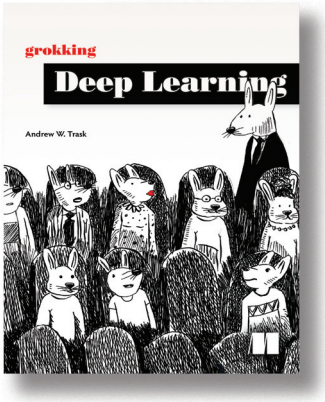
Author Picks

FREE

# Exploring Deep Learning for Language

Chapters selected by Jeff Smith

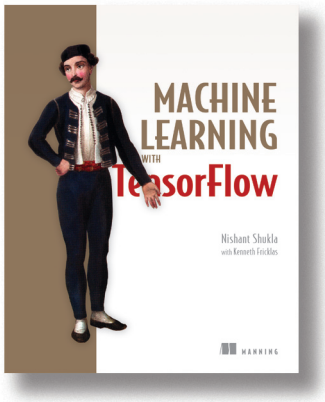
manning




*Grokking Deep Learning*

by Andrew Trask

ISBN: 9781617293702
325 pages
$39.99
2019




*Machine Learning with TensorFlow*

by Nishant Shukla

ISBN: 9781617293870
280 pages
$35.99
January 2018

# *Exploring Deep Learning for Language*

Chapters Selected by Jeff Smith

**Manning Author Picks**

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Candace Gillhoolley, cagi@manning.com



Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

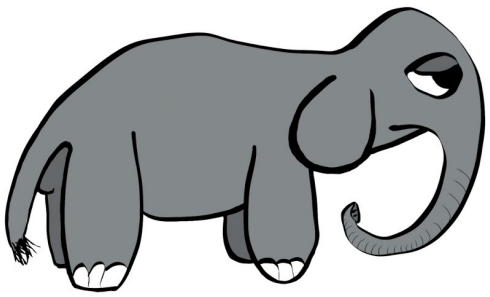
ISBN: 9781617296796
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 24 23 22 21 20 19

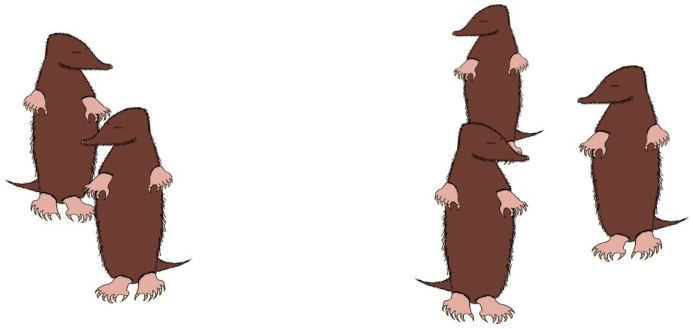
# *contents*

# *introduction*

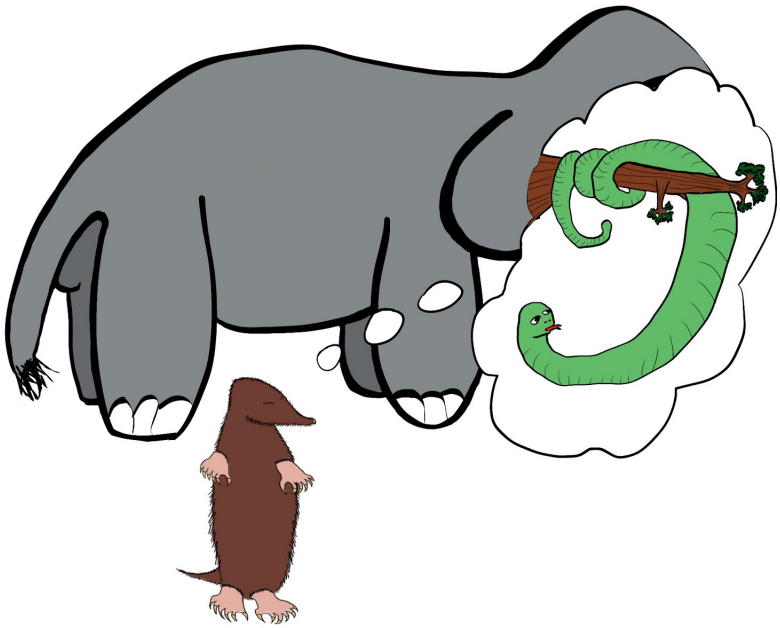
Whenever we learn a new topic, our perspective on the topic's always a bit incomplete. As learners, we're like the moles who encountered an elephant for the first time.

*Upon encountering an elephant for the first time, the moles were confused.*

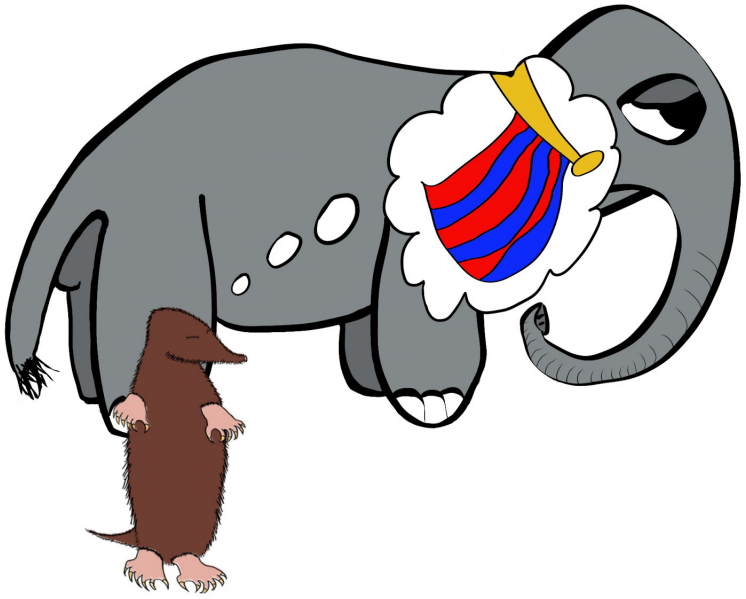




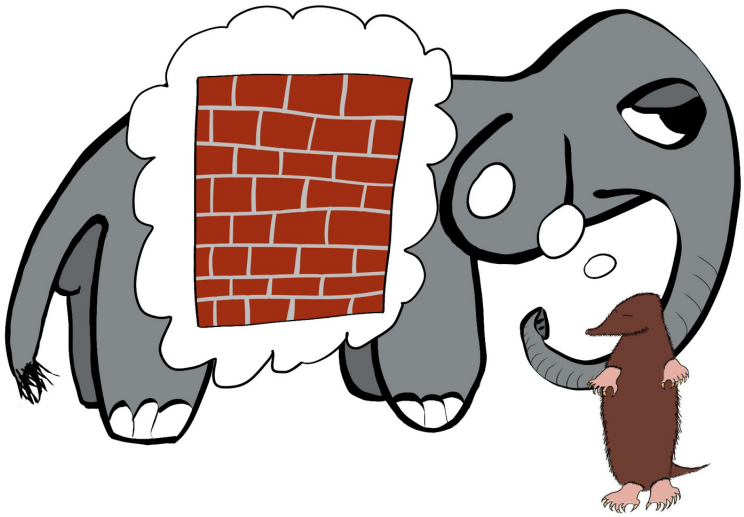
*The mole goes through life navigating by touch, and these moles touched this strange new animal, each in different places.*



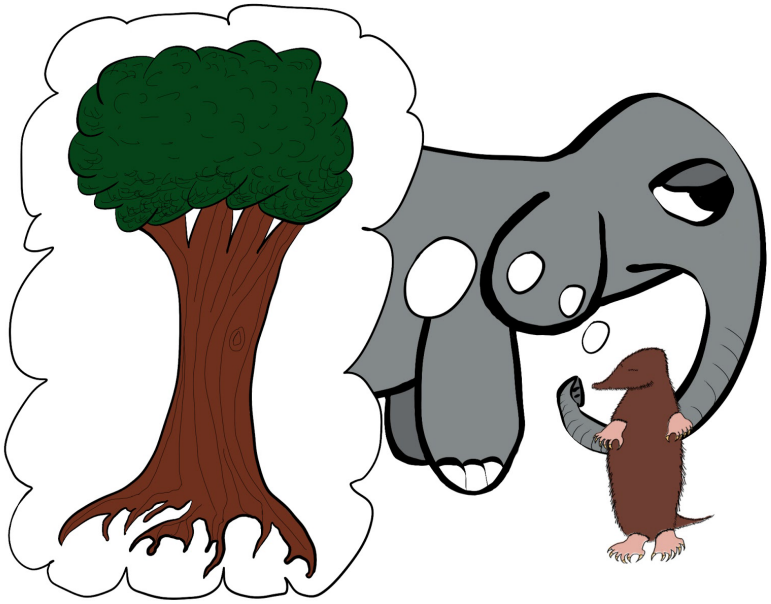
*"An elephant is like a snake," said the first mole. "Long and sinuous, yet strong."*



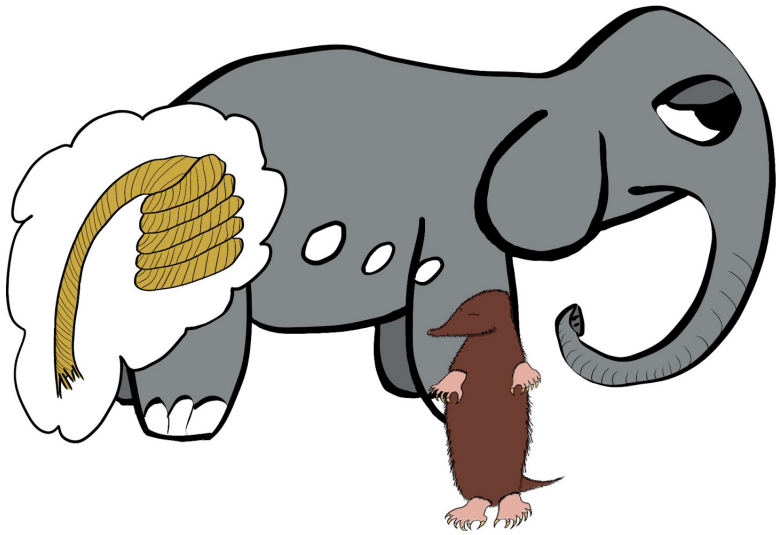
*"It is not," said the second mole. "The elephant is like a flag, hanging loosely."*



*"Both of you are wrong," said the third mole. "The elephant is much like a wall, broad and solid."*



*"What are these things you talk about?" asked the fourth mole. "They cannot be the elephant, because the elephant is like a tree, rooted firmly before climbing upward."*



*"I don't understand what they're fighting about," muttered the fifth mole. "The elephant is nothing much more than a limp rope, frayed at the end."*

It can go like this for the student of any topic. We see pieces of the whole, but it can take a long time to walk all the way through the elephant to understand it in its entirety.

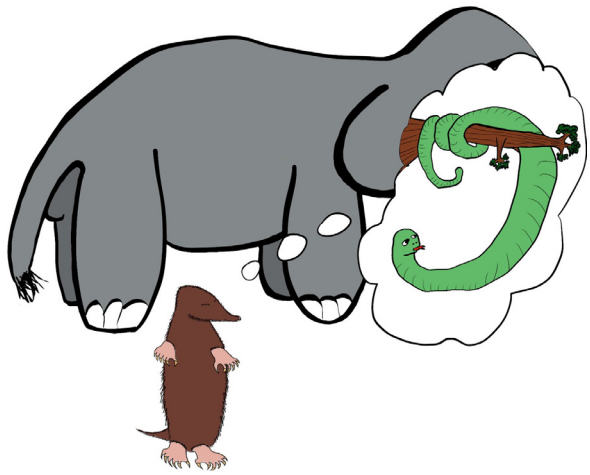
In my book, *Reactive Machine Learning Systems*, I try to walk readers all the way around the topic of building production-grade, real-world, machine learning systems. For all of the detail that I covered, there remain some important topics that I only scratched the surface of. In particular, I wasn't able to fully cover natural language processing or deep learning, beyond some basics.

Thankfully, other books by other authors cover these rich topics in some depth. This collection brings together chapters from great books that focus on working with language and deep learning. Taken as a whole, they'll give you a solid start on using the latest techniques in machine learning to solve the ubiquitous problems of language.

With all of the exciting things going on in the field of machine learning/artificial intelligence, it's worth asking why I chose to focus on language specifically. Language can look like an easy domain. Characters in a text file are much easier to work with than, say, 3D video. But this superficial simplicity gives way to an incredible richness. For all sorts of tasks around language—like speech transcription or translation—we humans don't generally do a perfect job, and building software that has human equivalent performance turns out to be extremely hard for a lot of these problems. You like extremely hard problems, don't you? I thought you did.

Another good reason to work on problems with language is because we use language for many real-world tasks, which makes machine learning systems that work with language useful. In my personal experience, I've built machine learning systems that reasoned about language to do things like trade stocks, schedule meetings, provide customer service, and even get my laundry done. Even if the biggest problem in your life's some smelly socks, tools for modeling language might help you.

# *The Language of Thought*



*"An elephant is like a snake," said the first mole. "Long and sinuous, yet strong."*

Like the moles, we must begin somewhere. In this chapter, we start our exploration of language by focusing on what language is and how we can derive insight from it—using software. It begins with basic techniques that use regular expressions. From there, it allows us to look forward to more sophisticated techniques for working with language data, such as embeddings. It also introduces a first system design for how to build a simple system for working with language for use in a chatbot. Let's grab hold of the trunk of the matter, and begin our exploration of learning from language.

Chapter 1 from *Natural Language Processing in Action* by Hobson Lane, Cole Howard, and Hannes Hapke

# *Packets of thought (NLP overview)*

***This chapter covers***

- What natural language processing (NLP) is
- Why NLP is hard and only recently has become widespread
- When word order and grammar is important and when it can be ignored
- How a chatbot combines many of the tools of NLP
- How to use a regular expression to build the start of a tiny chatbot

You are about to embark on an exciting adventure in natural language processing (NLP). First we show you what NLP is and all the things you can do with it. This will get your wheels turning, helping you think of ways to use NLP in your own life both at work and at home.

Then we dig into the details of exactly how to process a small bit of English text using a programming language like Python, which will help you build up your

NLP toolbox incrementally. In this chapter you'll write your first program that can read and write English statements. This Python snippet will be the first of many you'll use to learn all the tricks needed to assemble an English language dialog engine—a chatbot.

## 1.1 Natural language vs. programming language

Natural languages are different from computer programming languages. They aren't intended to be translated into a finite set of mathematical operations, like programming languages are. Natural languages are what humans use to share information with each other. We don't use programming languages to tell each other about our day or to give directions to the grocery store. A computer program written with a programming language tells a machine exactly what to do. But there are no compilers or interpreters for natural languages such as English and French.

**Important**

**Natural language processing**

Natural language processing is an area of research in computer science and artificial intelligence (AI) concerned with processing natural languages such as English or Mandarin. This processing generally involves translating natural language into data (numbers) that a computer can use to learn about the world. And this understanding of the world is sometimes used to generate natural language text that reflects that understanding.

Nonetheless, this chapter shows you how a machine can *process* natural language. You might even think of this as a natural language interpreter, just like the Python interpreter. When the computer program you develop processes natural language, it will be able to act on those statements or even reply to them. But these actions and replies are not precisely defined, which leaves more discretion up to you, the developer of the natural language pipeline.

**Important**

**Pipeline**

A natural language processing system is often referred to as a "pipeline" because it usually involves several stages of processing where natural language flows in one end and the processed output flows out the other.

You'll soon have the power to write software that does interesting, unpredictable things, like carry on a conversation, which can make machines seem a bit more human. It may seem a bit like magic—at first, all advanced technology does. But we pull back the curtain so you can explore backstage, and you'll soon discover all the props and tools you need to do the magic tricks yourself.

*"Everything is easy, once you know the answer."*

— Dave Magee, Georgia Tech 1995

## 1.2 The magic

What's so magical about a machine that can read and write in a natural language? Machines have been processing languages since computers were invented. However, these "formal" languages—such as early languages Ada, COBOL, and Fortran—were designed to be interpreted (or compiled) only one correct way. Today Wikipedia lists more than 700 programming languages. In contrast, *Ethnologue*[1] has identified ten times as many natural languages spoken by humans around the world. And Google's index of natural language documents is well over 100 million gigabytes.[2] And that's just the index. And it's incomplete. The size of the actual natural language content currently online must exceed 100 billion gigabytes.[3] But this massive amount of natural language text isn't the only reason it's important to build software that can process it.

The interesting thing about the process is that it's hard. Machines with the capability of processing something natural isn't natural. It's kind of like building a building that can do something useful with architectural diagrams. When software can process languages not designed for machines to understand, it seems magical—something we thought was a uniquely human capability.

The word "natural" in "natural language" is used in the same sense that it is used in "natural world". Natural, evolved things in the world about us are different from mechanical, artificial things designed and built by humans. Being able to design and build software that can read and process language like what you're reading here—language about building software that can process natural language… well that's very meta, very magical.

To make your job a little easier, we focus on only one natural language, English. But you can use the techniques you learn in this book to build software that can process any language, even a language you don't understand, or has yet to be deciphered by archaeologists and linguists. And we're going to show you how write software to process and generate that language using only one programming language, Python.

Python was designed from the ground up to be a readable language. It also exposes a lot of its own language processing "guts." Both of these characteristics make it a natural choice for learning natural language processing. It's a great language for building maintainable production pipelines for NLP algorithms in an enterprise environment, with many contributors to a single codebase. We even use Python in lieu of the "universal language" of mathematics and mathematical symbols, wherever possible. After all, Python is an unambiguous way to express mathematical algorithms,[4] and it's designed to be as readable as possible for programmers like you.

---

[1] *Ethnologue* is a web-based publication that maintains statistics about natural languages.

[2] https://www.google.com/search/howsearchworks/crawling-indexing/

[3] You can estimate the amount of actual natural language text out there to be at least 1000 times the size of Google's index.

[4] Mathematical notation is ambiguous: https://en.wikipedia.org/wiki/Ambiguity#Mathematical_notation

### 1.2.1 Machines that converse

Natural languages can't be directly translated into a precise set of mathematical operations, but they do contain information and instructions that can be extracted. Those pieces of information and instruction can be stored, indexed, searched, or immediately acted upon. One of those actions could be to generate a sequence of words in response to a statement. This is the function of the "dialog engine" or chatbot that you'll build.

We focus entirely on English text documents and messages, not spoken statements. We bypass the conversion of spoken statements into text—speech recognition, or speech to text (STT). We also ignore speech generation or text to speech, converting text back into some human-sounding voice utterance. But you can still use what you learn to build a voice interface or virtual assistant like Siri or Alexa, because speech-to-text and text-to-speech libraries are freely available. Android and iOS mobile operating systems provide high quality speech recognition and generation APIs, and there are Python packages to accomplish similar functionality on a laptop or server.

> **Speech recognition systems**
>
> If you want to build a customized speech recognition or generation system, that undertaking is a whole book in itself; we leave that as an "exercise for the reader." It requires a lot of high quality labeled data, voice recordings annotated with their phonetic spellings, and natural language transcriptions aligned with the audio files. Some of the algorithms you learn in this book might help, but most of the algorithms are quite different.

### 1.2.2 The math

Processing natural language to extract useful information can be difficult. It requires tedious statistical bookkeeping, but that's what machines are for. And like many other technical problems, solving it is a lot easier once you know the answer. Machines still cannot perform most practical NLP tasks, such as conversation and reading comprehension, as accurately and reliably as humans. So you might be able to tweak the algorithms you learn in this book to do some NLP tasks a bit better.

The techniques you'll learn, however, are powerful enough to create machines that can surpass humans in both accuracy and speed for some surprisingly subtle tasks. For example, you might not have guessed that recognizing sarcasm in an isolated Twitter message can be done more accurately by a machine than by a human.[5] Don't worry, humans are still better at recognizing humor and sarcasm within an ongoing dialog, due to our ability to maintain information about the context of a

[5] Gonzalo-Ibanez et al found that educated and trained human judges could not match the performance of their simple classification algorithm of 68% reported in their ACM paper. The Sarcasm Detector (https://github.com/MathieuCliche/Sarcasm_detector) and web app (http://www.thesarcasmdetector.com/) by Matthew Cliche at Cornell achieves similar accuracy (>70%).

statement. But machines are getting better and better at maintaining context. And this book helps you incorporate context (metadata) into your NLP pipeline if you want to try your hand at advancing the state of the art.

Once you extract structured numerical data, vectors, from natural language, you can take advantage of all the tools of mathematics and machine learning. We use the same linear algebra tricks as the projection of 3D objects onto a 2D computer screen, something that computers and drafters were doing long before natural language processing came into its own. These breakthrough ideas opened up a world of "semantic" analysis, allowing computers to interpret and store the "meaning" of statements rather than just word or character counts. Semantic analysis, along with statistics, can help resolve the ambiguity of natural language—the fact that words or phrases often have multiple meanings or interpretations.

So extracting information isn't at all like building a programming language compiler (fortunately for you). The most promising techniques bypass the rigid rules of regular grammars (patterns) or formal languages. You can rely on statistical relationships between words instead of a deep system of logical rules.[6] Imagine if you had to define English grammar and spelling rules in a nested tree of if…then statements. Could you ever write enough rules to deal with every possible way that words, letters, and punctuation can be combined to make a statement? Would you even begin to capture the semantics, the meaning of English statements? Even if it were useful for some kinds of statements, imagine how limited and brittle this software would be. Unanticipated spelling or punctuation would break or befuddle your algorithm.

Natural languages have an additional "decoding" challenge that is even harder to solve. Speakers and writers of natural languages assume that a human is the one doing the processing (listening or reading), not a machine. So when I say "good morning", I assume that you have some knowledge about what makes up a morning, including not only that mornings come before noons and afternoons and evenings but also after midnights. And you need to know they can represent times of day as well as general experiences of a period of time. The interpreter is assumed to know that "good morning" is a common greeting that doesn't contain much information at all about the morning. Rather it reflects the state of mind of the speaker and her readiness to speak with others.

This theory of mind about the human processor of language turns out to be a powerful assumption. It allows us to say a lot with few words if we assume that the "processor" has access to a lifetime of common sense knowledge about the world. This degree of compression is still out of reach for machines. There is no clear "theory of mind" you can point to in an NLP pipeline. However, we show you techniques in later chap-

[6] Some grammar rules can be implemented in a computer science abstraction called a finite state machine. Regular grammars can be implemented in regular expressions. There are two Python packages for running regular expression finite state machines, `re` which is built in, and `regex` which must be installed, but may soon replace `re`. Finite state machines are just trees of if… then…else statements for each token (character/word/n-gram) or action that a machine needs to react to or generate.

ters to help machines build ontologies, or knowledge bases, of common sense knowledge to help interpret statements that rely on this knowledge.

## 1.3 Practical applications

Natural language processing is everywhere. It's so ubiquitous that some of the examples in table 1.1 may surprise you.

**Table 1.1 Categorized NLP applications**

| | | | |
|---|---|---|---|
| **Search** | Web | Documents | Autocomplete |
| **Editing** | Spelling | Grammar | Style |
| **Dialog** | Chatbot | Assistant | Scheduling |
| **Writing** | Index | Concordance | Table of contents |
| **Email** | Spam filter | Classification | Prioritization |
| **Text mining** | Summarization | Knowledge extraction | Medical diagnoses |
| **Law** | Legal inference | Precedent search | Subpoena classification |
| **News** | Event detection | Fact checking | Headline composition |
| **Attribution** | Plagiarism detection | Literary forensics | Style coaching |
| **Sentiment analysis** | Community morale monitoring | Product review triage | Customer care |
| **Behavior prediction** | Finance | Election forecasting | Marketing |
| **Creative writing** | Movie scripts | Poetry | Song lyrics |

A search engine can provide more meaningful results if it indexes web pages or document archives in a way that takes into account the meaning of natural language text. Autocomplete uses NLP to complete your thought and is common among search engines and mobile phone keyboards. Many word processors, browser plugins, and text editors have spelling correctors, grammar checkers, concordance composers, and most recently, style coaches. Some dialog engines (chatbots) use natural language search to find a response to their conversation partner's message.

NLP pipelines that generate (compose) text can be used not only to compose short replies in chatbots and virtual assistants, but also to assemble much longer passages of text. The Associated Press uses NLP "robot journalists" to write entire financial news articles and sporting event reports.[7] Bots can compose weather forecasts that sound a lot like what your hometown weather person might say, perhaps because human meteorologists use word processors with NLP features to draft scripts.

[7] "AP's 'robot journalists' are writing their own stories now", The Verge, Jan 29, 2015, http://www.theverge.com/2015/1/29/7939067/ap-journalism-automation-robots-financial-reporting

NLP spam filters in early email programs helped email overtake telephone and fax communication channels in the '90s. And the spam filters have retained their edge in the cat and mouse game between spam filters and spam generators for email, but may be losing in other environments like social networks. An estimated 20% of the tweets about the 2016

US presidential election were composed by chatbots.[8] These bots amplify their owners' and developers' viewpoints with the resources and motivation to influence popular opinion. And these "puppet masters" tend to be foreign governments or large corporations.

NLP systems can generate more than just short social network posts. NLP can be used to compose lengthy movie and product reviews on Amazon and elsewhere. Many reviews are the creation of autonomous NLP pipelines that have never set foot in a movie theater or purchased the product they are reviewing.

There are chatbots on Slack, IRC, and even customer service websites—places where chatbots have to deal with ambiguous commands or questions. And chatbots paired with voice recognition and generation systems can even handle lengthy conversations with an indefinite goal or "objective function" such as making a reservation at a local restaurant.[9] NLP systems can answer phones for companies that want something better than a phone tree but don't want to pay humans to help their customers.

**Note**

With its **Duplex** demonstration at Google IO, engineers and managers overlooked concerns about the ethics of teaching chatbots to deceive humans. We all ignore this dilemma when we happily interact with chatbots on Twitter and other anonymous social networks, where bots do not share their pedigree. With bots that can so convincingly deceive us, the AI control problem[a] looms, and Yuval Harari's cautionary forecast of "Homo Deus"[b] may come sooner than we think.

a) https://en.wikipedia.org/wiki/AI_control_problem
b) WSJ Blog, March10, 2017 https://blogs.wsj.com/cio/2017/03/10/homo-deus-author-yuval-noah-harari-says-authority-shifting-from-people-to-ai/

NLP systems exist that can act as email "receptionists" for businesses or executive assistants for managers. These assistants schedule meetings and record summary details in an electronic Rolodex, or CRM (customer relationship management system), interacting with others by email on their boss's behalf. Companies are putting their brand and face in the hands of NLP systems, allowing bots to execute marketing and messaging campaigns. And some inexperienced daredevil NLP textbook authors are letting bots author several sentences in their book. More on that later.

