sentence_bleu
reference = [['this', 'is', 'a', 'test']]
candidate = ['this', 'is', 'a', 'test']
print('Individual 1-gram: %f' % sentence_bleu(reference, candidate, weights=(1, 0, 0, 0)))
print('Individual 2-gram: %f' % sentence_bleu(reference, candidate, weights=(0, 1, 0, 0)))
print('Individual 3-gram: %f' % sentence_bleu(reference, candidate, weights=(0, 0, 1, 0)))
print('Individual 4-gram: %f' % sentence_bleu(reference, candidate, weights=(0, 0, 0, 1)))
```

Listing 24.7: Example of calculating an individual n-gram BLEU scores.

Running the example gives the following results.

```
Individual 1-gram: 1.000000
Individual 2-gram: 1.000000
Individual 3-gram: 1.000000
Individual 4-gram: 1.000000
```

Listing 24.8: Sample output of calculating an individual n-gram BLEU scores.

Although we can calculate the individual BLEU scores, this is not how the method was intended to be used and the scores do not carry a lot of meaning, or seem that interpretable.

24.4.2 Cumulative n-gram Scores

Cumulative scores refer to the calculation of individual n-gram scores at all orders from 1 to n and weighting them by calculating the weighted geometric mean. By default, the `sentence_bleu()` and `corpus_bleu()` scores calculate the cumulative 4-gram BLEU score, also called BLEU-4. The weights for the BLEU-4 are $1/4$ (25%) or 0.25 for each of the 1-gram, 2-gram, 3-gram and 4-gram scores. For example:

```
# 4-gram cumulative BLEU
from nltk.translate.bleu_score import sentence_bleu
reference = [['this', 'is', 'small', 'test']]
candidate = ['this', 'is', 'a', 'test']
score = sentence_bleu(reference, candidate, weights=(0.25, 0.25, 0.25, 0.25))
print(score)
```

Listing 24.9: Example of calculating a cumulative 4-gram BLEU score.

Running this example prints the following score:

```
0.707106781187
```

Listing 24.10: Sample output of calculating a cumulative 4-gram BLEU score.

The cumulative and individual 1-gram BLEU use the same weights, e.g. $(1, 0, 0, 0)$. The 2-gram weights assign a 50% to each of 1-gram and 2-gram and the 3-gram weights are 33% for each of the 1, 2 and 3-gram scores. Let's make this concrete by calculating the cumulative scores for BLEU-1, BLEU-2, BLEU-3 and BLEU-4:

```
# cumulative BLEU scores
from nltk.translate.bleu_score import sentence_bleu
reference = [['this', 'is', 'small', 'test']]
candidate = ['this', 'is', 'a', 'test']
print('Cumulative 1-gram: %f' % sentence_bleu(reference, candidate, weights=(1, 0, 0, 0)))
```

```

print('Cumulative 2-gram: %f' % sentence_bleu(reference, candidate, weights=(0.5, 0.5, 0,
    0)))
print('Cumulative 3-gram: %f' % sentence_bleu(reference, candidate, weights=(0.33, 0.33,
    0.33, 0)))
print('Cumulative 4-gram: %f' % sentence_bleu(reference, candidate, weights=(0.25, 0.25,
    0.25, 0.25)))

```

Listing 24.11: Example of calculating cumulative n-gram BLEU scores.

Running the example prints the following scores. They are quite different and more expressive than the standalone individual n-gram scores.

```

Cumulative 1-gram: 0.750000
Cumulative 2-gram: 0.500000
Cumulative 3-gram: 0.632878
Cumulative 4-gram: 0.707107

```

Listing 24.12: Sample output of calculating cumulative n-gram BLEU scores.

It is common to report the cumulative BLEU-1 to BLEU-4 scores when describing the skill of a text generation system.

24.5 Worked Examples

In this section, we try to develop further intuition for the BLEU score with some examples. We work at the sentence level with a single reference sentence of the following:

```
the quick brown fox jumped over the lazy dog
```

Listing 24.13: Sample text for the worked example of calculating BLEU scores.

First, let's look at a perfect score.

```

# perfect match
from nltk.translate.bleu_score import sentence_bleu
reference = [['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']]
candidate = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
score = sentence_bleu(reference, candidate)
print(score)

```

Listing 24.14: Example of two matching cases.

Running the example prints a perfect match.

```
1.0
```

Listing 24.15: Sample output BLEU score for two matching cases.

Next, let's change one word, 'quick' to 'fast'.

```

# one word different
from nltk.translate.bleu_score import sentence_bleu
reference = [['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']]
candidate = ['the', 'fast', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
score = sentence_bleu(reference, candidate)
print(score)

```

Listing 24.16: Example of making one word different.

This result is a slight drop in score.

```
0.7506238537503395
```

Listing 24.17: Sample output BLEU score when making one word different.

Try changing two words, both ‘*quick*’ to ‘*fast*’ and ‘*lazy*’ to ‘*sleepy*’.

```
# two words different
from nltk.translate.bleu_score import sentence_bleu
reference = [['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']]
candidate = ['the', 'fast', 'brown', 'fox', 'jumped', 'over', 'the', 'sleepy', 'dog']
score = sentence_bleu(reference, candidate)
print(score)
```

Listing 24.18: Example of making two words different.

Running the example, we can see a linear drop in skill.

```
0.4854917717073234
```

Listing 24.19: Sample output BLEU score when making two words different.

What about if all words are different in the candidate?

```
# all words different
from nltk.translate.bleu_score import sentence_bleu
reference = [['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']]
candidate = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
score = sentence_bleu(reference, candidate)
print(score)
```

Listing 24.20: Example of making all words different.

We get the worse possible score.

```
0.0
```

Listing 24.21: Sample output BLEU score when making all words different.

Now, let’s try a candidate that has fewer words than the reference (e.g. drop the last two words), but the words are all correct.

```
# shorter candidate
from nltk.translate.bleu_score import sentence_bleu
reference = [['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']]
candidate = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the']
score = sentence_bleu(reference, candidate)
print(score)
```

Listing 24.22: Example of making one case shorter.

The score is much like the score when two words were wrong above.

```
0.7514772930752859
```

Listing 24.23: Sample output BLEU score when making one case shorter.

How about if we make the candidate two words longer than the reference?

```
# longer candidate
from nltk.translate.bleu_score import sentence_bleu
reference = [['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']]
candidate = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog',
             'from', 'space']
score = sentence_bleu(reference, candidate)
print(score)
```

Listing 24.24: Example of making one case longer.

Again, we can see that our intuition holds and the score is something like *two words wrong*.

```
0.7860753021519787
```

Listing 24.25: Sample output BLEU score when making one longer shorter.

Finally, let's compare a candidate that is way too short: only two words in length.

```
# very short
from nltk.translate.bleu_score import sentence_bleu
reference = [['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']]
candidate = ['the', 'quick']
score = sentence_bleu(reference, candidate)
print(score)
```

Listing 24.26: Example of making one case too short.

Running this example first prints a warning message indicating that the 3-gram and above part of the evaluation (up to 4-gram) cannot be performed. This is fair given we only have 2-grams to work with in the candidate.

```
UserWarning:
Corpus/Sentence contains 0 counts of 3-gram overlaps.
BLEU scores might be undesirable; use SmoothingFunction().
warnings.warn(_msg)
```

Listing 24.27: Sample warning message.

Next, we can a score that is very low indeed.

```
0.0301973834223185
```

Listing 24.28: Sample output BLEU score when making one case too short.

I encourage you to continue to play with examples. The math is pretty simple and I would also encourage you to read the paper and explore calculating the sentence-level score yourself in a spreadsheet.

24.6 Further Reading

This section provides more resources on the topic if you are looking go deeper.

- BLEU on Wikipedia.
<https://en.wikipedia.org/wiki/BLEU>
- *BLEU: a Method for Automatic Evaluation of Machine Translation*, 2002.
<http://www.aclweb.org/anthology/P02-1040.pdf>

- Source code for `nltk.translate.bleu_score`.
http://www.nltk.org/_modules/nltk/translate/bleu_score.html
- `nltk.translate` package API Documentation.
<http://www.nltk.org/api/nltk.translate.html>

24.7 Summary

In this tutorial, you discovered the BLEU score for evaluating and scoring candidate text to reference text in machine translation and other language generation tasks. Specifically, you learned:

- A gentle introduction to the BLEU score and an intuition for what is being calculated.
- How you can calculate BLEU scores in Python using the NLTK library for sentences and documents.
- How to can use a suite of small examples to develop an intuition for how differences between a candidate and reference text impact the final BLEU score.

24.7.1 Next

In the next chapter, you will discover how you can prepare data for training a caption generation model.

Chapter 25

How to Prepare a Photo Caption Dataset For Modeling

Automatic photo captioning is a problem where a model must generate a human-readable textual description given a photograph. It is a challenging problem in artificial intelligence that requires both image understanding from the field of computer vision as well as language generation from the field of natural language processing. It is now possible to develop your own image caption models using deep learning and freely available datasets of photos and their descriptions. In this tutorial, you will discover how to prepare photos and textual descriptions ready for developing a deep learning automatic photo caption generation model. After completing this tutorial, you will know:

- About the Flickr8K dataset comprised of more than 8,000 photos and up to 5 captions for each photo.
 - How to generally load and prepare photo and text data for modeling with deep learning.
 - How to specifically encode data for two different types of deep learning models in Keras.
- Let's get started.

25.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Download the Flickr8K Dataset
2. How to Load Photographs
3. Pre-Calculate Photo Features
4. How to Load Descriptions
5. Prepare Description Text
6. Whole Description Sequence Model
7. Word-By-Word Model
8. Progressive Loading

25.2 Download the Flickr8K Dataset

A good dataset to use when getting started with image captioning is the Flickr8K dataset. The reason is that it is realistic and relatively small so that you can download it and build models on your workstation using a CPU. The definitive description of the dataset is in the paper *Framing Image Description as a Ranking Task: Data, Models and Evaluation Metrics* from 2013. The authors describe the dataset as follows:

We introduce a new benchmark collection for sentence-based image description and search, consisting of 8,000 images that are each paired with five different captions which provide clear descriptions of the salient entities and events.

...

The images were chosen from six different Flickr groups, and tend not to contain any well-known people or locations, but were manually selected to depict a variety of scenes and situations.

— *Framing Image Description as a Ranking Task: Data, Models and Evaluation Metrics*, 2013.

The dataset is available for free. You must complete a request form and the links to the dataset will be emailed to you. I would love to link to them for you, but the email address expressly requests: *Please do not redistribute the dataset*. You can use the link below to request the dataset:

- Dataset Request Form.
<https://illinois.edu/fb/sec/1713398>

Within a short time, you will receive an email that contains links to two files:

- **Flickr8k_Dataset.zip** (1 Gigabyte) An archive of all photographs.
- **Flickr8k_text.zip** (2.2 Megabytes) An archive of all text descriptions for photographs.

Download the datasets and unzip them into your current working directory. You will have two directories:

- **Flicker8k_Dataset**: Contains more than 8000 photographs in JPEG format (yes the directory name spells it ‘Flicker’ not ‘Flickr’).
- **Flicker8k_text**: Contains a number of files containing different sources of descriptions for the photographs.

Next, let’s look at how to load the images.

25.3 How to Load Photographs

In this section, we will develop some code to load the photos for use with the Keras deep learning library in Python. The image file names are unique image identifiers. For example, here is a sample of image file names:

```
990890291_afc72be141.jpg
99171998_7cc800ceef.jpg
99679241_adc853a5c0.jpg
997338199_7343367d7f.jpg
997722733_0cb5439472.jpg
```

Listing 25.1: Example of photographs filenames.

Keras provides the `load_img()` function that can be used to load the image files directly as an array of pixels.

```
from keras.preprocessing.image import load_img
image = load_img('990890291_afc72be141.jpg')
```

Listing 25.2: Example of loading a single photograph

The pixel data needs to be converted to a NumPy array for use in Keras. We can use the `img_to_array()` Keras function to convert the loaded data.

```
from keras.preprocessing.image import img_to_array
image = img_to_array(image)
```

Listing 25.3: Example of converting a photograph to a NumPy array

We may want to use a pre-defined feature extraction model, such as a state-of-the-art deep image classification network trained on Image net. The Oxford Visual Geometry Group (VGG) model is popular for this purpose and is available in Keras. If we decide to use this pre-trained model as a feature extractor in our model, we can pre-process the pixel data for the model by using the `preprocess_input()` function in Keras, for example:

```
from keras.applications.vgg16 import preprocess_input

# reshape data into a single sample of an image
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
```

Listing 25.4: Prepare image data for the VGG16 model

We may also want to force the loading of the photo to have the same pixel dimensions as the VGG model, which are 224 x 224 pixels. We can do that in the call to `load_img()`, for example:

```
image = load_img('990890291_afc72be141.jpg', target_size=(224, 224))
```

Listing 25.5: Load an image in Keras to a specific size

We may want to extract the unique image identifier from the image filename. We can do that by splitting the filename string by the '.' (period) character and retrieving the first element of the resulting array:

```
image_id = filename.split('.')[0]
```

Listing 25.6: Retrieve image identifier from filename

We can tie all of this together and develop a function that, given the name of the directory containing the photos, will load and pre-process all of the photos for the VGG model and return them in a dictionary keyed on their unique image identifiers.

```
from os import listdir
from os import path
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input

def load_photos(directory):
    images = dict()
    for name in listdir(directory):
        # load an image from file
        filename = path.join(directory, name)
        image = load_img(filename, target_size=(224, 224))
        # convert the image pixels to a numpy array
        image = img_to_array(image)
        # reshape data for the model
        image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
        # prepare the image for the VGG model
        image = preprocess_input(image)
        # get image id
        image_id = name.split('.')[0]
        images[image_id] = image
    return images

# load images
directory = 'Flicker8k_Dataset'
images = load_photos(directory)
print('Loaded Images: %d' % len(images))
```

Listing 25.7: Complete example of loading photos from file.

Running this example prints the number of loaded images. It takes a few minutes to run.

```
Loaded Images: 8091
```

Listing 25.8: Example output of loading photos from file.

25.4 Pre-Calculate Photo Features

It is possible to use a pre-trained model to extract the features from photos in the dataset and store the features to file. This is an efficiency that means that the language part of the model that turns features extracted from the photo into textual descriptions can be trained standalone from the feature extraction model. The benefit is that the very large pre-trained models do not need to be loaded, held in memory, and used to process each photo while training the language model.

Later, the feature extraction model and language model can be put back together for making predictions on new photos. In this section, we will extend the photo loading behavior developed in the previous section to load all photos, extract their features using a pre-trained VGG model, and store the extracted features to a new file that can be loaded and used to train the language model. The first step is to load the VGG model. This model is provided directly in Keras and

can be loaded as follows. Note that this will download the 500-megabyte model weights to your computer, which may take a few minutes.

```
from keras.applications.vgg16 import VGG16
# load the model
in_layer = Input(shape=(224, 224, 3))
model = VGG16(include_top=False, input_tensor=in_layer, pooling='avg')
model.summary()
```

Listing 25.9: Load the VGG mode.

This will load the VGG 16-layer model. The two `Dense` output layers as well as the classification output layer are removed from the model by setting `include_top=False`. The output from the final pooling layer is taken as the features extracted from the image. Next, we can walk over all images in the directory of images as in the previous section and call `predict()` function on the model for each prepared image to get the extracted features. The features can then be stored in a dictionary keyed on the image id. The complete example is listed below.

```
from os import listdir
from os import path
from pickle import dump
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.layers import Input

# extract features from each photo in the directory
def extract_features(directory):
    # load the model
    in_layer = Input(shape=(224, 224, 3))
    model = VGG16(include_top=False, input_tensor=in_layer)
    model.summary()
    # extract features from each photo
    features = dict()
    for name in listdir(directory):
        # load an image from file
        filename = path.join(directory, name)
        image = load_img(filename, target_size=(224, 224))
        # convert the image pixels to a numpy array
        image = img_to_array(image)
        # reshape data for the model
        image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
        # prepare the image for the VGG model
        image = preprocess_input(image)
        # get features
        feature = model.predict(image, verbose=0)
        # get image id
        image_id = name.split('.')[0]
        # store feature
        features[image_id] = feature
        print('>%s' % name)
    return features

# extract features from all images
directory = 'Flicker8k_Dataset'
```

```

features = extract_features(directory)
print('Extracted Features: %d' % len(features))
# save to file
dump(features, open('features.pkl', 'wb'))

```

Listing 25.10: Complete example of pre-calculating VGG16 photo features.

The example may take some time to complete, perhaps one hour. After all features are extracted, the dictionary is stored in the file `features.pkl` in the current working directory. These features can then be loaded later and used as input for training a language model. You could experiment with other types of pre-trained models in Keras.

25.5 How to Load Descriptions

It is important to take a moment to talk about the descriptions; there are a number available. The file `Flickr8k.token.txt` contains a list of image identifiers (used in the image filenames) and tokenized descriptions. Each image has multiple descriptions. Below is a sample of the descriptions from the file showing 5 different descriptions for a single image.

```

1305564994_00513f9a5b.jpg#0 A man in street racer armor be examine the tire of another
racer 's motorbike .
1305564994_00513f9a5b.jpg#1 Two racer drive a white bike down a road .
1305564994_00513f9a5b.jpg#2 Two motorist be ride along on their vehicle that be oddly
design and color .
1305564994_00513f9a5b.jpg#3 Two person be in a small race car drive by a green hill .
1305564994_00513f9a5b.jpg#4 Two person in race uniform in a street car .

```

Listing 25.11: Sample of raw photo descriptions.

The file `ExpertAnnotations.txt` indicates which of the descriptions for each image were written by *experts* which were written by crowdsource workers asked to describe the image. Finally, the file `CrowdFlowerAnnotations.txt` provides the frequency of crowd workers that indicate whether captions suit each image. These frequencies can be interpreted probabilistically. The authors of the paper describe the annotations as follows:

... annotators were asked to write sentences that describe the depicted scenes, situations, events and entities (people, animals, other objects). We collected multiple captions for each image because there is a considerable degree of variance in the way many images can be described.

— *Framing Image Description as a Ranking Task: Data, Models and Evaluation Metrics*, 2013.

There are also lists of the photo identifiers to use in a train/test split so that you can compare results reported in the paper. The first step is to decide which captions to use. The simplest approach is to use the first description for each photograph. First, we need a function to load the entire annotations file (`Flickr8k.token.txt`) into memory. Below is a function to do this called `load_doc()` that, given a filename, will return the document as a string.

```

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')

```

```
# read all text
text = file.read()
# close the file
file.close()
return text
```

Listing 25.12: Function for loading a file into memory.

We can see from the sample of the file above that we need only split each line by white space and take the first element as the image identifier and the rest as the image description. For example:

```
# split line by white space
tokens = line.split()
# take the first token as the image id, the rest as the description
image_id, image_desc = tokens[0], tokens[1:]
```

Listing 25.13: Example of splitting a line into an identifier and a description.

We can then clean up the image identifier by removing the filename extension and the description number.

```
# remove filename from image id
image_id = image_id.split('.')[0]
```

Listing 25.14: Example of cleaning up the photo identifier.

We can also put the description tokens back together into a string for later processing.

```
# convert description tokens back to string
image_desc = ' '.join(image_desc)
```

Listing 25.15: Example of converting the description tokens into a string.

We can put all of this together into a function. Below defines the `load_descriptions()` function that will take the loaded file, process it line-by-line, and return a dictionary of image identifiers to their first description.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# extract descriptions for images
def load_descriptions(doc):
    mapping = dict()
    # process lines
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        if len(line) < 2:
            continue
        # take the first token as the image id, the rest as the description
```

```

image_id, image_desc = tokens[0], tokens[1:]
# remove filename from image id
image_id = image_id.split('.')[0]
# convert description tokens back to string
image_desc = ' '.join(image_desc)
# store the first description for each image
if image_id not in mapping:
    mapping[image_id] = image_desc
return mapping

filename = 'Flickr8k_text/Flickr8k.token.txt'
doc = load_doc(filename)
descriptions = load_descriptions(doc)
print('Loaded: %d' % len(descriptions))

```

Listing 25.16: Complete example of loading photo descriptions.

Running the example prints the number of loaded image descriptions.

```
Loaded: 8092
```

Listing 25.17: Example output of loading photo descriptions.

There are other ways to load descriptions that may turn out to be more accurate for the data. Use the above example as a starting point and let me know what you come up with.

25.6 Prepare Description Text

The descriptions are tokenized; this means that each token is comprised of words separated by white space. It also means that punctuation are separated as their own tokens, such as periods ('.') and apostrophes for word plurals ('s). It is a good idea to clean up the description text before using it in a model. Some ideas of data cleaning we can form include:

- Normalizing the case of all tokens to lowercase.
- Remove all punctuation from tokens.
- Removing all tokens that contain one or fewer characters (after punctuation is removed), e.g. 'a' and hanging 's' characters.

We can implement these simple cleaning operations in a function that cleans each description in the loaded dictionary from the previous section. Below defines the `clean_descriptions()` function that will clean each loaded description.

```

# clean description text
def clean_descriptions(descriptions):
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    for key, desc in descriptions.items():
        # tokenize
        desc = desc.split()
        # convert to lower case
        desc = [word.lower() for word in desc]
        # remove punctuation from each word
        desc = [re_punc.sub('', word) for word in desc]
        descriptions[key] = desc

```

```

desc = [re_punc.sub(' ', w) for w in desc]
# remove hanging 's' and 'a'
desc = [word for word in desc if len(word)>1]
# store as string
descriptions[key] = ' '.join(desc)

```

Listing 25.18: Function to clean photo descriptions.

We can then save the clean text to file for later use by our model. Each line will contain the image identifier followed by the clean description. Below defines the `save_doc()` function for saving the cleaned descriptions to file.

```

# save descriptions to file, one per line
def save_doc(descriptions, filename):
    lines = list()
    for key, desc in mapping.items():
        lines.append(key + ' ' + desc)
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()

```

Listing 25.19: Function to save clean descriptions.

Putting this all together with the loading of descriptions from the previous section, the complete example is listed below.

```

import string
import re

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# extract descriptions for images
def load_descriptions(doc):
    mapping = dict()
    # process lines
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        if len(line) < 2:
            continue
        # take the first token as the image id, the rest as the description
        image_id, image_desc = tokens[0], tokens[1:]
        # remove filename from image id
        image_id = image_id.split('.')[0]
        # convert description tokens back to string
        image_desc = ' '.join(image_desc)
        # store the first description for each image
        if image_id not in mapping:

```

```

mapping[image_id] = image_desc
return mapping

# clean description text
def clean_descriptions(descriptions):
    # prepare regex for char filtering
    re_punc = re.compile('[\s]' % re.escape(string.punctuation))
    for key, desc in descriptions.items():
        # tokenize
        desc = desc.split()
        # convert to lower case
        desc = [word.lower() for word in desc]
        # remove punctuation from each word
        desc = [re_punc.sub('', w) for w in desc]
        # remove hanging 's' and 'a'
        desc = [word for word in desc if len(word)>1]
        # store as string
        descriptions[key] = ' '.join(desc)

# save descriptions to file, one per line
def save_doc(descriptions, filename):
    lines = list()
    for key, desc in descriptions.items():
        lines.append(key + ' ' + desc)
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()

filename = 'Flickr8k_text/Flickr8k.token.txt'
# load descriptions
doc = load_doc(filename)
# parse descriptions
descriptions = load_descriptions(doc)
print('Loaded: %d' % len(descriptions))
# clean descriptions
clean_descriptions(descriptions)
# summarize vocabulary
all_tokens = ' '.join(descriptions.values()).split()
vocabulary = set(all_tokens)
print('Vocabulary Size: %d' % len(vocabulary))
# save descriptions
save_doc(descriptions, 'descriptions.txt')

```

Listing 25.20: Complete example of cleaning photo descriptions.

Running the example first loads 8,092 descriptions, cleans them, summarizes the vocabulary of 4,484 unique words, then saves them to a new file called `descriptions.txt`.

Loaded: 8092
Vocabulary Size: 4484

Listing 25.21: Example output of cleaning photo descriptions.

Open the new file `descriptions.txt` in a text editor and review the contents. You should see somewhat readable descriptions of photos ready for modeling.

```
...
3139118874_599b30b116 two girls pose for picture at christmastime
2065875490_a46b58c12b person is walking on sidewalk and skeleton is on the left inside of
fence
2682382530_f9f8fd1e89 man in black shorts is stretching out his leg
3484019369_354e0b88c0 hockey team in red and white on the side of the ice rink
505955292_026f1489f2 boy rides horse
```

Listing 25.22: Sample of clean photo descriptions.

The vocabulary is still relatively large. To make modeling easier, especially the first time around, I would recommend further reducing the vocabulary by removing words that only appear once or twice across all descriptions.

25.7 Whole Description Sequence Model

There are many ways to model the caption generation problem. One naive way is to create a model that outputs the entire textual description in a one-shot manner. This is a naive model because it puts a heavy burden on the model to both interpret the meaning of the photograph and generate words, then arrange those words into the correct order.