---

[8] New York Times, Oct 18, 2016, https://www.nytimes.com/2016/11/18/technology/automated-pro-trump-bots-overwhelmed-pro-clinton-messages-researchers-say.html and MIT Technology Review, Nov 2016, https://www.technologyreview.com/s/602817/how-the-bot-y-politic-influenced-this-election/

[9] Google Blog May 2018 about their Duplex system https://ai.googleblog.com/2018/05/advances-in-semantic-textual-similarity.html

## 1.4 Language through a computer's "eyes"

When you type "Good Morn'n Rosa", a computer sees only "01000111 01101111 01101111…". How can you program a chatbot to respond to this binary stream intelligently? Could a nested tree of conditionals (`if… else…`" statements) check each one of those bits and act on them individually? This would be equivalent to writing a special kind of program called a finite state machine (FSM). An FSM that outputs a sequence of new symbols as it runs, like the Python `str.translate` function, is called a finite state transducer (FST). You've probably already built a FSM without even knowing it. Have you ever written a regular expression? That's the kind of FSM we use in the next section to show you one possible approach to NLP: the pattern-based approach.

What if you decided to search a memory bank (database) for the exact same string of bits, characters, or words, and use one of the responses that other humans and authors have used for that statement in the past? But imagine if there was a typo or variation in the statement. Our bot would be sent off the rails. And bits aren't continuous or forgiving—they either match or they don't. There's no obvious way to find similarity between two streams of bits that takes into account what they signify. The bits for "good" will be just as similar to "bad!" as they are to "okay".

But let's see how this approach would work before we show you a better way. Let's build a small regular expression to recognize greetings like "Good morning Rosa" and respond appropriately—our first tiny chatbot!

### 1.4.1 The language of locks

Surprisingly the humble combination lock is actually a simple language processing machine. So, if you're mechanically inclined, this section may be illuminating. But if you don't need mechanical analogies to help you understand algorithms and how regular expressions work, then you can skip this section.

After finishing this section, you'll never think of your combination bicycle lock the same way again. A combination lock certainly can't read and understand the textbooks stored inside a school locker, but it can understand the language of locks. It can understand when you try to "tell" it a "password": a combination. A padlock combination is any sequence of symbols that matches the "grammar" (pattern) of lock language. Even more importantly, the padlock can tell if a lock "statement" matches a particularly meaningful statement, the one for which there is only one correct "response," to release the catch holding the U-shaped hasp so you can get into your locker.

This lock language (regular expressions) is a particularly simple one. But it's not so simple that we can't use it in a chatbot. We can use it to recognize a key phrase or command to unlock a particular action or behavior.

For example, we'd like our chatbot to recognize greetings such as "Hello Rosa," and respond to them appropriately. This kind of language, like the language of locks, is a formal language because it has strict rules about how an acceptable statement must be composed and interpreted. If you've ever written a math equation or coded a programming language expression, you've written a formal language statement.

Formal languages are a subset of natural languages. Many natural language statements can be matched or generated using a formal language grammar, like regular expressions. That's the reason for this diversion into the mechanical, "click, whirr"[10] language of locks.

### 1.4.2 Regular expressions

Regular expressions use a special kind (class) of formal language grammar called a regular grammar. Regular grammars have predictable, provable behavior, and yet are flexible enough to power some of the most sophisticated dialog engines and chatbots on the market. Amazon Alexa and Google Now are mostly pattern-based engines that rely on regular grammars. Deep, complex regular grammar rules can often be expressed in a single line of code called a regular expression. There are successful chatbot frameworks in Python, like `Will`, that rely exclusively on this kind of language to produce some useful and interesting behavior. Amazon Echo, Google Home, and similarly complex and useful assistants use this kind of language to encode the logic for most of their user interaction.

**Note**

Regular expressions implemented in Python and in Posix (Unix) applications such as `grep` are not true regular grammars. They have language and logic features such as look-ahead and look-back that make leaps of logic and recursion that aren't allowed in a regular grammar. As a result, regular expressions aren't provably halting; they can sometimes "crash" or run forever.[a]

a) Stack Exchange Went Down for 30 minutes on July 20, 2016 when a regex "crashed"

You may be saying to yourself, "I've heard of regular expressions. I use `grep`. But that's only for search!" And you're right. Regular Expressions are indeed used mostly for search, for sequence matching. But anything that can find matches within text is also great for carrying out a dialog. Some chatbots, like `Will`, use "search" to find sequences of characters within a user statement that they know how to respond to. These recognized sequences then trigger a scripted response appropriate to that particular regular expression match. And that same regular expression can also be used to extract a useful piece of information from a statement. A chatbot can add that bit of information to its knowledge base about the user or about the world the user is describing.

A machine that processes this kind of language can be thought of as a formal mathematical object called a finite state machine or deterministic finite automaton (DFA). FSMs come up again and again in this book. So you'll eventually get a good feel for what they're used for without digging into FSM theory and math. For those

[10] One of Cialdini's six psychology principles in his popular book *Influence* http://changingminds.org/techniques/general/cialdini/click-whirr.htm

who can't resist trying to understand a bit more about these computer science tools, figure 1.1 shows where FSMs fit into the nested world of automata (bots). And the side note that follows explains a bit more formal detail about formal languages.

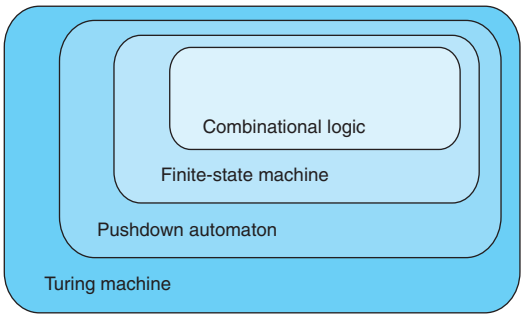



**Figure 1.1 Kinds of automata**

### Formal mathematical explanation of formal languages

Kyle Gorman describes programming languages this way:

- Most (if not all) programming languages are drawn from the class of context-free languages.
- Context free languages are parsed with context-free grammars, which provide efficient parsing.
- The regular languages are also efficiently parsable and used extensively in computing for string matching.
- String matching applications rarely require the expressiveness of context-free.
- There are a number of formal language classes, a few of which are shown here (in decreasing complexity):[a]
  - Recursively enumerable
  - Context-sensitive
  - Context-free
  - Regular

Natural languages are:

- Not regular[b]
- Not context-free[c]
- Can't be defined by any formal grammar[d]

a) https://en.wikipedia.org/wiki/Chomsky_hierarchy
b) http://cs.haifa.ac.il/~shuly/teaching/08/nlp/complexity.pdf#page=20
c) http://cs.haifa.ac.il/~shuly/teaching/08/nlp/complexity.pdf#page=24
d) http://interactivepython.org/runestone/static/CS152f17/GeneralIntro/FormalandNaturalLanguages.html

### 1.4.3 A simple chatbot

Let's build a quick and dirty chatbot. It won't be very capable, and it will require a lot of thinking about the English language. You will also have to hardcode regular expressions to match the ways people may try to say something. But don't worry if you think you couldn't have come up with this Python code yourself. You won't have to try to think of all the different ways people can say something, like we did in this example. You won't even have to write regular expressions (regexes) to build an awesome chatbot. We show you how to build a chatbot of your own in later chapters without hardcoding anything. A modern chatbot can learn from reading (processing) a bunch of English text. And we show you how to do that in later chapters.

This pattern matching chatbot is an example of a tightly controlled chatbot. Pattern matching chatbots were common before modern machine learning chatbot techniques were developed. And a variation of the pattern matching approach we show you here is used in chatbots like Amazon Alexa and other virtual assistants.

For now let's build a FSM, a regular expression, that can speak lock language (regular language). We could program it to understand lock language statements, such as "01-02-03." Even better, we'd like it to understand greetings, things like "open sesame" or "hello Rosa." An important feature for a prosocial chatbot is to be able to respond to a greeting. In high school, teachers often chastised me for being impolite when I'd ignore greetings like this while rushing to class. We surely don't want that for our benevolent chatbot.

In machine communication protocol, we'd define a simple handshake with an ACK (acknowledgement) signal after each message passed back and forth between two machines. But our machines are going to be interacting with humans who say things like "Good morning, Rosa". We don't want it sending out of bunch of chirps, beeps, or ACK messages, like it's syncing up a modem or HTTP connection at the start of a conversation or web browsing session. Instead let's use regular expressions to recognize several different human greetings at the start of a conversation handshake.

**There are two "official" regular expression packages in Python. We use the re package here just because it is installed with all versions of Python. The regex package comes with later versions of Python and is much more powerful, as you'll see in chapter 2.**

```
>>> import re
>>> r = "(hi|hello|hey)[ ]*([a-z]*)"
>>> re.match(r, 'Hello Rosa', flags=re.IGNORECASE)
<_sre.SRE_Match object; span=(0, 10), match='Hello Rosa'>
>>> re.match(r, "hi ho, hi ho, it's off to work ...", flags=re.IGNORECASE)
<_sre.SRE_Match object; span=(0, 5), match='hi ho'>
>>> re.match(r, "hey, what's up", flags=re.IGNORECASE)
<_sre.SRE_Match object; span=(0, 3), match='hey>
```

**Ignoring the case of text characters is common to keep the regular expressions simpler.**

**| means "OR", and * means the preceding character can occur 0 or more times and still match. So our regex will match greetings that start with "hi" or "hello" or "hey" followed by any number of <space> characters and then any number of letters.**

In regular expressions, you can specify a character class with square brackets. And you can use a dash (-) to indicate a range of characters without having to type them all out individually. So the regular expression `"[a-z]"` will match any single lowercase letter, "a" through "z". The star (*) after a character class means that the regular expression will match any number of consecutive characters if they are all within that character class.

Let's make our regular expression a lot more detailed to try to match more greetings.

**You can compile regular expressions so you don't have to specify the options (`flags`) each time you use it.**

**Notice that this regular expression cannot recognize (match) words with typos.**

```
>>> r = r"[^a-z]*([y]o|[h']?ello|ok|hey|(good[
])?(morn[gin']{0,3}|afternoon|even[gin']{0,3}))[\s,;:]{1,3}([a-z]{1,20})"
>>> re_greeting = re.compile(r, flags=re.IGNORECASE)
>>> re_greeting.match('Hello Rosa')
<_sre.SRE_Match object; span=(0, 10), match='Hello Rosa'>
>>> re_greeting.match('Hello Rosa').groups() ('Hello', None, None, 'Rosa')
>>> re_greeting.match("Good morning Rosa")
<_sre.SRE_Match object; span=(0, 17), match="Good morning Rosa">
>>> re_greeting.match("Good Manning Rosa")
>>> re_greeting.match('Good evening Rosa Parks').groups()
('Good evening', 'Good ', 'evening', 'Rosa')
>>> re_greeting.match("Good Morn'n Rosa")
<_sre.SRE_Match object; span=(0, 16), match="Good Morn'n Rosa">
>>> re_greeting.match("yo Rosa")
<_sre.SRE_Match object; span=(0, 7), match='yo Rosa'>
```

**Our chatbot can separate different parts of the greeting into `groups`, but it will be unaware of Rosa's famous last name, because we don't have a pattern to match any characters after the first name.**

**Tip**

The `"r"` before the quote specifies a raw string, not a regular expression. With a Python raw string, you can send backslashes directly to the regular expression compiler without having to double-backslash (`"\\"`) all the special regular expression characters such as spaces (`"\\ "`) and curly braces or handlebars (`"\\{\\}"`).

There's a lot of logic packed into that first line of code, the regular expression. It gets the job done for a surprising range of greetings. But it missed that "Manning" typo, which is one of the reasons NLP is hard. In machine learning and medical diagnostic testing, that's called a false negative classification error. Unfortunately, it will also match some statements that humans would be unlikely to ever say—a false positive, which is also a bad thing. Having both false positive and false negative errors means that our regular expression is both too liberal and too strict. These mistakes could make our bot sound a bit dull and mechanical. We'd have to do a lot more work to refine the phrases that it matches to be more human-like.

And this tedious work would be highly unlikely to ever succeed at capturing all the slang and misspellings people use. Fortunately, composing regular expressions by

hand isn't the only way to train a chatbot. Stay tuned for more on that later (the entire rest of the book). So we only use them when we need precise control over a chatbot's behavior, such as when issuing commands to a voice assistant on your mobile phone.

But let's go ahead and finish up our one-trick chatbot by adding an output generator. It needs to say something. We use Python's string formatter to create a "template" for our chatbot response.

```
>>> my_names = set(['rosa', 'rose', 'chatty', 'chatbot', 'bot',
     'chatterbot'])
>>> curt_names = set(['hal', 'you', 'u'])
>>> greeter_name = ''
...
>>> match = re_greeting.match(input())
...
>>> if match:
...     at_name = match.groups()[-1]
...     if at_name in curt_names:
...         print("Good one.")
...     elif at_name.lower() in my_names:
...         print("Hi {}, How are you?".format(greeter_name)
```

**We don't yet know who is chatting with the bot, and we won't worry about it here.**

So if you run this little script and chat to our bot with a phrase like "Hello Rosa", it will respond by asking about your day. If you use a slightly rude name to address the chatbot, she will be less responsive, but not inflammatory, to try to encourage politeness.[11] If you name someone else who might be monitoring the conversation on a party line or forum, the bot will keep quiet and allow you and whomever you are addressing to chat. Obviously there's no one else out there watching our `input()` line, but if this were a function within a larger chatbot, you want to deal with these sorts of things.

Because of the limitations of computational resources, early NLP researchers had to use their human brain's computational power to design and hand-tune complex logical rules to extract information from a natural language string. This is called a pattern-based approach to NLP. The patterns don't have to be merely character sequence patterns, like our regular expression. NLP also often involves patterns of word sequences, or parts of speech, or other "higher level" patterns. The core NLP building blocks like stemmers and tokenizers as well as sophisticated end-to-end NLP dialog engines (chatbots) like ELIZA were built this way, from regular expressions and pattern matching. The art of pattern-matching approaches to NLP is coming up with elegant patterns that capture just what you want, without too many lines of regular expression code.

[11] The idea for this defusing response originated with Viktor Frankl's Man's Search for Meaning, his Logotherapy approach to psychology and the many popular novels where a child protagonist like Owen Meany has the wisdom to respond to an insult with a response like this.

**Tip**

**Classical computational theory of mind**

This classical NLP pattern-matching approach is based on the computational theory of mind (CTM). CTM assumes that human-like NLP can be accomplished with a finite set of logical rules that are processed in series.[a] Advancements in neuroscience and NLP led to the development of a "connectionist" theory of mind around the turn of the century, which allows for parallel pipelines processing natural language simultaneously, as is done in artificial neural networks.[b] [c]

a) Stanford Encyclopedia of Philosophy, Computational Theory of Mind, https://plato.stanford.edu/entries/computational-mind/
b) Stanford Encyclopedia of Philosophy, Connectionism, https://plato.stanford.edu/entries/connectionism/]
c) Christiansen and Chater, 1999, Southern Illinois University, https://crl.ucsd.edu/~elman/Bulgaria/christiansen-chater-soa.pdf

You'll learn more about pattern-based approaches—such as the Porter stemmer or the Treebank tokenizer—to tokenizing and stemming in chapter 2. But in later chapters we take advantage of the exponentially greater computational resources, as well as our larger datasets, to shortcut this laborious hand programming and refining.

If you're new to regular expressions and want to learn more, you can check out appendix B or the online documentation for Python regular expressions. But you don't have to understand them just yet. We'll continue to provide you with example regular expressions as we use them for the building blocks of our NLP pipeline. So don't worry if they look like gibberish. Human brains are pretty good at generalizing from a set of examples, and I'm sure it will become clear by the end of this book. And it turns out machines can learn this way as well…

### 1.4.4 Another way

Is there a statistical or machine learning approach that might work in place of the pattern- based approach? If we had enough data could we do something different? What if we had a giant database containing sessions of dialog between humans, statements and responses for thousands or even millions of conversations? One way to build a chatbot would be to search that database for the exact same string of characters our chatbot user just "said" to our chatbot. Couldn't we then use one of the responses to that statement that other humans have said in the past?

But imagine how a single typo or variation in the statement would trip up our bot. Bit and character sequences are discrete. They either match or they don't. Instead, we'd like our bot to be able to measure the difference in **meaning** between character sequences.

When we use character sequence matches to measure distance between natural language phrases, we'll often get it wrong. Phrases with similar meaning, like "good" and "okay", can often have different character sequences and large distances when we count up character-by-character matches to measure distance. And sequences with completely different meanings, like "bad" and "bar", might be too close to one other

when we use metrics designed to measure distances between numerical sequences. Metrics like Jaccard, Levenshtein, and Euclidean vector distance can sometimes add enough "fuzziness" to prevent a chatbot from stumbling over minor spelling errors or typos. But these metrics fail to capture the essence of the relationship between two strings of characters when they are dissimilar. And they also sometimes bring small spelling differences close together that might not really be typos, like "bad" and "bar".

Distance metrics designed for numerical sequences and vectors are useful for a few NLP applications, like spelling correctors and recognizing proper nouns. So we use these distance metrics when they make sense. But for NLP applications where we are more interested in the meaning of the natural language than its spelling, there are better approaches. We use vector representations of natural language words and text and some distance metrics for those vectors for those NLP applications. We show you each approach, one by one, as we talk about these different applications and the kinds of vectors they are used with.

We don't stay in this confusing binary world of logic for long, but let's imagine we're famous World War II-era code-breaker Mavis Batey at Bletchley Park and we've just been handed that binary, Morse code message intercepted from communication between two German military officers. It could hold the key to winning the war. Where would we start? Well the first layer of deciding would be to do something statistical with that stream of bits to see if we can find patterns. We can first use the Morse code table (or ASCII table, in our case) to assign letters to each group of bits. Then, if the characters are gibberish to us, as they are to a computer or a cryptographer in WWII, we could start counting them up, looking up the short sequences in a dictionary of all the words we've seen before and putting a mark next to the entry every time it occurs. We might also make a mark in some other log book to indicate which message the word occurred in, creating an encyclopedic index to all the documents we've read before. This collection of documents is called a *corpus*, and the words or sequences we've listed in our index are called a *lexicon*.

If we're lucky, and we're not at war, and the messages we're looking at aren't strongly encrypted, we'll see patterns in those German word counts that mirror counts of English words used to communicate similar kinds of messages. Unlike a cryptographer trying to decipher German Morse code intercepts, we know that the symbols have consistent meaning and aren't changed with every key click to try to confuse us. This tedious counting of characters and words is just the sort of thing a computer can do without thinking. And surprisingly, it's nearly enough to make the machine appear to understand our language. It can even do math on these statistical vectors that coincides with our human understanding of those phrases and words. When we show you how to teach a machine our language using Word2Vec in later chapters, it may seem magical, but it's not. It's just math, computation.

But let's think for a moment about what information has been lost in our effort to count all the words in the messages we receive. We assign the words to bins and store them away as bit vectors like a coin or token sorter (see figure 1.2) directing different

kinds of tokens to one side or the other in a cascade of decisions that piles them in bins at the bottom. Our sorting machine must take into account hundreds of thousands if not millions of possible token "denominations," one for each possible word that a speaker or author might use. Each phrase or sentence or document we feed into our token sorting machine will come out the bottom, where we have a "vector" with a count of the tokens in each slot. Most of our counts are zero, even for large documents with verbose vocabulary. But we haven't lost any words yet. What have we lost? Could you, as a human understand a document that we presented you in this way, as a count of each possible word in your language, without any sequence or order associated with those words? I doubt it. But if it was a short sentence or tweet, you'd probably be able to rearrange them into their intended order and meaning most of the time.

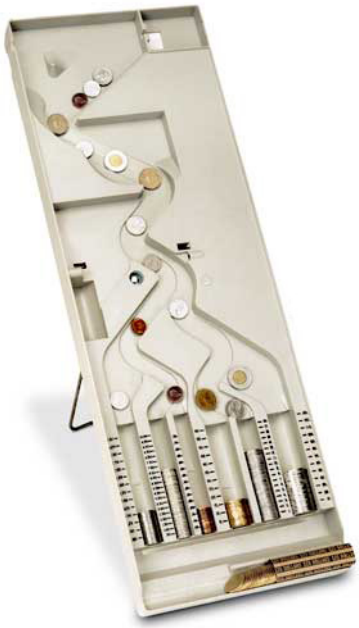


**Figure 1.2 Canadian coin sorter**

Here's how our token sorter fits into an NLP pipeline right after a tokenizer (see chapter 2). We've included a stopword filter as well as a "rare" word filter in our mechanical token sorter sketch. Strings flow in from the top, and bag-of-word vectors are created from the height profile of the token "stacks" at the bottom.

It turns out that machines can handle this bag of words quite well and glean most of the information content of even moderately long documents this way. Each document, after token sorting and counting, can be represented as a vector, a sequence of integers for each word or token in that document. You see a crude example in figure 1.3, and then chapter 2 shows some more useful data structures for bag-of-word vectors.

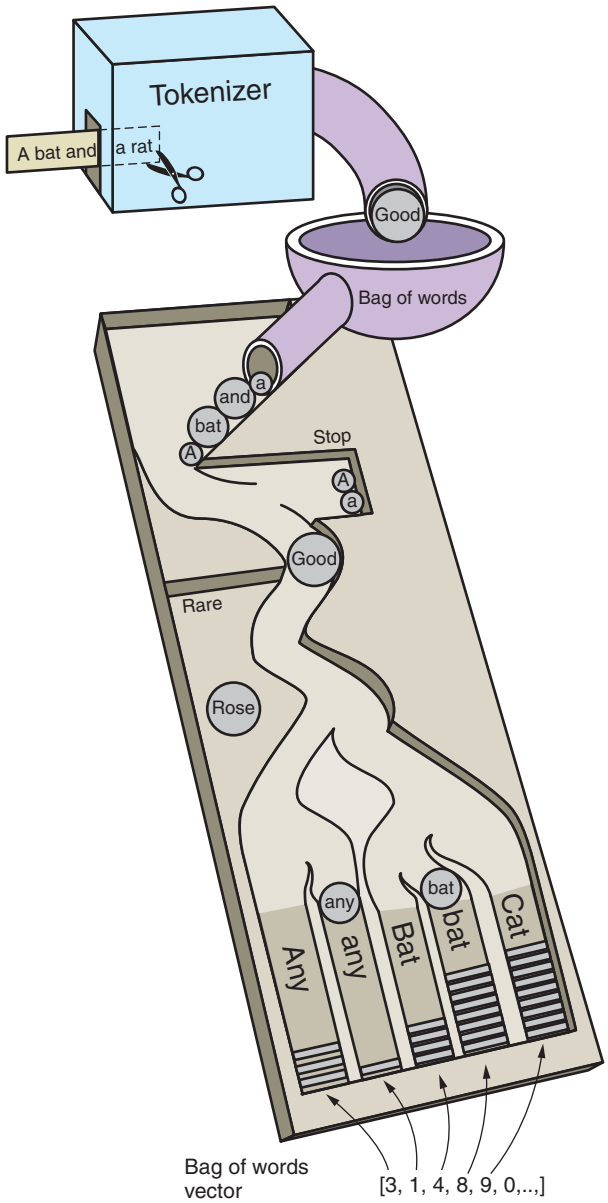



**Figure 1.3 Token sorting tray**

This is our first vector space model of a language. Those bins and the numbers they contain for each word are represented as long vectors containing a lot of zeros and a few ones or twos scattered around wherever the word for that bin occurred. All the different ways that words could be combined to create these vectors is called a *vector space*. And relationships between vectors in this space are what make up our model, which is attempting to predict combinations of these words occurring within a collec-

tion of various sequences of words (typically sentences or documents). In Python, we can represent these sparse (mostly empty) vectors (lists of numbers) as dictionaries. And a Python `Counter` is a special kind of dictionary that bins objects (including strings) and counts them just like we want.

```
>>> from collections import Counter

>>> Counter("Guten Morgen Rosa".split())
Counter({'Guten': 1, 'Rosa': 1, 'morgen': 1})
>>> Counter("Good morning, Rosa!".split())
Counter({'Good': 1, 'Rosa!': 1, 'morning,': 1})
```

You can probably imagine some ways to clean those tokens up. We do just that in the next chapter. But you might also think to yourself that these sparse, high-dimensional vectors (many bins, one for each possible word) aren't very useful for language processing. But they are good enough for some industry-changing tools like spam filters, which we discuss in chapter 3.

And we can imagine feeding into this machine, one at a time, all the documents, statements, sentences, and even single words we could find. We'd count up the tokens in each slot at the bottom after each of these statements was processed, and we'd call that a vector representation of that statement. All the possible vectors a machine might create this way is called a *vector space*. And this model of documents and statements and words is called a *vector space model*. It allows us to use linear algebra to manipulate these vectors and compute things like distances and statistics about natural language statements, which helps us solve a much wider range of problems with less human programming and less brittleness in the NLP pipeline. One statistical question that is asked of bag-of-words vector sequences is "What is the combination of words most likely to follow a particular bag of words." Or, even better, if a user enters a sequence of words, "What is the closest bag of words in our database to a bag-of-words vector provided by the user?" This is a search query. The input words are the words you might type into a search box, and the closest bag-of-words vector corresponds to the document or web page you were looking for. The ability to efficiently answer these two questions would be sufficient to build a machine learning chatbot that could get better and better as we gave it more and more data.

But wait a minute, perhaps these vectors aren't like any you've ever worked with before. They're extremely high-dimensional. It's possible to have millions of dimensions for a 3- gram vocabulary computed from a large corpus. In chapter 3, we discuss the curse of dimensionality and some other properties that make high dimensional vectors difficult to work with.

## 1.5 A brief overflight of hyperspace

In chapter 3, we show you how to consolidate words into a smaller number of vector dimensions to help mitigate the curse of dimensionality and maybe turn it to our advantage. When we project these vectors onto each other to determine the distance between pairs of vectors, this will be a reasonable estimate of the similarity in their

**meaning** rather than merely their statistical word usage. This vector distance metric is called *cosine distance metric*, which we talk about in chapter 3 and then reveal its true power on reduced dimension topic vectors in chapter 4. We can even project ("embed" is the more precise term) these vectors in a 2D plane to have a "look" at them in plots and diagrams to see if our human brains can find patterns. We can then teach a computer to recognize and act on these patterns in ways that reflect the underlying meaning of the words that produced those vectors.