This is not unlike the language translation problem used in an Encoder-Decoder recurrent neural network where the entire translated sentence is output one word at a time given an encoding of the input sequence. Here we would use an encoding of the image to generate the output sentence instead. The image may be encoded using a pre-trained model used for image classification, such as the VGG trained on the ImageNet model mentioned above.

The output of the model would be a probability distribution over each word in the vocabulary. The sequence would be as long as the longest photo description. The descriptions would, therefore, need to be first integer encoded where each word in the vocabulary is assigned a unique integer and sequences of words would be replaced with sequences of integers. The integer sequences would then need to be one hot encoded to represent the idealized probability distribution over the vocabulary for each word in the sequence. We can use tools in Keras to prepare the descriptions for this type of model. The first step is to load the mapping of image identifiers to clean descriptions stored in `descriptions.txt`.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load clean descriptions into memory
def load_clean_descriptions(filename):
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
```

```

# split id from description
image_id, image_desc = tokens[0], tokens[1:]
# store
mapping[image_id] = ' '.join(image_desc)
return descriptions

descriptions = load_clean_descriptions('descriptions.txt')
print('Loaded %d' % (len(descriptions)))

```

Listing 25.23: Load the cleaned descriptions.

Running this piece loads the 8,092 photo descriptions into a dictionary keyed on image identifiers. These identifiers can then be used to load each photo file for the corresponding inputs to the model.

```
Loaded 8092
```

Listing 25.24: Example output from loading the clean descriptions.

Next, we need to extract all of the description text so we can encode it.

```

# extract all text
desc_text = list(descriptions.values())

```

Listing 25.25: Convert loaded description text to a list.

We can use the Keras `Tokenizer` class to consistently map each word in the vocabulary to an integer. First, the object is created, then is fit on the description text. The fit tokenizer can later be saved to file for consistent decoding of the predictions back to vocabulary words.

```

from keras.preprocessing.text import Tokenizer
# prepare tokenizer
tokenizer = Tokenizer()
tokenizer.fit_on_texts(desc_text)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)

```

Listing 25.26: Fit a `Tokenizer` of the photo description text.

Next, we can use the fit tokenizer to encode the photo descriptions into sequences of integers.

```

# integer encode descriptions
sequences = tokenizer.texts_to_sequences(desc_text)

```

Listing 25.27: Example of integer encoding the description text.

The model will require all output sequences to have the same length for training. We can achieve this by padding all encoded sequences to have the same length as the longest encoded sequence. We can pad the sequences with 0 values after the list of words. Keras provides the `pad_sequences()` function to pad the sequences.

```

from keras.preprocessing.sequence import pad_sequences
# pad all sequences to a fixed length
max_length = max(len(s) for s in sequences)
print('Description Length: %d' % max_length)
padded = pad_sequences(sequences, maxlen=max_length, padding='post')

```

Listing 25.28: Pad descriptions to a maximum length.

Finally, we can one hot encode the padded sequences to have one sparse vector for each word in the sequence. Keras provides the `to_categorical()` function to perform this operation.

```
from keras.utils import to_categorical
# one hot encode
y = to_categorical(padded, num_classes=vocab_size)
```

Listing 25.29: Example of one hot encoding output text.

Once encoded, we can ensure that the sequence output data has the right shape for the model.

```
y = y.reshape((len(descriptions), max_length, vocab_size))
print(y.shape)
```

Listing 25.30: Example of reshaping encoded text.

Putting all of this together, the complete example is listed below.

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load clean descriptions into memory
def load_clean_descriptions(filename):
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        # split id from description
        image_id, image_desc = tokens[0], tokens[1:]
        # store
        descriptions[image_id] = ' '.join(image_desc)
    return descriptions

descriptions = load_clean_descriptions('descriptions.txt')
print('Loaded %d' % (len(descriptions)))
# extract all text
desc_text = list(descriptions.values())
# prepare tokenizer
tokenizer = Tokenizer()
tokenizer.fit_on_texts(desc_text)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# integer encode descriptions
sequences = tokenizer.texts_to_sequences(desc_text)
# pad all sequences to a fixed length
```

```

max_length = max(len(s) for s in sequences)
print('Description Length: %d' % max_length)
padded = pad_sequences(sequences, maxlen=max_length, padding='post')
# one hot encode
y = to_categorical(padded, num_classes=vocab_size)
y = y.reshape((len(descriptions), max_length, vocab_size))
print(y.shape)

```

Listing 25.31: Complete example of data preparation for a whole sequence model.

Running the example first prints the number of loaded image descriptions (8,092 photos), the dataset vocabulary size (4,485 words), the length of the longest description (28 words), then finally the shape of the data for fitting a prediction model in the form [samples, sequence length, features].

```

Loaded 8092
Vocabulary Size: 4485
Description Length: 28
(8092, 28, 4485)

```

Listing 25.32: Example output from preparing data for a whole sequence prediction model.

As mentioned, outputting the entire sequence may be challenging for the model. We will look at a simpler model in the next section.

25.8 Word-By-Word Model

A simpler model for generating a caption for photographs is to generate one word given both the image as input and the last word generated. This model would then have to be called recursively to generate each word in the description with previous predictions as input. Using the word as input, give the model a forced context for predicting the next word in the sequence.

This is the model used in prior research, such as: *Show and Tell: A Neural Image Caption Generator*, 2015. A word embedding layer can be used to represent the input words. Like the feature extraction model for the photos, this too can be pre-trained either on a large corpus or on the dataset of all descriptions.

The model would take a full sequence of words as input; the length of the sequence would be the maximum length of descriptions in the dataset. The model must be started with something. One approach is to surround each photo description with special tags to signal the start and end of the description, such as STARTDESC and ENDDESC. For example, the description:

```
boy rides horse
```

Listing 25.33: Example of a photo description.

Would become:

```
STARTDESC boy rides horse ENDDESC
```

Listing 25.34: Example of a wrapped photo description.

And would be fed to the model with the same image input to result in the following input-output word sequence pairs:

Input (X),	Output (y)
STARTDESC,	boy
STARTDESC, boy,	rides
STARTDESC, boy, rides,	horse
STARTDESC, boy, rides, horse	ENDDESC

Listing 25.35: Example input-output pairs for a wrapped description.

The data preparation would begin much the same as was described in the previous section. Each description must be integer encoded. After encoding, the sequences are split into multiple input and output pairs and only the output word (y) is one hot encoded. This is because the model is only required to predict the probability distribution of one word at a time. The code is the same up to the point where we calculate the maximum length of sequences.

```
...
descriptions = load_clean_descriptions('descriptions.txt')
print('Loaded %d' % (len(descriptions)))
# extract all text
desc_text = list(descriptions.values())
# prepare tokenizer
tokenizer = Tokenizer()
tokenizer.fit_on_texts(desc_text)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# integer encode descriptions
sequences = tokenizer.texts_to_sequences(desc_text)
# determine the maximum sequence length
max_length = max(len(s) for s in sequences)
print('Description Length: %d' % max_length)
```

Listing 25.36: Example of loading and encoding photo descriptions.

Next, we split the each integer encoded sequence into input and output pairs. Let's step through a single sequence called seq at the i 'th word in the sequence, where i more than or equal to 1. First, we take the first $i-1$ words as the input sequence and the i 'th word as the output word.

```
# split into input and output pair
in_seq, out_seq = seq[:i], seq[i]
```

Listing 25.37: Example of splitting a description sequence.

Next, the input sequence is padded to the maximum length of the input sequences. Pre-padding is used (the default) so that new words appear at the end of the sequence, instead of the input beginning.

```
# pad input sequence
in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
```

Listing 25.38: Example of padding a split description sequence.

The output word is one hot encoded, much like in the previous section.

```
# encode output sequence
out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
```

Listing 25.39: Example of one hot encoding the output word.

We can put all of this together into a complete example to prepare description data for the word-by-word model.

```

from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load clean descriptions into memory
def load_clean_descriptions(filename):
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        # split id from description
        image_id, image_desc = tokens[0], tokens[1:]
        # store
        descriptions[image_id] = ' '.join(image_desc)
    return descriptions

descriptions = load_clean_descriptions('descriptions.txt')
print('Loaded %d' % (len(descriptions)))
# extract all text
desc_text = list(descriptions.values())
# prepare tokenizer
tokenizer = Tokenizer()
tokenizer.fit_on_texts(desc_text)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# integer encode descriptions
sequences = tokenizer.texts_to_sequences(desc_text)
# determine the maximum sequence length
max_length = max(len(s) for s in sequences)
print('Description Length: %d' % max_length)

X, y = list(), list()
for img_no, seq in enumerate(sequences):
    # split one sequence into multiple X,y pairs
    for i in range(1, len(seq)):
        # split into input and output pair
        in_seq, out_seq = seq[:i], seq[i]
        # pad input sequence
        in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
        # encode output sequence
        out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
        X.append(in_seq)
        y.append(out_seq)

```

```

# store
X.append(in_seq)
y.append(out_seq)

# convert to numpy arrays
X, y = array(X), array(y)
print(X.shape)
print(y.shape)

```

Listing 25.40: Complete example of data preparation for a word-by-word model.

Running the example prints the same statistics, but prints the size of the resulting encoded input and output sequences. Note that the input of images must follow the exact same ordering where the same photo is shown for each example drawn from a single description. One way to do this would be to load the photo and store it for each example prepared from a single description.

```

Loaded 8092
Vocabulary Size: 4485
Description Length: 28
(66456, 28)
(66456, 4485)

```

Listing 25.41: Example output of data preparation for a word-by-word prediction model.

25.9 Progressive Loading

The Flickr8K dataset of photos and descriptions can fit into RAM, if you have a lot of RAM (e.g. 8 Gigabytes or more), and most modern systems do. This is fine if you want to fit a deep learning model using the CPU. Alternately, if you want to fit a model using a GPU, then you will not be able to fit the data into memory of an average GPU video card. One solution is to progressively load the photos and descriptions as-needed by the model.

Keras supports progressively loaded datasets by using the `fit_generator()` function on the model. A generator is the term used to describe a function used to return batches of samples for the model to train on. This can be as simple as a standalone function, the name of which is passed to the `fit_generator()` function when fitting the model. As a reminder, a model is fit for multiple epochs, where one epoch is one pass through the entire training dataset, such as all photos. One epoch is comprised of multiple batches of examples where the model weights are updated at the end of each batch.

A generator must create and yield one batch of examples. For example, the average sentence length in the dataset is 11 words; that means that each photo will result in 11 examples for fitting the model and two photos will result in about 22 examples on average. A good default batch size for modern hardware may be 32 examples, so that is about 2-3 photos worth of examples.

We can write a custom generator to load a few photos and return the samples as a single batch. Let's assume we are working with a word-by-word model described in the previous section that expects a sequence of words and a prepared image as input and predicts a single word. Let's design a data generator that given a loaded dictionary of image identifiers to clean descriptions, a trained tokenizer, and a maximum sequence length will load one-image worth of examples for each batch.

A generator must loop forever and yield each batch of samples. We can loop forever with a while loop and within this, loop over each image in the image directory. For each image filename, we can load the image and create all of the input-output sequence pairs from the image's description. Below is the data generator function.

```
def data_generator(mapping, tokenizer, max_length):
    # loop for ever over images
    directory = 'Flicker8k_Dataset'
    while 1:
        for name in listdir(directory):
            # load an image from file
            filename = directory + '/' + name
            image, image_id = load_image(filename)
            # create word sequences
            desc = mapping[image_id]
            in_img, in_seq, out_word = create_sequences(tokenizer, max_length, desc, image)
            yield [[in_img, in_seq], out_word]
```

Listing 25.42: Example of a generator for progressive loading.

You could extend it to take the name of the dataset directory as a parameter. The generator returns an array containing the inputs (X) and output (y) for the model. The input is comprised of an array with two items for the input images and encoded word sequences. The outputs are one hot encoded words. You can see that it calls a function called `load_photo()` to load a single photo and return the pixels and image identifier. This is a simplified version of the photo loading function developed at the beginning of this tutorial.

```
# load a single photo intended as input for the VGG feature extractor model
def load_photo(filename):
    image = load_img(filename, target_size=(224, 224))
    # convert the image pixels to a NumPy array
    image = img_to_array(image)
    # reshape data for the model
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
    # prepare the image for the VGG model
    image = preprocess_input(image)[0]
    # get image id
    image_id = filename.split('/')[-1].split('.')[0]
    return image, image_id
```

Listing 25.43: Example of a function for loading and preparing a photo.

Another function named `create_sequences()` is called to create sequences of images, input sequences of words, and output words that we then yield to the caller. This is a function that includes everything discussed in the previous section, and also creates copies of the image pixels, one for each input-output pair created from the photo's description.

```
# create sequences of images, input sequences and output words for an image
def create_sequences(tokenizer, max_length, descriptions, images):
    Ximages, XSeq, y = list(), list(), list()
    vocab_size = len(tokenizer.word_index) + 1
    for j in range(len(descriptions)):
        seq = descriptions[j]
        image = images[j]
        # integer encode
        seq = tokenizer.texts_to_sequences([seq])[0]
```

```
# split one sequence into multiple X,y pairs
for i in range(1, len(seq)):
    # select
    in_seq, out_seq = seq[:i], seq[i]
    # pad input sequence
    in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
    # encode output sequence
    out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
    # store
    Ximages.append(image)
    XSeq.append(in_seq)
    y.append(out_seq)
Ximages, XSeq, y = array(Ximages), array(XSeq), array(y)
return Ximages, XSeq, y
```

Listing 25.44: Example of a function for preparing description text.

Prior to preparing the model that uses the data generator, we must load the clean descriptions, prepare the tokenizer, and calculate the maximum sequence length. All 3 of must be passed to the `data_generator()` as parameters. We use the same `load_clean_descriptions()` function developed previously and a new `create_tokenizer()` function that simplifies the creation of the tokenizer. Tying all of this together, the complete data generator is listed below, ready for use to train a model.

```
from os import listdir
from os import path
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load clean descriptions into memory
def load_clean_descriptions(filename):
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        # split id from description
        image_id, image_desc = tokens[0], tokens[1:]
        # store
        descriptions[image_id] = ' '.join(image_desc)
    return descriptions
```

```
# fit a tokenizer given caption descriptions
def create_tokenizer(descriptions):
    lines = list(descriptions.values())
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# load a single photo intended as input for the VGG feature extractor model
def load_photo(filename):
    image = load_img(filename, target_size=(224, 224))
    # convert the image pixels to a numpy array
    image = img_to_array(image)
    # reshape data for the model
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
    # prepare the image for the VGG model
    image = preprocess_input(image)[0]
    # get image id
    image_id = path.basename(filename).split('.')[0]
    return image, image_id

# create sequences of images, input sequences and output words for an image
def create_sequences(tokenizer, max_length, desc, image):
    Ximages, XSeq, y = list(), list(), list()
    vocab_size = len(tokenizer.word_index) + 1
    # integer encode the description
    seq = tokenizer.texts_to_sequences([desc])[0]
    # split one sequence into multiple X,y pairs
    for i in range(1, len(seq)):
        # select
        in_seq, out_seq = seq[:i], seq[i]
        # pad input sequence
        in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
        # encode output sequence
        out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
        # store
        Ximages.append(image)
        XSeq.append(in_seq)
        y.append(out_seq)
    Ximages, XSeq, y = array(Ximages), array(XSeq), array(y)
    return [Ximages, XSeq, y]

# data generator, intended to be used in a call to model.fit_generator()
def data_generator(descriptions, tokenizer, max_length):
    # loop for ever over images
    directory = 'Flicker8k_Dataset'
    while 1:
        for name in listdir(directory):
            # load an image from file
            filename = path.join(directory, name)
            image, image_id = load_photo(filename)
            # create word sequences
            desc = descriptions[image_id]
            in_img, in_seq, out_word = create_sequences(tokenizer, max_length, desc, image)
            yield [[in_img, in_seq], out_word]
```

```
# load mapping of ids to descriptions
descriptions = load_clean_descriptions('descriptions.txt')
# integer encode sequences of words
tokenizer = create_tokenizer(descriptions)
# pad to fixed length
max_length = max(len(s.split()) for s in list(descriptions.values()))
print('Description Length: %d' % max_length)
# test the data generator
generator = data_generator(descriptions, tokenizer, max_length)
inputs, outputs = next(generator)
print(inputs[0].shape)
print(inputs[1].shape)
print(outputs.shape)
```

Listing 25.45: Complete example of progressive loading.

A data generator can be tested by calling the `next()` function. We can test the generator as follows.

```
# test the data generator
generator = data_generator(descriptions, tokenizer, max_length)
inputs, outputs = next(generator)
print(inputs[0].shape)
print(inputs[1].shape)
print(outputs.shape)
```

Listing 25.46: Example of testing the custom generator function.

Running the example prints the shape of the input and output example for a single batch (e.g. 13 input-output pairs):

```
(13, 224, 224, 3)
(13, 28)
(13, 4485)
```

Listing 25.47: Example output from testing the generator function.

The generator can be used to fit a model by calling the `fit_generator()` function on the model (instead of `fit()`) and passing in the generator. We must also specify the number of steps or batches per epoch. We could estimate this as (10 x training dataset size), perhaps 70,000 if 7,000 images are used for training.

```
# define model
# ...
# fit model
model.fit_generator(data_generator(descriptions, tokenizer, max_length),
    steps_per_epoch=70000, ...)
```

Listing 25.48: Example of using the progressive loading data generator when fitting a Keras model.

25.10 Further Reading

This section provides more resources on the topic if you are looking go deeper.

25.10.1 Flickr8K Dataset

- *Framing image description as a ranking task: data, models and evaluation metrics* (Homepage).
http://nlp.cs.illinois.edu/HockenmaierGroup/Framing_Image_Description/KCCA.html
- *Framing Image Description as a Ranking Task: Data, Models and Evaluation Metrics*, 013.
<https://www.jair.org/media/3994/live-3994-7274-jair.pdf>
- Dataset Request Form.
<https://illinois.edu/fb/sec/1713398>
- Old Flickr8K Homepage.
<http://nlp.cs.illinois.edu/HockenmaierGroup/8k-pictures.html>

25.10.2 API

- Python Generators.
<https://wiki.python.org/moin/Generators>
- Keras Model API.
<https://keras.io/models/model/>
- Keras pad_sequences() API.
https://keras.io/preprocessing/sequence/#pad_sequences
- Keras Tokenizer API.
<https://keras.io/preprocessing/text/#tokenizer>
- Keras VGG16 API.
<https://keras.io/applications/#vgg16>

25.11 Summary

In this tutorial, you discovered how to prepare photos and textual descriptions ready for developing an automatic photo caption generation model. Specifically, you learned:

- About the Flickr8K dataset comprised of more than 8,000 photos and up to 5 captions for each photo.
- How to generally load and prepare photo and text data for modeling with deep learning.
- How to specifically encode data for two different types of deep learning models in Keras.

25.11.1 Next

In the next chapter, you will discover how you can develop a model for automatic caption generation.

Chapter 26

Project: Develop a Neural Image Caption Generation Model

Caption generation is a challenging artificial intelligence problem where a textual description must be generated for a given photograph. It requires both methods from computer vision to understand the content of the image and a language model from the field of natural language processing to turn the understanding of the image into words in the right order. Recently, deep learning methods have achieved state-of-the-art results on examples of this problem.

Deep learning methods have demonstrated state-of-the-art results on caption generation problems. What is most impressive about these methods is a single end-to-end model can be defined to predict a caption, given a photo, instead of requiring sophisticated data preparation or a pipeline of specifically designed models. In this tutorial, you will discover how to develop a photo captioning deep learning model from scratch. After completing this tutorial, you will know:

- How to prepare photo and text data for training a deep learning model.
- How to design and train a deep learning caption generation model.
- How to evaluate a train caption generation model and use it to caption entirely new photographs.

Let's get started.

26.1 Tutorial Overview

This tutorial is divided into the following parts:

1. Photo and Caption Dataset
2. Prepare Photo Data
3. Prepare Text Data
4. Develop Deep Learning Model
5. Evaluate Model
6. Generate New Captions

26.2 Photo and Caption Dataset

In this tutorial, we will use the Flickr8k dataset. This dataset was introduced previously in Chapter 25. The dataset is available for free. You must complete a request form and the links to the dataset will be emailed to you. I would love to link to them for you, but the email address expressly requests: *Please do not redistribute the dataset.* You can use the link below to request the dataset:

- Dataset Request Form.
<https://illinois.edu/fb/sec/1713398>

Within a short time, you will receive an email that contains links to two files:

- **Flickr8k_Dataset.zip** (1 Gigabyte) An archive of all photographs.
- **Flickr8k_text.zip** (2.2 Megabytes) An archive of all text descriptions for photographs.

Download the datasets and unzip them into your current working directory. You will have two directories:

- **Flicker8k_Dataset**: Contains 8092 photographs in JPEG format (yes the directory name spells it ‘Flicker’ not ‘Flickr’).
- **Flickr8k_text**: Contains a number of files containing different sources of descriptions for the photographs.

The dataset has a pre-defined training dataset (6,000 images), development dataset (1,000 images), and test dataset (1,000 images). One measure that can be used to evaluate the skill of the model are BLEU scores. For reference, below are some ball-park BLEU scores for skillful models when evaluated on the test dataset (taken from the 2017 paper *Where to put the Image in an Image Caption Generator*):

- BLEU-1: 0.401 to 0.578.
- BLEU-2: 0.176 to 0.390.
- BLEU-3: 0.099 to 0.260.
- BLEU-4: 0.059 to 0.170.

We describe the BLEU metric more later when we work on evaluating our model. Next, let’s look at how to load the images.

26.3 Prepare Photo Data

We will use a pre-trained model to interpret the content of the photos. There are many models to choose from. In this case, we will use the Oxford Visual Geometry Group, or VGG, model that won the ImageNet competition in 2014. Keras provides this pre-trained model directly. Note, the first time you use this model, Keras will download the model weights from the Internet, which are about 500 Megabytes. This may take a few minutes depending on your internet connection. Note the use of the VGG pre-trained model was introduced in Chapter 23.

We could use this model as part of a broader image caption model. The problem is, it is a large model and running each photo through the network every time we want to test a new language model configuration (downstream) is redundant. Instead, we can pre-compute the *photo features* using the pre-trained model and save them to file. We can then load these features later and feed them into our model as the interpretation of a given photo in the dataset. It is no different to running the photo through the full VGG model; it is just we will have done it once in advance.

This is an optimization that will make training our models faster and consume less memory. We can load the VGG model in Keras using the VGG class. We will remove the last layer from the loaded model, as this is the model used to predict a classification for a photo. We are not interested in classifying images, but we are interested in the internal representation of the photo right before a classification is made. These are the *features* that the model has extracted from the photo.

Keras also provides tools for reshaping the loaded photo into the preferred size for the model (e.g. 3 channel 224 x 224 pixel image). Below is a function named `extract_features()` that, given a directory name, will load each photo, prepare it for VGG, and collect the predicted features from the VGG model. The image features are a 1-dimensional 4,096 element vector.

The function returns a dictionary of image identifier to image features.

```
# extract features from each photo in the directory
def extract_features(directory):
    # load the model
    model = VGG16()
    # re-structure the model
    model.layers.pop()
    model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
    # summarize
    model.summary()
    # extract features from each photo
    features = dict()
    for name in.listdir(directory):
        # load an image from file
        filename = directory + '/' + name
        image = load_img(filename, target_size=(224, 224))
        # convert the image pixels to a NumPy array
        image = img_to_array(image)
        # reshape data for the model
        image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
        # prepare the image for the VGG model
        image = preprocess_input(image)
        # get features
        feature = model.predict(image, verbose=0)
        # get image id
        features[name] = feature
```

```

image_id = name.split('.')[0]
# store feature
features[image_id] = feature
print('>%s' % name)
return features

```

Listing 26.1: Function to extract photo features

We can call this function to prepare the photo data for testing our models, then save the resulting dictionary to a file named `features.pkl`. The complete example is listed below.

```

from os import listdir
from os import path
from pickle import dump
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.models import Model

# extract features from each photo in the directory
def extract_features(directory):
    # load the model
    model = VGG16()
    # re-structure the model
    model.layers.pop()
    model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
    # summarize
    model.summary()
    # extract features from each photo
    features = dict()
    for name in listdir(directory):
        # load an image from file
        filename = path.join(directory, name)
        image = load_img(filename, target_size=(224, 224))
        # convert the image pixels to a numpy array
        image = img_to_array(image)
        # reshape data for the model
        image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
        # prepare the image for the VGG model
        image = preprocess_input(image)
        # get features
        feature = model.predict(image, verbose=0)
        # get image id
        image_id = name.split('.')[0]
        # store feature
        features[image_id] = feature
        print('>%s' % name)
    return features

# extract features from all images
directory = 'Flicker8k_Dataset'
features = extract_features(directory)
print('Extracted Features: %d' % len(features))
# save to file
dump(features, open('features.pkl', 'wb'))

```

Listing 26.2: Complete example of extracting photo features.

Running this data preparation step may take a while depending on your hardware, perhaps one hour on the CPU with a modern workstation. At the end of the run, you will have the extracted features stored in `features.pkl` for later use. This file will be a few hundred Megabytes in size.