Imagine all the possible tweets or messages or sentences that humans might write. Even though we do repeat ourselves a lot, that's still a lot of possibilities. And when those tokens are each treated as separate, distinct dimensions, there's no concept that "Good morning, Hobs" has some shared meaning with "Guten Morgen, Hannes." We need to create some reduced dimension vector space model of messages so we can label them with a set of continuous (float) values. We could rate messages and words for qualities like subject matter and sentiment. We could ask questions like:

- How likely is this message to be a question?
- How much is it about a person?
- How much is it about me?
- How angry or happy does it sound?
- Is it something I need to respond to?

Think of all the ratings we could give statements. We could put these ratings in order and "compute" them for each statement to compile a "vector" for each statement. The list of ratings or dimensions we could give a set of statements should be much smaller than the number of possible statements, and statements that mean the same thing should have similar values for all our questions.

These rating vectors become something that a machine can be programmed to react to. We can simplify and generalize vectors further by clumping (clustering) statements together, making them close on some dimensions and not on others.

But how can a computer assign values to each of these vector dimensions? Well, if we simplified our vector dimension questions to things like "does it contain the word 'good'?" Does it contain the word "morning?" And so on. You can see that we might be able to come up with a million or so questions resulting in numerical value assignments that a computer could make to a phrase. This is the first practical vector space model, called a bit vector language model, or the sum of "one-hot encoded" vectors. You can see why computers are just now getting powerful enough to make sense of natural language. The millions of million-dimensional vectors that humans might generate simply "Does not compute!" on a supercomputer of the 80s, but is no problem on a commodity laptop in the 21st century. More than just raw hardware power and capacity made NLP practical; incremental, constant-RAM, linear algebra algorithms were the final piece of the puzzle that allowed machines to crack the code of natural language.

There's an even simpler, but much larger representation that can be used in a chatbot. What if our vector dimensions completely described the exact sequence of charac-

ters. It would contain the answer to questions like, "Is the first letter an A? Is it a B? … Is the second letter an A?" and so on. This vector has the advantage that it retains all the information contained in the original text, including the order of the characters and words. Imagine a player piano that could only play a single note at a time, and it had 52 or more possible notes it could play. The "notes" for this natural language mechanical player piano are the 26 uppercase and lowercase letters plus any punctuation that the piano must know how to "play." The paper roll wouldn't have to be much wider than for a real player piano and the number of notes in some long piano songs doesn't exceed the number of characters in a small document. But this one-hot character sequence encoding representation is mainly useful for recording and then replaying an exact piece rather than composing something new or extracting the essence of a piece. We can't easily compare the piano paper roll for one song to that of another. And this representation is longer than the original ASCII-encoded representation of the document. The number of possible document representations just exploded in order to retain information about each sequence of characters. We retained the order of characters and words but expanded the dimensionality of our NLP problem.

These representations of documents don't cluster together well in this character-based vector world. The Russian mathematician Vladimir Levenshtein came up with a brilliant approach for quickly finding similarities between vectors (strings of characters) in this world. Levenshtein's algorithm made it possible to create some surprisingly fun and useful chatbots, with only this simplistic, mechanical view of language. But the real magic happened when we figured out how to compress/embed these higher dimensional spaces into a lower dimensional space of fuzzy meaning or topic vectors. We peek behind the magician's curtain in chapter 4 when we talk about latent semantic indexing and latent Dirichlet allocation, two techniques for creating much more dense and meaningful vector representations of statements and documents.

## 1.6 Word order and grammar

The order of words matters. Those rules that govern word order in a sequence of words (like a sentence) are called the grammar of a language. That's something that our bag of words or word vector discarded in the earlier examples. Fortunately, in most short phrases and even many complete sentences, this word vector approximation works OK. If you just want to encode the general sense and sentiment of a short sentence, word order is not terribly important. Take a look at all these orderings of our "Good morning Rosa" example.

```
>>> from itertools import permutations

>>> [" ".join(combo) for combo in permutations("Good morning Rosa!".split(),
     3)] ['Good morning Rosa!',
'Good Rosa! morning',
'morning Good Rosa!',
'morning Rosa! Good',
'Rosa! Good morning',
'Rosa! morning Good']
```

Now if you tried to interpret each of those strings in isolation (without looking at the others), you'd probably conclude that they all probably had similar intent or meaning. You might even notice the capitalization of the word "Good" and place the word at the front of the phrase in your mind. But you might also think that "Good Rosa" was some sort of proper noun, like the name of a restaurant or flower shop. Nonetheless, a smart chatbot or clever woman of the 1940s in Bletchley Park would likely respond to any of these six permutations with the same innocuous greeting, "Good morning my dear General."

Let's try that (in our heads) on a much longer, more complex phrase, a logical statement where the order of the words matters a lot:

```
>>> s = "Find textbooks with titles containing 'NLP', or 'natural' and
'language', or 'computational' and  'linguistics'."
>>> len(set(s.split()))
12
>>> import numpy as np
>>> np.arange(1, 12 + 1).prod() # factorial(12) = arange(1, 13).prod()
479001600
```

The number of permutations exploded from `factorial(3) == 6` in our simple greeting to `factorial(12) == 479001600` in our longer statement! And it's clear that the logic contained in the order of the words is important to any machine that would like to reply with the correct response. Even though common greetings are not usually garbled by bag-of-words processing, more complex statements can lose most of their meaning when thrown into a bag. A bag of words is not the best way to begin processing a database query, like the natural language query in the preceding example.

Whether a statement is written in a formal programming language like SQL, or in an informal natural language like English, word order and grammar are important when a statement intends to convey logical relationships between things. That's why computer languages depend on rigid grammar and syntax rule parsers. Fortunately, recent advances in natural language syntax tree parsers have made possible the extraction of syntactical and logical relationships from natural language with remarkable accuracy (greater than 90%).[12] In later chapters, we show you how to use packages like `SyntaxNet` (Parsey McParseface) and `SpaCy` to identify these relationships.

And just as in the Bletchley Park example greeting, even if a statement doesn't rely on word order for logical interpretation, sometimes paying attention to that word order can reveal subtle hints of meaning that might facilitate deeper responses. These deeper layers of natural language processing are discussed in the next section. And chapter 2 shows you a trick for incorporating some of the information conveyed by word order into our word- vector representation. It also shows you how to refine the crude tokenizer used in the previous examples (`str.split()`) to more accurately bin words into more appropriate slots within the word vector, so that strings like "good" and "Good" are assigned the same bin, and separate bins can be allocated for tokens like "rosa" and "Rosa" but not "Rosa!".

[12] A comparison of the syntax parsing accuracy of SpaCy (93%), SyntaxNet (94%), Stanford's CoreNLP (90%), and others is available at https://spacy.io/docs/api/

## 1.7 A chatbot natural language pipeline

The NLP pipeline required to build a dialog engine, or chatbot, is similar to the pipeline required to build a question answering system described in *Taming Text* (Manning, 2013).[13] However, some of the algorithms listed within the five subsystem blocks may be new to you. We help you implement these in Python to accomplish various NLP tasks essential for most applications, including chatbots.

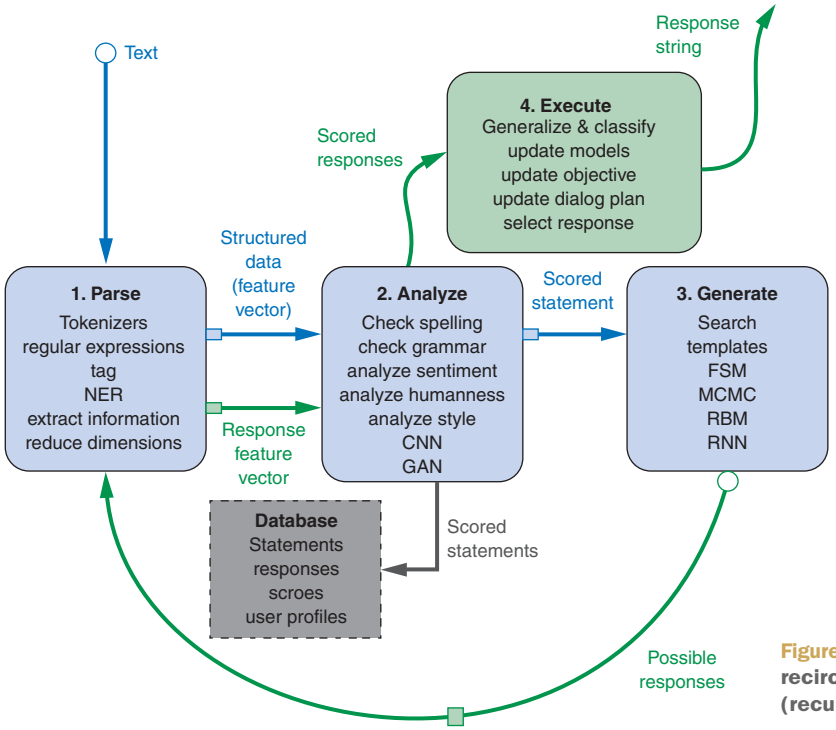



**Figure 1.4 Chatbot recirculating (recurrent) pipeline**

A chatbot requires four kinds of processing as well as a database to maintain a memory of past statements and responses. Each of the four processing stages can contain one or more processing algorithms working in parallel or in series (see figure 1.4).

1. *Parse*—Extract features, structured numerical data, from natural language text.
2. *Analyze*—Generate and combine features by scoring text for sentiment, grammaticality, semantics.
3. *Generate*—Compose possible responses using templates, search, or language models.
4. *Execute*—Plan statements based on conversation history and objectives, and select the next response.

[13] Ingersol, Morton, and Farris, http://www.manning.com/books/taming-text/?a_aid=totalgood

Each of these four stages can be implemented using one or more of the algorithms listed within the corresponding boxes in the block diagram. We show you how to use Python to accomplish near state-of-the-art performance for each of these processing steps. And we show you several alternative approaches to implementing these five subsystems.

Most chatbots will contain elements of all five of these subsystems (the four processing stages as well as the database). But many applications require only simple algorithms for many of these steps. Some chatbots are better at answering factual questions, and others are better at generating lengthy, complex, convincingly human responses. Each of these capabilities require different approaches; we show you techniques for both.

In addition, deep learning and data-driven programming (machine learning, or probabilistic language modeling) have rapidly diversified the possible applications for NLP and chatbots. This data-driven approach allows ever greater sophistication for an NLP pipeline by providing it with greater and greater amounts of data in the domain you want to apply it to. And when a new machine learning approach is discovered that makes even better use of this data, with more efficient model generalization or regularization, then large jumps in capability are possible.

The NLP pipeline for a chatbot shown in figure 1.4 contains all the building blocks for most of the NLP applications that we described at the start of this chapter. As in *Taming Text*, we break out our pipeline into four main subsystems or stages. In addition we've explicitly called out a database to record data required for each of these stages and persist their configuration and training sets over time. This can enable batch or online retraining of each of the stages as the chatbot interacts with the world. In addition we've shown a "feedback loop" on our generated text responses so that our responses can be processed using the same algorithms used to process the user statements. The response "scores" or features can then be combined in an objective function to evaluate and select the best possible response, depending on the chatbot's plan or goals for the dialog. This book is focused on configuring this NLP pipeline for a chatbot, but you may also be able to see the analogy to the NLP problem of text retrieval or "search," perhaps the most common NLP application. And our chatbot pipeline is certainly appropriate for the question answering application that was the focus of *Taming Text*.

The application of this pipeline to financial forecasting or business analytics may not be so obvious. But imagine the features generated by the analysis portion of your pipeline. These features of your analysis or feature generation can be optimized for your particular finance or business prediction. That way they can help you incorporate natural language data into a machine learning pipeline for forecasting. Despite focusing on building a chatbot, this book gives you the tools you need for a broad range of NLP applications, from search to financial forecasting.

One processing element in figure 1.4 that is not typically employed in search, forecasting, or question answering systems is natural language **generation**. For chatbots this is their central feature. Nonetheless, the text generation step is often incorporated into a search engine NLP application and can give such an engine a large com-

petitive advantage. The ability to consolidate or summarize search results is a winning feature for many popular search engines (DuckDuckGo, Bing, and Google). And you can imagine how valuable it is for a financial forecasting engine to be able to generate statements, tweets, or entire articles based on the business-actionable events it detects in natural language streams from social media networks and news feeds.

The next section shows how the layers of such a system can be combined to create greater sophistication and capability at each stage of the NLP pipeline.

## 1.8 Processing in depth

The stages of a natural language processing pipeline can be thought of as layers, like the layers in a feed-forward neural network. Deep learning is all about creating more complex models and behavior by adding additional processing layers to the conventional two-layer machine learning model architecture of feature extraction followed by modeling. In chapter 5 we explain how neural networks help spread the learning across layers by backpropagating model errors from the output layers back to the input layers. But here we talk about the top layers and what can be done by training each layer independently of the other layers.

The top four layers in figure 1.5 correspond to the first two stages in the chatbot pipeline (feature extraction and feature analysis) in the previous section. For example the part-of- speech tagging (POS tagging), is one way to generate features within the Analyze stage of our chatbot pipeline. POS tags are generated automatically by the default `SpaCY` pipeline, which includes all the top four layers in this diagram. POS tagging is typically accomplished with a finite state transducer like the methods in the `nltk.tag` package.

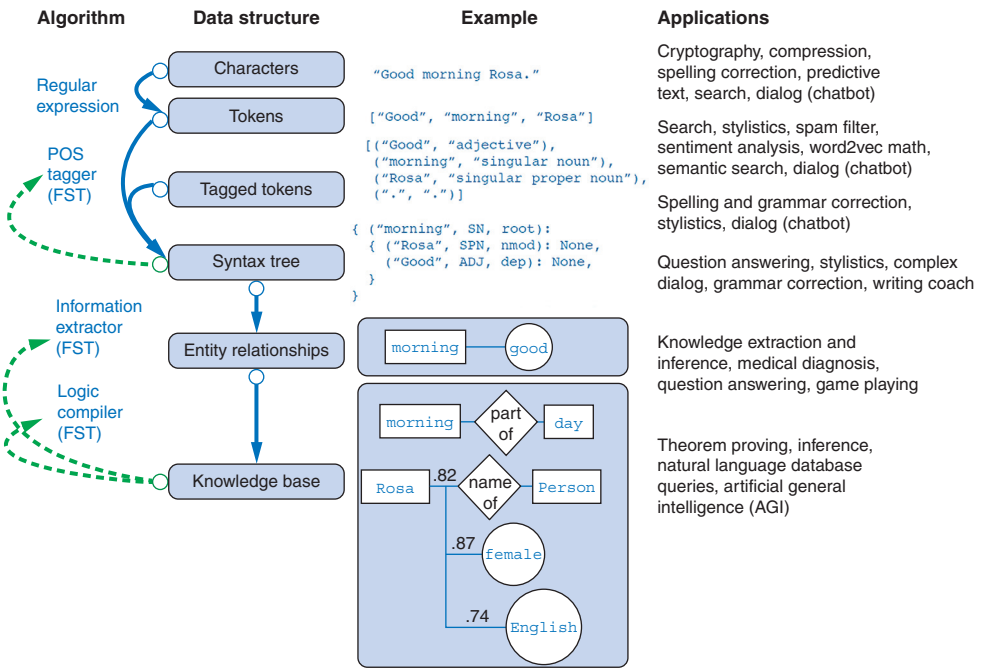



Figure 1.5 Example layers for an NLP pipeline

The bottom two layers (Entity Relationships and a Knowledge Base) are used to populate a database containing information (knowledge) about a particular domain. And the information extracted from a particular statement or document using all six of these layers can then be used in combination with that database to make inferences. Inferences are logical extrapolations from a set of conditions detected in the environment, like the logic contained in the statement of a chatbot user. This kind of "inference engine" in the deeper layers of this diagram are considered the domain of artificial intelligence, where machines can make inferences about their world and use those inferences to make logical decisions.

However, chatbots can make reasonable decisions without this knowledge database, using only the algorithms of the upper few layers. And these decisions can combine to produce surprisingly human-like behaviors.

Over the next few chapters, we dive down through the top few layers of NLP. The top three layers are all that is required to perform meaningful sentiment analysis and semantic search, and to build human-mimicking chatbots. In fact, it's possible to build a useful and interesting chatbot using only single layer of processing, using the text (character sequences) directly as the features for a language model. A chatbot that only does string matching and search is capable of participating in a reasonably convincing conversation, if given enough example statements and responses.

For example, the open source project `ChatterBot` simplifies this pipeline by merely computing the string "edit distance" (Levenshtein distance) between an input statement and the statements recorded in its database. If its database of statement-response pairs contains a matching statement, the corresponding reply (from a previously "learned" human or machine dialog) can be reused as the reply to the latest user statement. For this pipeline, all that is required is step 3 (Generate) of our chatbot pipeline. And within this stage, only a brute force search algorithm is required to find the best response. With this simple technique (no tokenization or feature generation required), `ChatterBot` can maintain a convincing conversion as the dialog engine for Salvius, a mechanical robot built from salvaged parts by Gunther Cox.[14]

`Will` is an open source Python chatbot framework by Steven Skoczen with a completely different approach.[15]. `Will` can only be trained to respond to statements by programming it with regular expressions. This is the labor-intensive and data- light approach to NLP. This grammar-based approach is especially effective for question answering systems and task-execution assistant bots, like Lex, Siri, and Google Now. These kinds of systems overcome the "brittleness" of regular expressions by employing "fuzzy regular expressions"[16] and other techniques for finding approximate grammar matches. Fuzzy regular expressions find the closest grammar matches among a list of

[14] ChatterBot by Gunther Cox and others at https://github.com/gunthercox/ChatterBot

[15] Will, a chatbot for HipChat by Steven Skoczen and the HipChat community https://github.com/skoczen/will

[16] The Python `regex` package is backward compatible with `re` and adds fuzziness among other features. It will replace the `re` in the future: https://pypi.python.org/pypi/regex. Similarly TRE `agrep` (approximate grep) is an alternative to the UNIX command-line application `grep`: https://github.com/laurikari/tre/

possible grammar rules (regular expressions) instead of exact matches by ignoring some maximum number of insertion, deletion, and substitution errors. However, expanding the breadth and complexity of behaviors for a grammar-based chatbot requires a lot of human development work. Even the most advanced grammar-based chatbots, built and maintained by some of the largest corporations on the planet (Google, Amazon, Apple, Microsoft), remain in the middle of the pack for depth and breadth of chatbot IQ.

A lot of powerful things can be done with shallow NLP. And little, if any, human supervision (labeling or curating of text) is required. Often a machine can be left to learn perpetually from its environment (the stream of words it can pull from Twitter or some other source).[17] We show you how to do this in chapter 6.

## 1.9 Natural language IQ

Like human brainpower, the power of an NLP pipeline cannot be easily gauged with a single IQ score without considering multiple "smarts" dimensions. A common way to measure the capability of a robotic system is along the dimensions of complexity of behavior and degree of human supervision required. But for a natural language processing pipeline, the goal is to build systems that fully automate the processing of natural language, eliminating all human supervision (once the model is trained and deployed). So a better pair of IQ dimensions should capture the breadth and depth of the complexity of the natural language pipeline.

A consumer product chatbot or virtual assistant like Alexa or Allo is usually designed to have extremely broad knowledge and capabilities. However, the logic used to respond to requests tends to be shallow, often consisting of a set of trigger phrases that all produce the same response with a single if-then decision branch. Alexa (and the underlying Lex engine) behave like a single layer, flat tree of (if, elif, elif, …) statements.[18] Google Dialogflow (which was developed independently of Google's Allo and Google Assistant) has similar capability to Amazon Lex, Contact Flow, and Lambda, but without the drag-and-drop user interface for designing your dialog tree.

On the other hand, the Google Translate pipeline (or any similar machine translation system) relies on a deep tree of feature extractors, decision trees, and knowledge graphs connecting bits of knowledge about the world. Sometimes these feature extractors, decision trees, and knowledge graphs are explicitly programmed into the system, as in figure 1.5. Another approach rapidly overtaking this "hand-coded" pipeline is the deep learning data-driven approach. Feature extractors for deep neural networks are learned rather than hard-coded, but they often require much more training data to achieve the same performance as intentionally designed algorithms.

[17] Simple neural networks are often used for unsupervised feature extraction from character and word sequences.

[18] More complicated logic and behaviors are now possible when you incorporate Lambdas into an AWS Contact Flow dialog tree: https://greenice.net/creating-call-center-bot-aws-connect-amazon-lex-can-speak-understand/

You will use both approaches (neural networks and hand-coded algorithms) as you incrementally build an NLP pipeline for a chatbot capable of conversing within a focused knowledge domain. This will give you the skills you need to accomplish the natural language processing tasks within your industry or business domain. Along the way you'll probably get ideas about how to expand the breadth of things this NLP pipeline can do. Figure 1.6 puts the chatbot in its place among the natural language processing systems that are already out there. Imagine the chatbots you have interacted with. Where do you think they might fit on a plot like this? Have you attempted to gauge their intelligence by probing them with difficult questions or something like an IQ test?[19] You'll get a chance to do exactly that in later chapters, to help you decide how your chatbot stacks up against some of the others in this diagram.

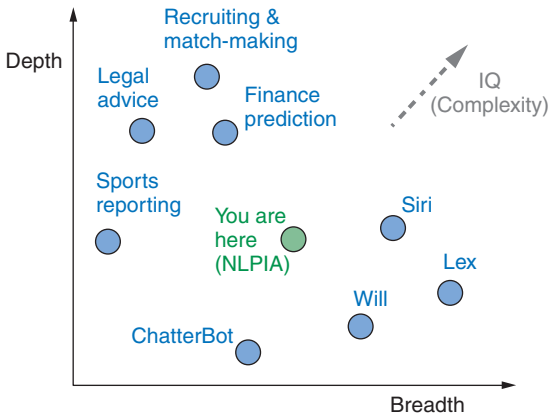



**Figure 1.6 2D IQ of some natural language processing systems**

As you progress through this book, you'll be building the elements of a chatbot. Chatbots require all the tools of NLP to work well:

- Feature extraction (usually to produce a vector space model)
- Information extraction to be able to answer factual questions
- Semantic search to learn from previously recorded natural language text or dialog
- Natural language generation to compose new, meaningful statements

Machine learning gives us a way to trick machines into behaving as if we'd spent a lifetime programming them with hundreds of complex regular expressions or algorithms. We can teach a machine to respond to patterns similar to the patterns defined in regular expressions by merely providing it examples of user statements and the responses we want the chatbot to mimic. And the "models" of language, the FSMs, produced by machine learning, are much better. They are less picky about mispelings and typoz.

[19] A good question suggested by Byron Reese is: "What's larger? The sun or a nickel?", Here are a couple more to get you started.

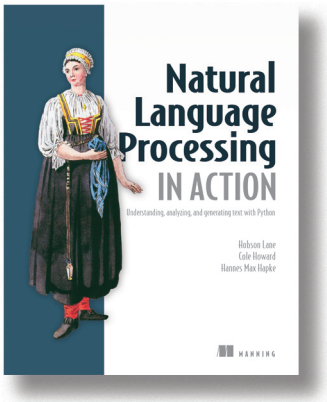
And machine learning NLP pipelines are easier to "program." We don't have to anticipate every possible use of symbols in our language. We just have to feed the training pipeline with examples of the phrases that match and example phrases that don't match. As long we label them during training, so that the chatbot knows which is which, it will learn to discriminate between them. And there are even machine learning approaches that require little if any "labeled" data.

We've given you some exciting reasons to learn about natural language processing. You want to help save the world, don't you? And we've attempted to pique your interest with some practical NLP applications that are revolutionizing the way we communicate, learn, do business, and even think. It won't be long before you're able to build a system that approaches human-like conversational behavior. And you 0 should be able to see in upcoming chapters how to train a chatbot or NLP pipeline with any domain knowledge that interests you—from finance and sports to psychology and literature. If you can find a corpus of writing about it, then you can train a machine to understand it.

The rest of this book is about using machine learning to save us from having to anticipate all the ways people can say things in natural language. Each chapter incrementally improves on the basic NLP pipeline for the chatbot introduced in this chapter. As you learn the tools of natural language processing, you'll be building an NLP pipeline that can not only carry on a conversation but help you accomplish your goals in business and in life.

## 1.10 Summary

- Good NLP may help save the world.
- The meaning and intent of words can be deciphered by machines.
- A smart NLP pipeline will be able to deal with ambiguity.
- We can teach machines common sense knowledge without spending a lifetime training them.
- Chatbots can be thought of as semantic search engines.
- Regular expressions are useful for more than just search.




Most humans are good at reading and interpreting text; computers...not so much. Natural Language Processing (NLP) is the discipline of teaching computers to read more like people, and you see examples of it in everything from chatbots to the speech-recognition software on your phone. Modern NLP techniques based on machine learning radically improve the ability of software to recognize patterns, use context to infer meaning, and accurately discern intent from poorly-structured text. NLP promises to help you improve customer interactions, save cost, and reinvent text-intensive applications like search or product support.

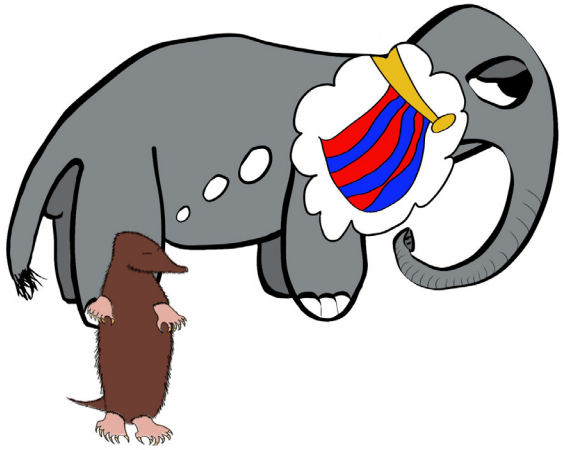
*Natural Language Processing in Action* is your guide to creating machines that understand human language using the power of Python with its ecosystem of packages dedicated to NLP and AI! You'll start with a mental model of how a computer learns to read and interpret language. Then, you'll discover how to train a Python-based NLP machine to recognize patterns and extract information from text. As you explore the carefully-chosen examples, you'll expand your machine's knowledge and apply it to a range of challenges, from building a search engine that can find documents based on their meaning rather than merely keywords, to training a chatbot that uses deep learning to answer questions and participate in a conversation.

***What's inside:***

- Working with Keras, TensorFlow, Gensim, scikit-learn, and more
- Parsing and normalizing text
- Rule-based (Grammar) NLP
- Data-based (Machine Learning) NLP
- Deep Learning NLP
- End-to-end chatbot pipeline with training data
- Scalable NLP pipelines
- Hyperparameter optimization algorithms

Although all examples are written in Python, experience with any modern programming language allows readers to get the most from this book. A basic understanding of machine learning's also helpful.

# Generating Features



*"The elephant is like a flag, hanging loosely."*

Now we continue our journey from trunk to tail moving on to consider how to engineer features. Features are semantically meaningful, derived representations of our raw data—which in our case is natural language. Despite all of the attention that model learning algorithms can get, much of real-world machine learning work's focused on features. When we work with language, features are crucial technique for injecting our knowledge of the structure of language into our machine learning system.