26.4 Prepare Text Data

The dataset contains multiple descriptions for each photograph and the text of the descriptions requires some minimal cleaning. Note, a fuller investigation into how this text data can be prepared was described in Chapter 25. First, we will load the file containing all of the descriptions.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

filename = 'Flickr8k_text/Flickr8k.token.txt'
# load descriptions
doc = load_doc(filename)
```

Listing 26.3: Example of loading photo descriptions into memory

Each photo has a unique identifier. This identifier is used on the photo filename and in the text file of descriptions. Next, we will step through the list of photo descriptions. Below defines a function `load_descriptions()` that, given the loaded document text, will return a dictionary of photo identifiers to descriptions. Each photo identifier maps to a list of one or more textual descriptions.

```
# extract descriptions for images
def load_descriptions(doc):
    mapping = dict()
    # process lines
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        if len(line) < 2:
            continue
        # take the first token as the image id, the rest as the description
        image_id, image_desc = tokens[0], tokens[1:]
        # remove filename from image id
        image_id = image_id.split('.')[0]
        # convert description tokens back to string
        image_desc = ' '.join(image_desc)
        # create the list if needed
```

```

if image_id not in mapping:
    mapping[image_id] = list()
# store description
mapping[image_id].append(image_desc)
return mapping

# parse descriptions
descriptions = load_descriptions(doc)
print('Loaded: %d' % len(descriptions))

```

Listing 26.4: Example of splitting descriptions from photo identifiers

Next, we need to clean the description text. The descriptions are already tokenized and easy to work with. We will clean the text in the following ways in order to reduce the size of the vocabulary of words we will need to work with:

- Convert all words to lowercase.
- Remove all punctuation.
- Remove all words that are one character or less in length (e.g. ‘a’).
- Remove all words with numbers in them.

Below defines the `clean_descriptions()` function that, given the dictionary of image identifiers to descriptions, steps through each description and cleans the text.

```

def clean_descriptions(descriptions):
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    for key, desc_list in descriptions.items():
        for i in range(len(desc_list)):
            desc = desc_list[i]
            # tokenize
            desc = desc.split()
            # convert to lower case
            desc = [word.lower() for word in desc]
            # remove punctuation from each token
            desc = [re_punc.sub('', w) for w in desc]
            # remove hanging 's' and 'a'
            desc = [word for word in desc if len(word)>1]
            # remove tokens with numbers in them
            desc = [word for word in desc if word.isalpha()]
            # store as string
            desc_list[i] = ' '.join(desc)

    # clean descriptions
    clean_descriptions(descriptions)

```

Listing 26.5: Example of cleaning description text

Once cleaned, we can summarize the size of the vocabulary. Ideally, we want a vocabulary that is both expressive and as small as possible. A smaller vocabulary will result in a smaller model that will train faster. For reference, we can transform the clean descriptions into a set and print its size to get an idea of the size of our dataset vocabulary.

```
# convert the loaded descriptions into a vocabulary of words
def to_vocabulary(descriptions):
    # build a list of all description strings
    all_desc = set()
    for key in descriptions.keys():
        [all_desc.update(d.split()) for d in descriptions[key]]
    return all_desc

# summarize vocabulary
vocabulary = to_vocabulary(descriptions)
print('Vocabulary Size: %d' % len(vocabulary))
```

Listing 26.6: Example of defining the description text vocabulary

Finally, we can save the dictionary of image identifiers and descriptions to a new file named `descriptions.txt`, with one image identifier and description per line. Below defines the `save_doc()` function that, given a dictionary containing the mapping of identifiers to descriptions and a filename, saves the mapping to file.

```
# save descriptions to file, one per line
def save_descriptions(descriptions, filename):
    lines = list()
    for key, desc_list in descriptions.items():
        for desc in desc_list:
            lines.append(key + ' ' + desc)
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()

# save descriptions
save_doc(descriptions, 'descriptions.txt')
```

Listing 26.7: Example of saving clean descriptions to file

Putting this all together, the complete listing is provided below.

```
import string
import re

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# extract descriptions for images
def load_descriptions(doc):
    mapping = dict()
    # process lines
    for line in doc.split('\n'):
        # split line by white space
```

```
tokens = line.split()
if len(line) < 2:
    continue
# take the first token as the image id, the rest as the description
image_id, image_desc = tokens[0], tokens[1:]
# remove filename from image id
image_id = image_id.split('.')[0]
# convert description tokens back to string
image_desc = ' '.join(image_desc)
# create the list if needed
if image_id not in mapping:
    mapping[image_id] = list()
# store description
mapping[image_id].append(image_desc)
return mapping

def clean_descriptions(descriptions):
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    for _, desc_list in descriptions.items():
        for i in range(len(desc_list)):
            desc = desc_list[i]
            # tokenize
            desc = desc.split()
            # convert to lower case
            desc = [word.lower() for word in desc]
            # remove punctuation from each token
            desc = [re_punc.sub('', w) for w in desc]
            # remove hanging 's' and 'a'
            desc = [word for word in desc if len(word)>1]
            # remove tokens with numbers in them
            desc = [word for word in desc if word.isalpha()]
            # store as string
            desc_list[i] = ' '.join(desc)

# convert the loaded descriptions into a vocabulary of words
def to_vocabulary(descriptions):
    # build a list of all description strings
    all_desc = set()
    for key in descriptions.keys():
        [all_desc.update(d.split()) for d in descriptions[key]]
    return all_desc

# save descriptions to file, one per line
def save_descriptions(descriptions, filename):
    lines = list()
    for key, desc_list in descriptions.items():
        for desc in desc_list:
            lines.append(key + ' ' + desc)
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()

filename = 'Flickr8k_text/Flickr8k.token.txt'
# load descriptions
```

```

doc = load_doc(filename)
# parse descriptions
descriptions = load_descriptions(doc)
print('Loaded: %d' % len(descriptions))
# clean descriptions
clean_descriptions(descriptions)
# summarize vocabulary
vocabulary = to_vocabulary(descriptions)
print('Vocabulary Size: %d' % len(vocabulary))
# save to file
save_descriptions(descriptions, 'descriptions.txt')

```

Listing 26.8: Complete example of text data preparation.

Running the example first prints the number of loaded photo descriptions (8,092) and the size of the clean vocabulary (8,763 words).

```

Loaded: 8,092
Vocabulary Size: 8,763

```

Listing 26.9: Example output from preparing the text data

Finally, the clean descriptions are written to `descriptions.txt`. Taking a look at the file, we can see that the descriptions are ready for modeling. The order of descriptions in your file may vary.

```

2252123185_487f21e336 bunch on people are seated in stadium
2252123185_487f21e336 crowded stadium is full of people watching an event
2252123185_487f21e336 crowd of people fill up packed stadium
2252123185_487f21e336 crowd sitting in an indoor stadium
2252123185_487f21e336 stadium full of people watch game
...

```

Listing 26.10: Sample of text from the clean photo descriptions

26.5 Develop Deep Learning Model

In this section, we will define the deep learning model and fit it on the training dataset. This section is divided into the following parts:

1. Loading Data.
2. Defining the Model.
3. Fitting the Model.
4. Complete Example.

26.5.1 Loading Data

First, we must load the prepared photo and text data so that we can use it to fit the model. We are going to train the data on all of the photos and captions in the training dataset. While

training, we are going to monitor the performance of the model on the development dataset and use that performance to decide when to save models to file.

The train and development dataset have been predefined in the `Flickr_8k.trainImages.txt` and `Flickr_8k.devImages.txt` files respectively, that both contain lists of photo file names. From these file names, we can extract the photo identifiers and use these identifiers to filter photos and descriptions for each set. The function `load_set()` below will load a pre-defined set of identifiers given the train or development sets filename.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load a pre-defined list of photo identifiers
def load_set(filename):
    doc = load_doc(filename)
    dataset = list()
    # process line by line
    for line in doc.split('\n'):
        # skip empty lines
        if len(line) < 1:
            continue
        # get the image identifier
        identifier = line.split('.')[0]
        dataset.append(identifier)
    return set(dataset)
```

Listing 26.11: Functions for loading the photo description text and identifiers

Now, we can load the photos and descriptions using the pre-defined set of train or development identifiers. Below is the function `load_clean_descriptions()` that loads the cleaned text descriptions from `descriptions.txt` for a given set of identifiers and returns a dictionary of identifiers to lists of text descriptions.

The model we will develop will generate a caption given a photo, and the caption will be generated one word at a time. The sequence of previously generated words will be provided as input. Therefore, we will need a *first word* to kick-off the generation process and a *last word* to signal the end of the caption. We will use the strings `startseq` and `endseq` for this purpose. These tokens are added to the loaded descriptions as they are loaded. It is important to do this now before we encode the text so that the tokens are also encoded correctly.

```
# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
    # load document
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        # split id from description
```

```

image_id, image_desc = tokens[0], tokens[1:]
# skip images not in the set
if image_id in dataset:
    # create list
    if image_id not in descriptions:
        descriptions[image_id] = list()
    # wrap description in tokens
    desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
    # store
    descriptions[image_id].append(desc)
return descriptions

```

Listing 26.12: Function for loading the clean photo descriptions

Next, we can load the photo features for a given dataset. Below defines a function named `load_photo_features()` that loads the entire set of photo descriptions, then returns the subset of interest for a given set of photo identifiers. This is not very efficient; nevertheless, this will get us up and running quickly.

```

# load photo features
def load_photo_features(filename, dataset):
    # load all features
    all_features = load(open(filename, 'rb'))
    # filter features
    features = {k: all_features[k] for k in dataset}
    return features

```

Listing 26.13: Function for loading pre-calculated photo features

We can pause here and test everything developed so far. The complete code example is listed below.

```

from pickle import load

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load a pre-defined list of photo identifiers
def load_set(filename):
    doc = load_doc(filename)
    dataset = list()
    # process line by line
    for line in doc.split('\n'):
        # skip empty lines
        if len(line) < 1:
            continue
        # get the image identifier
        identifier = line.split('.')[0]
        dataset.append(identifier)
    return set(dataset)

```

```

# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
    # load document
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        # split id from description
        image_id, image_desc = tokens[0], tokens[1:]
        # skip images not in the set
        if image_id in dataset:
            # create list
            if image_id not in descriptions:
                descriptions[image_id] = list()
            # wrap description in tokens
            desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
            # store
            descriptions[image_id].append(desc)
    return descriptions

# load photo features
def load_photo_features(filename, dataset):
    # load all features
    all_features = load(open(filename, 'rb'))
    # filter features
    features = {k: all_features[k] for k in dataset}
    return features

# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# photo features
train_features = load_photo_features('features.pkl', train)
print('Photos: train=%d' % len(train_features))

```

Listing 26.14: Complete example of loading the prepared data.

Running this example first loads the 6,000 photo identifiers in the test dataset. These features are then used to filter and load the cleaned description text and the pre-computed photo features. We are nearly there.

```

Dataset: 6,000
Descriptions: train=6,000
Photos: train=6,000

```

Listing 26.15: Example output from preparing the text data

The description text will need to be encoded to numbers before it can be presented to the model as input or compared to the model's predictions. The first step in encoding the data is to create a consistent mapping from words to unique integer values. Keras provides

the `Tokenizer` class that can learn this mapping from the loaded description data. Below defines the `to_lines()` to convert the dictionary of descriptions into a list of strings and the `create_tokenizer()` function that will fit a `Tokenizer` given the loaded photo description text.

```
# convert a dictionary of clean descriptions to a list of descriptions
def to_lines(descriptions):
    all_desc = []
    for key in descriptions.keys():
        [all_desc.append(d) for d in descriptions[key]]
    return all_desc

# fit a tokenizer given caption descriptions
def create_tokenizer(descriptions):
    lines = to_lines(descriptions)
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
```

Listing 26.16: Example of preparing the Tokenizer

We can now encode the text. Each description will be split into words. The model will be provided one word and the photo and generate the next word. Then the first two words of the description will be provided to the model as input with the image to generate the next word. This is how the model will be trained. For example, the input sequence “*little girl running in field*” would be split into 6 input-output pairs to train the model:

X1,	X2 (text sequence),	y (word)
photo	startseq,	little
photo	startseq, little,	girl
photo	startseq, little, girl,	running
photo	startseq, little, girl, running,	in
photo	startseq, little, girl, running, in,	field
photo	startseq, little, girl, running, in, field,	endseq

Listing 26.17: Example of how a photo description is transformed into input and output sequences

Later, when the model is used to generate descriptions, the generated words will be concatenated and recursively provided as input to generate a caption for an image. The function below named `create_sequences()`, given the tokenizer, a maximum sequence length, and the dictionary of all descriptions and photos, will transform the data into input-output pairs of data for training the model. There are two input arrays to the model: one for photo features and one for the encoded text. There is one output for the model which is the encoded next word in the text sequence.

The input text is encoded as integers, which will be fed to a word embedding layer. The photo features will be fed directly to another part of the model. The model will output a prediction, which will be a probability distribution over all words in the vocabulary. The output data will therefore be a one hot encoded version of each word, representing an idealized

probability distribution with 0 values at all word positions except the actual word position, which has a value of 1.

```
# create sequences of images, input sequences and output words for an image
def create_sequences(tokenizer, max_length, descriptions, photos):
    X1, X2, y = list(), list(), list()
    # walk through each image identifier
    for key, desc_list in descriptions.items():
        # walk through each description for the image
        for desc in desc_list:
            # encode the sequence
            seq = tokenizer.texts_to_sequences([desc])[0]
            # split one sequence into multiple X,y pairs
            for i in range(1, len(seq)):
                # split into input and output pair
                in_seq, out_seq = seq[:i], seq[i]
                # pad input sequence
                in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
                # encode output sequence
                out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
                # store
                X1.append(photos[key][0])
                X2.append(in_seq)
                y.append(out_seq)
    return array(X1), array(X2), array(y)
```

Listing 26.18: Function for creating input and output sequences

We will need to calculate the maximum number of words in the longest description. A short helper function named `max_length()` is defined below.

```
# calculate the length of the description with the most words
def max_length(descriptions):
    lines = to_lines(descriptions)
    return max(len(d.split()) for d in lines)
```

Listing 26.19: Function for calculating the maximum sequence length.

We now have enough to load the data for the training and development datasets and transform the loaded data into input-output pairs for fitting a deep learning model.

26.5.2 Defining the Model

We will define a deep learning based on the *merge-model* described by Marc Tanti, et al. in their 2017 papers. Note, the merge model for image captioning was introduced in Chapter 22. We will describe the model in three parts:

- **Photo Feature Extractor.** This is a 16-layer VGG model pre-trained on the ImageNet dataset. We have pre-processed the photos with the VGG model (without the output layer) and will use the extracted features predicted by this model as input.
- **Sequence Processor.** This is a word embedding layer for handling the text input, followed by a Long Short-Term Memory (LSTM) recurrent neural network layer.

- **Decoder** (for lack of a better name). Both the feature extractor and sequence processor output a fixed-length vector. These are merged together and processed by a **Dense** layer to make a final prediction.

The Photo Feature Extractor model expects input photo features to be a vector of 4,096 elements. These are processed by a **Dense** layer to produce a 256 element representation of the photo. The Sequence Processor model expects input sequences with a pre-defined length (34 words) which are fed into an **Embedding** layer that uses a mask to ignore padded values. This is followed by an LSTM layer with 256 memory units.

Both the input models produce a 256 element vector. Further, both input models use regularization in the form of 50% dropout. This is to reduce overfitting the training dataset, as this model configuration learns very fast. The Decoder model merges the vectors from both input models using an addition operation. This is then fed to a Dense 256 neuron layer and then to a final output Dense layer that makes a softmax prediction over the entire output vocabulary for the next word in the sequence. The function below named `define_model()` defines and returns the model ready to be fit.

```
# define the captioning model
def define_model(vocab_size, max_length):
    # feature extractor model
    inputs1 = Input(shape=(4096,))
    fe1 = Dropout(0.5)(inputs1)
    fe2 = Dense(256, activation='relu')(fe1)
    # sequence model
    inputs2 = Input(shape=(max_length,))
    se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
    se2 = Dropout(0.5)(se1)
    se3 = LSTM(256)(se2)
    # decoder model
    decoder1 = add([fe2, se3])
    decoder2 = Dense(256, activation='relu')(decoder1)
    outputs = Dense(vocab_size, activation='softmax')(decoder2)
    # tie it together [image, seq] [word]
    model = Model(inputs=[inputs1, inputs2], outputs=outputs)
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam')
    # summarize model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model
```

Listing 26.20: Function to define the caption generation model.

A plot of the model is created and helps to better understand the structure of the network and the two streams of input.

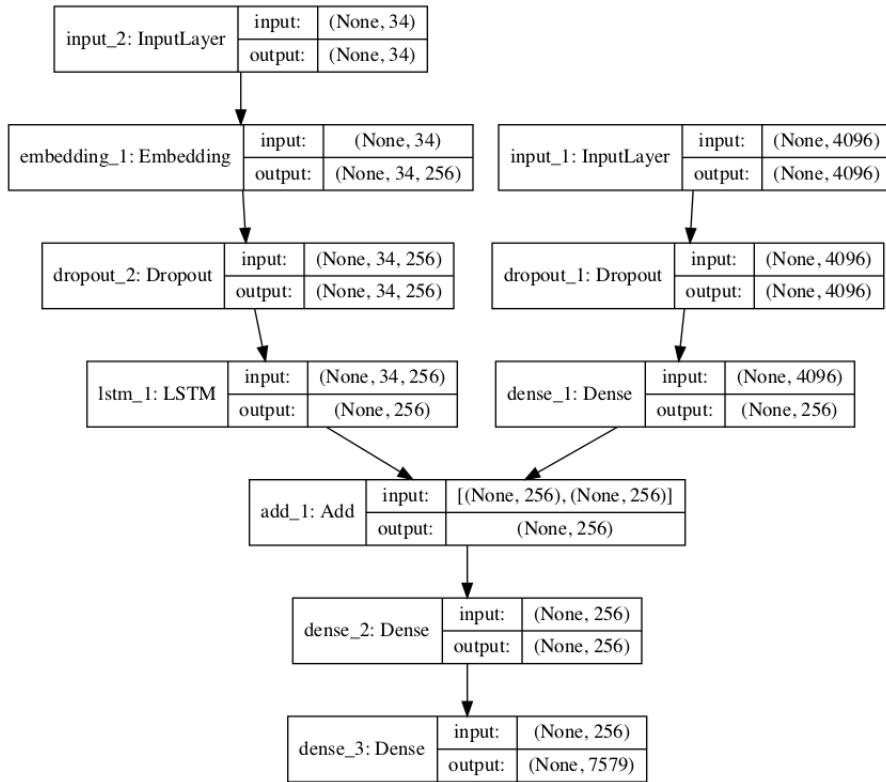


Figure 26.1: Plot of the defined caption generation model.

26.5.3 Fitting the Model

Now that we know how to define the model, we can fit it on the training dataset. The model learns fast and quickly overfits the training dataset. For this reason, we will monitor the skill of the trained model on the holdout development dataset. When the skill of the model on the development dataset improves at the end of an epoch, we will save the whole model to file.

At the end of the run, we can then use the saved model with the best skill on the training dataset as our final model. We can do this by defining a `ModelCheckpoint` in Keras and specifying it to monitor the minimum loss on the validation dataset and save the model to a file that has both the training and validation loss in the filename.

```
# define checkpoint callback
checkpoint = ModelCheckpoint('model.h5', monitor='val_loss', verbose=1,
                             save_best_only=True, mode='min')
```

Listing 26.21: Example of checkpoint configuration.

We can then specify the checkpoint in the call to `fit()` via the `callbacks` argument. We must also specify the development dataset in `fit()` via the `validation_data` argument. We will only fit the model for 20 epochs, but given the amount of training data, each epoch may take 30 minutes on modern hardware.

```
# fit model
model.fit([X1train, X2train], ytrain, epochs=20, verbose=2, callbacks=[checkpoint],
           validation_data=(X1test, X2test), ytest))
```

Listing 26.22: Example of fitting the caption generation model.

26.5.4 Complete Example

The complete example for fitting the model on the training data is listed below. Note, running this example may require a machine with 8 or more Gigabytes of RAM. See the appendix for using AWS, if needed.

```
from numpy import array
from pickle import load
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding
from keras.layers import Dropout
from keras.layers.merge import add
from keras.callbacks import ModelCheckpoint

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load a pre-defined list of photo identifiers
def load_set(filename):
    doc = load_doc(filename)
    dataset = list()
    # process line by line
    for line in doc.split('\n'):
        # skip empty lines
        if len(line) < 1:
            continue
        # get the image identifier
        identifier = line.split('.')[0]
        dataset.append(identifier)
    return set(dataset)

# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
    # load document
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        # split id from description
        image_id, image_desc = tokens[0], tokens[1:]
        # skip images not in the set
        if image_id in dataset:
            descriptions[image_id] = image_desc
    return descriptions
```

```
if image_id in dataset:
    # create list
    if image_id not in descriptions:
        descriptions[image_id] = list()
    # wrap description in tokens
    desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
    # store
    descriptions[image_id].append(desc)
return descriptions

# load photo features
def load_photo_features(filename, dataset):
    # load all features
    all_features = load(open(filename, 'rb'))
    # filter features
    features = {k: all_features[k] for k in dataset}
    return features

# convert a dictionary of clean descriptions to a list of descriptions
def to_lines(descriptions):
    all_desc = list()
    for key in descriptions.keys():
        [all_desc.append(d) for d in descriptions[key]]
    return all_desc

# fit a tokenizer given caption descriptions
def create_tokenizer(descriptions):
    lines = to_lines(descriptions)
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# calculate the length of the description with the most words
def max_length(descriptions):
    lines = to_lines(descriptions)
    return max(len(d.split()) for d in lines)

# create sequences of images, input sequences and output words for an image
def create_sequences(tokenizer, max_length, descriptions, photos):
    X1, X2, y = list(), list(), list()
    # walk through each image identifier
    for key, desc_list in descriptions.items():
        # walk through each description for the image
        for desc in desc_list:
            # encode the sequence
            seq = tokenizer.texts_to_sequences([desc])[0]
            # split one sequence into multiple X,y pairs
            for i in range(1, len(seq)):
                # split into input and output pair
                in_seq, out_seq = seq[:i], seq[i]
                # pad input sequence
                in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
                # encode output sequence
                out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
                # store
                X1.append(photos[key][0])
```

```
X2.append(in_seq)
y.append(out_seq)
return array(X1), array(X2), array(y)

# define the captioning model
def define_model(vocab_size, max_length):
    # feature extractor model
    inputs1 = Input(shape=(4096,))
    fe1 = Dropout(0.5)(inputs1)
    fe2 = Dense(256, activation='relu')(fe1)
    # sequence model
    inputs2 = Input(shape=(max_length,))
    se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
    se2 = Dropout(0.5)(se1)
    se3 = LSTM(256)(se2)
    # decoder model
    decoder1 = add([fe2, se3])
    decoder2 = Dense(256, activation='relu')(decoder1)
    outputs = Dense(vocab_size, activation='softmax')(decoder2)
    # tie it together [image, seq] [word]
    model = Model(inputs=[inputs1, inputs2], outputs=outputs)
    # compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam')
    # summarize model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model

# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# photo features
train_features = load_photo_features('features.pkl', train)
print('Photos: train=%d' % len(train_features))
# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# determine the maximum sequence length
max_length = max_length(train_descriptions)
print('Description Length: %d' % max_length)
# prepare sequences
X1train, X2train, ytrain = create_sequences(tokenizer, max_length, train_descriptions,
                                             train_features)

# load test set
filename = 'Flickr8k_text/Flickr_8k.devImages.txt'
test = load_set(filename)
print('Dataset: %d' % len(test))
# descriptions
test_descriptions = load_clean_descriptions('descriptions.txt', test)
print('Descriptions: test=%d' % len(test_descriptions))
```

```

# photo features
test_features = load_photo_features('features.pkl', test)
print('Photos: test=%d' % len(test_features))
# prepare sequences
X1test, X2test, ytest = create_sequences(tokenizer, max_length, test_descriptions,
                                         test_features)

# define the model
model = define_model(vocab_size, max_length)
# define checkpoint callback
checkpoint = ModelCheckpoint('model.h5', monitor='val_loss', verbose=1,
                             save_best_only=True, mode='min')
# fit model
model.fit([X1train, X2train], ytrain, epochs=20, verbose=2, callbacks=[checkpoint],
           validation_data=([X1test, X2test], ytest))

```

Listing 26.23: Complete example of training the caption generation model.

Running the example first prints a summary of the loaded training and development datasets.

```

Dataset: 6,000
Descriptions: train=6,000
Photos: train=6,000
Vocabulary Size: 7,579
Description Length: 34
Dataset: 1,000
Descriptions: test=1,000
Photos: test=1,000
Train on 306,404 samples, validate on 50,903 samples

```

Listing 26.24: Sample output from fitting the caption generation model

After the summary of the model, we can get an idea of the total number of training and validation (development) input-output pairs. The model then runs, saving the best model to .h5 files along the way. Note, that even on a modern CPU, each epoch may take 20 minutes. You may want to consider running the example on a GPU, such as on AWS. See the appendix for details on how to set this up. When I ran the example, the best model was saved at the end of epoch 2 with a loss of 3.245 on the training dataset and a loss of 3.612 on the development dataset.