This chapter from my book, *Machine Learning Systems*, breaks down the work of feature engineering into two phases: extraction of the initial feature, and transformation of extracted features into other features. It takes advantage of some of the feature engineering capabilities built into Spark's MLlib, a widely used machine learning library capable of operating at enormous scale. The chapter also discusses how we work with sets of features. Finally, it discusses how to compose pipelines of operations on data to ensure that the system behaves as designed and reliably executes the feature generation task.

Chapter 4 from *Machine Learning Systems* by Jeff Smith.

# *Generating features*

***This chapter covers***

- Extracting features from raw data
- Transforming features to make them more useful
- Selecting among the features you've created
- How to organize feature-generation code

This chapter is the next step on our journey through the components, or phases, of a machine learning system, shown in figure 4.1. The chapter focuses on turning raw data into useful representations called *features*. The process of building systems that can generate features from data, sometimes called *feature engineering*, can be deceptively complex. Often, people begin with an intuitive understanding of *what* they want the features used in a system to be, with few plans for *how* those features will be produced. Without a solid plan, the process of feature engineering can easily get off track, as you saw in the Sniffable example from chapter 1.

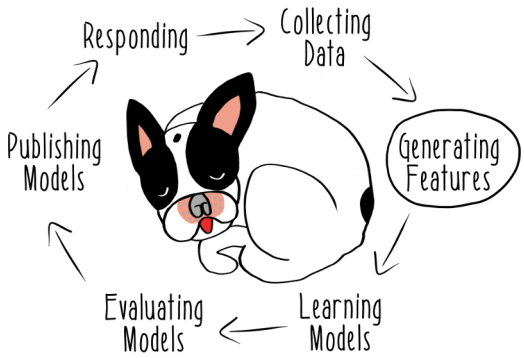



**Figure 4.1** **Phases of machine learning**

In this chapter, I'll guide you through the three main types of operations in a feature pipeline: extraction, transformation, and selection. Not all systems do all the types of operations shown in this chapter, but all feature engineering techniques can be thought of as falling into one of these three buckets. I'll use type signatures to assign techniques to groups and give our exploration some structure, as shown in table 4.1.

**Table 4.1 Phases of feature generation**

| Phase | Input | Output |
|---|---|---|
| Extract | `RawData` | `Feature` |
| Transform | `Feature` | `Feature` |
| Select | `Set[Feature]` | `Set[Feature]` |

Real-world feature pipelines can have very complex structures. You'll use these groupings to help you understand how you can build a feature-generation pipeline in the best way possible. As we explore these three types of feature-processing operations, I'll introduce common techniques and design patterns that will keep your machine learning system from becoming a tangled, unmaintainable mess. Finally, we'll consider some general properties of data pipelines when discussing the next component of machine learning systems discussed in chapter 5, the model-learning pipeline.

### Type signatures

You may not be familiar with the use of *types* to guide how you think about and implement programs. This technique is common in statically typed languages like Scala and Java. In Scala, functions are defined in terms of the inputs they take, the outputs they return, and the types of both. This is called a *type signature*. In this book, I mostly use a fairly simple form of signature notation that looks like this: `Grass => Milk`. You can read this as, "A function from an input of type Grass to an output of type Milk." This would be the type signature of some function that behaves much like a cow.


To cover this enormous scope of functionality, we need to rise above it all to gain some perspective on what features are all about. To that end, we'll join the team of Pidg'n, a microblogging social network for tree-dwelling animals, not too different from Twitter.

We'll look at how we can take the chaos of a short-form, text-based social network and build meaningful representations of that activity. Much like the forest itself, the world of features is diverse and rich, full of hidden complexity. We can, however, begin to peer through the leaves and capture insights about the lives of tree-dwelling animals using the power of reactive machine learning.

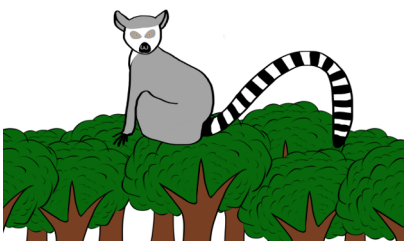
## 4.1 Spark ML

Before we get started building features, I want to introduce you to more Spark functionality. The `spark.ml` package, sometimes called Spark ML, defines some high-level APIs that can be used to create machine learning pipelines. This functionality can reduce the amount of machine learning–specific code that you need to implement yourself, but using it does involve a change in how you structure your data.

The Spark ML API uses mostly the same nomenclature for feature extraction, transformation, and selection that I use in this chapter, though there are subtle differences. If and when you read the Spark ML documentation, you may see something called a *transformation* operation, which I call an *extraction* operation. These are generally minor, unimportant differences that you can ignore. Different technologies name and structure this functionality differently, and you'll see all sorts of different naming conventions in the machine learning literature. The type signature–based framework for dividing feature-generation functionality that I use in this chapter is just a tool to help you implement and organize your code. Once you've mastered the feature-generation concepts in this chapter, you'll be better equipped to see through differences in nomenclature to the similarities in functionality.

Much of the machine learning functionality in Spark is designed to be used with DataFrames, instead of the RDDs that you've seen up until this point. DataFrames are simply a higher-level API on top of RDDs that give you a richer set of operations. You can think of DataFrames as something like tables in relational databases. They have different columns, which you can define and then query. Much of the recent progress in performance and functionality within Spark has been focused on DataFrames, so to get access to the full power of things like MLlib's machine learning capabilities, you'll need to use DataFrames for some operations. The good news is that they're very similar to the RDDs you've been working with and tabular data structures you may have used in other languages, such as pandas DataFrames in Python or R's data frames.

## 4.2 Extracting features



Now that I've introduced some of the tools, let's begin to solve the problem. We'll start our exploration of the feature engineering process at the very beginning, with raw data. In this chapter, you'll take on the role of Lemmy, an engineer on the Pidg'n data team.

Your team knows it wants to build all sorts of predictive models about user activity. You're just getting started, though, and all you have are the basics of application data: squawks (text posts of 140 characters or less), user profiles, and the follower relationships. This is a rich dataset, for sure, but you've never put it to much analytical use. To start with, you've decided to focus on the problem of predicting which new users will become *Super Squawkers*, users with more than a million followers.

To start this project, you'll extract some features to use in the rest of your machine learning system. I define the process of feature extraction as taking in raw data of some sort and returning a feature. Using Scala type signatures, feature extraction can be represented like this: `RawData => Feature`. That type signature can be read as, "A function that takes raw data and returns a feature." If you define a function that satisfies that type signature, it might look something like the stub in the following listing.

**Listing 4.1 Extracting features**

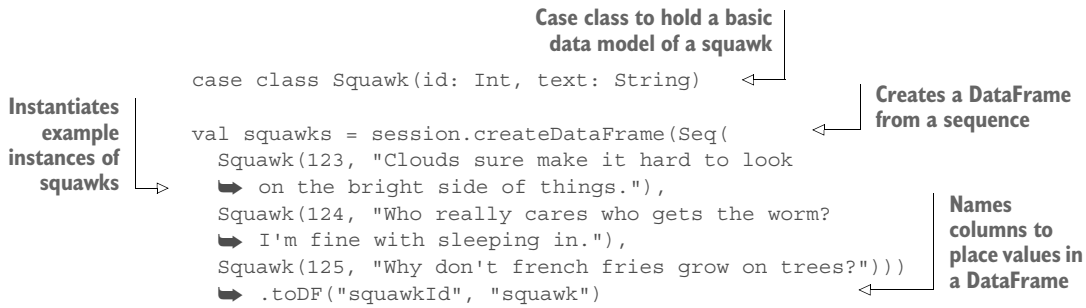
```
def extract(rawData: RawData): Feature = ???
```

Put differently, any output produced from raw data is a potential feature, regardless of whether it ever gets used to learn a model.

The Pidg'n data team has been collecting data since day one of the app as part of keeping the network running. You have the complete, unaltered record of all the actions ever taken by Pidg'n users, much like the data model discussed in chapter 3. Your team has built a few aggregates of that data for basic analytical purposes. Now you want to take that system to the next level by generating semantically meaningful derived representations of that raw data—features. Once you have features of any kind, you can begin learning models to predict user behavior. In particular, you're interested in seeing if you can understand what makes particular squawks and squawkers more popular than others. If a squawker has the potential to become very popular, you want to provide them with a more streamlined experience, free of advertisements, to encourage them to squawk more.

Let's begin by extracting features from the raw data of the text of squawks. You can start by defining a simple case class and extracting a single feature for a few squawks. Listing 4.2 shows how to extract a feature consisting of the words in the text of a given squawk. This implementation will use Spark's `Tokenizer` to break sentences into words. Tokenization is just one of several common text-processing utilities that come built into Spark that make writing code like this fast and easy. For advanced use cases, you may want to use a more sophisticated text-parsing library, but having common utilities easily available can be very helpful.

**Listing 4.2 Extracting word features from squawks**




```
case class Squawk(id: Int, text: String)

val squawks = session.createDataFrame(Seq(
  Squawk(123, "Clouds sure make it hard to look
  ➥ on the bright side of things."),
  Squawk(124, "Who really cares who gets the worm?
  ➥ I'm fine with sleeping in."),
  Squawk(125, "Why don't french fries grow on trees?")))
  ➥ .toDF("squawkId", "squawk")
```

```
val tokenizer = new Tokenizer().setInputCol("squawk")
➥ .setOutputCol("words")

val tokenized = tokenizer.transform(squawks)

tokenized.select("words", "squawkId").show()
```

Sets up a Tokenizer to split the text of squawks into words and put them in an output column

Executes the Tokenizer and populates the words column in a DataFrame

Prints the results for inspection

The operations in listing 4.2 give you a `DataFrame` that contains a column called `words`, which has all the words in the text of the squawk. You could call the values in the `words` column a *feature*. These values could be used to learn a model. But let's make the semantics of the pipeline clearer using the Scala type system.

Using the code in listing 4.3, you can define what a feature is and what specific sort of feature you've produced. Then, you can take the `words` column from that `DataFrame` and use it to instantiate instances of those feature classes. It's the same words that the `Tokenizer` produced for you, but now you have richer representations that you can use to help build up a feature-generation pipeline.

**Listing 4.3 Extracting word features from squawks**

```
trait FeatureType {
  val name: String
  type V
}

trait Feature extends FeatureType {
  val value: V
}

case class WordSequenceFeature(name: String, value: Seq[String])
➥ extends Feature {
  type V = Seq[String]
}

val wordsFeatures = tokenized.select("words")
  .map(row =>
  WordSequenceFeature("words",
    row.getSeq[String](0)))

wordsFeatures.show()
```

Defines a base trait for all types of features

Requires feature types to have names

Type parameter to hold the type of values generated by feature

Defines a base trait for all features as an extension of feature types

Requires that features have values of the type specified in the feature type

Defines a case class for features consisting of word sequences

Specifies that the type of features being generated is a sequence of strings (words)

Maps over rows and applies a function to each

Selects a words column from the DataFrame

Gets extracted words out of a row

Creates an instance of WordSequenceFeature named words

Prints features for inspection

With this small bit of extra code, you can define your features in a way that's more explicit and less tied to the specifics of the raw data in the original `DataFrame`. The resulting value is an RDD of `WordSequenceFeature`. You'll see later how you can con-

tinue to use this `Feature` trait with specific case classes defining the different types of features in your pipeline.


Also note that, when operating over the `DataFrame`, you can use a pure, anonymous, higher-order function to create instances of your features. The concepts of purity, anonymous functions, and higher-order functions may have sounded quite abstract when I introduced them in chapter 1. But now that you've seen them put to use in several places, I hope it's clear that they can be very simple to write. Now that you've gotten some Scala and Spark programming under your belt, I hope you're finding it straightforward to think of data transformations like feature extraction in terms of pure functions with no side effects.

You and the rest of the Pidg'n data team could now use these features in the next phase of the machine learning pipeline—model learning—but they probably wouldn't be good enough to learn a model of Super Squawkers. These initial word features are just the beginning. You can encode far more of your understanding of what makes a squawker super into the features themselves.

To be clear, there are sophisticated model-learning algorithms, such as neural networks, that require very little feature engineering on the data that they consume. You *could* use the values you've just produced as features in a model-learning process. But many machine learning systems will require you to do far more with your features before using them in model learning if you want acceptable predictive performance. Different model-learning algorithms have different strengths and weaknesses, as we'll explore in chapter 5, but all of them will benefit from having base features transformed in ways that make the process of model learning simpler. We need to move on to see how to make features out of other features.

## 4.3 Transforming features

Now that you've extracted some basic features, let's figure out how to make them useful. This process of taking a feature and producing a new feature from it is called *feature transformation*. In this section, I'll introduce you to some common transform functions and discuss how they can be structured. Then I'll show you a very important class of feature transformations: transforming features into concept labels.

What is feature transformation? In the form of a type signature, feature transformation can be expressed as `Feature => Feature`, a function that takes a feature and returns a feature. A stub implementation of a transformation function (sometimes called a *transform*) is shown in the next listing.

**Listing 4.4 Transforming features**

```
def transform(feature: Feature): Feature = ???
```

In the case of the Pidg'n data team, you've decided to build on your previous feature-engineering work by creating a feature consisting of the frequencies of given words in a squawk. This quantity is sometimes called a *term frequency*. Spark has built-in functionality that makes calculating this value easy.

**Listing 4.5 Transforming words to term frequencies**

```
val hashingTF = new HashingTF()
  .setInputCol("words")
  .setOutputCol("termFrequencies")

val tfs = hashingTF.transform(tokenized)

tfs.select("termFrequencies").show()
```

Instantiates an instance of a class to calculate term frequencies

Defines an input column to read from when consuming DataFrames

Defines an output to put term frequencies in

Executes the transformation

Prints term frequencies for inspection

It's worth noting that the `hashingTF` implementation of term frequencies was implemented to consume the `DataFrame` you previously produced, not the features you designed later. Spark ML's concept of a pipeline is focused on connecting operations on `DataFrames`, so it can't consume the features you produced before without more conversion code.


### Feature hashing

The use of the term *hashing* in the Spark library refers to the technique of *feature hashing*. Although it's not always used in feature-generation pipelines, feature hashing can be a critically important technique for building large numbers of features. In text-based features like term frequencies, there's no way of knowing a priori what all the possible features could be. Squawkers can write anything they want in a squawk on Pidg'n. Even an English-language dictionary wouldn't contain all the slang terms squawkers might use. Free-text input means that the universe of possible terms is effectively infinite.

One solution is to define a hash range of the size of the total number of distinct features you want to use in your model. Then you can apply a deterministic hashing function to each input to produce a distinct value within the hash range, giving you a unique identifier for each feature. For example, suppose `hash("trees")` returns `65381`. That value will be passed to the model-learning function as the identifier of the feature. This might not seem much more useful than just using `"trees"` as the identifier, but it is. When I discuss prediction services in chapter 7, I'll talk about why you'll want to be able to identify features that the system has possibly never seen before.

Let's take a look at how Spark ML's `DataFrame`-focused API is intended to be used in connecting operations like this. You won't be able to take full advantage of Spark ML until chapter 5, where you'll start learning models, but it's still useful for feature generation. Some of the preceding code can be reimplemented using a `Pipeline` from Spark ML. That will allow you to set the tokenizer and the term frequency operations as stages within a pipeline.

**Listing 4.6 Using Spark ML pipelines**

```
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, hashingTF))

val pipelineHashed = pipeline.fit(squawksDF)

println(pipelineHashed.getClass)
```

Instantiates a new pipeline

Sets the two stages of this pipeline

Executes the pipeline

Prints the type of the result of the pipeline, a PipelineModel

This `Pipeline` doesn't result in a set of features, or even a `DataFrame`. Instead, it returns a `PipelineModel`, which in this case won't be able to do anything useful, because you haven't learned a model yet. We'll revisit this code in chapter 5, where we can go all the way from feature generation through model learning. The main thing to note about this code at this point is that you can encode a pipeline as a clear abstraction within your application. A large fraction of machine learning work involves working with pipeline-like operations. With the Spark ML approach to pipelines, you can be very explicit about how your pipeline is composed by setting the stages of the pipeline in order.

### 4.3.1 Common feature transforms

Sometimes you don't have library implementations of the feature transform that you need. A given feature transform might have semantics that are specific to your application, so you'll often need to implement feature transforms yourself.

Consider how you could build a feature to indicate that a given Pidg'n user was a Super Squawker (user with more than a million followers). The feature-extraction process will give you the raw data about the number of followers a given squawker has. If you used the number of followers as a feature, that would be called a *numerical* feature. That number would be an accurate snapshot of the data from the follower graph, but it wouldn't necessarily be easy for all model-learning algorithms to use. Because your intention is to express the idea of a Super Squawker, you could use a far simpler representation: a Boolean value representing whether or not the squawker has more than a million followers.

The squirrel, a rather ordinary user, has very few followers. But the sloth is an terrific Super Squawker. To produce meaningful features about the differences between these two squawkers, you'll follow the same process of going from raw data, to numeric features, and then to Boolean features. This series of data transformations is shown for the two users in figure 4.2.

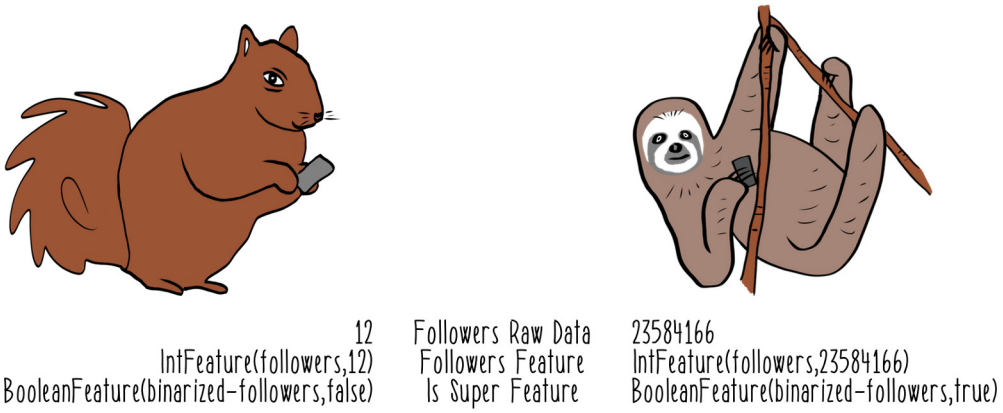



**Figure 4.1 Feature transformations**

The following listing shows how to implement this approach to binarization to produce a Super Squawker feature.

**Listing 4.7 Binarizing a numerical feature**

Specifies that these are integer features

Case class representing a numerical feature where the value is an integer

```
case class IntFeature(name: String, value: Int) extends Feature {
  type V = Int
}
```

Case class representing a Boolean feature

```
case class BooleanFeature(name: String, value: Boolean) extends Feature {
  type V = Boolean
}
```

Specifies that these are Boolean features

Function that takes a numeric integer feature and threshold and returns a Boolean feature

```
def binarize(feature: IntFeature, threshold: Double): BooleanFeature = {
  BooleanFeature("binarized-" + feature.name, feature.value > threshold)
}
```

Adds the name of the transform function to the resulting feature name

Constant defining the cutoff for a squawker to be super

```
val SUPER_THRESHOLD = 1000000
```

Raw numbers of followers for the squirrel and the sloth

```
val squirrelFollowers = 12
val slothFollowers = 23584166
```

Numeric integer feature representing the number of followers

```
val squirrelFollowersFeature = IntFeature("followers", squirrelFollowers)
val slothFollowersFeature = IntFeature("followers", slothFollowers)
```

```
val squirrelIsSuper = binarize(squirrelFollowers, SUPER_THRESHOLD)
val slothIsSuper = binarize(slothFollowers, SUPER_THRESHOLD)
```

Boolean feature indicating the squirrel is not a Super Squawker

Boolean feature indicating the sloth is a Super Squawker


The `binarize` function is a good example of a reusable transform function. It also ensures the resulting feature is somewhat self-describing by appending the name of the transform function to the resulting feature. Ensuring that we can identify the operations that were applied to produce a feature is an idea we'll revisit in later chapters. Finally, note that the transformation function `binarize` is a pure function.

Using only pure functions in feature transforms is an important part of establishing a coherent structure for feature-generation code. Separating feature extraction and feature transformation within a code base can be difficult, and the boundaries between the two can be hard to draw. Ideally, any I/O or side-effecting operations should be contained in the feature-extraction phase of the pipeline, with all transformations' functionality being implemented as pure functions. As you'll see later, pure transforms are simple to scale and easy to reuse across features and feature-extraction contexts (model learning and predicting).

There's a huge range of commonly used transformation functions. Similar to binarization, some approaches reduce continuous values to discrete labels. For example, a feature designed to express the time of day when a squawk was posted might not use the full timestamp. Instead, a more useful representation could be to transform all times into a limited set of labels, as shown in table 4.2.

**Table 4.2 Transforming times into time labels**

| Time | Label |
|---|---|
| 7:02 | Morning |
| 12:53 | Midday |
| 19:12 | Night |

The implementation of a transform to do this is trivial and is naturally a pure function.

There's another variation on reducing continuous data to labels, called *binning*, in which the source feature is reduced to some arbitrary label defined by the range of values that it falls into. For example, you could take the number of squawks a given user has made and reduce it to one of three labels indicating how active the squawker is, as shown in table 4.3.

**Table 4.3 Binning**

| Squawks | Label | Activity level |
|---|---|---|
| 7 | 0_99 | Least active squawkers |
| 1,204 | 1000_99999 | Moderately active squawkers |
| 2,344,910 | 1000000_UP | Most active squawkers |

Again, an implementation of such a transform would be trivial and naturally a pure function. Transforms *should* be easy to write and should correspond closely to their formulation in mathematical notation. When it comes to implementing transforms, you should always abide by the KISS principle: Keep It Simple, Sparrow. Reactive machine learning systems are hard enough to implement without implementing complicated transforms. Usually, an overly long transform implementation is a smell that someone has laid a rotten egg. In a few special cases, you may want to implement something like a transformer with more involved semantics. We'll consider such circumstances later in this chapter and later in the book.

### 4.3.2 Transforming concepts

Before we leave the topic of transformations, we need to consider one very common and critical class of feature transformations: the ones that produce concepts. As mentioned in chapter 1, *concepts* are the things that a machine learning model is trying to predict. Although some machine learning algorithms can learn models of continuous concepts, such as the number of squawks a given user will write over the course of the next month, many machine learning systems are built to perform classification. In *classification* problems, the learning algorithm is trying to learn a discrete number of class labels, not continuous values. In such systems, the concept has to be produced from the raw data, during feature extraction, and then reduced to a class label via transformation. Concept class labels aren't exactly the same thing as features, but often the difference is just a matter of how we use the piece of data. Typically, and ideally, the same code that might binarize a feature will also binarize a concept.

Building on the code in listing 4.7, in the next listing, take the Boolean feature about Super Squawkers and produce a Boolean concept label that classifies squawkers into super or not.

**Listing 4.8 Creating concept labels from features**

```
trait Label extends Feature

case class BooleanLabel(name: String, value: Boolean) extends Label {
  type V = Boolean
}

def toBooleanLabel(feature: BooleanFeature) = {
  BooleanLabel(feature.name, feature.value)
}

val squirrelLabel = toBooleanLabel(squirrelIsSuper)
val slothLabel = toBooleanLabel(slothIsSuper)

Seq(squirrelLabel, slothLabel).foreach(println)
```

Defines labels as subtypes of features

Creates a case class for Boolean labels

Defines a simple conversion function from Boolean features to Boolean labels

Converts Super Squawker feature values into concept labels

Prints label values for inspection

In this code, you've defined concept labels as a special subtype of features. That's not how features and labels are generally discussed, but it can be a helpful convention for code reuse in machine learning systems. Whether you intend to do so or not, any given feature value could be used as a concept label if it represents the concept class to be learned. The `Label` trait in listing 4.8 doesn't change the underlying structure of the data in a feature, but it does allow you to annotate when you're using a feature as a concept label. The rest of the code is quite simple, and you arrive at the same conclusion again: people just aren't that interested in what squirrels have to say.

## 4.4 Selecting features

Again, you find yourself in the same situation: if you've done all the work so far, you might now be finished. You could use the features you've already produced to learn a model. But sometimes it's worthwhile to perform additional processing on features before beginning to learn a model. In the previous two phases of the feature-generation process, you produced all the features you *might* want to use to learn a model, sometimes called a *feature set*. Now that you have that feature set, you could consider throwing some of those features in the trash.

The process of choosing from a feature set which features to use is known as *feature selection*. In type-signature form, it can be expressed `Set[Feature] => Set[Feature]`, a function that takes a set of features and returns another set of features. The next listing shows a stub implementation of a feature selector.

**Listing 4.9 A feature selector**

```
def select(featureSet: Set[Feature]): Set[Feature] = ???
```


Why would you ever want to discard features? Aren't they useful and valuable? In theory, a robust machine learning algorithm could take as input feature vectors containing arbitrary numbers of features and learn a model of the given concept. In reality, providing a machine learning algorithm with too many features is just going to make it take longer to learn a model and potentially degrade that model's performance. You can find yourself needing to choose among features quite easily. By varying the parameters used in the transformation process, you could create an infinite number of features with a very small amount of code.

Using a modern distributed data-processing framework like Spark makes handling arbitrarily sized datasets easy. It's definitely to your benefit to consider a huge range of features during the feature extraction and transformation phases of your pipeline. And once you've produced all the features in your feature set, you can use some of the facilities in Spark to cut that feature set down to just those features that your model-learning algorithm will use to learn the model. There are implementations of feature-selection functionality in other machine learning libraries; Spark's MLlib is one of many options and certainly not the oldest one. For some cases, the feature-selection functionality provided by MLlib might not be sufficient, but the principles of feature

selection are the same whether you use a library implementation or something more bespoke. If you end up writing your own version of feature selection, it will still be conceptually similar to MLlib's implementations.

Using the Spark functionality will again require you to leave behind your feature-case classes and the guarantees of static typing to use the machine learning functionality implemented around the high-level DataFrame API. To begin, you'll need to construct a `DataFrame` of training instances. These instances will consist of three parts: an arbitrary identifier, a feature vector, and a concept label. The following listing shows how to build up this collection of instances. Instead of using real features, you'll use some synthetic data, which you can imagine being about various properties of Squawkers.

**Listing 4.10 A `DataFrame` of instances**

```
val instances = Seq(
  (123, Vectors.dense(0.2, 0.3, 16.2, 1.1), 0.0),
  (456, Vectors.dense(0.1, 1.3, 11.3, 1.2), 1.0),
  (789, Vectors.dense(1.2, 0.8, 14.5, 0.5), 0.0)
)

val featuresName = "features"
val labelName = "isSuper"

val instancesDF = session.createDataFrame(instances)
  .toDF("id", featuresName, labelName)
```

Defines a collection of instances

Hardcodes some synthetic feature and concept label data

Names for features and label columns

Creates a DataFrame from the instances collection

Sets the name of each column in the DataFrame

Once you have a `DataFrame` of instances, you can take advantage of the feature-selection functionality built into MLlib. You can apply a chi-squared statistical test to rank the impact of each feature on the concept label. This is sometimes called *feature importance*. After the features are ranked by this criterion, the less impactful features can be discarded prior to model learning. The next listing shows how you can select the two most important features from your feature vectors.

**Listing 4.11 Chi-squared-based feature selection**

```
val selector = new ChiSqSelector()
  .setNumTopFeatures(2)
  .setFeaturesCol(featuresName)
  .setLabelCol(labelName)
  .setOutputCol("selectedFeatures")

val selectedFeatures = selector.fit(instancesDF)
  .transform(instancesDF)

selectedFeatures.show()
```

Creates a new feature selector

Sets the number of features to retain to 2

Sets the column where features are

Sets the column where concept labels are

Sets the column to place results, the selected features

Fits a chi-squared model to the data

Selects the most important features and returns a new DataFrame

Prints the resulting DataFrame for inspection

As you can see, having standard feature-selection functionality available at a library call makes feature selection pretty convenient. If you had to implement chi-squared-based feature selection yourself, you'd find that the implementation was a lot longer than the code you just wrote.

## 4.5 Structuring feature code

In this chapter, you've written example implementations of all the most common components of a feature-generation pipeline. As you've seen, some of these components are simple and easy to build, and you could probably see yourself building quite a few of them without any difficulty. If you've Kept It Simple, Sparrow, you shouldn't be intimidated by the prospect of producing lots of feature extraction, transformation, and selection functionality in your system. Or should you?

Within a machine learning system, feature-generation code can often wind up being the largest part of the codebase by some measures. A typical Scala implementation might have a class for each extraction and transformation operation, and that can quickly become unwieldy as the number of classes grows. To prevent feature-generation code from becoming a confusing grab bag of various arbitrary operations, you need to start putting more of your understanding of the semantics of feature generation into the structure of your implementation of feature-generation functionality. The next section introduces one such strategy for structuring your feature-generation code.

### 4.5.1 Feature generators

At the most basic level, you need to define an implementation of what is a unit of feature-generation functionality. Let's call this a *feature generator*. A feature generator can encompass either extraction or both extraction and transformation operations. The implementation of the extraction and transformation operations may not be very different from what you've seen before, but these operations will all be encapsulated in an independently executable unit of code that produces a feature. Your feature generators will be things that can take raw data and produce features that you want to use to learn a model.

Let's implement your feature generators using a trait. In Scala, *traits* are used to define behaviors in the form of a type. A typical trait will include the signatures and possibly implementations of methods that define the common behavior to the trait. Scala traits are very similar to interfaces in Java, C++, and C# but are much easier and more flexible to use than interfaces in any of those languages.

For the purpose of this section, let's say that your raw data, from the perspective of your feature-generation system, consists of squawks. Feature generation will be the process of going from squawks to features. The corresponding feature-generator trait can be defined.

**Listing 4.12 A feature-generator trait**

```
trait Generator {

  def generate(squawk: Squawk): Feature

}
```

The `Generator` trait defines a feature generator to be an object that implements a method, `generate`, that takes a squawk and returns a feature. This is a concrete way of defining the behavior of feature generation. A given implementation of feature generation might need all sorts of other functionality, but this is the part that will be common across all implementations of feature generation. Let's look at one implementation of this trait.

Your team is interested in understanding how squawk length affects squawk popularity. There's an intuition that even 140 characters is too much to read for some squawkers, such as hummingbirds. They just get bored too quickly. Conversely, vultures have been known to stare at the same squawk for hours on end, so long posts are rarely a problem for them. For you to be able to build a recommendation model that will surface relevant content to these disparate audiences, you'll need to encode some of the data around squawk length as a feature. This can easily be implemented using the `Generator` trait.

As discussed before, the idea of length can be captured using the technique of binning to reduce your numeric data to categories. There's not much difference between a 72-character squawk and a 73-character squawk; you're just trying to capture the approximate size of a squawk. You'll divide squawks into three categories based on length: short, moderate, and long. You'll define your thresholds between the categories to be at the thirds of the total possible length. Implemented according to your `Generator` trait, you get something like the following listing.

**Listing 4.13 A categorical feature generator**

```
object SquawkLengthCategory extends Generator {

  val ModerateSquawkThreshold = 47
  val LongSquawkThreshold = 94

  private def extract(squawk: Squawk): IntFeature = {
    IntFeature("squawkLength", squawk.text.length)
  }

  private def transform(lengthFeature: IntFeature): IntFeature = {
```

**Defines a generator as an object that extends the Generator trait**

**Constant thresholds to compare against**

**Extracting: uses the length of the squawk to instantiate an IntFeature**

**Transforming: takes the IntFeature of length, returns the IntFeature of category**

```
    val squawkLengthCategory = lengthFeature match {
      case IntFeature(_, length) if length < ModerateSquawkThreshold => 1
      case IntFeature(_, length) if length < LongSquawkThreshold => 2
      case _ => 3
    }

    IntFeature("squawkLengthCategory", squawkLengthCategory)
  }

  def generate(squawk: Squawk): IntFeature = {
    transform(extract(squawk))
  }
}
```

**Uses a pattern-matching structure to determine which category the squawk length falls into**

**Returns Int for a category (for ease of use in model learning)**

**Returns a category of 3, a long squawk, in all other cases**

**Returns a category as a new IntFeature**

**Generating: extracts a feature from the squawk and then transforms it to a categorical IntFeature**


This generator is defined in terms of a singleton object. You don't need to use instances of a class, because all the generation operations are themselves pure functions.

Internal to your implementation of the feature generator, you still used a concept of extraction and transformation, even though you now only expose a `generate` method as the public API to this object. Though that may not always seem necessary, it can be helpful to define all extraction and transformation operations in a consistent manner using feature-based type signatures. This can make it easier to compose and reuse code.

Reuse of code is a huge issue in feature-generation functionality. In a given system, many feature generators will be performing operations very similar to each other.

A given transform might be used dozens of times if it's factored out and reusable. If you don't think about such concerns up front, you may find that your team has reimplemented some transform, like averaging five different times in subtly different ways across your feature-generation codebase. That can lead to tricky bugs and bloated code.

You don't want your feature-generation code to be messier than a tree full of marmosets! Let's take a closer look at the structure of your generator functionality. The `transform` function in listing 4.13 was doing something you might wind up doing a lot in your codebase: categorizing according to some threshold. Let's look at it again.

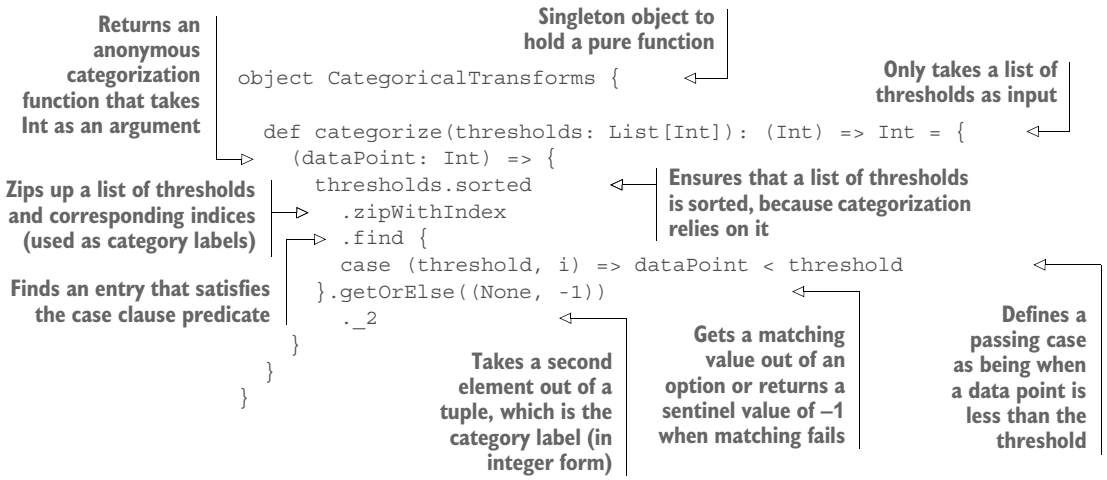
**Listing 4.14 Categorization using pattern matching**

```
private def transform(lengthFeature: IntFeature): IntFeature = {
  val squawkLengthCategory = lengthFeature match {
    case IntFeature(_, length) if length < ModerateSquawkThreshold => 1
    case IntFeature(_, length) if length < LongSquawkThreshold => 2
    case _ => 3
  }
```

You definitely shouldn't be implementing a comparison against thresholds more than once, so let's find a way to pull that code out and make it reusable. It's also weird that you had to define the class label integers yourself. Ideally, you'd just have to worry about your thresholds and nothing else.

Let's pull out the common parts of this code for reuse and make it more general in the process. The code in the next listing shows one way of doing this. It's a little dense, so we'll walk through it in detail.

**Listing 4.15 Generalized categorization**




This solution uses a few techniques that you may not have seen before. For one, this function's return type is `(Int) => Int`, a function that takes an integer and returns an integer. In this case, the function returned will categorize a given integer according to the thresholds previously provided.

The thresholds and categories are also zipped together so they can be operated on as a pair of related values (in the form of a tuple). *Zipping*, or *convolution* as it's sometimes called, is a powerful technique that's commonly used in Scala and other languages in the functional programming tradition. The name *zip* comes from the similarity to the action of a zipper. In this case, you're using a special sort of zip operation that conveniently provides you indices corresponding to the number the elements in the collection being zipped over. This approach to producing indices is far more elegant than C-style iteration using a mutable counter, which you may have seen in other languages, such as Java and C++.

After zipping over the values, you use another new function, `find`, with which you can define the element of a collection you're looking for in terms of a *predicate*. Predicates are Boolean functions that are either true or false, depending on their values. They're commonly used in mathematics, logic, and various forms of programming

such as logic and functional programming. In this usage, the predicate gives you a clear syntax for defining what constitutes falling into a category bucket.

This code also deals with uncertainty in external usage in ways that you haven't before. Specifically, it sorts the categories, because they might not be provided in a sorted list, but your algorithm relies on operating on them in order. Also, the `find` function returns an `Option` because the `find` operation may or may not find a matching value. In this case, you use the value `-1` to indicate an unusable category, but how a categorization failure should be handled depends a lot on how the functionality will be integrated in the client generator code. When you factor out common feature transforms to shared functions like this, you should take into account the possibilities of future broad usage of the transform. By implementing it with these extra guarantees, you reduce the chances that someone will use your categorization functionality in the future and not get the results they wanted.

The code in listing 4.15 might be a bit harder to understand than the original implementation in listings 4.13 and 4.14. Your refactored version does more work to give you a more general and robust version of categorization. You may not expect every implementer of a feature generator to go through this much work for a simple transform, but because you've factored out this functionality to shared, reusable code, they don't have to. Any feature-generation functionality needing to categorize values according to a list of thresholds can now call this function. The transform from listings 4.13 and 4.14 can now be replaced with the very simple version in listing 4.16. You still have a relatively complex implementation of categorization in listing 4.15, but now, that complex implementation has been factored out to a separate component, which is more general and reusable. As you can see in the next listing, the callers of that functionality, like this transform function, can be quite simple.

**Listing 4.16 Refactored categorization transform**

```
import CategoricalTransforms.categorize

private def transform(lengthFeature: IntFeature): IntFeature = {
  val squawkLengthCategory = categorize(Thresholds)
  ➥ (lengthFeature.value)
  IntFeature("squawkLengthCategory", squawkLengthCategory)
}
```

**Creates the categorization function and applies it to the value for categorization**

Once you have dozens of categorical features, this sort of design strategy will make your life a lot easier. Categorization is now simple to plug in and easy to refactor should you decide to change how you want it implemented.

### 4.5.2 Feature set composition

You've seen how you can choose among the features you produced, but there's actually a zeroth step that's necessary in some machine learning systems. Before you even

begin the process of feature generation, you may want to choose which feature generators should be executed. Different models need different features provided to them. Moreover, sometimes you need to apply specific overrides to your normal usage of data because of business rules, privacy concerns, or legal reasons.

In the case of Pidg'n, you have some unique challenges due to your global scale. Different regions have different regulatory regimes governing the use of their citizens' data. Recently, a new government has come to power in the rainforests of Panama.

The new minister of commerce, an implacable poison-dart frog, has announced new regulation restricting the use of social-media user data for non-rainforest purposes. After consultation with your lawyers, you decide that the new law means that features using data from rainforest users should only be used in the context of models to be applied on recommendations for residents of the rainforest.

Let's look at what impact this change might have on your codebase. To make things a bit more concise, let's define a simple trait to allow you to make simplified generators quickly. This will be a helper to allow you to skip over generator-implementation details that aren't relevant to feature-set composition. The next listing defines a stub feature generator that returns random integers.

**Listing 4.17 A stub feature-generator trait**

```
trait StubGenerator extends Generator {
  def generate(squawk: Squawk) = {
    IntFeature("dummyFeature", Random.nextInt())
  }
}
```

Implementation of the generate method for implementers of trait to use

Returns random integers

Using this simple helper trait, you can now explore some of the possible impacts that the rainforest data-usage rules might have on your feature-generation code. Let's say the code responsible for assembling your feature generators looks like the following listing.

**Listing 4.18 Initial feature set composition**

```
object SquawkLanguage extends StubGenerator {}

object HasImage extends StubGenerator {}

object UserData extends StubGenerator {}

val featureGenerators = Set(SquawkLanguage, HasImage, UserData)
```

Normal feature generator about the language the squawk was written in

Normal feature generator about whether the squawk contains an image

User-data feature generator that must be changed

Set of all the feature generators to execute to produce data

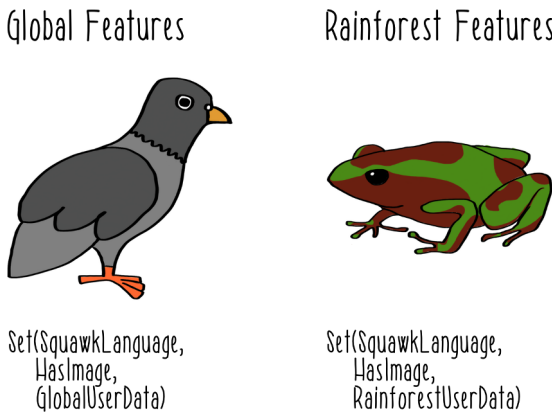



Figure 4.2 Multiple feature-generator sets

Now you need to restructure this code to have one feature set produced for your normal, global models and one feature set for your rainforest models, as shown in figure 4.3. The following listing shows an approach to defining these two different sets of feature generators.

**Listing 4.19 Multiple feature sets**

```
object GlobalUserData extends StubGenerator {}

object RainforestUserData extends StubGenerator {}

val globalFeatureGenerators = Set(SquawkLanguage, HasImage,
➥ GlobalUserData)

val rainforestFeatureGenerators = Set(SquawkLanguage, HasImage,
➥ RainforestUserData)
```

User-data feature generator that will only access non-rainforest data

User-data feature generator that will only access rainforest data

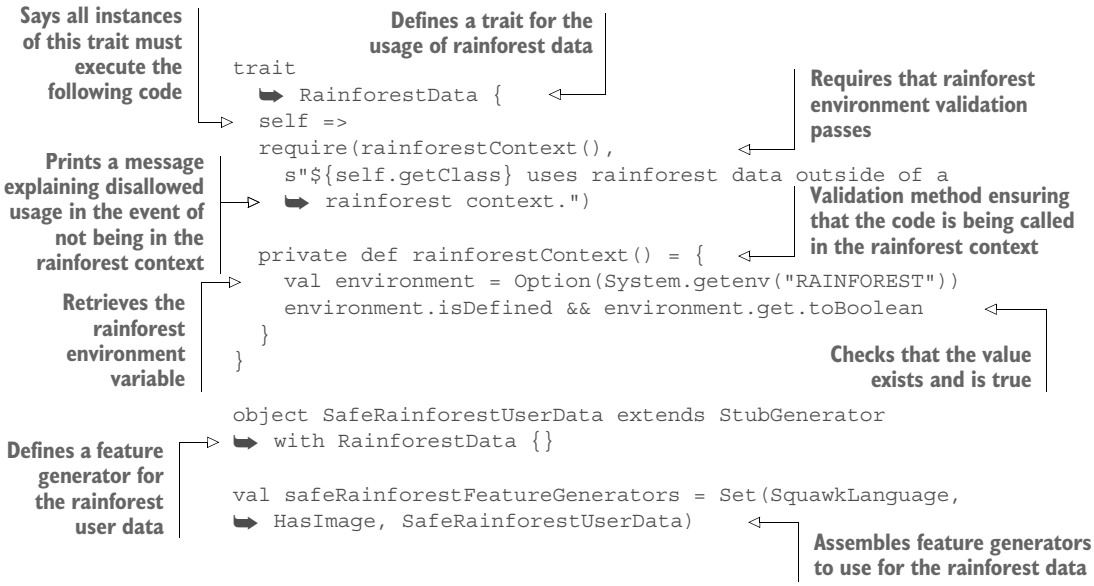
Set of features available to be used on global models

Set of features available to be used on rainforest models

You could stop with this implementation if you chose. As long as the rainforest feature generators are being used for rainforest models, you've done what the frog asked. But there are reasons to keep working on this problem. Machine learning systems are incredibly complicated to implement. Common feature-generation functionality can get reused in all sorts of places. The implementation in listing 4.19 is correct, but with Pidg'n's rapid growth, new engineers unfamiliar with this data-usage issue might refactor this code in such a way as to misuse rainforest feature data.

Let's see if you can make misusing this data even harder by defining a trait that allows you to mark code as having rainforest user data in it.

**Listing 4.20 Ensuring correct usage of rainforest data**




```
trait
  ➥ RainforestData {
  self =>
  require(rainforestContext(),
    s"${self.getClass} uses rainforest data outside of a
    ➥ rainforest context.")

  private def rainforestContext() = {
    val environment = Option(System.getenv("RAINFOREST"))
    environment.isDefined && environment.get.toBoolean
  }
}

object SafeRainforestUserData extends StubGenerator
➥ with RainforestData {}

val safeRainforestFeatureGenerators = Set(SquawkLanguage,
➥ HasImage, SafeRainforestUserData)
```

This code will throw an exception unless you've explicitly defined an environment variable `RAINFOREST` and set it to `TRUE`. If you want to see this switch in action, you can export that variable in a terminal window, if you're using macOS or Linux.

**Listing 4.21 Exporting an environment variable**

```
export RAINFOREST=TRUE
```

Then you can execute the code from listing 4.20 again, in the same terminal window, without getting exceptions. That's similar to how you can use this in your production feature-generation jobs. Using any of several different mechanisms in your configuration, build, or job-orchestration functionality, you can ensure that this variable is set properly for rainforest feature-generation jobs and not set for global feature-generation jobs. A new engineer creating a new feature-generation job for some other purpose would have no reason to set this variable. If that engineer misused the rainforest feature generator, that misuse would immediately manifest the first time the job was executed in any form.

### Configuration

Using environment variables is one of many different methods to configure components of your machine learning system. It has the advantage of being simple to get started with and broadly supported.

**(continued)**

As your machine learning system grows in complexity, you'll want to ensure that you have a well-thought-out plan for dealing with configuration. After all, properties of your machine learning system set as configurations can determine a lot about whether it remains responsive in the face of errors or changes in load. Part 3 of this book addresses most of these issues, where we consider the challenges of operating a machine learning system. The good news is that you'll find a lot of versatile tools from the Scala and big data ecosystems that will help you tame some of the complexity of dealing with configurations.

## 4.6 Applications

You're probably not an arboreal animal, and you may not even operate a microblogging service. But if you're doing machine learning, you're probably building features at some point.

In advertising systems, you can build features that capture users' past interactions with various types of products. If a user spends all afternoon looking at different laptops, you probably want to show them an ad for a laptop or maybe a case, but an ad for a sweater wouldn't make a lot of sense. That feature about which types of products the user had been looking at would help the machine-learned model figure that out and make the right recommendation.

At a political polling organization, you could build features pertaining to the demographics of different voters. Things like the average income, education, and home property value could be encoded into features about voting districts. Then those features could be used to learn models about which party a given voting district is likely to vote for.

The applications of features are as endless as the applications of machine learning as a technique. They allow you to encode human intelligence about the problem in a way that a model-learning algorithm can use that intelligence. Machine learning systems are not black-box systems that perform magic tricks. You, the system developer, are the one instructing it how to solve the problem, and features are a big part of how you encode that information.

## 4.7 Reactivities

This chapter covered a lot, but if you're still interested in learning more about features, there's definitely more to explore. Here are some reactivities to take you even deeper into the world of features:

- *Implement two or more feature extractors of your own.* To do this, you'll probably want to choose some sort of base dataset to work with. If you don't have anything meaningful at hand, you can often use text files and then extract features from the text. Spark has some basic text-processing functionality built in, which you may find helpful. Alternatively, random numbers organized into tabular data can work just

as well for an activity like this. If you do want to use real data, the UCI Machine Learning Repository at https://archive.ics.uci.edu/ml/index.php is one of the best sources of datasets. Whatever data you use, the point is to decide for yourself what might be some interesting transformations to apply to this dataset.

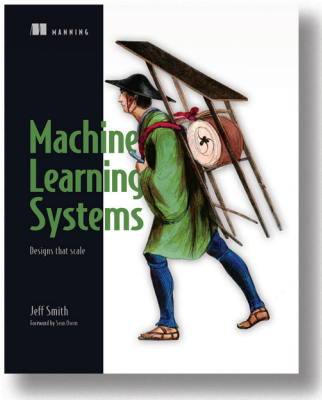
- *Implement feature-selection functionality.* Using the feature extractors you created in the previous reactivity (or some other extractors), define some basis for including or excluding a given feature within the final output. This could include criteria like the following:
  - Proportion of nonzero values.
  - Number of distinct values.
  - Externally defined business rule/policy. The goal is to ensure that the instances produced by your feature-extraction functionality only include the features that you define as valid.
- *Evaluate the reactivity of an existing feature-extraction pipeline.* If you did the previous two exercises, you can evaluate your own implementation. Alternatively, you can examine examples from open source projects like Spark. As you examine the feature-extraction pipeline, ask yourself questions like the following:
  - Can I find the feature-transform function? Is it implemented as a pure function, or does it have some sort of side effects? Can I easily reuse this transform in other feature extractors?
  - How will bad inputs be handled? Will errors be returned to the user?
  - How will the pipeline behave when it has to handle a thousand records? A million? A billion?
  - What can I discern about the feature extractors from the persisted output? Can I determine when the features were extracted? With which feature extractors?
  - How could I use these feature extractors to make a prediction on a new instance of unseen data?

## 4.8 Summary

- Like chicks cracking through eggs and entering the world of real birds, features are our entry points into the process of building intelligence into a machine learning system. Although they haven't always gotten the attention they deserve, features are a large and crucial part of a machine learning system.
- It's easy to begin writing feature-generation functionality. But that doesn't mean your feature-generation pipeline should be implemented with anything less than the same rigor you'd apply to your real-time predictive application. Feature-generation pipelines can and should be awesome applications that live up to all the reactive traits.
- Feature extraction is the process of producing semantically meaningful, derived representations of raw data.

- Features can be transformed in various ways to make them easier to learn from.
- You can select among all the features you have to make the model-learning process easier and more successful.
- Feature extractors and transformers should be well structured for composition and reuse.
- Feature-generation pipelines should be assembled into a series of immutable transformations (pure functions) that can easily be serialized and reused.
- Features that rely on external resources should be built with resilience in mind.

We're not remotely done with features. In chapter 5, you'll use features in the learning of models. In chapter 6, you'll generate features when you make predictions about unseen data. Beyond that, in part 3 of the book, we'll get into more-advanced aspects of generating and using features.




Machine learning applications autonomously reason about data at massive scale. It's important that they remain responsive in the face of failure and changes in load. And the best way to keep applications responsive, resilient, and elastic is to incorporate reactive design. But machine learning systems are different than other applications when it comes to testing, building, deploying, and monitoring. They also have unique challenges when you need to change the semantics or architecture of the system. To make machine learning systems reactive, you need to understand both reactive design patterns and modern data architecture patterns.

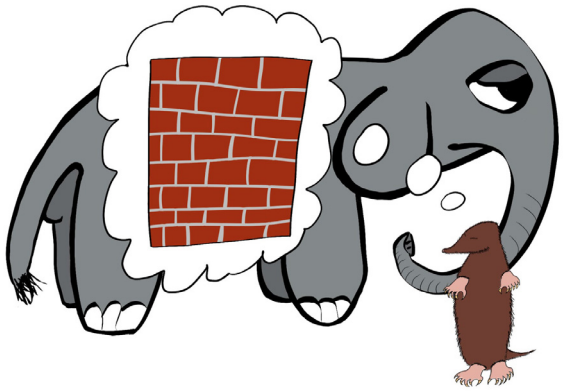
*Machine Learning Systems* teaches you how to implement reactive design solutions in your machine learning systems to make them as reliable as a well-built web app. This example-rich guide starts with an overview of machine learning systems while focusing on where reactive design fits. Then you'll discover how to develop design patterns to implement and coordinate ML subsystems. Using Scala and powerful frameworks such as Spark, MLlib, and Akka, you'll learn to quickly and reliably move from a single machine to a massive cluster. Finally, you'll see how you can operate a large-scale machine learning system over time. By the end, you'll be employing the principles of reactive systems design to build machine learning applications which are responsive, resilient, and elastic.

***What's inside:***

- Functional programming for distributed systems
- Reactive techniques like futures, actors, and supervision
- Spark and MLlib, and Akka
- Scala-based examples
- Predictive microservices
- Data models for uncertain data
- Design patterns for machine learning systems

Readers should have intermediate skills in Java or Scala. No previous machine learning experience is required.

# Generating Synonyms



*"The elephant is much like a wall, broad and solid."*

Now that we've a handle on some core techniques for working with language, it's time to consider a broad and powerful set of techniques that can be used for all sorts of language problems: deep neural networks. This chapter and the ones that follow use deep learning to model language. This chapter's part of a larger exploration into how to build systems for search applications. In particular, it focuses on how to build a scalable, maintainable system for matching words up as synonyms. To build this system, you'll learn about how to use word2vec, a powerful and widely used technique for representing data about language. Like the side of an elephant, the concepts in this chapter provide a solid base for the remainder of our explorations.