26.6 Evaluate Model

Once the model is fit, we can evaluate the skill of its predictions on the holdout test dataset. We will evaluate a model by generating descriptions for all photos in the test dataset and evaluating those predictions with a standard cost function. First, we need to be able to generate a description for a photo using a trained model.

This involves passing in the start description token `startseq`, generating one word, then calling the model recursively with generated words as input until the end of sequence token is reached `endseq` or the maximum description length is reached. The function below named `generate_desc()` implements this behavior and generates a textual description given a trained model, and a given prepared photo as input. It calls the function `word_for_id()` in order to map an integer prediction back to a word.

```

# map an integer to a word
def word_for_id(integer, tokenizer):
    for word, index in tokenizer.word_index.items():
        if index == integer:
            return word
    return None

# generate a description for an image
def generate_desc(model, tokenizer, photo, max_length):
    # seed the generation process
    in_text = 'startseq'
    # iterate over the whole length of the sequence
    for i in range(max_length):
        # integer encode input sequence
        sequence = tokenizer.texts_to_sequences([in_text])[0]
        # pad input
        sequence = pad_sequences([sequence], maxlen=max_length)
        # predict next word
        yhat = model.predict([photo, sequence], verbose=0)
        # convert probability to integer
        yhat = argmax(yhat)
        # map integer to word
        word = word_for_id(yhat, tokenizer)
        # stop if we cannot map the word
        if word is None:
            break
        # append as input for generating the next word
        in_text += ' ' + word
        # stop if we predict the end of the sequence
        if word == 'endseq':
            break
    return in_text

```

Listing 26.25: Functions for generating a description for a photo.

When generating and comparing photo descriptions, we will need to strip off the special start and end of sequence words. The function below named `cleanup_summary()` will perform this operation.

```

# remove start/end sequence tokens from a summary
def cleanup_summary(summary):
    # remove start of sequence token
    index = summary.find('startseq ')
    if index > -1:
        summary = summary[len('startseq '):]
    # remove end of sequence token
    index = summary.find(' endseq')
    if index > -1:
        summary = summary[:index]
    return summary

```

Listing 26.26: Functions to remove start and end of sequence words.

We will generate predictions for all photos in the test dataset. The function below named `evaluate_model()` will evaluate a trained model against a given dataset of photo descriptions and photo features. The actual and predicted descriptions are collected and evaluated collectively

using the corpus BLEU score that summarizes how close the generated text is to the expected text.

```
# evaluate the skill of the model
def evaluate_model(model, descriptions, photos, tokenizer, max_length):
    actual, predicted = list(), list()
    # step over the whole set
    for key, desc_list in descriptions.items():
        # generate description
        yhat = generate_desc(model, tokenizer, photos[key], max_length)
        # clean up prediction
        yhat = cleanup_summary(yhat)
        # store actual and predicted
        references = [cleanup_summary(d).split() for d in desc_list]
        actual.append(references)
        predicted.append(yhat.split())
    # calculate BLEU score
    print('BLEU-1: %f' % corpus_bleu(actual, predicted, weights=(1.0, 0, 0, 0)))
    print('BLEU-2: %f' % corpus_bleu(actual, predicted, weights=(0.5, 0.5, 0, 0)))
    print('BLEU-3: %f' % corpus_bleu(actual, predicted, weights=(0.3, 0.3, 0.3, 0)))
    print('BLEU-4: %f' % corpus_bleu(actual, predicted, weights=(0.25, 0.25, 0.25, 0.25)))
```

Listing 26.27: Functions for evaluating a caption generation model.

BLEU scores are used in text translation for evaluating translated text against one or more reference translations. Here, we compare each generated description against all of the reference descriptions for the photograph. We then calculate BLEU scores for 1, 2, 3 and 4 cumulative n-grams. The NLTK Python library implements the BLEU score calculation in the `corpus_bleu()` function. A higher score close to 1.0 is better, a score closer to zero is worse. Note that the BLEU score and NLTK API were introduced in Chapter 24.

We can put all of this together with the functions from the previous section for loading the data. We first need to load the training dataset in order to prepare a `Tokenizer` so that we can encode generated words as input sequences for the model. It is critical that we encode the generated words using exactly the same encoding scheme as was used when training the model. We then use these functions for loading the test dataset. The complete example is listed below.

```
from numpy import argmax
from pickle import load
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model
from nltk.translate.bleu_score import corpus_bleu

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load a pre-defined list of photo identifiers
def load_set(filename):
```

```
doc = load_doc(filename)
dataset = list()
# process line by line
for line in doc.split('\n'):
    # skip empty lines
    if len(line) < 1:
        continue
    # get the image identifier
    identifier = line.split('.')[0]
    dataset.append(identifier)
return set(dataset)

# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
    # load document
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        # split id from description
        image_id, image_desc = tokens[0], tokens[1:]
        # skip images not in the set
        if image_id in dataset:
            # create list
            if image_id not in descriptions:
                descriptions[image_id] = list()
            # wrap description in tokens
            desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
            # store
            descriptions[image_id].append(desc)
    return descriptions

# load photo features
def load_photo_features(filename, dataset):
    # load all features
    all_features = load(open(filename, 'rb'))
    # filter features
    features = {k: all_features[k] for k in dataset}
    return features

# convert a dictionary of clean descriptions to a list of descriptions
def to_lines(descriptions):
    all_desc = list()
    for key in descriptions.keys():
        [all_desc.append(d) for d in descriptions[key]]
    return all_desc

# fit a tokenizer given caption descriptions
def create_tokenizer(descriptions):
    lines = to_lines(descriptions)
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# calculate the length of the description with the most words
```

```
def max_length(descriptions):
    lines = to_lines(descriptions)
    return max(len(d.split()) for d in lines)

# map an integer to a word
def word_for_id(integer, tokenizer):
    for word, index in tokenizer.word_index.items():
        if index == integer:
            return word
    return None

# generate a description for an image
def generate_desc(model, tokenizer, photo, max_length):
    # seed the generation process
    in_text = 'startseq'
    # iterate over the whole length of the sequence
    for _ in range(max_length):
        # integer encode input sequence
        sequence = tokenizer.texts_to_sequences([in_text])[0]
        # pad input
        sequence = pad_sequences([sequence], maxlen=max_length)
        # predict next word
        yhat = model.predict([photo, sequence], verbose=0)
        # convert probability to integer
        yhat = argmax(yhat)
        # map integer to word
        word = word_for_id(yhat, tokenizer)
        # stop if we cannot map the word
        if word is None:
            break
        # append as input for generating the next word
        in_text += ' ' + word
        # stop if we predict the end of the sequence
        if word == 'endseq':
            break
    return in_text

# remove start/end sequence tokens from a summary
def cleanup_summary(summary):
    # remove start of sequence token
    index = summary.find('startseq ')
    if index > -1:
        summary = summary[len('startseq '):]
    # remove end of sequence token
    index = summary.find(' endseq')
    if index > -1:
        summary = summary[:index]
    return summary

# evaluate the skill of the model
def evaluate_model(model, descriptions, photos, tokenizer, max_length):
    actual, predicted = list(), list()
    # step over the whole set
    for key, desc_list in descriptions.items():
        # generate description
        yhat = generate_desc(model, tokenizer, photos[key], max_length)
```

```

# clean up prediction
yhat = cleanup_summary(yhat)
# store actual and predicted
references = [cleanup_summary(d).split() for d in desc_list]
actual.append(references)
predicted.append(yhat.split())
# calculate BLEU score
print('BLEU-1: %f' % corpus_bleu(actual, predicted, weights=(1.0, 0, 0, 0)))
print('BLEU-2: %f' % corpus_bleu(actual, predicted, weights=(0.5, 0.5, 0, 0)))
print('BLEU-3: %f' % corpus_bleu(actual, predicted, weights=(0.3, 0.3, 0.3, 0)))
print('BLEU-4: %f' % corpus_bleu(actual, predicted, weights=(0.25, 0.25, 0.25, 0.25)))

# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# determine the maximum sequence length
max_length = max_length(train_descriptions)
print('Description Length: %d' % max_length)

# load test set
filename = 'Flickr8k_text/Flickr_8k.testImages.txt'
test = load_set(filename)
print('Dataset: %d' % len(test))
# descriptions
test_descriptions = load_clean_descriptions('descriptions.txt', test)
print('Descriptions: test=%d' % len(test_descriptions))
# photo features
test_features = load_photo_features('features.pkl', test)
print('Photos: test=%d' % len(test_features))

# load the model
filename = 'model.h5'
model = load_model(filename)
# evaluate model
evaluate_model(model, test_descriptions, test_features, tokenizer, max_length)

```

Listing 26.28: Complete example of evaluating the caption generation model.

Running the example prints the BLEU scores. We can see that the scores fit within the expected range of a skillful model on the problem. The chosen model configuration is by no means optimized.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```

BLEU-1: 0.438805
BLEU-2: 0.230646

```

```
BLEU-3: 0.150245
BLEU-4: 0.062847
```

Listing 26.29: Sample output from evaluating the caption generation model

26.7 Generate New Captions

Now that we know how to develop and evaluate a caption generation model, how can we use it? Almost everything we need to generate captions for entirely new photographs is in the model file. We also need the `Tokenizer` for encoding generated words for the model while generating a sequence, and the maximum length of input sequences, used when we defined the model (e.g. 34).

We can hard code the maximum sequence length. With the encoding of text, we can create the tokenizer and save it to a file so that we can load it quickly whenever we need it without needing the entire Flickr8K dataset. An alternative would be to use our own vocabulary file and mapping to integers function during training. We can create the `Tokenizer` as before and save it as a pickle file `tokenizer.pkl`. The complete example is listed below.

```
from keras.preprocessing.text import Tokenizer
from pickle import dump

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# load a pre-defined list of photo identifiers
def load_set(filename):
    doc = load_doc(filename)
    dataset = list()
    # process line by line
    for line in doc.split('\n'):
        # skip empty lines
        if len(line) < 1:
            continue
        # get the image identifier
        identifier = line.split('.')[0]
        dataset.append(identifier)
    return set(dataset)

# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
    # load document
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
```

```

# split id from description
image_id, image_desc = tokens[0], tokens[1:]
# skip images not in the set
if image_id in dataset:
    # create list
    if image_id not in descriptions:
        descriptions[image_id] = list()
    # wrap description in tokens
    desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
    # store
    descriptions[image_id].append(desc)
return descriptions

# convert a dictionary of clean descriptions to a list of descriptions
def to_lines(descriptions):
    all_desc = list()
    for key in descriptions.keys():
        [all_desc.append(d) for d in descriptions[key]]
    return all_desc

# fit a tokenizer given caption descriptions
def create_tokenizer(descriptions):
    lines = to_lines(descriptions)
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# load training dataset
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
# save the tokenizer
dump(tokenizer, open('tokenizer.pkl', 'wb'))

```

Listing 26.30: Complete example of preparing and saving the `Tokenizer`.

We can now load the tokenizer whenever we need it without having to load the entire training dataset of annotations. Now, let's generate a description for a new photograph. Below is a new photograph that I chose randomly on Flickr (available under a permissive license)¹.

¹<https://www.flickr.com/photos/bambe1964/7837618434/>



Figure 26.2: Photo of a dog at the beach. Photo by bambe1964, some rights reserved.

We will generate a description for it using our model. Download the photograph and save it to your local directory with the filename `example.jpg`. First, we must load the `Tokenizer` from `tokenizer.pkl` and define the maximum length of the sequence to generate, needed for padding inputs.

```
# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))
# pre-define the max sequence length (from training)
max_length = 34
```

Listing 26.31: Example of loading the saved `Tokenizer`

Then we must load the model, as before.

```
# load the model
model = load_model('model.h5')
```

Listing 26.32: Example of loading the saved model

Next, we must load the photo we wish to describe and extract the features. We could do this by re-defining the model and adding the VGG-16 model to it, or we can use the VGG model to predict the features and use them as inputs to our existing model. We will do the latter and use a modified version of the `extract_features()` function used during data preparation, but adapted to work on a single photo.

```
# extract features from each photo in the directory
def extract_features(filename):
    # load the model
    model = VGG16()
    # re-structure the model
    model.layers.pop()
    model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
    # load the photo
    image = load_img(filename, target_size=(224, 224))
    # convert the image pixels to a NumPy array
    image = img_to_array(image)
    # reshape data for the model
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
```

```

# prepare the image for the VGG model
image = preprocess_input(image)
# get features
feature = model.predict(image, verbose=0)
return feature

# load and prepare the photograph
photo = extract_features('example.jpg')

```

Listing 26.33: Example of extracting features for the provided photo.

We can then generate a description using the `generate_desc()` function defined when evaluating the model. The complete example for generating a description for an entirely new standalone photograph is listed below.

```

from pickle import load
from numpy import argmax
from keras.preprocessing.sequence import pad_sequences
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.models import Model
from keras.models import load_model

# extract features from each photo in the directory
def extract_features(filename):
    # load the model
    model = VGG16()
    # re-structure the model
    model.layers.pop()
    model = Model(inputs=model.inputs, outputs=model.layers[-1].output)
    # load the photo
    image = load_img(filename, target_size=(224, 224))
    # convert the image pixels to a numpy array
    image = img_to_array(image)
    # reshape data for the model
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
    # prepare the image for the VGG model
    image = preprocess_input(image)
    # get features
    feature = model.predict(image, verbose=0)
    return feature

# map an integer to a word
def word_for_id(integer, tokenizer):
    for word, index in tokenizer.word_index.items():
        if index == integer:
            return word
    return None

# remove start/end sequence tokens from a summary
def cleanup_summary(summary):
    # remove start of sequence token
    index = summary.find('startseq ')
    if index > -1:

```

```

summary = summary[len('startseq'):]
# remove end of sequence token
index = summary.find(' endseq')
if index > -1:
    summary = summary[:index]
return summary

# generate a description for an image
def generate_desc(model, tokenizer, photo, max_length):
    # seed the generation process
    in_text = 'startseq'
    # iterate over the whole length of the sequence
    for _ in range(max_length):
        # integer encode input sequence
        sequence = tokenizer.texts_to_sequences([in_text])[0]
        # pad input
        sequence = pad_sequences([sequence], maxlen=max_length)
        # predict next word
        yhat = model.predict([photo, sequence], verbose=0)
        # convert probability to integer
        yhat = argmax(yhat)
        # map integer to word
        word = word_for_id(yhat, tokenizer)
        # stop if we cannot map the word
        if word is None:
            break
        # append as input for generating the next word
        in_text += ' ' + word
        # stop if we predict the end of the sequence
        if word == 'endseq':
            break
    return in_text

# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))
# pre-define the max sequence length (from training)
max_length = 34
# load the model
model = load_model('model.h5')
# load and prepare the photograph
photo = extract_features('example.jpg')
# generate description
description = generate_desc(model, tokenizer, photo, max_length)
description = cleanup_summary(description)
print(description)

```

Listing 26.34: Complete example of generating a description for a new photo.

In this case, the description generated was as follows:

```
dog is running across the beach
```

Listing 26.35: Sample output from generating a caption for the new photograph

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

26.8 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Alternate Pre-Trained Image Models.** A small 16-layer VGG model was used for feature extraction. Consider exploring larger models that offer better performance on the ImageNet dataset, such as Inception.
- **Smaller Vocabulary.** A larger vocabulary of nearly eight thousand words was used in the development of the model. Many of the words supported may be misspellings or only used once in the entire dataset. Refine the vocabulary and reduce the size, perhaps by half.
- **Pre-trained Word Vectors.** The model learned the word vectors as part of fitting the model. Better performance may be achieved by using word vectors either pre-trained on the training dataset or trained on a much larger corpus of text, such as news articles or Wikipedia.
- **Train Word2Vec Vectors.** Pre-train word vectors using Word2Vec on the description data and explore models that allow and don't allow fine tuning of the vectors during training, then compare skill.
- **Tune Model.** The configuration of the model was not tuned on the problem. Explore alternate configurations and see if you can achieve better performance.
- **Inject Architecture.** Explore the inject architecture for caption generation and compare performance to the merge architecture used in this tutorial.
- **Alternate Framings.** Explore alternate framings of the problems such as generating the entire sequence from the photo alone.
- **Pre-Train Language Model.** Pre-train a language model for generating description text, then use it in the caption generation model and evaluate the impact on model training time and skill.
- **Truncate Descriptions.** Only train the model on description at or below a specific number of words and explore truncating long descriptions to a preferred length. Evaluate the impact on training time and model skill.
- **Alternate Measure.** Explore alternate performance measures beside BLEU such as ROGUE. Compare scores for the same descriptions to develop an intuition for how the measures differ in practice.

If you explore any of these extensions, I'd love to know.

26.9 Further Reading

This section provides more resources on the topic if you are looking go deeper.

26.9.1 Caption Generation Papers

- *Show and Tell: A Neural Image Caption Generator*, 2015.
<https://arxiv.org/abs/1411.4555>
- *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*, 2015.
<https://arxiv.org/abs/1502.03044>
- *Where to put the Image in an Image Caption Generator*, 2017.
<https://arxiv.org/abs/1703.09137>
- *What is the Role of Recurrent Neural Networks (RNNs) in an Image Caption Generator?*, 2017.
<https://arxiv.org/abs/1708.02043>
- *Automatic Description Generation from Images: A Survey of Models, Datasets, and Evaluation Measures*, 2016.
<https://arxiv.org/abs/1601.03896>

26.9.2 Flickr8K Dataset

- *Framing image description as a ranking task: data, models and evaluation metrics* (Homepage).
http://nlp.cs.illinois.edu/HockenmaierGroup/Framing_Image_Description/KCCA.html
- *Framing Image Description as a Ranking Task: Data, Models and Evaluation Metrics*, 2013.
<https://www.jair.org/media/3994/live-3994-7274-jair.pdf>
- Dataset Request Form.
<https://illinois.edu/fb/sec/1713398>
- Old Flickr8K Homepage.
<http://nlp.cs.illinois.edu/HockenmaierGroup/8k-pictures.html>

26.9.3 API

- Keras Model API.
<https://keras.io/models/model/>
- Keras pad_sequences() API.
https://keras.io/preprocessing/sequence/#pad_sequences
- Keras Tokenizer API.
<https://keras.io/preprocessing/text/#tokenizer>
- Keras VGG16 API.
<https://keras.io/applications/#vgg16>

- Gensim Word2Vec API.
<https://radimrehurek.com/gensim/models/word2vec.html>
- `nltk.translate` package API Documentation.
<http://www.nltk.org/api/nltk.translate.html>

26.10 Summary

In this tutorial, you discovered how to develop a photo captioning deep learning model from scratch. Specifically, you learned:

- How to prepare photo and text data ready for training a deep learning model.
- How to design and train a deep learning caption generation model.
- How to evaluate a train caption generation model and use it to caption entirely new photographs.

26.10.1 Next

This is the last chapter in the image captioning part. In the next part you will discover how to develop neural machine translation models.

Part IX

Machine Translation

Chapter 27

Neural Machine Translation

One of the earliest goals for computers was the automatic translation of text from one language to another. Automatic or machine translation is perhaps one of the most challenging artificial intelligence tasks given the fluidity of human language. Classically, rule-based systems were used for this task, which were replaced in the 1990s with statistical methods. More recently, deep neural network models achieve state-of-the-art results in a field that is aptly named neural machine translation. In this chapter, you will discover the challenge of machine translation and the effectiveness of neural machine translation models. After reading this chapter, you will know:

- Machine translation is challenging given the inherent ambiguity and flexibility of human language.
- Statistical machine translation replaces classical rule-based systems with models that learn to translate from examples.
- Neural machine translation models fit a single model rather than a pipeline of fine-tuned models and currently achieve state-of-the-art results.

Let's get started.

27.1 What is Machine Translation?

Machine translation is the task of automatically converting source text in one language to text in another language.

In a machine translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language.

— Page 98, *Deep Learning*, 2016.

Given a sequence of text in a source language, there is no one single best translation of that text to another language. This is because of the natural ambiguity and flexibility of human language. This makes the challenge of automatic machine translation difficult, perhaps one of the most difficult in artificial intelligence:

The fact is that accurate translation requires background knowledge in order to resolve ambiguity and establish the content of the sentence.

— Page 21, *Artificial Intelligence, A Modern Approach*, 3rd Edition, 2009.

Classical machine translation methods often involve rules for converting text in the source language to the target language. The rules are often developed by linguists and may operate at the lexical, syntactic, or semantic level. This focus on rules gives the name to this area of study: Rule-based Machine Translation, or RBMT.

RBMT is characterized with the explicit use and manual creation of linguistically informed rules and representations.

— Page 133, *Handbook of Natural Language Processing and Machine Translation*, 2011.

The key limitations of the classical machine translation approaches are both the expertise required to develop the rules, and the vast number of rules and exceptions required.

27.2 What is Statistical Machine Translation?

Statistical machine translation, or SMT for short, is the use of statistical models that learn to translate text from a source language to a target language given a large corpus of examples. This task of using a statistical model can be stated formally as follows:

Given a sentence T in the target language, we seek the sentence S from which the translator produced T. We know that our chance of error is minimized by choosing that sentence S that is most probable given T. Thus, we wish to choose S so as to maximize $\Pr(S|T)$.

— *A Statistical Approach to Machine Translation*, 1990.

This formal specification makes the maximizing of the probability of the output sequence given the input sequence of text explicit. It also makes the notion of there being a suite of candidate translations explicit and the need for a search process or decoder to select the one most likely translation from the model's output probability distribution.

Given a text in the source language, what is the most probable translation in the target language? [...] how should one construct a statistical model that assigns high probabilities to “good” translations and low probabilities to “bad” translations?

— Page xiii, *Syntax-based Statistical Machine Translation*, 2017.

The approach is data-driven, requiring only a corpus of examples with both source and target language text. This means linguists are not longer required to specify the rules of translation.

This approach does not need a complex ontology of interlingua concepts, nor does it need handcrafted grammars of the source and target languages, nor a hand-labeled treebank. All it needs is data-sample translations from which a translation model can be learned.

— Page 909, *Artificial Intelligence, A Modern Approach*, 3rd Edition, 2009.

Quickly, the statistical approach to machine translation outperformed the classical rule-based methods to become the de-facto standard set of techniques.

Since the inception of the field at the end of the 1980s, the most popular models for statistical machine translation [...] have been sequence-based. In these models, the basic units of translation are words or sequences of words [...] These kinds of models are simple and effective, and they work well for many language pairs

— *Syntax-based Statistical Machine Translation*, 2017.

The most widely used techniques were phrase-based and focus on translating sub-sequences of the source text piecewise.

Statistical Machine Translation (SMT) has been the dominant translation paradigm for decades. Practical implementations of SMT are generally phrase-based systems (PBMT) which translate sequences of words or phrases where the lengths may differ

— *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, 2016.

Although effective, statistical machine translation methods suffered from a narrow focus on the phrases being translated, losing the broader nature of the target text. The hard focus on data-driven approaches also meant that methods may have ignored important syntax distinctions known by linguists. Finally, the statistical approaches required careful tuning of each module in the translation pipeline.

27.3 What is Neural Machine Translation?

Neural machine translation, or NMT for short, is the use of neural network models to learn a statistical model for machine translation. The key benefit to the approach is that a single system can be trained directly on source and target text, no longer requiring the pipeline of specialized systems used in statistical machine learning.

Unlike the traditional phrase-based translation system which consists of many small sub-components that are tuned separately, neural machine translation attempts to build and train a single, large neural network that reads a sentence and outputs a correct translation.

— *Neural Machine Translation by Jointly Learning to Align and Translate*, 2014.

As such, neural machine translation systems are said to be end-to-end systems as only one model is required for the translation.

The strength of NMT lies in its ability to learn directly, in an end-to-end fashion, the mapping from input text to associated output text.

— *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, 2016.

27.3.1 Encoder-Decoder Model

Multilayer Perceptron neural network models can be used for machine translation, although the models are limited by a fixed-length input sequence where the output must be the same length. These early models have been greatly improved upon recently through the use of recurrent neural networks organized into an encoder-decoder architecture that allow for variable length input and output sequences.

An encoder neural network reads and encodes a source sentence into a fixed-length vector. A decoder then outputs a translation from the encoded vector. The whole encoder-decoder system, which consists of the encoder and the decoder for a language pair, is jointly trained to maximize the probability of a correct translation given a source sentence.

— *Neural Machine Translation by Jointly Learning to Align and Translate*, 2014.

Key to the encoder-decoder architecture is the ability of the model to encode the source text into an internal fixed-length representation called the context vector. Interestingly, once encoded, different decoding systems could be used, in principle, to translate the context into different languages.

... one model first reads the input sequence and emits a data structure that summarizes the input sequence. We call this summary the “context” C. [...] A second mode, usually an RNN, then reads the context C and generates a sentence in the target language.

— Page 461, *Deep Learning*, 2016.

27.3.2 Encoder-Decoders with Attention

Although effective, the Encoder-Decoder architecture has problems with long sequences of text to be translated. The problem stems from the fixed-length internal representation that must be used to decode each word in the output sequence. The solution is the use of an attention mechanism that allows the model to learn where to place attention on the input sequence as each word of the output sequence is decoded.

Using a fixed-sized representation to capture all the semantic details of a very long sentence [...] is very difficult. [...] A more efficient approach, however, is to read the whole sentence or paragraph [...], then to produce the translated words one at a time, each time focusing on a different part of the input sentence to gather the semantic details required to produce the next output word.

— Page 462, *Deep Learning*, 2016.

The encoder-decoder recurrent neural network architecture with attention is currently the state-of-the-art on some benchmark problems for machine translation. And this architecture is used in the heart of the Google Neural Machine Translation system, or GNMT, used in their Google Translate service.

... current state-of-the-art machine translation systems are powered by models that employ attention.

— Page 209, *Neural Network Methods in Natural Language Processing*, 2017.

Although effective, the neural machine translation systems still suffer some issues, such as scaling to larger vocabularies of words and the slow speed of training the models. There are the current areas of focus for large production neural translation systems, such as the Google system.

Three inherent weaknesses of Neural Machine Translation [...]: its slower training and inference speed, ineffectiveness in dealing with rare words, and sometimes failure to translate all words in the source sentence.

— *Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, 2016.

27.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

27.4.1 Books

- *Neural Network Methods in Natural Language Processing*, 2017.
<http://amzn.to/2wPrW37>
- *Syntax-based Statistical Machine Translation*, 2017.
<http://amzn.to/2xCrl3p>
- *Deep Learning*, 2016.
<http://amzn.to/2xBEsBJ>
- *Statistical Machine Translation*, 2010.
<http://amzn.to/2xCe1vP>
- *Handbook of Natural Language Processing and Machine Translation*, 2011.
<http://amzn.to/2jYUFFy>
- *Artificial Intelligence, A Modern Approach*, 3rd Edition, 2009.
<http://amzn.to/2wUZesr>

27.4.2 Papers

- *A Statistical Approach to Machine Translation*, 1990.
<https://dl.acm.org/citation.cfm?id=92860>
- *Review Article: Example-based Machine Translation*, 1999.
<https://link.springer.com/article/10.1023/A:1008109312730>

- *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.
<https://arxiv.org/abs/1406.1078>
- *Neural Machine Translation by Jointly Learning to Align and Translate*, 2014.
<https://arxiv.org/abs/1409.0473>
- *Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, 2016.
<https://arxiv.org/abs/1609.08144>
- *Sequence to sequence learning with neural networks*, 2014.
<https://arxiv.org/abs/1409.3215>
- *Recurrent Continuous Translation Models*, 2013.
<http://www.aclweb.org/anthology/D13-1176>
- *Continuous space translation models for phrase-based statistical machine translation*, 2013.
<https://aclweb.org/anthology/C/C12/C12-2104.pdf>

27.4.3 Additional

- Machine Translation Archive.
<http://www.mt-archive.info/>
- Neural machine translation on Wikipedia.
https://en.wikipedia.org/wiki/Neural_machine_translation
- *Neural Machine Translation, Statistical Machine Translation*, 2017.
<https://arxiv.org/abs/1709.07809>

27.5 Summary

In this chapter, you discovered the challenge of machine translation and the effectiveness of neural machine translation models. Specifically, you learned:

- Machine translation is challenging given the inherent ambiguity and flexibility of human language.
- Statistical machine translation replaces classical rule-based systems with models that learn to translate from examples.
- Neural machine translation models fit a single model rather than a pipeline of fine tuned models and currently achieve state-of-the-art results.

27.5.1 Next

In the next chapter, you will discover how you can design neural machine translation models.

Chapter 28

What are Encoder-Decoder Models for Neural Machine Translation

The encoder-decoder architecture for recurrent neural networks is the standard neural machine translation method that rivals and in some cases outperforms classical statistical machine translation methods. This architecture is very new, having only been pioneered in 2014, although, has been adopted as the core technology inside Google's translate service. In this chapter, you will discover the two seminal examples of the encoder-decoder model for neural machine translation. After reading this chapter, you will know:

- The encoder-decoder recurrent neural network architecture is the core technology inside Google's translate service.
- The so-called *Sutskever model* for direct end-to-end machine translation.
- The so-called *Cho model* that extends the architecture with GRU units and an attention mechanism.

Let's get started.

28.1 Encoder-Decoder Architecture for NMT

The Encoder-Decoder architecture with recurrent neural networks has become an effective and standard approach for both neural machine translation (NMT) and sequence-to-sequence (seq2seq) prediction in general. The key benefits of the approach are the ability to train a single end-to-end model directly on source and target sentences and the ability to handle variable length input and output sequences of text. As evidence of the success of the method, the architecture is the core of the Google translation service.

Our model follows the common sequence-to-sequence learning framework with attention. It has three components: an encoder network, a decoder network, and an attention network.

— *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, 2016

In this chapter, we will take a closer look at two different research projects that developed the same Encoder-Decoder architecture at the same time in 2014 and achieved results that put the spotlight on the approach. They are:

- Sutskever NMT Model
- Cho NMT Model

28.2 Sutskever NMT Model

In this section, we will look at the neural machine translation model developed by Ilya Sutskever, et al. as described in their 2014 paper *Sequence to Sequence Learning with Neural Networks*. We will refer to it as the *Sutskever NMT Model*, for lack of a better name. This is an important paper as it was one of the first to introduce the Encoder-Decoder model for machine translation and more generally sequence-to-sequence learning. It is an important model in the field of machine translation as it was one of the first neural machine translation systems to outperform a baseline statistical machine learning model on a large translation task.

28.2.1 Problem

The model was applied to English to French translation, specifically the WMT 2014 translation task. The translation task was processed one sentence at a time, and an end-of-sequence (`<EOS>`) token was added to the end of output sequences during training to signify the end of the translated sequence. This allowed the model to be capable of predicting variable length output sequences.

Note that we require that each sentence ends with a special end-of-sentence symbol `<EOS>`, which enables the model to define a distribution over sequences of all possible lengths.

— *Sequence to Sequence Learning with Neural Networks*, 2014.

The model was trained on a subset of the 12 Million sentences in the dataset, comprised of 348 Million French words and 304 Million English words. This set was chosen because it was pre-tokenized. The source vocabulary was reduced to the 160,000 most frequent source English words and 80,000 of the most frequent target French words. All out-of-vocabulary words were replaced with the UNK token.

28.2.2 Model

An Encoder-Decoder architecture was developed where an input sequence was read in entirety and encoded to a fixed-length internal representation. A decoder network then used this internal representation to output words until the end of sequence token was reached. LSTM networks were used for both the encoder and decoder.

The idea is to use one LSTM to read the input sequence, one timestep at a time, to obtain large fixed-dimensional vector representation, and then to use another LSTM to extract the output sequence from that vector

— *Sequence to Sequence Learning with Neural Networks*, 2014.

The final model was an ensemble of 5 deep learning models. A left-to-right beam search was used during the inference of the translations.

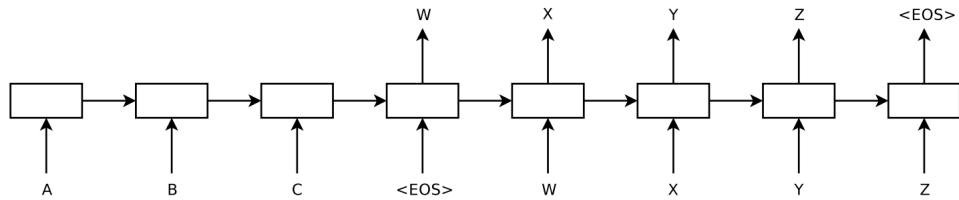


Figure 28.1: Depiction of Sutskever Encoder-Decoder Model for Text Translation. Taken from *Sequence to Sequence Learning with Neural Networks*.

28.2.3 Model Configuration

The following provides a summary of the model configuration taken from the paper:

- Input sequences were reversed.
- A 1000-dimensional word embedding layer was used to represent the input words.
- Softmax was used on the output layer.
- The input and output models had 4 layers with 1,000 units per layer.
- The model was fit for 7.5 epochs where some learning rate decay was performed.
- A batch-size of 128 sequences was used during training.
- Gradient clipping was used during training to mitigate the chance of gradient explosions.
- Batches were comprised of sentences with roughly the same length to speed-up computation.

The model was fit on an 8-GPU machine where each layer was run on a different GPU. Training took 10 days.

The resulting implementation achieved a speed of 6,300 (both English and French) words per second with a minibatch size of 128. Training took about ten days with this implementation.

— *Sequence to Sequence Learning with Neural Networks*, 2014.

28.2.4 Result

The system achieved a BLEU score of 34.81, which is a good score compared to the baseline score developed with a statistical machine translation system of 33.30. Importantly, this is the first example of a neural machine translation system that outperformed a phrase-based statistical machine translation baseline on a large scale problem.

... we obtained a BLEU score of 34.81 [...] This is by far the best result achieved by direct translation with large neural networks. For comparison, the BLEU score of an SMT baseline on this dataset is 33.30

— *Sequence to Sequence Learning with Neural Networks*, 2014.

The final model was used to re-score the list of best translations and improved the score to 36.5 which brings it close to the best result at the time of 37.0.

28.3 Cho NMT Model

In this section, we will look at the neural machine translation system described by Kyunghyun Cho, et al. in their 2014 paper titled *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. We will refer to it as the *Cho NMT Model* model for lack of a better name. Importantly, the Cho Model is used only to score candidate translations and is not used directly for translation like the Sutskever model above. Although extensions to the work to better diagnose and improve the model do use it directly and alone for translation.

28.3.1 Problem

As above, the problem is the English to French translation task from the WMT 2014 workshop. The source and target vocabulary were limited to the most frequent 15,000 French and English words which covers 93% of the dataset, and out of vocabulary words were replaced with UNK.

... called RNN Encoder-Decoder that consists of two recurrent neural networks (RNN). One RNN encodes a sequence of symbols into a fixed-length vector representation, and the other decodes the representation into another sequence of symbols.

— *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.

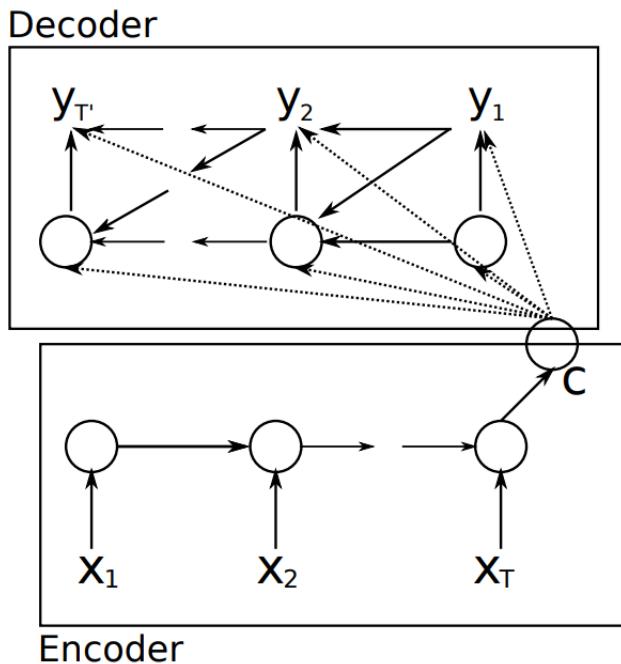


Figure 28.2: Depiction of the Encoder-Decoder architecture. Taken from *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*.

The implementation does not use LSTM units; instead, a simpler recurrent neural network unit is developed called the gated recurrent unit or GRU.

... we also propose a new type of hidden unit that has been motivated by the LSTM unit but is much simpler to compute and implement.

— *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.

28.3.2 Model Configuration

The following provides a summary of the model configuration taken from the paper:

- A 100-dimensional word embedding was used to represent the input words.
- The encoder and decoder were configured with 1 layer of 1000 GRU units.
- 500 Maxout units pooling 2 inputs were used after the decoder.
- A batch size of 64 sentences was used during training.

The model was trained for approximately 2 days.

28.3.3 Extensions

In the paper *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*, Cho, et al. investigate the limitations of their model. They discover that performance degrades quickly with the increase in the length of input sentences and with the number of words outside of the vocabulary.

Our analysis revealed that the performance of the neural machine translation suffers significantly from the length of sentences.

— *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*, 2014.

They provide a useful graph of the performance of the model as the length of the sentence is increased that captures the graceful loss in skill with increased difficulty.

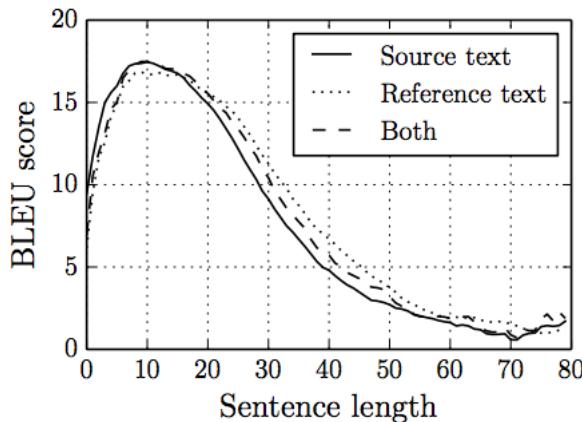


Figure 28.3: Loss in model skill with increased sentence length. Taken from *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*.

To address the problem of unknown words, they suggest dramatically increasing the vocabulary of known words during training. They address the problem of sentence length in a follow-up paper titled *Neural Machine Translation by Jointly Learning to Align and Translate* in which they propose the use of an attention mechanism. Instead of encoding the input sentence to a fixed length vector, a fuller representation of the encoded input is kept and the model learns to use to pay attention to different parts of the input for each word output by the decoder.

Each time the proposed model generates a word in a translation, it (soft-)searches for a set of positions in a source sentence where the most relevant information is concentrated. The model then predicts a target word based on the context vectors associated with these source positions and all the previous generated target words.

— *Neural Machine Translation by Jointly Learning to Align and Translate*, 2015.

A wealth of technical details are provided in the paper; for example:

- A similarly configured model is used, although with bidirectional layers.

- The data is prepared such that 30,000 of the most common words are kept in the vocabulary.
- The model is first trained with sentences with a length up to 20 words, then with sentences with a length up to 50 words.
- A batch size of 80 sentences is used and the model was fit for 4-6 epochs.
- A beam search was used during the inference to find the most likely sequence of words for each translation.

This time the model takes approximately 5 days to train. The code for this follow-up work is also made available. As with the Sutskever, the model achieved results within the reach of classical phrase-based statistical approaches.

Perhaps more importantly, the proposed approach achieved a translation performance comparable to the existing phrase-based statistical machine translation. It is a striking result, considering that the proposed architecture, or the whole family of neural machine translation, has only been proposed as recently as this year. We believe the architecture proposed here is a promising step toward better machine translation and a better understanding of natural languages in general.

— *Neural Machine Translation by Jointly Learning to Align and Translate*, 2015.

Kyunghyun Cho is also the author of a 2015 series of posts on the Nvidia developer blog on the topic of the encoder-decoder architecture for neural machine translation titled *Introduction to Neural Machine Translation with GPUs*. The series provides a good introduction to the topic and the model.

28.4 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- *Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, 2016.
<https://arxiv.org/abs/1609.08144>
- *Sequence to Sequence Learning with Neural Networks*, 2014.
<https://arxiv.org/abs/1409.3215>
- *Presentation for Sequence to Sequence Learning with Neural Networks*, 2016.
<https://www.youtube.com/watch?v=-uyXE7dY5H0>
- Ilya Sutskever Homepage.
<http://www.cs.toronto.edu/~ilya/>
- *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.
<https://arxiv.org/abs/1406.1078>

- *Neural Machine Translation by Jointly Learning to Align and Translate*, 2014.
<https://arxiv.org/abs/1409.0473>
- *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*, 2014.
<https://arxiv.org/abs/1409.1259>
- Kyunghyun Cho Homepage.
<http://www.kyunghyuncho.me/>
- *Introduction to Neural Machine Translation with GPUs*, 2015.
<https://goo.gl/GmWjvX>

28.5 Summary

In this chapter, you discovered two examples of the encoder-decoder model for neural machine translation. Specifically, you learned:

- The encoder-decoder recurrent neural network architecture is the core technology inside Google’s translate service.
- The so-called *Sutskever model* for direct end-to-end machine translation.
- The so-called *Cho model* that extends the architecture with GRU units and an attention mechanism.

28.5.1 Next

In the next chapter, you will discover how you can configure neural machine translation models.

Chapter 29

How to Configure Encoder-Decoder Models for Machine Translation

The encoder-decoder architecture for recurrent neural networks is achieving state-of-the-art results on standard machine translation benchmarks and is being used in the heart of industrial translation services. The model is simple, but given the large amount of data required to train it, tuning the myriad of design decisions in the model in order get top performance on your problem can be practically intractable. Thankfully, research scientists have used Google-scale hardware to do this work for us and provide a set of heuristics for how to configure the encoder-decoder model for neural machine translation and for sequence prediction generally.

In this chapter, you will discover the details of how to best configure an encoder-decoder recurrent neural network for neural machine translation and other natural language processing tasks. After reading this chapter, you will know:

- The Google study that investigated each model design decision in the encoder-decoder model to isolate their effects.
- The results and recommendations for design decisions like word embeddings, encoder and decoder depth, and attention mechanisms.
- A set of base model design decisions that can be used as a starting point on your own sequence-to-sequence projects.

Let's get started.

29.1 Encoder-Decoder Model for Neural Machine Translation

The Encoder-Decoder architecture for recurrent neural networks is displacing classical phrase-based statistical machine translation systems for state-of-the-art results. As evidence, by their 2016 paper *Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, Google now uses the approach in their core of their Google Translate service.

A problem with this architecture is that the models are large, in turn requiring very large datasets on which to train. This has the effect of model training taking days or weeks and

requiring computational resources that are generally very expensive. As such, little work has been done on the impact of different design choices on the model and their impact on model skill. This problem is addressed explicitly by Denny Britz, et al. in their 2017 paper *Massive Exploration of Neural Machine Translation Architectures*. In the paper, they design a baseline model for a standard English-to-German translation task and enumerate a suite of different model design choices and describe their impact on the skill of the model. They claim that the complete set of experiments consumed more than 250,000 GPU compute hours, which is impressive, to say the least.

We report empirical results and variance numbers for several hundred experimental runs, corresponding to over 250,000 GPU hours on the standard WMT English to German translation task. Our experiments lead to novel insights and practical advice for building and extending NMT architectures.

— *Massive Exploration of Neural Machine Translation Architectures*, 2017.

In this chapter, we will look at some of the findings from this paper that we can use to tune our own neural machine translation models, as well as sequence-to-sequence models in general.

29.2 Baseline Model

We can start-off by describing the baseline model used as the starting point for all experiments. A baseline model configuration was chosen such that the model would perform reasonably well on the translation task.

- Embedding: 512-dimensions.
- RNN Cell: Gated Recurrent Unit or GRU.
- Encoder: Bidirectional.
- Encoder Depth: 2-layers (1 layer in each direction).
- Decoder Depth: 2-layers.
- Attention: Bahdanau-style.
- Optimizer: Adam.
- Dropout: 20% on input.

Each experiment started with the baseline model and varied one element in an attempt to isolate the impact of the design decision on the model skill, in this case, BLEU scores.

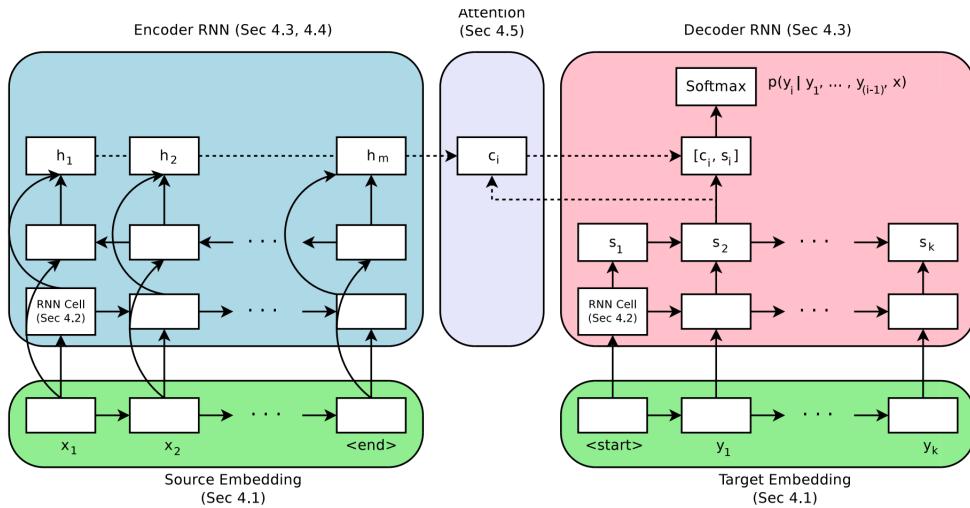


Figure 29.1: Encoder-Decoder Architecture for Neural Machine Translation. Taken from *Massive Exploration of Neural Machine Translation Architectures*.

29.3 Word Embedding Size

A word-embedding is used to represent words input to the encoder. This is a distributed representation where each word is mapped to a fixed-sized vector of continuous values. The benefit of this approach is that different words with similar meaning will have a similar representation. This distributed representation is often learned while fitting the model on the training data. The embedding size defines the length of the vectors used to represent words. It is generally believed that a larger dimensionality will result in a more expressive representation, and in turn, better skill. Interestingly, the results show that the largest size tested did achieve the best results, but the benefit of increasing the size was minor overall.

[results show] that 2048-dimensional embeddings yielded the overall best result, they only did so by a small margin. Even small 128-dimensional embeddings performed surprisingly well, while converging almost twice as quickly.

— *Massive Exploration of Neural Machine Translation Architectures*, 2017.

Recommendation: Start with a small embedding, such as 128, perhaps increase the size later for a minor lift in skill.

29.4 RNN Cell Type

There are generally three types of recurrent neural network cells that are commonly used:

- Simple RNN.
- Long Short-Term Memory or LSTM.
- Gated Recurrent Unit or GRU.

The LSTM was developed to address the vanishing gradient problem of the Simple RNN that limited the training of deep RNNs. The GRU was developed in an attempt to simplify the LSTM. Results showed that both the GRU and LSTM were significantly better than the Simple RNN, but the LSTM was generally better overall.

In our experiments, LSTM cells consistently outperformed GRU cells

— *Massive Exploration of Neural Machine Translation Architectures*, 2017.

Recommendation: Use LSTM RNN units in your model.

29.5 Encoder-Decoder Depth

Generally, deeper networks are believed to achieve better performance than shallow networks. The key is to find a balance between network depth, model skill, and training time. This is because we generally do not have infinite resources to train very deep networks if the benefit to skill is minor. The authors explore the depth of both the encoder and decoder models and the impact on model skill. When it comes to encoders, it was found that depth did not have a dramatic impact on skill and more surprisingly, a 1-layer bidirectional model performs only slightly better than a 4-layer bidirectional configuration. A two-layer bidirectional encoder performed slightly better than other configurations tested.

We found no clear evidence that encoder depth beyond two layers is necessary.

— *Massive Exploration of Neural Machine Translation Architectures*, 2017.

Recommendation: Use a 1-layer bidirectional encoder and extend to 2 bidirectional layers for a small lift in skill.

A similar story was seen when it came to decoders. The skill between decoders with 1, 2, and 4 layers was different by a small amount where a 4-layer decoder was slightly better. An 8-layer decoder did not converge under the test conditions.

On the decoder side, deeper models outperformed shallower ones by a small margin.

— *Massive Exploration of Neural Machine Translation Architectures*, 2017.

Recommendation: Use a 1-layer decoder as a starting point and use a 4-layer decoder for better results.

29.6 Direction of Encoder Input

The order of the sequence of source text can be provided to the encoder a number of ways:

- Forward or as-normal.
- Reversed.
- Both forward and reversed at the same time.

The authors explored the impact of the order of the input sequence on model skill comparing various unidirectional and bidirectional configurations. Generally, they confirmed previous findings that a reversed sequence is better than a forward sequence and that bidirectional is slightly better than a reversed sequence.

... bidirectional encoders generally outperform unidirectional encoders, but not by a large margin. The encoders with reversed source consistently outperform their non-reversed counterparts.

— *Massive Exploration of Neural Machine Translation Architectures*, 2017.

Recommendation: Use a reversed order input sequence or move to bidirectional for a small lift in model skill.

29.7 Attention Mechanism

A problem with the naive Encoder-Decoder model is that the encoder maps the input to a fixed-length internal representation from which the decoder must produce the entire output sequence. Attention is an improvement to the model that allows the decoder to *pay attention* to different words in the input sequence as it outputs each word in the output sequence. The authors look at a few variations on simple attention mechanisms. The results show that having attention results in dramatically better performance than not having attention.

While we did expect the attention-based models to significantly outperform those without an attention mechanism, we were surprised by just how poorly the [no attention] models fared.

— *Massive Exploration of Neural Machine Translation Architectures*, 2017.

The simple weighted average style attention described by Bahdanau, et al. in their 2015 paper *Neural machine translation by jointly learning to align and translate* was found to perform the best.

Recommendation: Use attention and prefer the Bahdanau-style weighted average style attention.

29.8 Inference

It is common in neural machine translation systems to use a beam-search to sample the probabilities for the words in the sequence output by the model. The wider the beam width, the more exhaustive the search, and, it is believed, the better the results. The results showed that a modest beam-width of 3-5 performed the best, which could be improved only very slightly through the use of length penalties. The authors generally recommend tuning the beam width on each specific problem.

We found that a well-tuned beam search is crucial to achieving good results, and that it leads to consistent gains of more than one BLEU point

— *Massive Exploration of Neural Machine Translation Architectures*, 2017.

Recommendation: Start with a greedy search (beam=1) and tune based on your problem.

29.9 Final Model

The authors pull together their findings into a single *best model* and compare the results of this model to other well-performing models and state-of-the-art results. The specific configurations of this model are summarized in the table below, taken from the paper. These parameters may be taken as a good or best starting point when developing your own encoder-decoder model for an NLP application.

Hyperparameter	Value
embedding dim	512
rnn cell variant	LSTMCell
encoder depth	4
decoder depth	4
attention dim	512
attention type	Bahdanau
encoder	bidirectional
beam size	10
length penalty	1.0

Figure 29.2: Summary of Model Configuration for the Final NMT Model. Taken from *Massive Exploration of Neural Machine Translation Architectures*.

The results of the system were shown to be impressive and achieve skill close to state-of-the-art with a simpler model, which was not the goal of the paper.

... we do show that through careful hyperparameter tuning and good initialization, it is possible to achieve state-of-the-art performance on standard WMT benchmarks

— *Massive Exploration of Neural Machine Translation Architectures*, 2017.

Importantly, the authors provide all of their code as an open source project called tf-seq2seq. Because two of the authors were members of the Google Brain residency program, their work was announced on the Google Research blog with the title *Introducing tf-seq2seq: An Open Source Sequence-to-Sequence Framework in TensorFlow*, 2017.

29.10 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- *Massive Exploration of Neural Machine Translation Architectures*, 2017.
<https://arxiv.org/abs/1703.03906>

- Denny Britz Homepage.
<http://blog.dennybritz.com/>
- WildML Blog.
<http://www.wildml.com/>
- *Introducing tf-seq2seq: An Open Source Sequence-to-Sequence Framework in TensorFlow*, 2017.
<https://research.googleblog.com/2017/04/introducing-tf-seq2seq-open-source.html>
- tf-seq2seq: A general-purpose encoder-decoder framework for TensorFlow.
<https://github.com/google/seq2seq>
- tf-seq2seq Project Documentation.
<https://google.github.io/seq2seq/>
- tf-seq2seq Tutorial: Neural Machine Translation Background.
<https://google.github.io/seq2seq/nmt/>
- *Neural machine translation by jointly learning to align and translate*, 2015.
<https://arxiv.org/abs/1409.0473>

29.11 Summary

In this chapter, you discovered how to best configure an encoder-decoder recurrent neural network for neural machine translation and other natural language processing tasks. Specifically, you learned:

- The Google study that investigated each model design decision in the encoder-decoder model to isolate their effects.
- The results and recommendations for design decisions like word embeddings, encoder and decoder depth, and attention mechanisms.
- A set of base model design decisioning that can be used as a starting point on your own sequence to sequence projects.

29.11.1 Next

In the next chapter, you will discover how you can develop a neural machine translation model.

Chapter 30

Project: Develop a Neural Machine Translation Model

Machine translation is a challenging task that traditionally involves large statistical models developed using highly sophisticated linguistic knowledge. Neural machine translation is the use of deep neural networks for the problem of machine translation. In this tutorial, you will discover how to develop a neural machine translation system for translating German phrases to English. After completing this tutorial, you will know:

- How to clean and prepare data ready to train a neural machine translation system.
- How to develop an encoder-decoder model for machine translation.
- How to use a trained model for inference on new input phrases and evaluate the model skill.

Let's get started.

30.1 Tutorial Overview

This tutorial is divided into the following parts:

1. German to English Translation Dataset
2. Preparing the Text Data
3. Train Neural Translation Model
4. Evaluate Neural Translation Model

30.2 German to English Translation Dataset

In this tutorial, we will use a dataset of German to English terms used as the basis for flashcards for language learning. The dataset is available from the ManyThings.org website, with examples drawn from the Tatoeba Project. The dataset is comprised of German phrases and their English counterparts and is intended to be used with the Anki flashcard software.

- Download the English-German pairs dataset.
<http://www.manythings.org/anki/deu-eng.zip>

Download the dataset to your current working directory and decompress it; for example:

```
unzip deu-eng.zip
```

Listing 30.1: Unzip the dataset

You will have a file called `deu.txt` that contains 152,820 pairs of English to German phrases, one pair per line with a tab separating the language. For example, the first 5 lines of the file look as follows:

```
Hi.    Hallo!
Hi.    GruB Gott!
Run!   Lauf!
Wow!   Potzdonner!
Wow!   Donnerwetter!
```

Listing 30.2: Sample of the raw dataset (with Unicode characters normalized).

We will frame the prediction problem as given a sequence of words in German as input, translate or predict the sequence of words in English. The model we will develop will be suitable for some beginner German phrases.

30.3 Preparing the Text Data

The next step is to prepare the text data ready for modeling. Take a look at the raw data and note what you see that we might need to handle in a data cleaning operation. For example, here are some observations I note from reviewing the raw data:

- There is punctuation.
- The text contains uppercase and lowercase.
- There are special characters in the German.
- There are duplicate phrases in English with different translations in German.
- The file is ordered by sentence length with very long sentences toward the end of the file.

A good text cleaning procedure may handle some or all of these observations. Data preparation is divided into two subsections:

1. Clean Text
2. Split Text

30.3.1 Clean Text

First, we must load the data in a way that preserves the Unicode German characters. The function below called `load_doc()` will load the file as a blob of text.

```
# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, mode='rt', encoding='utf-8')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text
```

Listing 30.3: Function to load a file into memory

Each line contains a single pair of phrases, first English and then German, separated by a tab character. We must split the loaded text by line and then by phrase. The function `to_pairs()` below will split the loaded text.

```
# split a loaded document into sentences
def to_pairs(doc):
    lines = doc.strip().split('\n')
    pairs = [line.split('\t') for line in lines]
    return pairs
```

Listing 30.4: Function to split lines into pairs

We are now ready to clean each sentence. The specific cleaning operations we will perform are as follows:

- Remove all non-printable characters.
- Remove all punctuation characters.
- Normalize all Unicode characters to ASCII (e.g. Latin characters).
- Normalize the case to lowercase.
- Remove any remaining tokens that are not alphabetic.

We will perform these operations on each phrase for each pair in the loaded dataset. The `clean_pairs()` function below implements these operations.

```
# clean a list of lines
def clean_pairs(lines):
    cleaned = list()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    re_print = re.compile('[^%s]' % re.escape(string.printable))
    for pair in lines:
        clean_pair = list()
        for line in pair:
            # normalize unicode characters
            line = normalize('NFD', line).encode('ascii', 'ignore')
            clean_pair.append(re_print.sub('', line))
        cleaned.append(clean_pair)
    return cleaned
```

```

line = line.decode('UTF-8')
# tokenize on white space
line = line.split()
# convert to lowercase
line = [word.lower() for word in line]
# remove punctuation from each token
line = [re_punc.sub('', w) for w in line]
# remove non-printable chars form each token
line = [re_print.sub('', w) for w in line]
# remove tokens with numbers in them
line = [word for word in line if word.isalpha()]
# store as string
clean_pair.append(' '.join(line))
cleaned.append(clean_pair)
return array(cleaned)

```

Listing 30.5: Function to clean text

Finally, now that the data has been cleaned, we can save the list of phrase pairs to a file ready for use. The function `save_clean_data()` uses the pickle API to save the list of clean text to file. Pulling all of this together, the complete example is listed below.

```

import string
import re
from pickle import dump
from unicodedata import normalize
from numpy import array

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, mode='rt', encoding='utf-8')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

# split a loaded document into sentences
def to_pairs(doc):
    lines = doc.strip().split('\n')
    pairs = [line.split('\t') for line in lines]
    return pairs

# clean a list of lines
def clean_pairs(lines):
    cleaned = list()
    # prepare regex for char filtering
    re_punc = re.compile('[%s]' % re.escape(string.punctuation))
    re_print = re.compile('[^%s]' % re.escape(string.printable))
    for pair in lines:
        clean_pair = list()
        for line in pair:
            # normalize unicode characters
            line = normalize('NFD', line).encode('ascii', 'ignore')
            line = line.decode('UTF-8')
            clean_pair.append(line)
        cleaned.append(clean_pair)
    return cleaned

```

```

# tokenize on white space
line = line.split()
# convert to lowercase
line = [word.lower() for word in line]
# remove punctuation from each token
line = [re_punc.sub('', w) for w in line]
# remove non-printable chars form each token
line = [re_print.sub('', w) for w in line]
# remove tokens with numbers in them
line = [word for word in line if word.isalpha()]
# store as string
clean_pair.append(' '.join(line))
cleaned.append(clean_pair)
return array(cleaned)

# save a list of clean sentences to file
def save_clean_data(sentences, filename):
    dump(sentences, open(filename, 'wb'))
    print('Saved: %s' % filename)

# load dataset
filename = 'deu.txt'
doc = load_doc(filename)
# split into english-german pairs
pairs = to_pairs(doc)
# clean sentences
clean_pairs = clean_pairs(pairs)
# save clean pairs to file
save_clean_data(clean_pairs, 'english-german.pkl')
# spot check
for i in range(100):
    print('[%s] => [%s]' % (clean_pairs[i,0], clean_pairs[i,1]))

```

Listing 30.6: Complete example of text data preparation.

Running the example creates a new file in the current working directory with the cleaned text called `english-german.pkl`. Some examples of the clean text are printed for us to evaluate at the end of the run to confirm that the clean operations were performed as expected.

30.3.2 Split Text

The clean data contains a little over 150,000 phrase pairs and some of the pairs toward the end of the file are very long. This is a good number of examples for developing a small translation model. The complexity of the model increases with the number of examples, length of phrases, and size of the vocabulary. Although we have a good dataset for modeling translation, we will simplify the problem slightly to dramatically reduce the size of the model required, and in turn the training time required to fit the model.

You can explore developing a model on the fuller dataset as an extension; I would love to hear how you do. We will simplify the problem by reducing the dataset to the first 10,000 examples in the file; these will be the shortest phrases in the dataset. Further, we will then stake the first 9,000 of those as examples for training and the remaining 1,000 examples to test the fit model.

Below is the complete example of loading the clean data, splitting it, and saving the split portions of data to new files.

```
from pickle import load
from pickle import dump
from numpy.random import shuffle

# load a clean dataset
def load_clean_sentences(filename):
    return load(open(filename, 'rb'))

# save a list of clean sentences to file
def save_clean_data(sentences, filename):
    dump(sentences, open(filename, 'wb'))
    print('Saved: %s' % filename)

# load dataset
raw_dataset = load_clean_sentences('english-german.pkl')

# reduce dataset size
n_sentences = 10000
dataset = raw_dataset[:n_sentences, :]
# random shuffle
shuffle(dataset)
# split into train/test
train, test = dataset[:9000], dataset[9000:]
# save
save_clean_data(dataset, 'english-german-both.pkl')
save_clean_data(train, 'english-german-train.pkl')
save_clean_data(test, 'english-german-test.pkl')
```

Listing 30.7: Complete example of splitting text data.

Running the example creates three new files: the `english-german-both.pkl` that contains all of the train and test examples that we can use to define the parameters of the problem, such as max phrase lengths and the vocabulary, and the `english-german-train.pkl` and `english-german-test.pkl` files for the train and test dataset. We are now ready to start developing our translation model.

30.4 Train Neural Translation Model

In this section, we will develop the translation model. This involves both loading and preparing the clean text data ready for modeling and defining and training the model on the prepared data. Let's start off by loading the datasets so that we can prepare the data. The function below named `load_clean_sentences()` can be used to load the train, test, and both datasets in turn.

```
# load a clean dataset
def load_clean_sentences(filename):
    return load(open(filename, 'rb'))

# load datasets
dataset = load_clean_sentences('english-german-both.pkl')
train = load_clean_sentences('english-german-train.pkl')
```

```
test = load_clean_sentences('english-german-test.pkl')
```

Listing 30.8: Load cleaned data from file.

We will use the *both* or combination of the train and test datasets to define the maximum length and vocabulary of the problem. This is for simplicity. Alternately, we could define these properties from the training dataset alone and truncate examples in the test set that are too long or have words that are out of the vocabulary. We can use the Keras Tokenizer class to map words to integers, as needed for modeling. We will use separate tokenizer for the English sequences and the German sequences. The function below-named `create_tokenizer()` will train a tokenizer on a list of phrases.

```
# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer
```

Listing 30.9: Fit a tokenizer on the clean text data.

Similarly, the function named `max_length()` below will find the length of the longest sequence in a list of phrases.

```
# max sentence length
def max_length(lines):
    return max(len(line.split()) for line in lines)
```

Listing 30.10: Calculate the maximum sequence length.

We can call these functions with the combined dataset to prepare tokenizers, vocabulary sizes, and maximum lengths for both the English and German phrases.

```
# prepare english tokenizer
eng_tokenizer = create_tokenizer(dataset[:, 0])
eng_vocab_size = len(eng_tokenizer.word_index) + 1
eng_length = max_length(dataset[:, 0])
print('English Vocabulary Size: %d' % eng_vocab_size)
print('English Max Length: %d' % (eng_length))
# prepare german tokenizer
ger_tokenizer = create_tokenizer(dataset[:, 1])
ger_vocab_size = len(ger_tokenizer.word_index) + 1
ger_length = max_length(dataset[:, 1])
print('German Vocabulary Size: %d' % ger_vocab_size)
print('German Max Length: %d' % (ger_length))
```

Listing 30.11: Prepare Tokenizers for source and target sequences.

We are now ready to prepare the training dataset. Each input and output sequence must be encoded to integers and padded to the maximum phrase length. This is because we will use a word embedding for the input sequences and one hot encode the output sequences. The function below named `encode_sequences()` will perform these operations and return the result.

```
# encode and pad sequences
def encode_sequences(tokenizer, length, lines):
    # integer encode sequences
    X = tokenizer.texts_to_sequences(lines)
    # pad sequences with 0 values
```

```
X = pad_sequences(X, maxlen=length, padding='post')
return X
```

Listing 30.12: Function to encode and pad sequences.

The output sequence needs to be one hot encoded. This is because the model will predict the probability of each word in the vocabulary as output. The function `encode_output()` below will one hot encode English output sequences.

```
# one hot encode target sequence
def encode_output(sequences, vocab_size):
    ylist = list()
    for sequence in sequences:
        encoded = to_categorical(sequence, num_classes=vocab_size)
        ylist.append(encoded)
    y = array(ylist)
    y = y.reshape(sequences.shape[0], sequences.shape[1], vocab_size)
    return y
```

Listing 30.13: One hot encode output sequences.

We can make use of these two functions and prepare both the train and test dataset ready for training the model.

```
# prepare training data
trainX = encode_sequences(ger_tokenizer, ger_length, train[:, 1])
trainY = encode_sequences(eng_tokenizer, eng_length, train[:, 0])
trainY = encode_output(trainY, eng_vocab_size)
# prepare validation data
testX = encode_sequences(ger_tokenizer, ger_length, test[:, 1])
testY = encode_sequences(eng_tokenizer, eng_length, test[:, 0])
testY = encode_output(testY, eng_vocab_size)
```

Listing 30.14: Prepare training and test data for modeling.

We are now ready to define the model. We will use an encoder-decoder LSTM model on this problem. In this architecture, the input sequence is encoded by a front-end model called the encoder then decoded word by word by a backend model called the decoder. The function `define_model()` below defines the model and takes a number of arguments used to configure the model, such as the size of the input and output vocabularies, the maximum length of input and output phrases, and the number of memory units used to configure the model.

The model is trained using the efficient Adam approach to stochastic gradient descent and minimizes the categorical loss function because we have framed the prediction problem as multiclass classification. The model configuration was not optimized for this problem, meaning that there is plenty of opportunity for you to tune it and lift the skill of the translations. I would love to see what you can come up with.

```
# define NMT model
def define_model(src_vocab, tar_vocab, src_timesteps, tar_timesteps, n_units):
    model = Sequential()
    model.add(Embedding(src_vocab, n_units, input_length=src_timesteps, mask_zero=True))
    model.add(LSTM(n_units))
    model.add(RepeatVector(tar_timesteps))
    model.add(LSTM(n_units, return_sequences=True))
    model.add(TimeDistributed(Dense(tar_vocab, activation='softmax')))
# compile model
```

```

model.compile(optimizer='adam', loss='categorical_crossentropy')
# summarize defined model
model.summary()
plot_model(model, to_file='model.png', show_shapes=True)
return model

```

Listing 30.15: Define and summarize the model.

Finally, we can train the model. We train the model for 30 epochs and a batch size of 64 examples. We use checkpointing to ensure that each time the model skill on the test set improves, the model is saved to file.

```

# fit model
checkpoint = ModelCheckpoint('model.h5', monitor='val_loss', verbose=1,
    save_best_only=True, mode='min')
model.fit(trainX, trainY, epochs=30, batch_size=64, validation_data=(testX, testY),
    callbacks=[checkpoint], verbose=2)

```

Listing 30.16: Fit the defined model and save models using checkpointing.

We can tie all of this together and fit the neural translation model. The complete working example is listed below.

```

from pickle import load
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Embedding
from keras.layers import RepeatVector
from keras.layers import TimeDistributed
from keras.callbacks import ModelCheckpoint

# load a clean dataset
def load_clean_sentences(filename):
    return load(open(filename, 'rb'))

# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# max sentence length
def max_length(lines):
    return max(len(line.split()) for line in lines)

# encode and pad sequences
def encode_sequences(tokenizer, length, lines):
    # integer encode sequences
    X = tokenizer.texts_to_sequences(lines)
    # pad sequences with 0 values
    X = pad_sequences(X, maxlen=length, padding='post')

```

```

return X

# one hot encode target sequence
def encode_output(sequences, vocab_size):
    ylist = list()
    for sequence in sequences:
        encoded = to_categorical(sequence, num_classes=vocab_size)
        ylist.append(encoded)
    y = array(ylist)
    y = y.reshape(sequences.shape[0], sequences.shape[1], vocab_size)
    return y

# define NMT model
def define_model(src_vocab, tar_vocab, src_timesteps, tar_timesteps, n_units):
    model = Sequential()
    model.add(Embedding(src_vocab, n_units, input_length=src_timesteps, mask_zero=True))
    model.add(LSTM(n_units))
    model.add(RepeatVector(tar_timesteps))
    model.add(LSTM(n_units, return_sequences=True))
    model.add(TimeDistributed(Dense(tar_vocab, activation='softmax')))
    # compile model
    model.compile(optimizer='adam', loss='categorical_crossentropy')
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model

# load datasets
dataset = load_clean_sentences('english-german-both.pkl')
train = load_clean_sentences('english-german-train.pkl')
test = load_clean_sentences('english-german-test.pkl')
# prepare english tokenizer
eng_tokenizer = create_tokenizer(dataset[:, 0])
eng_vocab_size = len(eng_tokenizer.word_index) + 1
eng_length = max_length(dataset[:, 0])
print('English Vocabulary Size: %d' % eng_vocab_size)
print('English Max Length: %d' % (eng_length))
# prepare german tokenizer
ger_tokenizer = create_tokenizer(dataset[:, 1])
ger_vocab_size = len(ger_tokenizer.word_index) + 1
ger_length = max_length(dataset[:, 1])
print('German Vocabulary Size: %d' % ger_vocab_size)
print('German Max Length: %d' % (ger_length))
# prepare training data
trainX = encode_sequences(ger_tokenizer, ger_length, train[:, 1])
trainY = encode_sequences(eng_tokenizer, eng_length, train[:, 0])
trainY = encode_output(trainY, eng_vocab_size)
# prepare validation data
testX = encode_sequences(ger_tokenizer, ger_length, test[:, 1])
testY = encode_sequences(eng_tokenizer, eng_length, test[:, 0])
testY = encode_output(testY, eng_vocab_size)
# define model
model = define_model(ger_vocab_size, eng_vocab_size, ger_length, eng_length, 256)
# fit model
checkpoint = ModelCheckpoint('model.h5', monitor='val_loss', verbose=1,
    save_best_only=True, mode='min')

```

```
model.fit(trainX, trainY, epochs=30, batch_size=64, validation_data=(testX, testY),  
callbacks=[checkpoint], verbose=2)
```

Listing 30.17: Complete example of training the neural machine translation model.

Running the example first prints a summary of the parameters of the dataset such as vocabulary size and maximum phrase lengths.

```
English Vocabulary Size: 2404  
English Max Length: 5  
German Vocabulary Size: 3856  
German Max Length: 10
```

Listing 30.18: Summary of the loaded data

Next, a summary of the defined model is printed, allowing us to confirm the model configuration.

```
-----  
Layer (type)          Output Shape         Param #  
=====-----  
embedding_1 (Embedding)    (None, 10, 256)      987136  
-----  
lstm_1 (LSTM)           (None, 256)          525312  
-----  
repeat_vector_1 (RepeatVector) (None, 5, 256)     0  
-----  
lstm_2 (LSTM)           (None, 5, 256)          525312  
-----  
time_distributed_1 (TimeDistributed) (None, 5, 2404) 617828  
=====-----  
Total params: 2,655,588  
Trainable params: 2,655,588  
Non-trainable params: 0  
-----
```

Listing 30.19: Summary of the defined model

A plot of the model is also created providing another perspective on the model configuration.

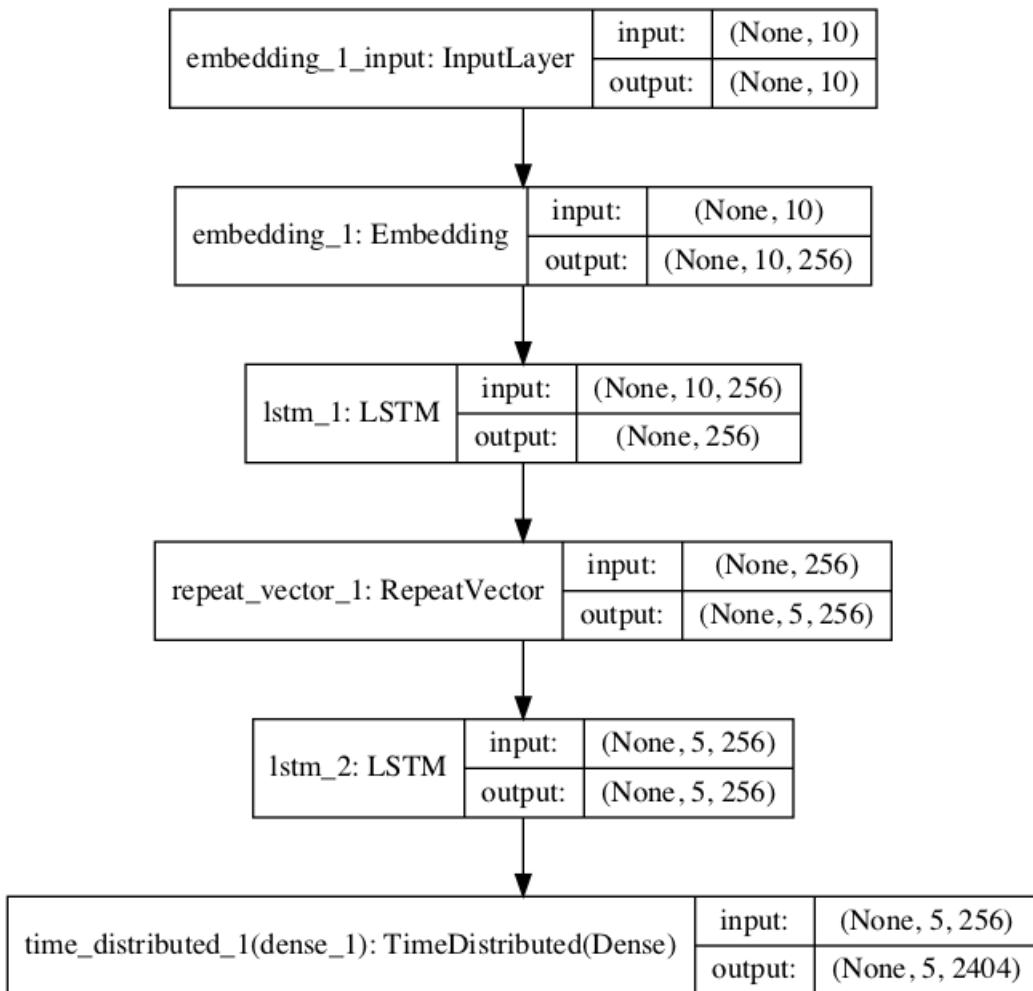


Figure 30.1: Plot of the defined neural machine translation model

Next, the model is trained. Each epoch takes about 30 seconds on modern CPU hardware; no GPU is required. During the run, the model will be saved to the file `model.h5`, ready for inference in the next step.