Chapter 2 from *Deep Learning for Search*
by Tommaso Teofili.

# *Generating synonyms*

***This chapter covers***

- Why and how synonyms are used in search
- A brief introduction to Apache Lucene
- Fundamentals of feed-forward neural networks
- Using a word2vec algorithm
- Generating synonyms using word2vec

Chapter 1 gave you a high-level overview of the kinds of possibilities that open up when deep learning is applied to search problems. Those possibilities include using deep neural networks to search for images via a text query based on its content, generating text queries in natural language, and so on. You also learned about the basics of search engines and how they conduct searches from queries and deliver relevant results. You're now ready to start applying deep neural networks to solve search problems.

In this chapter, we'll begin with a shallow (not deep) neural network that can help you identify when two words are similar in semantics. This seemingly easy task is crucial for giving a search engine the ability to understand language.

In information retrieval, a common technique to improve the number of relevant results for a query is to use *synonyms*. Synonyms allow you to expand the number of potential ways a query or piece of indexed document is expressed. For example, you can express the sentence "I like living in Rome" as "I enjoy living in

the Eternal City": the terms *living* and *enjoying* as well as *Rome* and *the Eternal City* are semantically similar, so the information conveyed by both sentences is mostly the same. Synonyms could help with the problem discussed in chapter 1, of a librarian and a student looking for a book to understand one another. That's because using synonyms allows people to express the same concept in different ways—and still retrieve the same search results!

In this chapter, we'll start working with synonyms using *word2vec*, one of the most common neural network-based algorithms for learning word representations. Learning about *word2vec* will give you a closer look at how neural networks work in practice. To do this, you'll first get an understanding of how *feed-forward* neural networks work. Feed-forward neural networks, one of the most basic types of neural networks, are the basic building blocks of deep learning. Next, you'll learn about two specific feed-forward neural network architectures: skip-gram and continuous bag of words (CBOW). They make it possible to learn when two words are similar in meaning, and hence they're a good fit for understanding whether two given words are synonyms. You'll see how to apply them to improve search engine *recall* by helping the search engine avoid missing relevant search results.

Finally, you'll measure how much the search engine can be enhanced this way and what trade-offs you'll need to consider for production systems. Understanding these costs and benefits is important when deciding when and where to apply these techniques to real-life scenarios.

## 2.1 Introducing synonym expansion

In the previous chapter, you saw how important it is to have good algorithms for performing text analysis: these algorithms specify the way text is broken into smaller fragments or *terms*. When it comes to executing a query, the terms generated at indexing time need to match those extracted from the query. This matching allows a document to be found and then appear in the search results.

One of the most frequent hurdles that prevents matching is the fact that people can express a concept in multiple different ways. For example, "going for a walk in the mountains" can be also expressed using the words "hiking" or "trekking." If the author of the text to be indexed uses "hike," but the user doing the search enters "trek," the user won't find the document. This is why you need to make the search engine aware of synonyms.

We'll explain how you can use a technique called *synonym expansion* to make it possible to express the same information need in several ways. Although synonym expansion is a popular technique, it has some limitations: in particular, the need to maintain a dictionary of synonyms that will likely change over time and that often isn't perfectly suited to the data to be indexed (such dictionaries are often obtained from publicly available data). You'll see how you can use algorithms like word2vec to learn word representations that help generate synonyms accurately based on the data that needs to be indexed.

By the end of the chapter, you'll have a search engine that can use a neural network to generate synonyms that can then be used to *decorate* the text to be indexed. To

show how this works, we'll use an example in which a user sends the query "music is my aircraft" through the search engine user interface. (I'll explain why the user is using that particular query in a moment.) Figure 2.1 shows what you'll end up with.

Here are the major steps, as shown in the figure. In the search engine, the query is first processed by the text-analysis pipeline. A *synonym filter* in the pipeline uses a neural network to generate synonyms. In the example, the neural network returns *airplane*, *aeroplane*, and *plane* as synonyms of *aircraft*. The generated synonyms are then used together with the tokens from the user query to find matches in the inverted index. And finally, the search results are collected. That's the big picture. Don't worry: we'll now go through each step in detail.

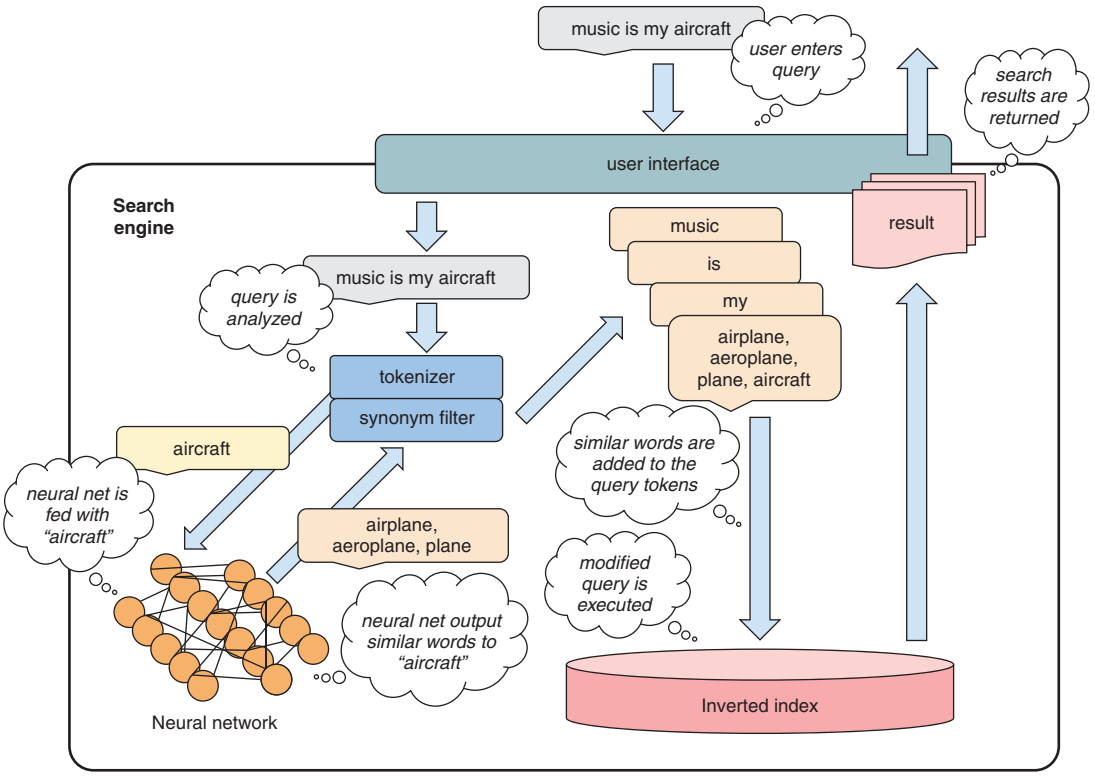



**Figure 2.1** **Synonym expansion at search time, with a neural network**

## 2.2 Why synonyms?

*Synonyms* are words that differ in spelling and pronunciation, but that have the same or a very close meaning. For example, *aircraft* and *airplane* are both synonyms of the word *plane*. In information retrieval, it's common to use synonyms to decorate text in order to increase the probability that an appropriate query will match. Yes, we're talking about probability here, because we can't anticipate all the possible ways of expressing an information need. This technique isn't a silver bullet that will let you *understand* all user queries, but it will reduce the number of queries that give too few or zero results.

Let's look at an example where synonyms can be useful. This has probably happened to you: you vaguely remember a short piece of a song, or you remember something about the meaning of a lyric, but not the exact wording from the song you have in mind. Suppose you liked a song whose chorus was along the lines of, "Music is my … *something*." What was it? A *car*? A *boat*? A *plane*? Now imagine you have a system that collects song lyrics, and you want users to be able to search through it. If you have synonym expansion enabled in the search engine, searching for "music is my plane" will yield the phrase you're looking for: "music is my aeroplane"! In this case, using synonyms let you find a relevant document (the song "Aeroplane" by Red Hot Chili Peppers) using a fragment and an incorrect word. Without synonym expansion, it would haven't been possible to retrieve this relevant response with queries like "music is my boat," "music is my plane," and "music is my car."

This is considered an improvement in recall. As briefly mentioned in chapter 1, *recall* is a number between 0 and 1 that equals the number of documents that are retrieved and relevant, divided by the number of relevant documents. If none of the retrieved documents are relevant, recall is 0. And if all the retrieved documents are relevant, recall is 1.

The overall idea of synonym expansion is that when the search engine receives a stream of terms, it can enrich them by adding their synonyms, if they exist, at the same position. In the "Aeroplane" example, synonyms of the query terms have been expanded: they were silently decorated with the word *aeroplane* at the same position as *plane* in the stream of text, see figure 2.2.

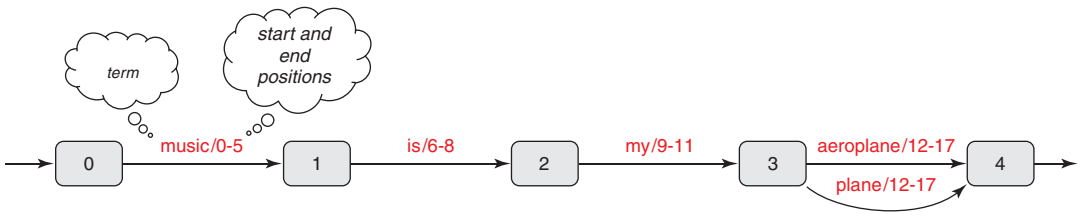



**Figure 2.2** **Synonym expansion graph**

You can apply the same technique during indexing of the "Aeroplane" lyrics. Expanding synonyms at indexing time will make indexing slightly slower (because of the calls to word2vec), and the index will inevitably be bigger (because it will contain more terms to store). On the plus side, searching will be faster because the word2vec call won't happen during search. Deciding whether to do index-time or search-time synonym expansion may have a noticeable impact on the performance of the system as its size and load grow.

Now that you've seen why synonyms are useful in the context of search, let's look at how to implement synonym expansion, first by using common techniques and then by using word2vec. This will help you appreciate the advantages of using the latter over the former.

## 2.3 Vocabulary-based synonym matching

Let's start by seeing how to implement a search engine with synonym expansion enabled at indexing time. The simplest and most common approach for implementing synonyms is based on feeding the search engine a vocabulary that contains the mapping between all the words and their related synonyms. Such a vocabulary can look like a table, where each key is a word and the corresponding values are its synonyms:

```
aeroplane -> plane, airplane, aircraft
boat -> ship, vessel
car -> automobile
...
```

Imagine that you feed the lyrics of "Aeroplane" into the search engine for indexing, and you use synonym expansion with the previous vocabulary. Let's pick the chorus of the song—"music is my aeroplane"—and see how synonym expansion handles it. You have a simple text-analysis pipeline composed of a tokenizer, which creates a token every time it encounters a white space, resulting in creating a token for each of the words in the sentence. The index-time text-analysis pipeline will thus create these tokens. Then you'll use a *token filter* for synonym expansion: for each received token, it will look at the vocabulary of synonyms and see if any of the keys (aeroplane, boat, car) is equal to the token text. The posting list for the fragment "music is my aeroplane" (sorted in ascending alphabetical order) will look like table 2.1.

**Table 2.1 Posting list for the fragment "music is my aeroplane"**

| Term | DOC(POS) |
|---|---|
| aeroplane | 1(12,17) |
| aircraft | 1(12,17) |
| airplane | 1(12,17) |
| is | 1(6,8) |
| music | 1(0,5) |
| my | 1(9,11) |
| plane | 1(12,17) |

This particular posting list also records information about the position of the occurrence of a term in a specific document. This information helps you visualize the fact that the terms *plane*, *airplane*, and *aircraft*, which weren't included in the original text fragment, were added to the index with the same position information attached to the original term (*aeroplane*).

You can record the *positions* of the terms in an inverted index in order to reconstruct the order in which a term appears in the text of a document. If you look at the inverted index table and pick the terms that have the lower positions in ascending order, you'll get *music is my aeroplane/aircraft/airplane/plane*. The synonyms can be seamlessly replaced with one another, so in the index you can imagine having four different pieces of text: *music is my aeroplane*, *music is my aircraft*, *music is my airplane*, and *music is my plane*.

It's important to emphasize that although you found four different forms in which the sentence can be indexed and searched, if any of them matches, only one document will be returned by a search engine: they all reference `DOC 1 in the posting list.

Now that you understand how synonyms can be indexed into the search engine, you're ready to try things out and build your first Apache Lucene-based search engine that indexes lyrics, setting up proper text analysis with synonym expansion at indexing time.

**NOTE** Going forward, I'll use *Lucene* and *Apache Lucene* interchangeably, but the proper trademarked name is Apache Lucene.

### 2.3.1 A quick look at Apache Lucene

I'll briefly introduce Lucene before diving into synonym expansion. This will allow you to focus more on the concepts rather than on the Lucene APIs and implementation details.

#### Obtaining Apache Lucene

You can download the latest release of Apache Lucene at https://lucene.apache.org/core/mirrors-core-latest-redir.html?. You can download either a binary package (.tgz or .zip) or the source release. The binary distribution is recommended if you just want to use Lucene within your own project; the .tgz/.zip package contains the jar files of the Lucene components. Lucene is made of various artifacts: the only mandatory one is `lucene-core`, and the others are optional parts that you can use if needed.

You can find the basics you need to know to get started with Lucene in the official documentation, available at https://lucene.apache.org/core/7_4_0/index.html. The source package is suitable for developers who want to look at the code or enhance it. (Patches for improvements, new features, bug fixes, documentation, and so on are always welcome at https://issues.apache.org/jira/browse/LUCENE.) If you use a build tool like Maven, Ant, or Gradle, you can include Lucene in your project, because all the components are released in public repositories like Maven Central (http://mng.bz/vN1x).

Apache Lucene is an open source search library written in Java, licensed under Apache License 2. In Lucene, the main entities to be indexed and searched are represented by `Document`s. A `Document`, depending on your use case, can represent anything: a page, a book, a paragraph, an image, and so on. Whatever it is, that's what

you'll get in your search results. A `Document` is composed of a number of `Field`s, which can be used to capture different portions of the `Document`. For example, if your document is a web page, you can think of having a separate `Field` for the page title, the page contents, the page size, the creation time, and so on. The main reasons for the existence of fields are that you can do the following:

- Configure per-field text-analysis pipelines
- Configure indexing options, such as whether to store in the posting list the term positions or the value of the original text each term refers to

A Lucene search engine can be accessed via a `Directory`: a list of files where the inverted indexes (and other data structures, used for example to record positions) are persisted. A view on a `Directory` for reading an inverted index can be obtained by opening an `IndexReader`:

```
Path path = Paths.get("/home/lucene/luceneidx");

Directory directory = FSDirectory.open(path);

IndexReader reader = DirectoryReader.open(directory);
```

**Target path where the inverted indexes are stored on the filesystem**

**Opens a Directory on the target path**

**Obtains a read-only view of the search engine via an IndexReader**

You can also use an `IndexReader` to obtain useful statistics for an index, such as the number of documents currently indexed, or if there are any documents that have been deleted. You can also obtain statistics for a field or a particular term. Also, if you know the *identifier* of the document you want to retrieve, you can get `Document`s from an `IndexReader` directly:

```
int identifier = 123;
Document document = reader.document(identifier);
```

An `IndexReader` is needed in order to search as it allows reading an index. Therefore you need an `IndexReader` to create an `IndexSearcher`. An `IndexSearcher` is the entry point for performing search and collecting results; the queries that will be performed via such an `IndexSearcher` will run on the index data, exposed by the `IndexReader`.

Without getting too much into coding queries programmatically, you can run a user-entered query using a `QueryParser`. You need to specify (search-time) text analysis when searching. In Lucene, the text-analysis task is performed by implementing the Analyzer API. An `Analyzer` can be made up of a `Tokenizer` and, optionally, `TokenFilter` components; or you can use out-of-the-box implementations, as in this example:

```
QueryParser parser = new QueryParser("title",
   new WhitespaceAnalyzer());
Query query = parser.parse("+Deep +search");
```

**Creates a query parser for the "title" field with a `WhitespaceAnalyzer`**

**Parses the user-entered query and obtains a Lucene Query**

In this case, you tell the query parser to split tokens when they find a whitespace and run queries against the field named `title`. Suppose a user types in the query "+Deep +search". You pass it to the `QueryParser` and obtain a Lucene `Query` object. Now you can run the query:

```
TopDocs hits = searcher.search(query, 10);

for (int i = 0; i < hits.scoreDocs.length; i++) {
  ScoreDoc scoreDoc = hits.scoreDocs[i]; #C
  Document doc = reader.document(scoreDoc.doc);
  System.out.println(doc.get("title") + " : " + scoreDoc.score);
}
```

**Performs the query against the IndexSearcher, returning the first 10 documents**

**Iterates over the results**

**Retrieves a ScoreDoc, which holds the returned document identifier and its score (given by the underlying retrieval model)**

**Outputs the value of the title field of the returned document**

**Obtains a Document in which you can inspect fields using the document ID**

If you run this, you'll get no results, because haven't indexed anything yet! Let's fix this and learn how to index `Document`s with Lucene. First you have to decide which fields to put into your documents and how their (index-time) text-analysis pipelines should look. We'll use books for this example. Assume you want to remove some useless words from the books' contents while using a simpler text-analysis pipeline for the title that doesn't remove anything.

**Listing 2.1 Building per-field analyzers**

```
Map<String, Analyzer> perFieldAnalyzers = new HashMap<>();

CharArraySet stopWords = n`ew CharArraySet(Arrays
    .asList("a", "an", "the"), true);

perFieldAnalyzers.put("pages", new StopAnalyzer(stopWords));

perFieldAnalyzers.put("title", new WhitespaceAnalyzer());

Analyzer analyzer = new PerFieldAnalyzerWrapper(
    new EnglishAnalyzer(), perFieldAnalyzers);
```

**Sets up a map where the keys are the names of fields and the values are the Analyzers to be used for the fields**

**Creates a stop-word list of the tokens to remove from the books' contents while indexing**

**Uses a StopAnalyzer with the given stop words for the pages field**

**Uses a WhitespaceAnalyzer for the title field**

**Creates a per-field Analyzer, which also requires a default analyzer (EnglishAnalyzer in this case) for any other field that may be added to a Document**

The inverted indexes for a Lucene-based search engine are written on disk in a `Directory` by an `IndexWriter` that will persist `Document`s according to an `IndexWriterConfig`. This config contains many options, but for you the most important bit is the required index-time analyzer. Once the IndexWriter is ready, you can create `Document`s and add `Field`s.

**Listing 2.2 Adding documents to the Lucene index**

```
IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter writer = new IndexWriter(directory, config);

Document dl4s = new Document();
dl4s.add(new TextField("title", "DL for search",
   Field.Store.YES));
dl4s.add(new TextField("page", "Living in the information age ...",
   Field.Store.YES));

Document rs = new Document();
rs.add(new TextField("title", "Relevant search", Field.Store.YES));
rs.add(new TextField("page", "Getting a search engine to behave ...",
   Field.Store.YES));

writer.addDocument(dl4s);
writer.addDocument(rs);
```

Creates a configuration for indexing

Creates an `IndexWriter` to write `Documents` into a `Directory`, based on an `IndexWriterConfig`

Creates `Document` instances

Adds `Fields`, each of which has a name, a value, and an option to store the value with the terms

Adds `Documents` to the search engine

After you've added a few documents to the `IndexWriter`, you can persist them on the filesystem by issuing a `commit`. Until you do, new `IndexReader`s won't see the added documents:

```
writer.commit();
writer.close();
```

Commits the changes

Closes the `IndexWriter` (releases resources)

Run the search code again, and this is what you'll get:

```
Deep learning for search : 0.040937614
```

The code finds a match for the query "+Deep +search" and prints its title and score.

Now that you've been introduced to Lucene, let's get back to the topic of synonym expansion.

### 2.3.2 Setting up a Lucene index with synonym expansion

You'll first define the algorithms to use for text analysis at indexing and search time. Then you'll add some lyrics to an inverted index. In many cases, it's a good practice to use the same tokenizer at both indexing and search time, so the text is split according to the same algorithm. This makes it easier for queries to match fragments of documents. You'll start simple and set up the following:

- A search-time `Analyzer` that uses a tokenizer that splits tokens when it encounters a whitespace character (also called a *whitespace tokenizer*)
- An index-time `Analyzer` that uses the whitespace tokenizer and a synonym filter

The reason to do this is that you don't need synonym expansion at both query time and index time. For two synonyms to match, it's sufficient to do expansion once.

Assuming you have the two synonyms *aeroplane* and *plane*, the following listing will build a text-analysis chain that can take a term from an original token (for example,

*plane*) and generate another term for its synonym (for example, *aeroplane*). Both the original and the new term will be generated.

**Listing 2.3 Configuring synonym expansion**

```
SynonymMap.Builder b = new SynonymMap.Builder();
builder.add(new CharsRef("aeroplane"), new CharsRef("plane"), true);
final SynonymMap map = b.build();

Analyzer indexTimeAnalyzer = new Analyzer() {
  @Override
  protected TokenStreamComponents createComponents(
        String fieldName) {
    Tokenizer tokenizer = new WhitespaceTokenizer();
    SynonymGraphFilter synFilter = new
        SynonymGraphFilter(tokenizer, map, true);
    return new TokenStreamComponents(tokenizer, synFilter);
  }
};

Analyzer searchTimeAnalyzer = new WhitespaceAnalyzer();
```

Programmatically defines synonyms

Creates a custom Analyzer, for indexing

Creates a synonym filter that receives terms from the whitespace tokenizer and expands synonyms according to a map word, ignoring case

Whitespace analyzer for the search time

This simplistic example creates a synonym vocabulary with just one entry. Normally you'll have more entries, or you'll read them from an external file so you don't have to write the code for each synonym.

You're just about ready to put some song lyrics into the index using the `indexTimeAnalyzer`. Before doing that, let's look at how song lyrics are structured. Each song has an author, a title, a publication year, lyrics text, and so on. As I said earlier, it's important to examine the data to be indexed, to see what kind of data you have and possibly come up with reasoned text-analysis chains that you expect to work well on that data. Here's an example:

```
author: Red Hot Chili Peppers
title: Aeroplane
year: 1995
album: One Hot Minute
text: I like pleasure spiked with pain and music is my aeroplane ...
```

Can you keep track of such a structure in a search engine? Would doing so be useful?

In most cases, it's handy to keep a lightweight document structure, because each part of it conveys different semantics, and therefore different requirements in the way it's hit by search. For example, the year will always be a numeric value; it makes no sense to use a whitespace tokenizer on it, because it's unlikely that any whitespace will appear in that field. For all the other fields, you can probably use the Analyzer you defined earlier for indexing. Putting it all together, you'll have multiple inverted indexes (one for each attribute) that address indexing of different parts of a document, all within the same search engine; see figure 2.3.

With Lucene, you can define a field for each of the attributes in the example (author, title, year, album, text). You specify that you want a separate `Analyzer` for the

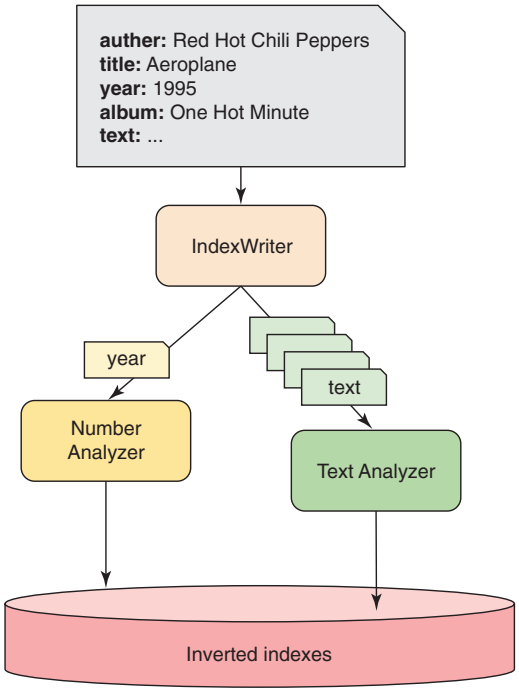



**Figure 2.3** Splitting portions of the text depending on the type of data

year field that doesn't touch the value; for all the other values, it will use the previously defined indexTimeAnalyzer with synonym expansion enabled.

**Listing 2.4 Separate analysis chains for indexing and search**

```
Directory directory = FSDirectory.open(Paths.get("/path/to/index"));

Map<String, Analyzer> perFieldAnalyzers = new HashMap<>();

perFieldAnalyzers.put("year", new KeywordAnalyzer());

Analyzer analyzer = new PerFieldAnalyzerWrapper(
    indexTimeAnalyzer, perFieldAnalyzers);

IndexWriterConfig config = new IndexWriterConfig(analyzer);

IndexWriter writer = new IndexWriter(directory, config);
```

Creates a map whose keys are the names of the fields and the values in the corresponding analysis chain to be used

Opens a Directory for indexing

Sets up a different analyzer (keyword; doesn't touch the value) for the year

Creates a wrapping analyzer that can work with per-field analyzers

Build all the above in a configuration object

Creates an IndexWriter to be used for indexing

This mechanism allows indexing to be flexible in the way content is analyzed before being written into the inverted indexes; it's common to play with different `Analyzer`s for different portions of `Document`s and to change them several times before finding the best combination for a data corpus. Even then, in the real world, it's likely that such configurations will need adjustments over time. For instance, you may only index English songs and then, at some later point, begin to add songs in English and Chinese

languages. In this case, you'll have to adjust the analyzers to work with both languages. (For example, you can't expect a whitespace tokenizer to work well on Chinese, Japanese, and Korean (CJK) languages, where words often aren't separated by a space).

Let's put your first document into the Lucene index.

**Listing 2.5 Indexing documents**

```
Document aeroplaneDoc = new Document();
aeroplaneDoc.add(new Field("title", "Aeroplane", type));
aeroplaneDoc.add(new Field("author", "Red Hot Chili Peppers", type));
aeroplaneDoc.add(new Field("year", "1995", type));
aeroplaneDoc.add(new Field("album", "One Hot Minute", type));
aeroplaneDoc.add(new Field("text",
    "I like pleasure spiked with pain and music is my aeroplane ...", type));

writer.addDocument(aeroplaneDoc);
writer.commit();
```

Creates a document for the song "Aeroplane"

Adds all the fields from the song lyrics

Adds the document

Persists the updated inverted index to the filesystem, making the changes durable (and searchable)

You create a document composed of multiple fields, one per song attribute, and then add it to the `writer`. In order to search, you open the `Directory` (again) and obtain a view on the index, an `IndexReader`, on which you can search via an `IndexSearcher`. To make sure the synonym expansion works as expected, enter the query with the word *plane*; you'll expect the "Aeroplane" song to be retrieved.

**Listing 2.6 Searching for the word plane**

```
IndexReader reader = DirectoryReader.open(directory);

IndexSearcher searcher = new IndexSearcher(reader);

QueryParser parser = new QueryParser(null, searchTimeAnalyzer);

Query query = parser.parse("plane");

TopDocs hits = searcher.search(query, 10);

for (int i = 0; i < hits.scoreDocs.length; i++) {
    ScoreDoc scoreDoc = hits.scoreDocs[i];
    Document doc = searcher.doc(scoreDoc.doc);
    System.out.println(doc.get("title") + " by "
        + doc.get("author"));
}
```

Opens a view on the index

Instantiates a searcher

Creates a query parser that uses the search-time analyzer with the user-entered query to produce search terms

Transforms a user-entered query (as a String) into a proper Lucene query object using the QueryParser

Iterates over the results

Searches, and obtains the first 10 results

Obtains the search result

Outputs the title and author of the returned song

As expected, the result is as follows:

```
Aeroplane by Red Hot Chili Peppers
```

We've gone through a quick tour of how to set up text analysis for index and search, and how to index documents and retrieve them. You've also learned how to add the synonym expansion capability. But it should be clear that this code can't be maintained in real life:

- You can't write code for each and every synonym you want to add.
- You need a synonym vocabulary that can be plugged in and managed separately, to avoid having to modify the search code every time you need to update it.
- You need to manage the evolution of languages—new words (and synonyms) are added constantly.

A first step toward resolving these issues is to write the synonyms into a file and let the synonym filter read them from there. You'll do that by putting synonyms on the same line, separated by commas. You'll build the Analyzer in a more compact way, by using a builder pattern (see https://en.wikipedia.org/wiki/Builder_pattern).

**Listing 2.7 Feeding synonyms from a file**

```
Map<String, String> sffargs = new HashMap<>();
sffargs.put("synonyms", "synonyms.txt"); #A

CustomAnalyzer.Builder builder = CustomAnalyzer.builder()
    .withTokenizer(WhitespaceTokenizerFactory.class)
    .addTokenFilter(SynonymGraphFilterFactory.class, sffargs)
return builder.build();
```

Defines the file that contains the synonyms

Defines an analyzer

Lets the analyzer use a whitespace tokenizer

Lets the analyzer use a synonym filter

Set up synonyms in the synonyms file:

```
plane,aeroplane,aircraft,airplane
boat,vessel,ship
...
```

This way, the code remains unchanged regardless of any change in the synonyms file; you can update the file as much as you need to. Although this is much better than having to write code for synonyms, you don't want to write the synonyms file by hand, unless you know that you'll have just a few fixed synonyms. Fortunately, these days there's lots of open data that you can use for free or for a very low cost. A good, large resource for natural language processing in general is the WordNet project (http://wordnet.princeton.edu), a lexical database for the English language from Princeton University. You can take advantage of WordNet's large synonym vocabulary, which is constantly updated, and include it in your indexing analysis pipeline by downloading it as a file (for example, called synonyms-wn.txt) and specifying that you want to use the WordNet format.

**Listing 2.8 Using synonyms from WordNet**

```
Map<String, String> sffargs = new HashMap<>();
sffargs.put("synonyms", "synonyms-wn.txt");
sffargs.put("format", "wordnet");
CustomAnalyzer.Builder builder = CustomAnalyzer.builder()
    .withTokenizer(WhitespaceTokenizerFactory.class)
    .addTokenFilter(SynonymGraphFilterFactory.class, sffargs)
return builder.build();
```

Sets up a synonym file using the WordNet vocabulary

Specifies the WordNet format for the synonym file

With the WordNet dictionary plugged in, you have a very large, high-quality source of synonym expansion that should work well for English. But there are still a few problems. First, there's not a WordNet-type resource for every language. Second, even you stick to English, the synonym expansion for a word is based on its *denotation* as defined by the rules of English grammar and dictionaries; this doesn't take into account its *connotation* as defined by the context in which those words appear.

I'm describing the difference between what linguists define as a synonym, based on strict dictionary definitions (denotation), versus how people commonly use language and words in real life (connotation). In informal contexts like social networks, chatrooms, and meeting friends in real life, people may use two words as if they were synonyms even if, by grammar rules, they aren't synonyms. To handle this issue, word2vec will kick in and provide a more advanced level of search than just expanding synonyms based on the strict syntax of a language. You'll see that using word2vec enables you to build synonym expansions that are language agnostic; it learns from the data which words are similar, without caring too much about the language used and whether it's formal or informal. This is a helpful feature of word2vec: words with similar contexts are considered similar exactly because of their context. There's no grammar or syntax involved. For each word, word2vec looks at the surrounding words, assuming that semantically similar words will appear in similar contexts.

## 2.4 Generating synonyms

The main problem with the approach outlined so far is that synonym mappings are static and not bound to the indexed data. For example, in the case of WordNet, synonyms strictly obey English grammar semantics but don't take into account slang or informal contexts where words are often used as synonyms even if they aren't synonyms according to strict rules of grammar. Another example is acronyms used in chat sessions and emails. For instance, it's not uncommon to see acronyms like ICYMI ("in case you missed it") and AKA ("also known as") in email. *ICYMI* and *in case you missed it* can't be called synonyms, and you won't probably find them in a dictionary, but they mean the same thing.

One approach to overcome these limitations is to have a way to generate synonyms from the data to be ingested into the search engine. The basic concept is that it should be possible to extract the *nearest neighbors* of a word by looking at the context of the word, which means analyzing the patterns of surrounding words that occur together with the word itself. A nearest neighbor of a word in this case should be its synonym, even it's not strictly a synonym from the grammar perspective.

This idea that words that are used, and occur, in the same contexts tend to have similar meanings is called the *distributional hypothesis* (see https://aclweb.org/aclwiki/Distributional_Hypothesis) and is the basis of many deep learning algorithms for text representations. The interesting thing about this idea is that it disregards language, slang, style, and grammar: every bit of information about a word is inferred from the word contexts that appear in the text. Think, for example, of how words rep-

resenting cities (Rome, Cape Town, Oakland, and so on) are often used. Let's look at a few sentences:

- I like to live in Rome because …
- People who love surfing should go to Cape Town because …
- I would like to visit Oakland to see …
- Traffic is crazy in Rome …

Often the city names are used near the word *in* or a short distance from verbs like *live*, *visit*, and so on. This is the basic intuition behind the fact that the context provides a lot of information about each word.

With this in mind, you want to learn word representations for the words in the data to be indexed, so that you can generate synonyms from the data rather than manually building or downloading a synonym vocabulary. In the library example in chapter 1, I mentioned that it's best to have insight about what's in the library; with these additional insights, the librarian could help you more effectively. A student coming to the library could ask the librarian, say, for "books about artificial intelligence." Let's also suppose the library has only one book on the topic, and it's called *AI Principles*. If the librarian (or the student) were searching through book titles, they would miss this book, unless they knew that AI is an acronym (and, given previous assumptions, a synonym) of *artificial intelligence*. An assistant knowledgeable about these synonyms would be useful in this situation.

Let's imagine two hypothetical types of such an assistant: John, an English language expert who has studied English grammar and syntax for years; and Robbie, another student who collaborates weekly with the librarian and has the chance to read most of the books. John couldn't tell you that *AI* stands for *artificial intelligence*, because his background doesn't give him this information. Robbie, on the other hand, has far less formal knowledge of English, but he's an expert on the books in the library; he could easily tell you that *AI* stands for *artificial intelligence*, because he's read the book *AI Principles* and knows it's about the principles of artificial intelligence. In this scenario, John is acting like the WordNet vocabulary, and Robbie is the word2vec algorithm. Although John has proven knowledge of the language, Robbie may be more helpful in this particular situation.

In chapter 1, I mentioned that neural networks are good at learning representations (in this case, representations of words) that are sensitive to the context. That's the kind of capability you'll use with word2vec. In short, you'll use the word2vec neural network to learn a representation of the words that can tell you the most similar (or nearest neighbor) word for *plane*: *aeroplane*. Before we get deeper into that, let's take a closer look at one of the simplest form of neural networks: the *feed-forward neural network*. These are the basis for most more-complex neural network architectures.

## 2.5 Feed-forward neural networks

Neural networks are the key tool for neural search, and many neural network architectures extend from feed-forward networks. A *feed-forward neural network* is a neural net-

work in which information flows from the input layer to hidden layers, if any, and finally to the output layer; there are no loops, because the connections among neurons don't form a cycle. Think of it as a magic black box with inputs and outputs. The magic mostly happens inside the net, thanks to the way neurons are connected to each other and how they react to their inputs. If you were looking for a house to buy in a specific country, for instance, you could use the "magic box" to predict a fair price you could expect to pay for a specific house. As you can see in figure 2.4, the magic box would learn to make predictions using input features such as house size, location, and a rating given by the seller.

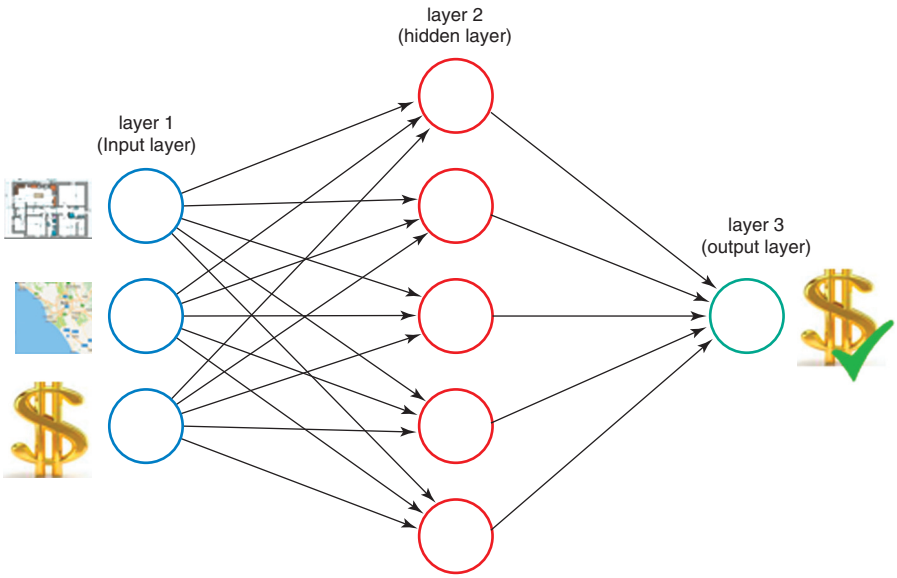



**Figure 2.4 Predicting price with a feed-forward neural network with three inputs, five hidden units, and one output unit**

A feed-forward neural network is composed of the following:

- *An input layer*—Responsible for gathering the inputs provided by the user. These inputs are usually in the form of real numbers. In the example of predicting a house price, you have three inputs: house size, location, and amount of money required by the seller. You'll encode these inputs as three real numbers, so the input you'll pass to the network will be a three-dimensional vector: `[size, location, price]`.
- *Optionally, one or more hidden layers*—Represents a more mysterious part of the network. Think of it as the part of the network that allows it to be so good at learning and predicting. In the example, there are five units in the hidden layer, all of which are connected to the units in the input layer and also to all the units in the output layer. The connectivity in the network plays a fundamen-

tal role in the network activity dynamics. Most of the time, all units in a layer ($x$) are fully connected (forward) to the units in the next layer ($x$+1).

- *An output layer*—Responsible for providing the final output of the network. In the price example, it will provide a real number representing what the network estimates the right price should be.

**NOTE** Usually, it's a good idea to scale inputs so they're more or less in the same range of values—for example, between -1 and 1. In the example, a house's size in square meters is between 10 and 200, and its price range is in the order of tens of thousands. Preprocessing the input data so it's all in similar ranges of values allows the network to learn more quickly.

### 2.5.1 How it works: Weights and activation functions

As you've seen, a feed-forward neural network receives inputs and produces outputs. The fundamental building blocks of these networks are called *neurons* (even though a brain neuron is much more complex). Every neuron in a feed-forward neural network

- Belongs to a layer
- Smooths each input by its incoming weight
- Propagates its output according to an activation function

Looking at the example of the feed-forward neural network in figure 2.5, the second layer is composed of only one neuron. This neuron receives input from three neurons in layer 1 and propagates output to only one neuron in layer 3. It has an associated activation function, and its incoming links with the previous layer have associated weights (often, real numbers between -1 and 1).

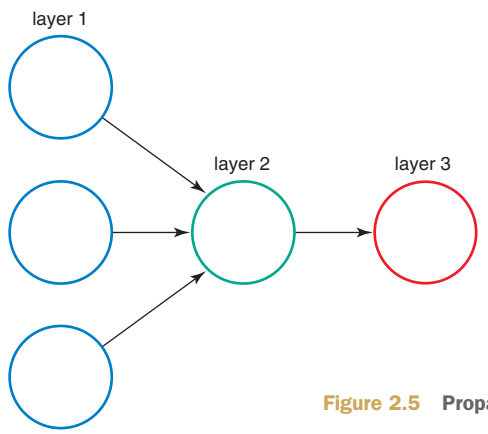



**Figure 2.5 Propagating signals through the network**

Let's assume that all the incoming weights of the neuron in layer 2 are set to 0.3 and that it receives from the first layer the inputs 0.4, 0.5, and 0.6. Each weight is mul-

tiplied by its input, and the results are summed together: $0.3 \times 0.4 + 0.3 \times 0.5 + 0.3 \times 0.6 = 0.45$. The activation function is applied to this intermediate result and then propagated to the outgoing links of the neuron. Common activation functions are hyperbolic tangent (`tanh`), `sigmoid`, and rectified linear unit (`ReLU`).

In the current example, let's use the *tanh* function. You'll have *tanh(0.45) = 0.4218990053*, so the neuron in the third layer will receive this number as an input on its only incoming link. The output neuron will perform exactly the same steps the neuron from layer 2 does, using its own weights. For this reason, these networks are called *feed forward*: each neuron transforms and propagates its inputs in order to feed the neurons in the next layer.

### 2.5.2 Backpropagation in a nutshell

In chapter 1, I mentioned that neural networks and deep learning belong to the field of machine learning. I also touched on the main algorithm used for training neural networks: backpropagation. In this section, we'll give it a somewhat closer look.

A fundamental point when discussing the rise of deep learning is related to how well and how quickly neural networks can learn. Although artificial neural networks are an old computing paradigm (circa 1950), they became popular again recently (around 2011) as modern computers' performance improved to a level that allowed neural nets to perform effective learning in a reasonable time.

In the previous section, you saw how a network propagates information from the input layer to the output layer in a feed-forward fashion. On the other hand, after a feed-forward pass, backpropagation lets the signal flow backward from the output layer to the input layer.

The values of the activations of the neurons in the output layer, generated by a feed-forward pass on a input, are compared the values in the desired output. This comparison is performed by a *cost function* that calculates a loss or cost and represents a measure of how much the network is wrong in that particular case. Such an error is sent backward through the incoming connections of the output neurons to the corresponding units in the hidden layer. You can see in figure 2.6 that the neuron in the output layer sends back its portion of error to the connected units in the hidden layer.

Once a unit receives an error, it updates its weights according to an *update algorithm*; usually, the algorithm used is *stochastic gradient descent*. This backward update of weights happens until the weights on the input layer connections are adjusted, and then the update stops. So, a run of backpropagation updates all the weights associated with the existing connections. The rationale behind this algorithm is that each weight is responsible for a portion of the error and, therefore, backpropagation tries to adjust such weights in order to reduce the error for that particular input/output pair.

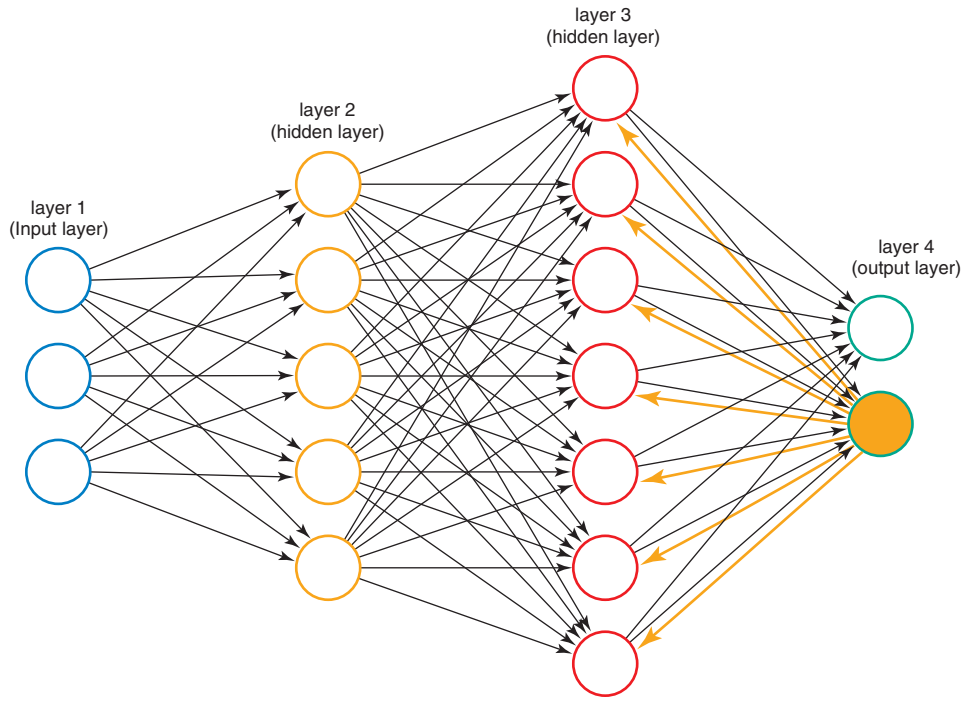



**Figure 2.6 Backpropagating signal from the output layer to the hidden layer**

The gradient descent algorithm (or any other update algorithm for adjusting the weights) decides *how* the weights are changed with respect to the portion of error each weight contributes.

There is a lot of math behind this concept, but you can think of it as if the cost function defines a shape like the one in figure 2.7, where the height of the hill defines the amount of error. A very low point corresponds to the combination of the neural network weights having a very low error:

- *Low*—The point with the lowest possible error, having optimal values for the neural network weights
- *High*—A point with high error; gradient descent tries to perform descent toward points with lower error

The coordinates of a point are given by the value of the weights in the neural network, so the gradient descent tries to find a value of the weights (a point) with very low error (a very low height) in the shape.

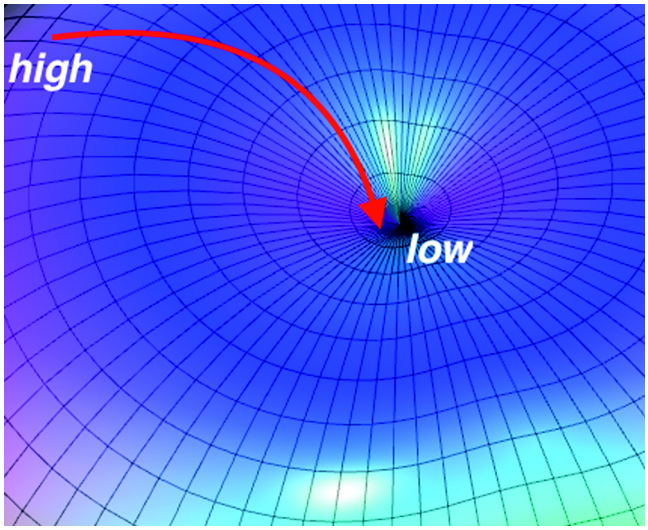



Figure 2.7 Geometric interpretation of backpropagation with gradient descent

## 2.6 Using word2vec

Now that you understand what a generic feed-forward network is, we can focus on a more specific neural network algorithm based on feed-forward neural networks: word2vec. Although its basics are fairly easy to understand, it's fascinating to see the good results (in terms of capturing the semantics of words in a text) you can achieve. But what does it do, and how is it useful for the synonym expansion use case?

Word2vec takes a piece text and outputs a series of vectors, one for each word in the text. When the output vectors of word2vec are plotted on a two-dimensional graph, vectors whose words are very similar in term of semantics are very close to one another. You can use a distance measures like the cosine distance to find the most similar words with respect to a given word. Thus, you can use this technique to find word synonyms. In short, in this section you'll set up a word2vec model, feed it the text of the song lyrics you want to index, get output vectors for each word, and use them to find synonyms.

Chapter 1 discussed using vectors in the context of search, when we talked about the vector space model and term frequency - inverted document frequency (TF-IDF). In a sense, word2vec also generates a vector space model whose vectors (one for each word) are weighted by the neural network during the learning process. Word vectors generated by algorithms like word2vec are often referred to as *word embeddings* because they map static, discrete, high-dimensional word representations (such as TF-IDF or one-hot encoding) into a different (continuous) vector space with fewer dimensions involved.

Let's get back at the example of the song "Aeroplane". If you feed its text to word2vec, you'll get a vector for each word:

```
0.7976110753441061, -1.300175666666296, i
-1.1589942649711316, 0.2550385962680938, like
-1.9136814615251492, 0.0, pleasure
```

```
-0.178102361461314, -5.778459658617458, spiked
0.11344064895365787, 0.0, with
0.3778008406249243, -0.11222894354254397, pain
-2.0494382050792344, 0.5871714329463343, and
-1.3652666102221962, -0.4866885862322685, music
-12.878251690899361, 0.7094618209959707, is
0.8220355668636578, -1.2088098678855501, my
-0.37314503461270637, 0.4801501371764839, aeroplane
...
```

You can see these in the coordinate plan shown in figure 2.8.

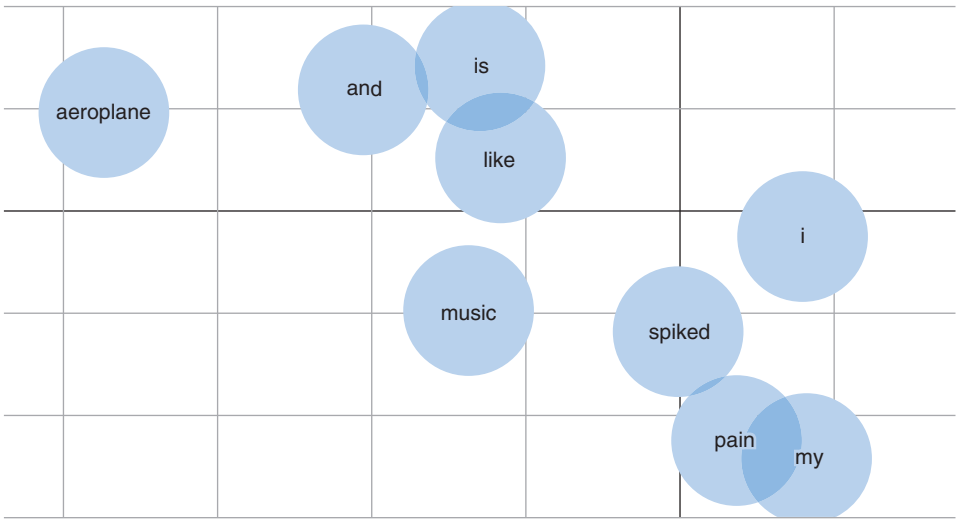



**Figure 2.8** **Plotted word vectors for "Aeroplane"**

In the example output, two dimensions were used so those vectors are more easily plottable on a graph. But in practice, it's common to use 100 or more dimensions, and to use a dimensionality-reduction algorithm like Principal Component Analysis or t-SNE to obtain two- or three-dimensional vectors that can be more easily plotted. (Using many dimensions lets you capture more information as the amount of data grows.) At this point, we won't discuss this tuning in detail, but we'll return to it later in the book as you learn more about neural networks.

Using cosine similarity to measure the distance among each of the generated vectors gives some interesting results:

```
music -> song, view
looking -> view, better
in -> the, like
sitting -> turning, could
```

As you can see, extracting the two nearest vectors for a few random vectors gives some good results and some, not so much:

- *Music* and *song* are very close semantically; you could even say they're synonyms. But the same isn't true for *view*.
- *Looking* and *view* are related, but *better* has nothing to do with *looking*.
- *In*, *the*, and *like* aren't close to each other.
- *Sitting* and *turning* are both verbs of the *ing* form, but their semantics are loosely coupled. *Could* is also a verb, but it doesn't have much else to do with *sitting*.

What's the problem? Isn't *word2vec* up to the task?

There are two factors at play:

- The number of dimensions (two) of the generated word vectors is probably too low.
- Feeding the word2vec model the text of a single song probably doesn't provide enough contexts for each of the words to come with an accurate representation. The model needs more examples of the contexts in which the words *better* and *view* occur.

Let's assume you again build the word2vec model, this time using 100 dimensions and a larger set of song lyrics taken from the Billboard Hot 100 Dataset (https://www.kaylinpavlik.com/50-years-of-pop-music):

```
music -> song, sing
view -> visions, gaze
sitting -> hanging, lying
in -> with, into
looking -> lookin, lustin
```

The results are much better and more appropriate: you could use almost all of them as synonyms in the context of search. You can imagine using such a technique at either query or indexing time. There would be no more dictionaries or vocabularies to keep up to date; the search engine could learn to generate synonyms from the data it handles.

You may have a couple of questions right about now: How does word2vec work? And how can you integrate it, in practice, into a search engine? The paper "Efficient Estimation of Word Representations in Vector Space"[1] describes two different neural network models for learning such word representations: *continuous bag of words* (CBOW) and *continuous skip-gram*. I'll discuss both of them, and how to implement them, in a moment. Word2vec performs unsupervised learning of word representations; the mentioned CBOW and skip-gram models just need to be fed a sufficiently large text, properly encoded.

The main concept behind word2vec is that the neural network is given a piece of text, which is split into fragments of a certain size (also called the *window*). Every fragment is

[1] Tomas Mikolov et al., 2013, https://arxiv.org/pdf/1301.3781.pdf.

fed to the network as a pair consisting of a *target word* and a *context*. In the case of figure 2.9, the target word is *aeroplane*, and the context consists of the words *music*, *is*, and *my*.