```

...
Epoch 26/30
Epoch 00025: val_loss improved from 2.20048 to 2.19976, saving model to model.h5
17s - loss: 0.7114 - val_loss: 2.1998
Epoch 27/30
Epoch 00026: val_loss improved from 2.19976 to 2.18255, saving model to model.h5
17s - loss: 0.6532 - val_loss: 2.1826
Epoch 28/30
Epoch 00027: val_loss did not improve
17s - loss: 0.5970 - val_loss: 2.1970
Epoch 29/30
Epoch 00028: val_loss improved from 2.18255 to 2.17872, saving model to model.h5
17s - loss: 0.5474 - val_loss: 2.1787
Epoch 30/30
Epoch 00029: val_loss did not improve
17s - loss: 0.5023 - val_loss: 2.1823

```

Listing 30.20: Summary output from training the neural machine translation model.

30.5 Evaluate Neural Translation Model

We will evaluate the model on the train and the test dataset. The model should perform very well on the train dataset and ideally have been generalized to perform well on the test dataset. Ideally, we would use a separate validation dataset to help with model selection during training instead of the test set. You can try this as an extension. The clean datasets must be loaded and prepared as before.

```
...
# load datasets
dataset = load_clean_sentences('english-german-both.pkl')
train = load_clean_sentences('english-german-train.pkl')
test = load_clean_sentences('english-german-test.pkl')
# prepare english tokenizer
eng_tokenizer = create_tokenizer(dataset[:, 0])
eng_vocab_size = len(eng_tokenizer.word_index) + 1
eng_length = max_length(dataset[:, 0])
# prepare german tokenizer
ger_tokenizer = create_tokenizer(dataset[:, 1])
ger_vocab_size = len(ger_tokenizer.word_index) + 1
ger_length = max_length(dataset[:, 1])
# prepare data
trainX = encode_sequences(ger_tokenizer, ger_length, train[:, 1])
testX = encode_sequences(ger_tokenizer, ger_length, test[:, 1])
```

Listing 30.21: Load and prepare data.

Next, the best model saved during training must be loaded.

```
# load model
model = load_model('model.h5')
```

Listing 30.22: Load and the saved model.

Evaluation involves two steps: first generating a translated output sequence, and then repeating this process for many input examples and summarizing the skill of the model across multiple cases. Starting with inference, the model can predict the entire output sequence in a one-shot manner.

```
translation = model.predict(source, verbose=0)
```

Listing 30.23: Predict the target sequence given the source sequence.

This will be a sequence of integers that we can enumerate and lookup in the tokenizer to map back to words. The function below, named `word_for_id()`, will perform this reverse mapping.

```
# map an integer to a word
def word_for_id(integer, tokenizer):
    for word, index in tokenizer.word_index.items():
        if index == integer:
            return word
```

```
    return None
```

Listing 30.24: Map a predicted word index to the word in the vocabulary.

We can perform this mapping for each integer in the translation and return the result as a string of words. The function `predict_sequence()` below performs this operation for a single encoded source phrase.

```
# generate target given source sequence
def predict_sequence(model, tokenizer, source):
    prediction = model.predict(source, verbose=0)[0]
    integers = [argmax(vector) for vector in prediction]
    target = list()
    for i in integers:
        word = word_for_id(i, tokenizer)
        if word is None:
            break
        target.append(word)
    return ' '.join(target)
```

Listing 30.25: Predict and interpret the target sequence.

Next, we can repeat this for each source phrase in a dataset and compare the predicted result to the expected target phrase in English. We can print some of these comparisons to screen to get an idea of how the model performs in practice. We will also calculate the BLEU scores to get a quantitative idea of how well the model has performed. The `evaluate_model()` function below implements this, calling the above `predict_sequence()` function for each phrase in a provided dataset.

```
# evaluate the skill of the model
def evaluate_model(model, tokenizer, sources, raw_dataset):
    actual, predicted = list(), list()
    for i, source in enumerate(sources):
        # translate encoded source text
        source = source.reshape((1, source.shape[0]))
        translation = predict_sequence(model, eng_tokenizer, source)
        raw_target, raw_src = raw_dataset[i]
        if i < 10:
            print('src=[%s], target=[%s], predicted=[%s]' % (raw_src, raw_target, translation))
        actual.append(raw_target.split())
        predicted.append(translation.split())
    # calculate BLEU score
    print('BLEU-1: %f' % corpus_bleu(actual, predicted, weights=(1.0, 0, 0, 0)))
    print('BLEU-2: %f' % corpus_bleu(actual, predicted, weights=(0.5, 0.5, 0, 0)))
    print('BLEU-3: %f' % corpus_bleu(actual, predicted, weights=(0.3, 0.3, 0.3, 0)))
    print('BLEU-4: %f' % corpus_bleu(actual, predicted, weights=(0.25, 0.25, 0.25, 0.25)))
```

Listing 30.26: Function to evaluate a fit model.

We can tie all of this together and evaluate the loaded model on both the training and test datasets. The complete code listing is provided below.

```
from pickle import load
from numpy import argmax
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model
```

```
from nltk.translate.bleu_score import corpus_bleu

# load a clean dataset
def load_clean_sentences(filename):
    return load(open(filename, 'rb'))

# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer

# max sentence length
def max_length(lines):
    return max(len(line.split()) for line in lines)

# encode and pad sequences
def encode_sequences(tokenizer, length, lines):
    # integer encode sequences
    X = tokenizer.texts_to_sequences(lines)
    # pad sequences with 0 values
    X = pad_sequences(X, maxlen=length, padding='post')
    return X

# map an integer to a word
def word_for_id(integer, tokenizer):
    for word, index in tokenizer.word_index.items():
        if index == integer:
            return word
    return None

# generate target given source sequence
def predict_sequence(model, tokenizer, source):
    prediction = model.predict(source, verbose=0)[0]
    integers = [argmax(vector) for vector in prediction]
    target = list()
    for i in integers:
        word = word_for_id(i, tokenizer)
        if word is None:
            break
        target.append(word)
    return ' '.join(target)

# evaluate the skill of the model
def evaluate_model(model, sources, raw_dataset):
    actual, predicted = list(), list()
    for i, source in enumerate(sources):
        # translate encoded source text
        source = source.reshape((1, source.shape[0]))
        translation = predict_sequence(model, eng_tokenizer, source)
        raw_target, raw_src = raw_dataset[i]
        if i < 10:
            print('src=[%s], target=[%s], predicted=[%s]' % (raw_src, raw_target, translation))
        actual.append(raw_target.split())
        predicted.append(translation.split())
    # calculate BLEU score
    corpus_bleu(actual, predicted)
```

```

print('BLEU-1: %f' % corpus_bleu(actual, predicted, weights=(1.0, 0, 0, 0)))
print('BLEU-2: %f' % corpus_bleu(actual, predicted, weights=(0.5, 0.5, 0, 0)))
print('BLEU-3: %f' % corpus_bleu(actual, predicted, weights=(0.3, 0.3, 0.3, 0)))
print('BLEU-4: %f' % corpus_bleu(actual, predicted, weights=(0.25, 0.25, 0.25, 0.25)))

# load datasets
dataset = load_clean_sentences('english-german-both.pkl')
train = load_clean_sentences('english-german-train.pkl')
test = load_clean_sentences('english-german-test.pkl')
# prepare english tokenizer
eng_tokenizer = create_tokenizer(dataset[:, 0])
eng_vocab_size = len(eng_tokenizer.word_index) + 1
eng_length = max_length(dataset[:, 0])
# prepare german tokenizer
ger_tokenizer = create_tokenizer(dataset[:, 1])
ger_vocab_size = len(ger_tokenizer.word_index) + 1
ger_length = max_length(dataset[:, 1])
# prepare data
trainX = encode_sequences(ger_tokenizer, ger_length, train[:, 1])
testX = encode_sequences(ger_tokenizer, ger_length, test[:, 1])
# load model
model = load_model('model.h5')
# test on some training sequences
print('train')
evaluate_model(model, trainX, train)
# test on some test sequences
print('test')
evaluate_model(model, testX, test)

```

Listing 30.27: Complete example of translating text with a fit neural machine translation model.

Running the example first prints examples of source text, expected and predicted translations, as well as scores for the training dataset, followed by the test dataset. Your specific results will differ given the random shuffling of the dataset and the stochastic nature of neural networks. Looking at the results for the test dataset first, we can see that the translations are readable and mostly correct. For example: ‘*ich liebe dich*’ was correctly translated to ‘*i love you*’.

We can also see that the translations were not perfect, with ‘*ich konnte nicht gehen*’ translated to *i cant go* instead of the expected ‘*i couldnt walk*’. We can also see the BLEU-4 score of 0.51, which provides an upper bound on what we might expect from this model.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```

src=[ich liebe dich], target=[i love you], predicted=[i love you]
src=[ich sagte du sollst den mund halten], target=[i said shut up], predicted=[i said stop
    up]
src=[wie geht es eurem vater], target=[hows your dad], predicted=[hows your dad]
src=[das gefallt mir], target=[i like that], predicted=[i like that]
src=[ich gehe immer zu fu], target=[i always walk], predicted=[i will to]
src=[ich konnte nicht gehen], target=[i couldnt walk], predicted=[i cant go]
src=[er ist sehr jung], target=[he is very young], predicted=[he is very young]
src=[versucht es doch einfach], target=[just try it], predicted=[just try it]
src=[sie sind jung], target=[youre young], predicted=[youre young]
src=[er ging surfen], target=[he went surfing], predicted=[he went surfing]

```

```
BLEU-1: 0.085682
BLEU-2: 0.284191
BLEU-3: 0.459090
BLEU-4: 0.517571
```

Listing 30.28: Sample output translation on the training dataset.

Looking at the results on the test set, do see readable translations, which is not an easy task. For example, we see ‘*ich mag dich nicht*’ correctly translated to ‘*i dont like you*’. We also see some poor translations and a good case that the model could support from further tuning, such as ‘*ich bin etwas beschwipst*’ translated as ‘*i a bit bit*’ instead of the expected *im a bit tipsy*. A BLEU-4 score of 0.076238 was achieved, providing a baseline skill to improve upon with further improvements to the model.

```
src=[tom erblasste], target=[tom turned pale], predicted=[tom went pale]
src=[bring mich nach hause], target=[take me home], predicted=[let us at]
src=[ich bin etwas beschwipst], target=[im a bit tipsy], predicted=[i a bit bit]
src=[das ist eine frucht], target=[its a fruit], predicted=[thats a a]
src=[ich bin pazifist], target=[im a pacifist], predicted=[im am]
src=[unser plan ist aufgegangen], target=[our plan worked], predicted=[who is a man]
src=[hallo tom], target=[hi tom], predicted=[hello tom]
src=[sei nicht nervos], target=[dont be nervous], predicted=[dont be crazy]
src=[ich mag dich nicht], target=[i dont like you], predicted=[i dont like you]
src=[tom stellte eine falle], target=[tom set a trap], predicted=[tom has a cough]

BLEU-1: 0.082088
BLEU-2: 0.006182
BLEU-3: 0.046129
BLEU-4: 0.076238
```

Listing 30.29: Sample output translation on the test dataset.

30.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Data Cleaning.** Different data cleaning operations could be performed on the data, such as not removing punctuation or normalizing case, or perhaps removing duplicate English phrases.
- **Vocabulary.** The vocabulary could be refined, perhaps removing words used less than 5 or 10 times in the dataset and replaced with *unk*.
- **More Data.** The dataset used to fit the model could be expanded to 50,000, 100,000 phrases, or more.
- **Input Order.** The order of input phrases could be reversed, which has been reported to lift skill, or a Bidirectional input layer could be used.
- **Layers.** The encoder and/or the decoder models could be expanded with additional layers and trained for more epochs, providing more representational capacity for the model.

- **Units.** The number of memory units in the encoder and decoder could be increased, providing more representational capacity for the model.
- **Regularization.** The model could use regularization, such as weight or activation regularization, or the use of dropout on the LSTM layers.
- **Pre-Trained Word Vectors.** Pre-trained word vectors could be used in the model.
- **Alternate Measure.** Explore alternate performance measures beside BLEU such as ROGUE. Compare scores for the same translations to develop an intuition for how the measures differ in practice.
- **Recursive Model.** A recursive formulation of the model could be used where the next word in the output sequence could be conditional on the input sequence and the output sequence generated so far.

If you explore any of these extensions, I'd love to know.

30.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

30.7.1 Dataset

- Tab-delimited Bilingual Sentence Pairs.
<http://www.manythings.org/anki/>
- German - English deu-eng.zip.
<http://www.manythings.org/anki/deu-eng.zip>

30.7.2 Neural Machine Translation

- *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, 2016.
<https://arxiv.org/abs/1609.08144>
- *Sequence to Sequence Learning with Neural Networks*, 2014.
<https://arxiv.org/abs/1409.3215>
- *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.
<https://arxiv.org/abs/1406.1078>
- *Neural Machine Translation by Jointly Learning to Align and Translate*, 2014.
<https://arxiv.org/abs/1409.0473>
- *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*, 2014.
<https://arxiv.org/abs/1409.1259>
- *Massive Exploration of Neural Machine Translation Architectures*, 2017.
<https://arxiv.org/abs/1703.03906>

30.8 Summary

In this tutorial, you discovered how to develop a neural machine translation system for translating German phrases to English. Specifically, you learned:

- How to clean and prepare data ready to train a neural machine translation system.
- How to develop an encoder-decoder model for machine translation.
- How to use a trained model for inference on new input phrases and evaluate the model skill.

30.8.1 Next

This is the final chapter for the machine translation part. In the next part you will discover helpful information in the appendix.

Part X

Appendix

Appendix A

Getting Help

This is just the beginning of your journey with deep learning for natural language processing in Python. As you start to work on projects and expand your existing knowledge of the techniques, you may need help. This appendix points out some of the best sources of help.

A.1 Official Keras Destinations

This section lists the official Keras sites that you may find helpful.

- Keras Official Blog.
<https://blog.keras.io/>
- Keras API Documentation.
<https://keras.io/>
- Keras Source Code Project.
<https://github.com/fchollet/keras>

A.2 Where to Get Help with Keras

This section lists the 9 best places I know where you can get help with Keras and LSTMs.

- Keras Users Google Group.
<https://groups.google.com/forum/#!forum/keras-users>
- Keras Slack Channel (you must request to join).
<https://keras-slack-autojoin.herokuapp.com/>
- Keras on Gitter.
<https://gitter.im/Keras-io/Lobby#>
- Keras tag on StackOverflow.
<https://stackoverflow.com/questions/tagged/keras>
- Keras tag on CrossValidated.
<https://stats.stackexchange.com/questions/tagged/keras>

- Keras tag on DataScience.
<https://datascience.stackexchange.com/questions/tagged/keras>
- Keras Topic on Quora.
<https://www.quora.com/topic/Keras>
- Keras Github Issues.
<https://github.com/fchollet/keras/issues>
- Keras on Twitter.
<https://twitter.com/hashtag/keras>

A.3 Where to Get Help with Natural Language

This section lists the best places I know where you can get help with natural language processing.

- Language Technology Reddit.
<https://www.reddit.com/r/LanguageTechnology/>
- NLP Tag on StackOverflow.
<https://stackoverflow.com/questions/tagged/nlp>
- Linguistics Stack Exchange.
<https://linguistics.stackexchange.com/>
- Natural Language Processing Topic on Quora.
<https://www.quora.com/topic/Natural-Language-Processing>

A.4 How to Ask Questions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

- Boil your question down to the simplest form. E.g. not something broad like *my model does not work* or *how does x work*.
- Search for answers before asking questions.
- Provide complete code and error messages.
- Boil your code down to the smallest possible working example that demonstrates the issue.

A.5 Contact the Author

You are not alone. If you ever have any questions about deep learning, natural language processing, or this book, please contact me directly. I will do my best to help.

Jason Brownlee

Jason@MachineLearningMastery.com

Appendix B

How to Setup a Workstation for Deep Learning

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda.

After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning and deep learning software. These instructions are suitable for Windows, Mac OS X, and Linux platforms. I will demonstrate them on OS X, so you may see some mac dialogs and file extensions.

B.1 Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda
2. Install Anaconda
3. Start and Update Anaconda
4. Install Deep Learning Libraries

Note: The specific versions may differ as the software and libraries are updated frequently.

B.2 Download Anaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

- 1. Visit the Anaconda homepage.
<https://www.continuum.io/>
- 2. Click **Anaconda** from the menu and click **Download** to go to the download page.
<https://www.continuum.io/downloads>

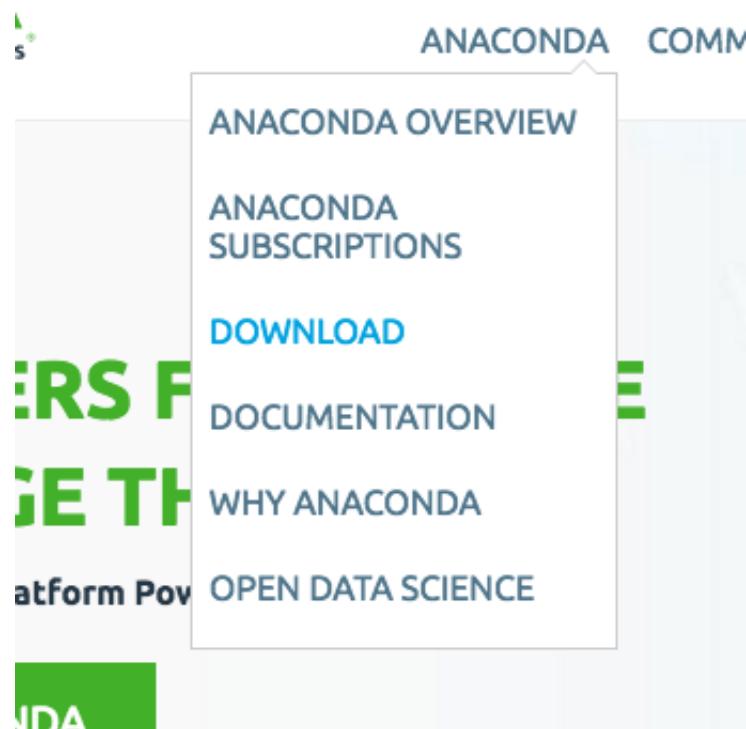


Figure B.1: Click Anaconda and Download.

- 3. Choose the download suitable for your platform (Windows, OSX, or Linux):
 - Choose Python 3.6
 - Choose the Graphical Installer

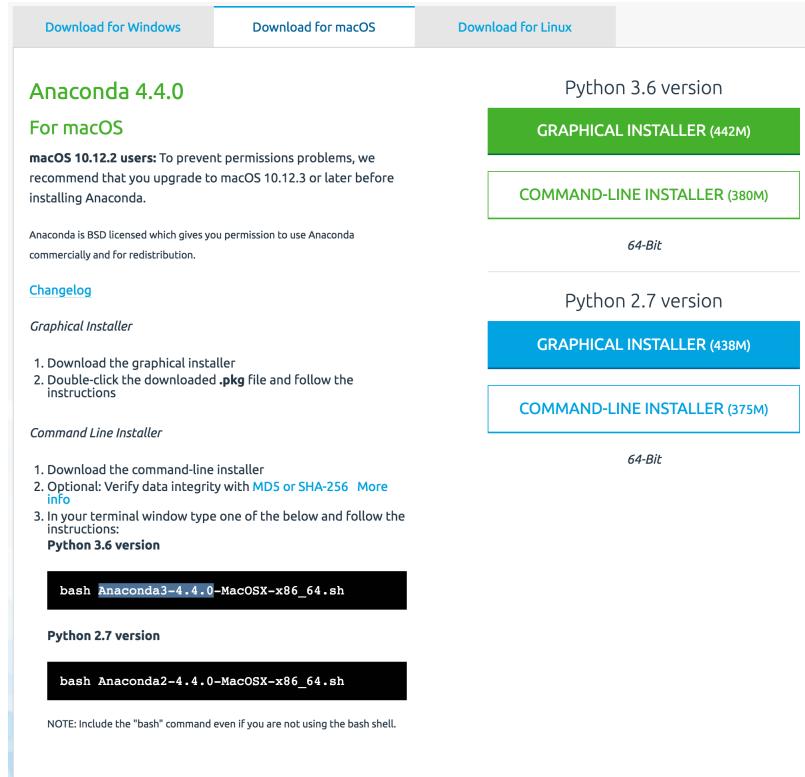


Figure B.2: Choose Anaconda Download for Your Platform.

This will download the Anaconda Python package to your workstation. I'm on OS X, so I chose the OS X version. The file is about 426 MB. You should have a file with a name like:

`Anaconda3-4.4.0-MacOSX-x86_64.pkg`

Listing B.1: Example filename on Mac OS X.

B.3 Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

- 1. Double click the downloaded file.
- 2. Follow the installation wizard.

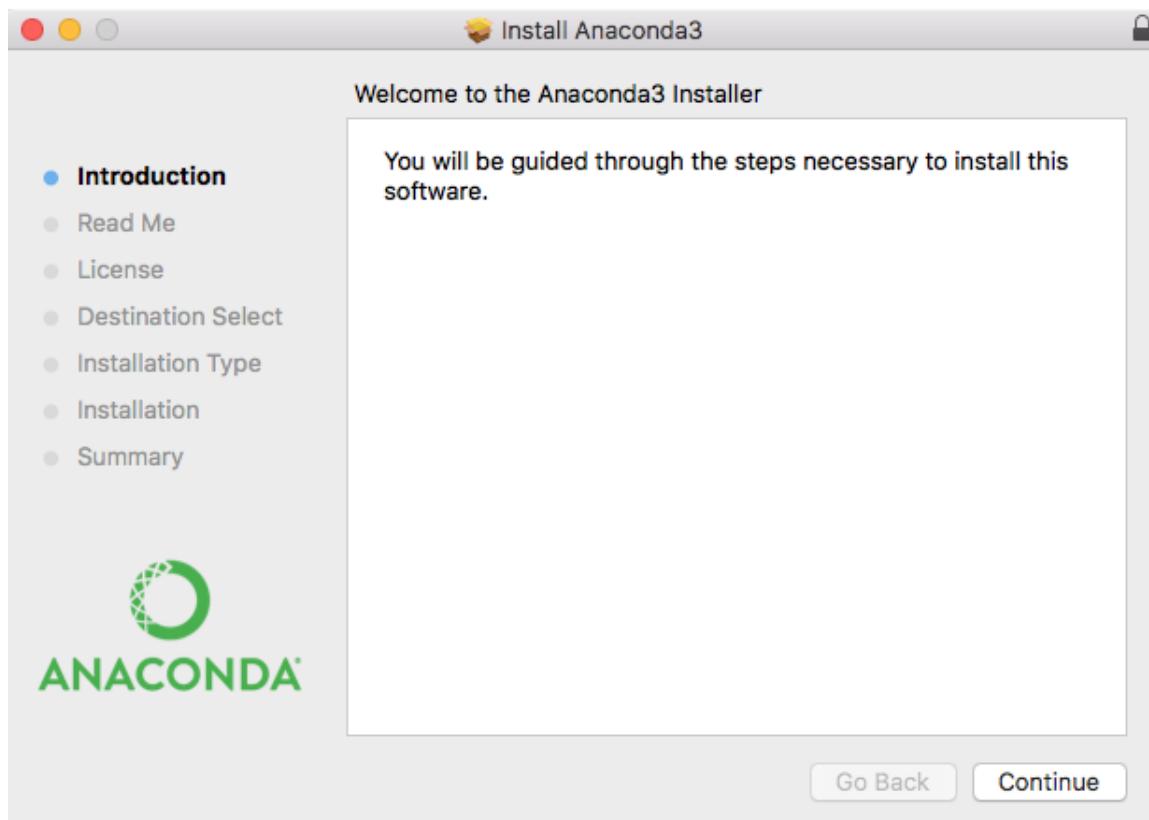


Figure B.3: Anaconda Python Installation Wizard.

Installation is quick and painless. There should be no tricky questions or sticking points.

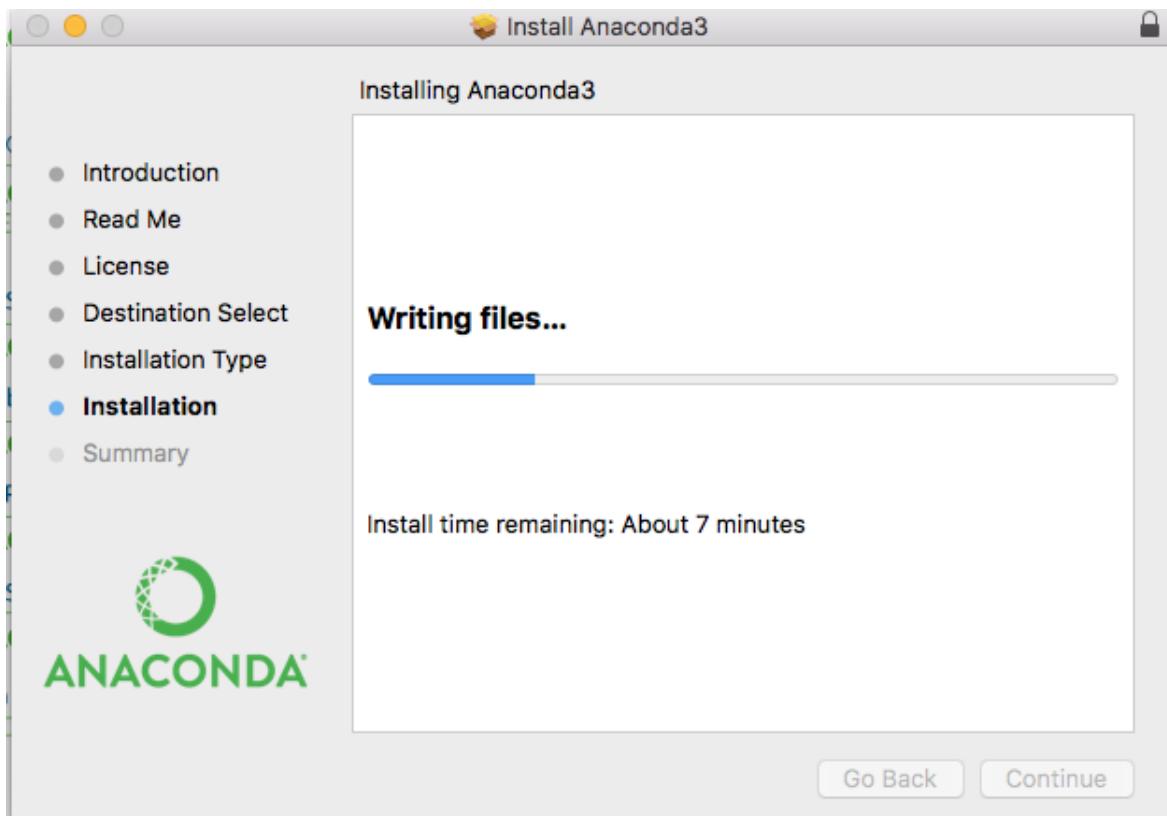


Figure B.4: Anaconda Python Installation Wizard Writing Files.

The installation should take less than 10 minutes and take up a little more than 1 GB of space on your hard drive.

B.4 Start and Update Anaconda

In this step, we will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.

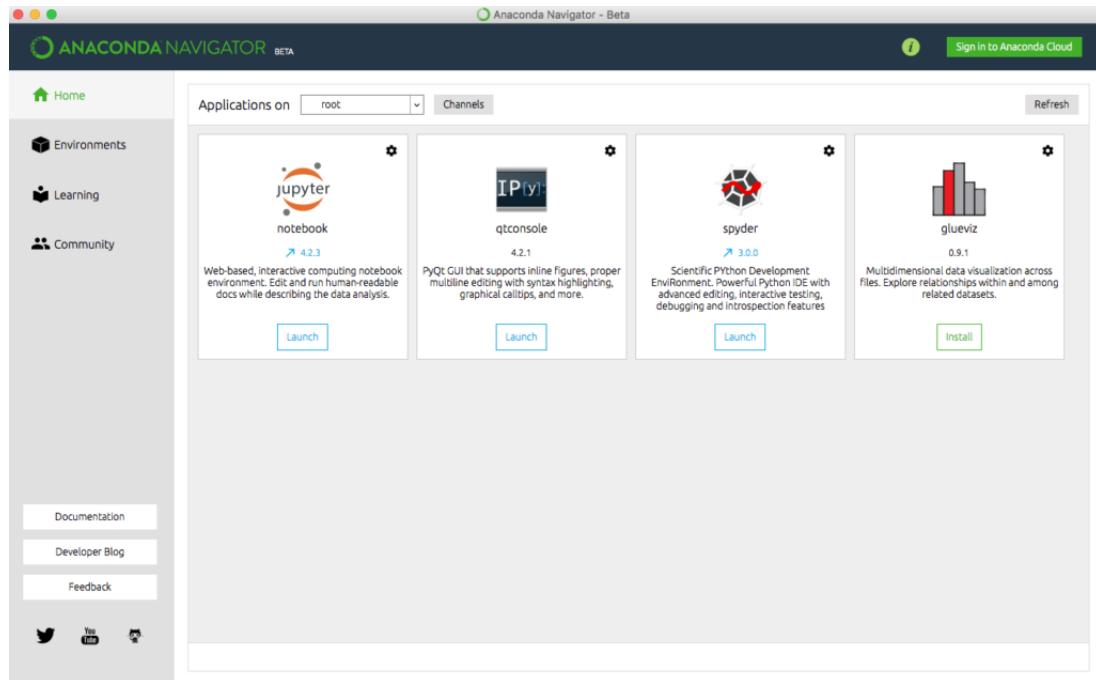


Figure B.5: Anaconda Navigator GUI.

You can use the Anaconda Navigator and graphical development environments later; for now, I recommend starting with the Anaconda command line environment called conda. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

- 1. Open a terminal (command line window).
- 2. Confirm conda is installed correctly, by typing:

```
conda -v
```

Listing B.2: Check the conda version.

You should see the following (or something similar):

```
conda 4.3.21
```

Listing B.3: Example conda version.

- 3. Confirm Python is installed correctly by typing:

```
python -v
```

Listing B.4: Check the Python version.

You should see the following (or something similar):

```
Python 3.6.1 :: Anaconda 4.4.0 (x86_64)
```

Listing B.5: Example Python version.

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the *Further Reading* section.

- 4. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

Listing B.6: Update conda and anaconda.

You may need to install some packages and confirm the updates.

- 5. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type `python` and type the commands in directly. Alternatively, I recommend opening a text editor and copy-pasting the script into your editor.

```
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing B.7: Code to check that key Python libraries are installed.

Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

Listing B.8: Run the script from the command line.

You should see output like the following:

```
scipy: 0.19.1
numpy: 1.13.3
matplotlib: 2.1.0
pandas: 0.20.3
statsmodels: 0.8.0
sklearn: 0.19.0
```

Listing B.9: Sample output of versions script.

B.5 Install Deep Learning Libraries

In this step, we will install Python libraries used for deep learning, specifically: Theano, TensorFlow, and Keras. Note: I recommend using Keras for deep learning and Keras only requires one of Theano or TensorFlow to be installed. You do not need both. There may be problems installing TensorFlow on some Windows machines.

- 1. Install the Theano deep learning library by typing:

```
conda install theano
```

Listing B.10: Install Theano with conda.

- 2. Install the TensorFlow deep learning library by typing:

```
conda install -c conda-forge tensorflow
```

Listing B.11: Install TensorFlow with conda.

Alternatively, you may choose to install using pip and a specific version of TensorFlow for your platform.

- 3. Install Keras by typing:

```
pip install keras
```

Listing B.12: Install Keras with pip.

- 4. Confirm your deep learning environment is installed and working correctly.

Create a script that prints the version numbers of each library, as we did before for the SciPy environment.

```
# theano
import theano
print('theano: %s' % theano.__version__)
# tensorflow
import tensorflow
print('tensorflow: %s' % tensorflow.__version__)
# keras
import keras
print('keras: %s' % keras.__version__)
```

Listing B.13: Code to check that key deep learning libraries are installed.

Save the script to a file `deep_versions.py`. Run the script by typing:

```
python deep_versions.py
```

Listing B.14: Run script from the command line.

You should see output like:

```
theano: 0.9.0
tensorflow: 1.3.0
keras: 2.0.8
```

Listing B.15: Sample output of the deep learning versions script.

B.6 Further Reading

This section provides resources if you want to know more about Anaconda.

- Anaconda homepage.
<https://www.continuum.io/>
- Anaconda Navigator.
<https://docs.continuum.io/anaconda/navigator.html>
- The conda command line tool.
<http://conda.pydata.org/docs/index.html>
- Instructions for installing TensorFlow in Anaconda.
https://www.tensorflow.org/get_started/os_setup#anaconda_installation

B.7 Summary

Congratulations, you now have a working Python development environment for machine learning and deep learning. You can now learn and practice machine learning and deep learning on your workstation.

Appendix C

How to Use Deep Learning 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.

C.1 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 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.

Note: The specific versions may differ as the software and libraries are updated frequently.

C.2 Setup Your AWS Account

You need an account on Amazon Web Services¹.

- 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.

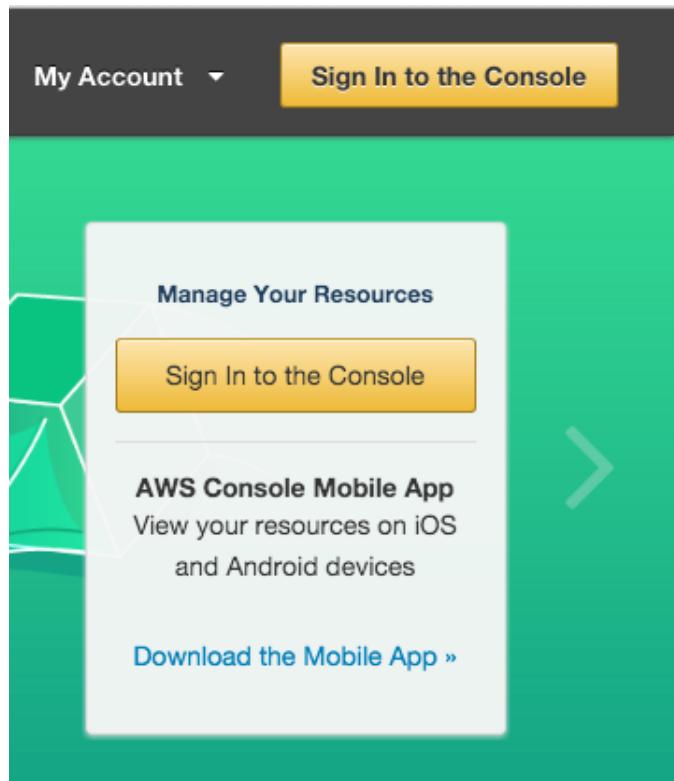


Figure C.1: AWS Sign-in Button

- 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>



The image shows the AWS Sign-In or Create an AWS Account form. At the top is the Amazon Web Services logo. Below it is the heading "Sign In or Create an AWS Account". A question "What is your email (phone for mobile accounts)? " is followed by a text input field. Below that is a question "E-mail or mobile number:" followed by another text input field. There are two radio button options: "I am a new user." (unselected) and "I am a returning user and my password is:" (selected). Below the selected radio button is a text input field. At the bottom right is a yellow "Sign in using our secure server" button with a circular arrow icon. To its left is a link "Forgot your password?".

Figure C.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.

C.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 is called the **Deep Learning AMI Amazon Linux Version** and was created and is maintained by Amazon. Let's launch it as an instance.

- 1. Login to your AWS console if you have not already.
<https://console.aws.amazon.com/console/home>

Amazon Web Services

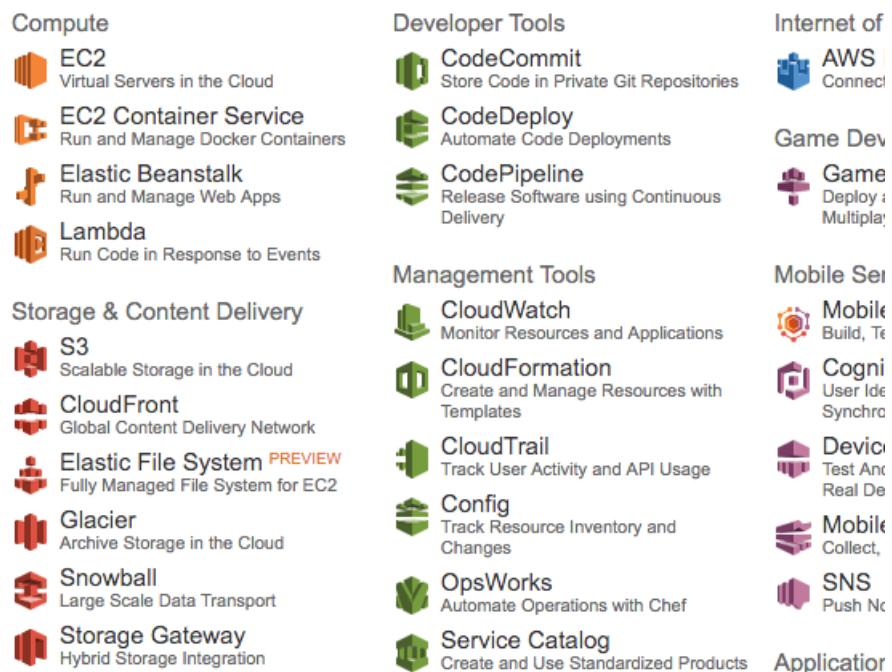


Figure C.3: AWS Console

- 2. Click on EC2 for launching a new virtual server.
- 3. Select *US West Oregon* 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.
- 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.

Step 1: Choose an Amazon Machine Image (AMI)

[Cancel and Exit](#)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. You can select an AMI provided by AWS, our user community, or the AWS Marketplace; or you can select one of your own AMIs.

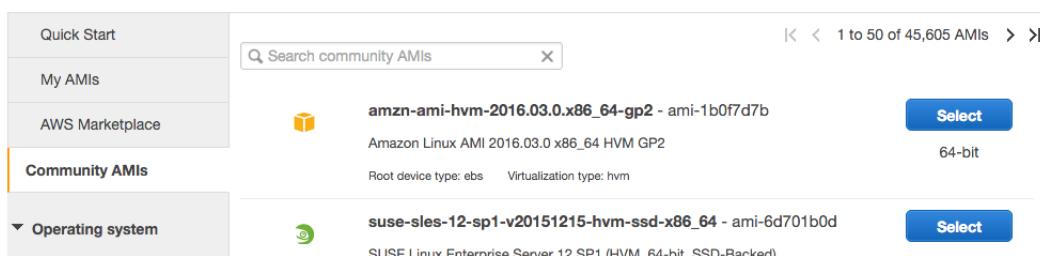


Figure C.4: Community AMIs

- 6. Enter `ami-df77b6a7` in the *Search community AMIs* search box and press enter (this is the current AMI id for v3.3 but the AMI may have been updated since, you check for a more recent id²). You should be presented with a single result.

²<https://aws.amazon.com/marketplace/pp/B01MOAXXQB>

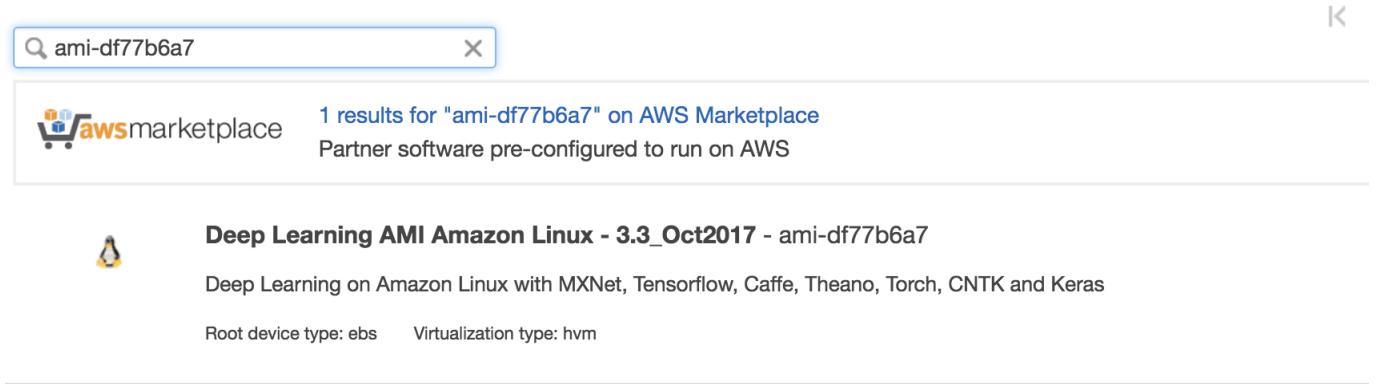


Figure C.5: Select a Specific AMI

- 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. The choice of hardware will impact the price, I generally recommend g2 and p2 hardware. See the AMI page for estimated pricing per hour for different hardware configurations³.

	Compute Optimized	g2.2xlarge	8	15	2 x 60 (SSD)	Yes	High
	GPU instances	g2.2xlarge	8	15	2 x 60 (SSD)	Yes	High
	GPU instances	g2.2xlarge	8	15	2 x 60 (SSD)	Yes	High
	CPU instances	g2.2xlarge	8	15	2 x 60 (SSD)	Yes	High

Figure C.6: 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.

³<https://aws.amazon.com/marketplace/pp/B01MOAXXQB>

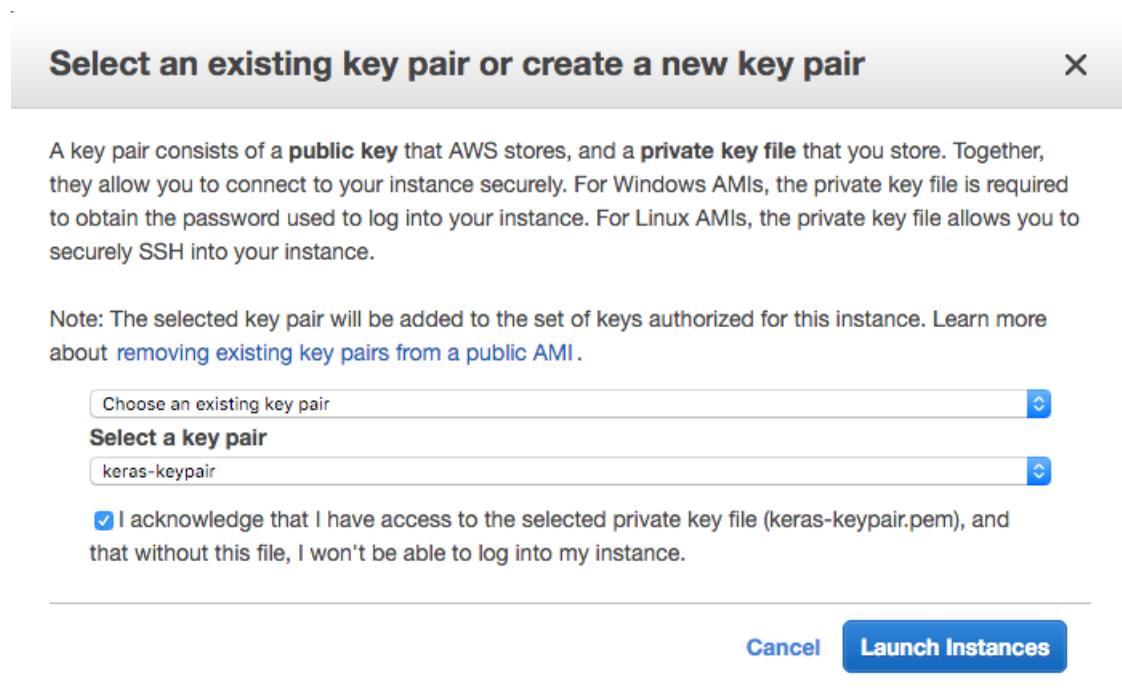


Figure C.7: Select Your Key Pair

- 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 C.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.

Description		Status Checks	Monitoring	Tags
Instance ID	i-0852e21f4779bb063			
Instance state	running			
Instance type	g2.2xlarge			
Elastic IPs				
Availability zone	us-west-2b			
Security groups	launch-wizard-2, view inbound rules			
Scheduled events	No scheduled events			
AMI ID	Deep Learning AMI AmazonLinux - 2.0 (ami-dfb13ebf)			
Platform	-			
IAM role	-			
Key pair name	test-aws-deep			
Owner	959845963779			
Launch time	March 22, 2017 at 8:13:53 AM UTC+11 (less than one hour)			
Termination protection	False			
Lifecycle	normal			
Monitoring	basic			
Alarm status	None			
Kernel ID	-			
RAM disk ID	-			
Placement group	-			
Virtualization	hvm			
Reservation	r-01d15becdf2de436d			
AMI launch index	0			
Tenancy	default			
Host ID	-			
Affinity	-			
State transition reason	-			
State transition reason message	-			
Public DNS (IPv4)	ec2-54-186-97-77.us-west-2.compute.amazonaws.com			
IPv4 Public IP	54.186.97.77			
IPv6 IPs	-			
Private DNS	ip-172-31-45-13.us-west-2.compute.internal			
Private IPs	172.31.45.13			
Secondary private IPs				
VPC ID	vpc-7d09db18			
Subnet ID	subnet-ddc962b8			
Network interfaces	eth0			
Source/dest. check	True			
EBS-optimized	False			
Root device type	ebs			
Root device	/dev/xvda			
Block devices	/dev/xvda			

Figure C.8: Review Your Running Instance

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

C.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 54.186.97.77. **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 to your server using SSH, for example:

```
ssh -i keras-aws-keypair.pem ec2-user@54.186.97.77
```

Listing C.2: Log-in To Your AWS Instance.

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

You are now logged into your server.

```
=====
  _\ ( __ )   Deep Learning AMI for Amazon Linux
  \_\|_|_
=====
The README file for the AMI ----- /home/ec2-user/src/README.md
Tests for deep learning frameworks ----- /home/ec2-user/src/bin
=====
[ec2-user@ip-172-31-45-13 ~]$ █
```

Figure C.9: 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.

C.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 2.0.8. We can specify this version as part of the upgrade of Keras via pip.

```
sudo pip install --upgrade keras==2.0.8
```

Listing C.3: Update Keras Using pip.

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

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

Listing C.4: Script To Check Keras Configuration.

You should see:

```
Using TensorFlow backend.
I tensorflow/stream_executor/dso_loader.cc:135] successfully opened CUDA library
    libcublas.so.7.5 locally
I tensorflow/stream_executor/dso_loader.cc:135] successfully opened CUDA library
    libcudnn.so.5 locally
I tensorflow/stream_executor/dso_loader.cc:135] successfully opened CUDA library
    libcufft.so.7.5 locally
I tensorflow/stream_executor/dso_loader.cc:135] successfully opened CUDA library
    libcuda.so.1 locally
I tensorflow/stream_executor/dso_loader.cc:135] successfully opened CUDA library
    libcurand.so.7.5 locally
2.0.8
```

Listing C.5: Sample Output of Script to Check Keras Configuration.

You are now free to run your code.

C.5 Build and Run Models on AWS

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

C.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 ec2-user@54.186.97.77:~/
```

Listing C.6: 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.

C.5.2 Run Models on AWS

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

```
python filename.py
```

Listing C.7: 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 C.8: Run Script as a Background Process.

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

C.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 C.9: 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.

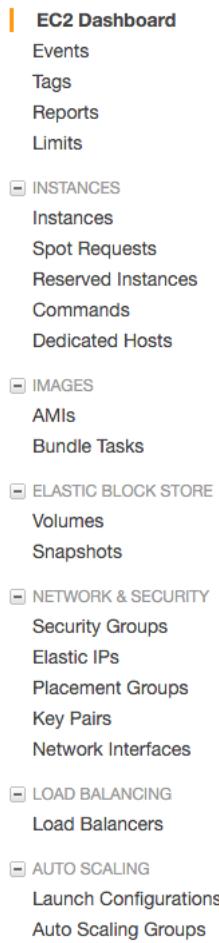


Figure C.10: Review Your List of Running Instances

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

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks
	i-bd12ae08	g2.2xlarge	us-west-1a	running	2/2 checks ...

Figure C.11: 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.

C.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.

C.8 Further Reading

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>
- Deep Learning AMI Amazon Linux Version on the AMI Marketplace.
<https://aws.amazon.com/marketplace/pp/B01M0AXXQB>

C.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.

Part XI

Conclusions

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come. You now know:

1. What natural language processing is and why it is challenging.
2. What deep learning is and how it is different from other machine learning methods.
3. The promise of deep learning methods for natural language processing problems.
4. How to prepare text data for modeling using best-of-breed Python libraries.
5. How to develop distributed representations of text using word embedding models.
6. How to develop a bag-of-words model, a representation technique that can be used for machine learning and deep learning methods.
7. How to develop a neural sentiment analysis model for automatically predicting the class label for a text document.
8. How to develop a neural language model, required for any text generating neural network.
9. How to develop a photo captioning system to automatically generate textual descriptions of photographs.
10. How to develop a neural machine translation system for translating text from one language to another.

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 implement and work through natural language prediction problems using deep learning in Python. You can now confidently bring deep learning models to your own natural language processing problems. The sky's the limit.

Thank You!

I want to take a moment and sincerely thank you for letting me help you start your deep learning for natural language processing journey. I hope you keep learning and have fun as you continue to master machine learning.

Jason Brownlee
2017