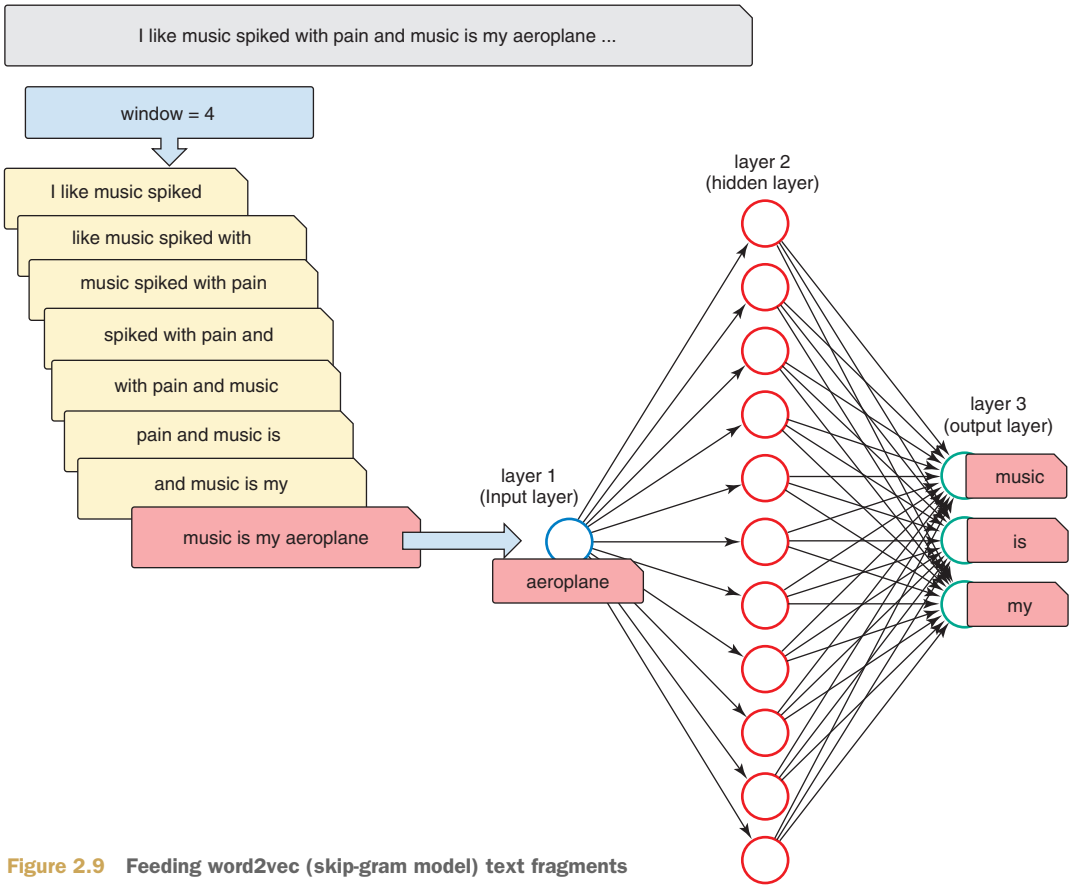



**Figure 2.9 Feeding word2vec (skip-gram model) text fragments**

The hidden layer of the network contains a set of weights (in this case, 11 of them—the number of neurons in the hidden layer) for each word. These vectors will be used as the word representations when learning ends.

An important trick about word2vec is that you don't care much about the outputs of the neural network. Instead, you extract the internal state of the hidden layer at the end of the training phase, which yields exactly one vector representations for each word.

During training, a portion of each fragment is used as target word, and the rest is used as context. With the CBOW model, the target word is used as the output of the network, and the remaining words of the text fragment (the context) are used as inputs. The opposite is true with the continuous skip-gram model: the target word is used as input and the context words as outputs (as in the example). In practice, both work well, but skip-gram is usually preferred because it works slightly better with infrequently used words.

For example, given the text "she keeps moet et chandon in her pretty cabinet let them eat cake she says" from the song "Killer Queen" (by the band Queen), and a window of 5, a word2vec model based on CBOW will receive a sample for each five-word fragment. For example, for the fragment | `she` | `keeps` | `moet` | `et` | `chandon` |, the input will consist of the words | `she` | `keeps` | `et` | `chandon` | and the output will consist of the word `moet`.

As you can see from figure 2.10, the neural network is composed of an input layer, a hidden layer, and an output layer. This kind of neural network, with one hidden layer, is referred to as *shallow*. Neural networks with more than one hidden layer are referred to as *deep*.

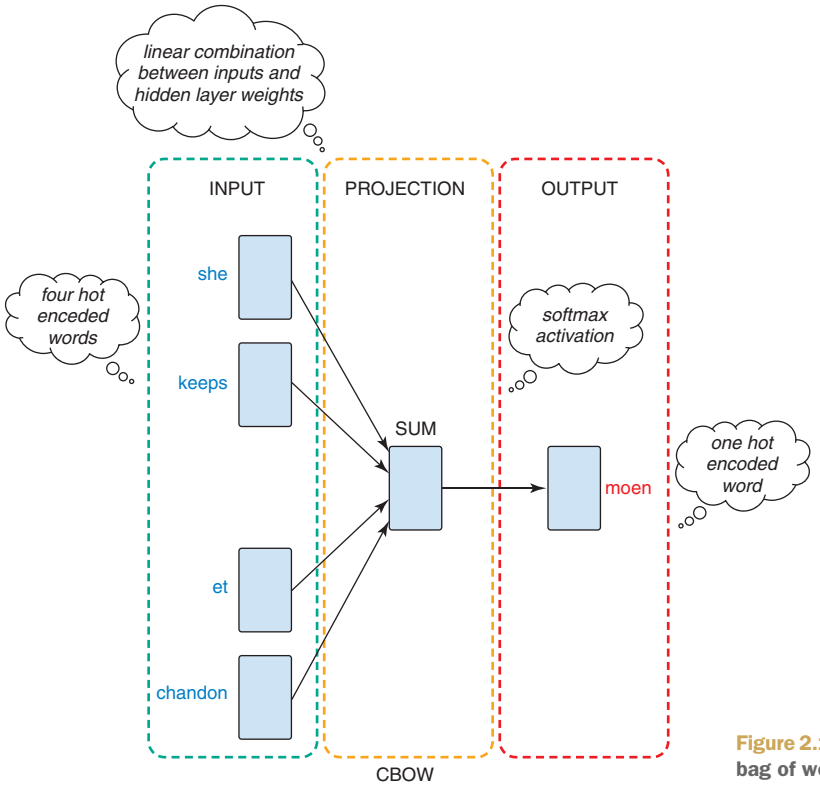



**Figure 2.10 Continuous bag of words model**

The neurons in the hidden layer have no activation function, so they linearly combine weights and inputs (multiply each input by its weight and sum all of the results together). The input layer has a number of neurons equal to the number of words in the text for each word; word2vec requires each word to be represented as a *one-hot encoded* vector.

Let's see what a one-hot encoded vector looks like. Imagine that you have a dataset with three words: `[cat, dog, mouse]`. You have three vectors, each with all the values set to 0 except one, which is set to 1 (that one identifies that specific word):

```
dog   : [0,0,1]
cat   : [0,1,0]
mouse : [1,0,0]
```

If you add the word *lion* to the dataset, one-hot encoded vectors for this dataset will have dimension 4:

```
lion  : [0,0,0,1]
dog   : [0,0,1,0]
cat   : [0,1,0,0]
mouse : [1,0,0,0]
```

If you have 100 words in your input text, each word will be represented as a 100-dimensional vector. Consequently, in the CBOW model, you'll have 100 input neurons multiplied by the value of the `window` parameter minus 1. So, if `window` is 4, you'll have 300 input neurons.

The hidden layer has a number of neurons equal to the desired dimensionality of the resulting word vectors. This parameter must be set by whoever sets up the network.

The size of the output layer is equal to the number of words in the input text: in this example, 100. A word2vec CBOW model for an input text with 100 words, embeddings dimensionality equals to 50 and `window` set to 4 will have 300 input neurons, 50 hidden neurons, and 100 output neurons. Note that, while input and output dimensionality depend on the size of the vocabulary (e.g. 100 in this case) and the `window` parameter, the dimensionality of the word embeddings generated by the CBOW model is a parameter, to be chosen by the user. For example, in figure 2.11 you can see the following:

- The input layer has a dimensionality of $C \times V$, where $C$ is the length of the context (corresponding to the `window` parameter minus one) and $V$ is the size of the vocabulary.
- The hidden layer has a dimensionality of $N$, defined by the user
- The output layer has a dimensionality equal to $V$.

For word2vec, CBOW model inputs are propagated through the network by first multiplying the one-hot encoded vectors of the input words by their input-to-hidden weights; you can imagine that as a matrix containing a weight for each connection between an input and a hidden neuron. Those are combined (multiplied) with the hidden-to-output weights, producing the outputs; and these outputs are then passed through a softmax function. Softmax "squashes" a K-dimensional vector (the output vector) of arbitrary real values to a K-dimensional vector of real values in the range (0, 1) that add up to 1, so that they can represent a probability distribution. Your network tells you the probability that each output word will be selected, given the context (the network input).

You now have a neural network that can predict the most likely word to appear in the text, given a context of a few words (the `window` parameter). This neural network can tell you that given a context like "I like eating," you should expect the next word to be something like *pizza*. Note that because word order isn't taken into account, you

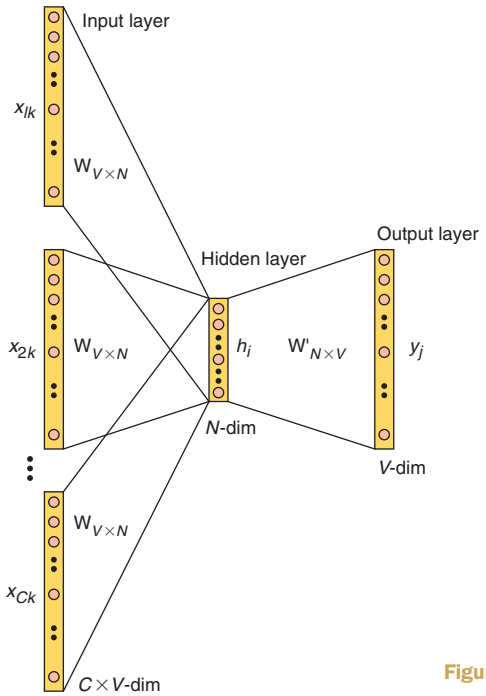



**Figure 2.11 Continuous bag of words model weights**

could also say that given the context "I eating pizza," the next word most likely to appear in the text is *like*.

But the most important part of this neural network for the goal of generating synonyms isn't learning to predict words given a context. The surprising beauty of this method is that internally, the weights of the hidden layer adjust in a way that makes it possible to determine when two words are semantically similar (because they appear in the same or similar contexts).

After forward propagation, the backpropagation learning algorithm adjusts the weights of each neuron in the different layers so it will produce a more accurate result for each new fragment. When the learning process has finished, the hidden-to-output weights represent the vector representation (embedding) for each word in the text.

Skip-gram looks reversed with respect to the CBOW model. The same concepts apply: the input vectors are one-hot encoded (one for each word) so the input layer has a number of neurons equals to the number of words in the input text. The hidden layer has the dimensionality of the desired resulting word vectors, and the output layer has a number of neurons equal to the number of words multiplied by `window` minus 1. Using the same example as before, given the text "she keeps moet et chandon in her pretty cabinet let them eat cake she says" and a `window` value of 5, a word2vec model based on the skip-gram model will receive a first sample for | `she` | `keeps` | `moet` | `et` | `chandon` | with the input `moet` and the output | `she` | `keeps` | `et` | `chandon` | (see figure 2.12).

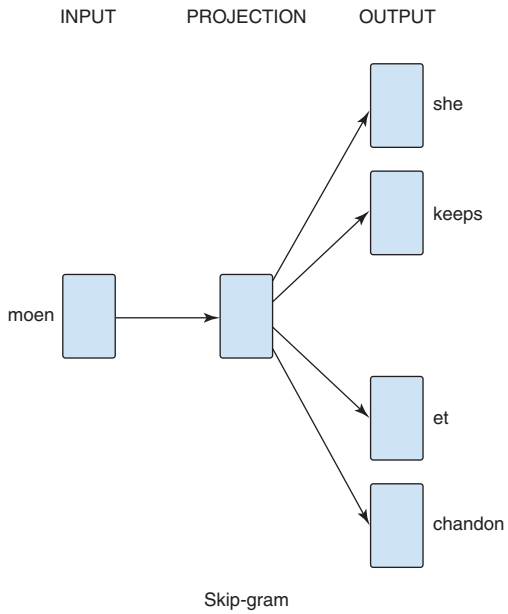



**Figure 2.12 Skip-gram model**

Figure 2.13 is an example excerpt of word vectors calculated by word2vec for the text of the Hot 100 Billboard dataset. It shows a small subset of words plotted, for the sake of appreciating word semantics being expressed geometrically.

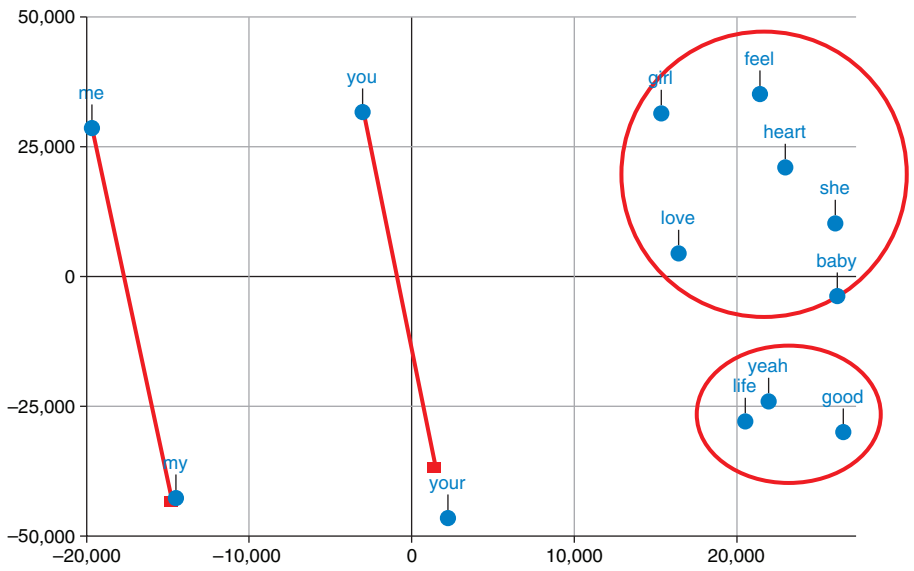



**Figure 2.13 Highlights of word2vec vectors for the Hot 100 Billboard dataset**

Notice the expected regularities between *me* and *my* with respect to *you* and *your*. Also note the groups of similar words, or words used in similar contexts, which are good candidates for synonyms.

Now that you've learned a bit about how the word2vec algorithm works, let's write some code and see it in action. Then you'll be able to combine it with the search engine for synonym expansion.

**Deeplearning4j**

Deeplearning4j (DL4J) is a deep learning library for the Java Virtual Machine (JVM). It has good adoption among the Java users and a not-too-steep learning curve for early adopters. It also comes with an Apache 2 license, which is handy if you want to use it within a company and include it in a possibly non-open source product. Additionally, DL4J has tools to import models created with other frameworks such as Keras, Caffe, Tensorflow, Theano, and so on.

### 2.6.1 Setting up word2vec in Deeplearning4J

In this book, we'll use DL4J to implement neural network-based algorithms. Let's see how to use it to set up a word2vec model.

DL4J has an out-of-the-box implementation of word2vec, based on the skip-gram model. You need to set up its configuration parameters and pass the input text you want to feed the search engine.

Keeping the song lyrics use case in mind, let's feed word2vec the Billboard Hot 100 text file. You want output word vectors of a suitable dimension, so you set that configuration parameter to 100 and the window size to 5.

**Listing 2.9 DL4J word2vec example**

```
String filePath = new ClassPathResource(
    "billboard_lyrics_1964-2015.txt").getFile().getAbsolutePath();
SentenceIterator iter = new BasicLineIterator(filePath);

Word2Vec vec = new Word2Vec.Builder()
      .layerSize(100)
      .windowSize(5)
      .iterate(iter)
      .elementsLearningAlgorithm(new CBOW<>())
      .build();
vec.fit();

String[] words = new String[]{"guitar", "love", "rock"};
```

Reads the corpus of text containing the lyrics

Sets up an iterator over the corpus

Creates a configuration for word2vec

Sets the number of dimensions the vector representations should have

Sets word2vec to iterate over the selected corpus

Sets the window parameter

Uses CBOW model

Performs training

```
for (String w : words) {
    Collection<String> lst = vec.wordsNearest(w, 2);   #I
    System.out.println("2 Words closest to '"
        + w + "': " + lst);
}
```

Obtains the closest words to an input word

Prints the nearest words

You obtain the following output, which seems good enough:

```
2 Words closest to 'guitar': [giggle, piano]
2 Words closest to 'love': [girl, baby]
2 Words closest to 'rock': [party, hips]
```

Note that you can alternatively use the skip-gram model by changing the `elementsLearningAlgorithm`.

**Listing 2.10 Using the skip-gram model**

```
Word2Vec vec = new Word2Vec.Builder()
    .layerSize(...)
    .windowSize(...)
    .iterate(...)
    .elementsLearningAlgorithm(new SkipGram<>())
    .build();
vec.fit();
```

Uses the skip-gram model

As you can see, it's straightforward to set up such a model and obtain results in a reasonable time (training the word2vec model took around 30 seconds on a "normal" laptop). Keep in mind that you'll now aim to use this in conjunction with the search engine, which should yield a better synonym-expansion algorithm.

### 2.6.2 Word2vec-based synonym expansion

Now that you have this powerful tool in your hands, you need to be careful! When using WordNet, you have a constrained set of synonyms, so you can't blow up the index. With word vectors generated by word2vec, you can ask the model to return the closest words for each word to be indexed. This might be not acceptable from a performance perspective (for both runtime and storage), so you have to come up with a strategy for using word2vec responsibly. One thing you can do is constrain the type of words for which you ask word2vec to get the nearest words. In natural language processing, it's common to tag each word with a *part of speech* (PoS) that labels its syntactic role in a sentence. Common parts of speech are `NOUN`, `VERB`, and `ADJ`; there are also finer-grained ones like `NP` and `NC` (proper and common noun, respectively). For example, you might decide to use word2vec only for words whose PoS is either `NC` or `VERB`, to avoid bloating the index with synonyms for adjectives. Another technique would be to look at how informative the document is. A short text has a relatively poor probability of being hit with a query because it's composed of only a few terms. So, you might decide to focus on such documents and be expand their synonyms, rather than focusing on longer documents.

On the other hand, the "informativeness" of a document doesn't only depend on its size. Thus you might use other techniques, such as looking at term *weights* (the

number of times a term appears in a piece of text) and skipping those that have a low weight.

You could also choose to use word2vec results only if they have a good similarity score. If you use cosine distance to measure the nearest neighbors of a word vector, such neighbors may be too far away (a low similarity score) but still be the nearest. In that case, you could decide not to use those neighbors.

Now that you've trained a word2vec model on the Hot 100 Billboard Dataset using Deeplearning4J, let's use it in conjunction with the search engine to generate synonyms. As explained in chapter 1, a token filter takes the terms provided by a tokenizer and performs operations on the terms, such as filtering them or, as in this case, adding other terms to be indexed. A Lucene `TokenFilter` is based on the `incrementToken` API, which returns a `boolean` value that is false at the end of the token stream. Implementors of this API consume one token at a time (for example, by filtering or expanding a token). Figure 2.14 shows a diagram of how word2vec-based synonym expansion is expected to work.

You're finished with word2vec training, so you can create a synonym filter that will use the learned model to predict term synonyms during filtering. You'll build a Lucene `TokenFilter` that can use DL4J word2vec on input tokens. This means implementing the left side of figure 2.14.

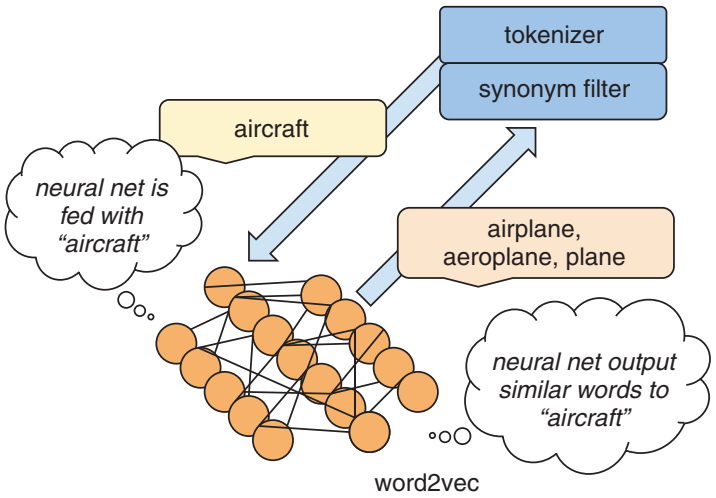



**Figure 2.14 Synonym expansion at search time, with word2vec**

The Lucene APIs for token filtering require you to implement the `incrementToken` method. This method will return `true` if there are still tokens to consume from the token stream or `false` if there are no more tokens left to consider for filtering. The basic idea is that the token filter will return `true` for all original tokens and `false` for all the related synonyms you get from word2vec.

**Listing 2.11 Word2vec-based synonym expansion filter**

```
protected W2VSynonymFilter(TokenStream input,
    Word2Vec word2Vec) {
  super(input);
  this.word2Vec = word2Vec;
}

@Override
public boolean incrementToken()
    throws IOException {
  if (!outputs.isEmpty()) {
    ...
  }

  if (!SynonymFilter.TYPE_SYNONYM.equals(typeAtt.type())) {
    String word = new String(termAtt.buffer())
        .trim();
    List<String> list = word2Vec.
        similarWordsInVocabTo(word, minAcc);
    int i = 0;
    for (String syn : list) {
      if (i == 2) {
        break;
      }
      if (!syn.equals(word)) {
        CharsRefBuilder charsRefBuilder = new CharsRefBuilder();
        CharsRef cr = charsRefBuilder.append(syn)
            .get();

        State state = captureState();
        outputs.add(new PendingOutput(state, cr));
        i++;
      }
    }
  }
  return !outputs.isEmpty() || input.incrementToken();
}
```

Creates a token filter that takes an already-trained word2vec model

Implements the Lucene API for token filtering

Adds cached synonyms to the token stream (see the next code listing)

Expands a token only if it's not a synonym (to avoid loops in the expansion)

For each term, uses word2vec to find the closest words that have an accuracy higher than a `minAcc` (for example, 0.35)

Records no more than two synonyms for each token

Records the synonym value

Records the current state of the original term (not the synonym) in the token stream (for example, starting and ending position)

Creates an object to contain the synonyms to be added to the token stream after all the original terms have been consumed

This code traverses all the terms and, when it finds a synonym, puts the synonym in a list of pending outputs to expand (the `outputs` List). You apply those pending terms to be added (the actual synonyms) after each original term has been processed, as shown next.

**Listing 2.12 Expanding pending synonyms**

```
...
  if (!outputs.isEmpty()) {
    PendingOutput output = outputs.remove(0);

    restoreState(output.state);
```

Gets the first pending output to expand

Retrieves the state of the original term, including its text, its position in the text stream, and so on

```
            termAtt.copyBuffer(output.charsRef.chars, output
               .charsRef.offset, output.charsRef.length);
            typeAtt.setType(SynonymFilter.TYPE_SYNONYM);
            return true;
        }
```

Sets the synonym text to that given by word2vec and previously saved in the pending output

Sets the type of the term as synonym

You use the word2vec output results as synonyms only if they have an accuracy greater than a certain threshold, as discussed in the previous section. The filter picks only the two words closest to the given term (according to word2vec) having an accuracy of at least 0.35 (which isn't that high), for each term passed by the tokenizer. If you pass the sentence "I like pleasure spiked with pain and music is my airplane" to the filter, it will expand the word *airplane* with two additional words: *airplanes* and *aeroplane* (see related token stream in Figure 2.15).

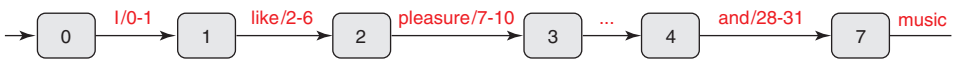



**Figure 2.15 Token stream after word2vec synonym expansion**

## 2.7 Evaluations and comparisons

As mentioned in chapter 1, you can usually capture metrics, including precision, recall, query with zero results, and so on, both before and after the introduction of query expansion. It's also usually good to determine the best configuration set for all the parameters of a neural network. A generic neural network has many parameters you can adjust:

- The general network architecture, such as using one or more hidden layers
- The transformations performed in each layer
- The number of neurons in each layer
- The connections between neurons belonging to different layers
- The number of times (also called *epochs*) the network should read through all the training sets in order to reach its final state (possibly with a low error and high accuracy)

These parameters also apply to other machine learning techniques. In the case of word2vec, you can decide

- The size of the generated word embeddings
- The window used to create fragments for unsupervised training of models
- Which architecture to use: CBOW or skip-gram

As you can see, there are many possible parameter settings to try.

*Cross validation* is a method to optimize the parameters while making sure a machine learning model performs well enough on data that's different from the one used for training. With cross validation, the original data set is split into three subsets: a

training set, a validation set, and a test set. The training set is used as the data source to train the model. In practice, it's often used to train a bunch of separate models with different settings for the available parameters. The cross-validation set is used to select the model that has the best-performing parameters. This can be done, for example, by taking each pair of input and desired output in the cross-validation set and seeing whether a model gives results equal or close to the desired output, when given that particular input. The test set is used the same way as the cross-validation set, except it's only used by the model selected by testing on the cross-validation set. The accuracy of results on the test set can be considered a good measure of the model's overall effectiveness.

## 2.8 Considerations for production systems

In this chapter, you've seen how to use word2vec to generate synonyms from data to be indexed and searched. Most existing production systems already contain lots of indexed documents, and in such cases it's often impossible to access the original data as it existed before it was indexed. In the case of indexing the top 100 songs of the year to build a search engine of song lyrics, you have to take into account that the rankings of the most popular songs change every day, week, month, and year. This implies that the dataset will change over time; therefore, if you don't keep old copies in separate storage, you won't be able to build a word2vec model for all indexed documents (song lyrics) later.

The solution to this problem is to work with the search engine as the primary data source. When you set up word2vec using DL4J, you fetched sentences from a single file:

```
String filePath = new ClassPathResource("billboard_lyrics.txt").getFile()
   .getAbsolutePath();
SentenceIterator iter = new BasicLineIterator(filePath);
```

Given an evolving system that's fed song lyrics from different files daily, weekly, or monthly, you'll need to take the sentences directly from the search engine. For this reason, you'll build a `SentenceIterator` that reads stored values from the Lucene index.

**Listing 2.13 Fetching sentences for word2vec from the Lucene index**

```
public class FieldValuesSentenceIterator implements
    SentenceIterator {

  private final IndexReader reader;
  private final String field;
  private int currentId;

  public FieldValuesSentenceIterator(
      IndexReader reader, String field) {
    this.reader = reader;
    this.field = field;
    this.currentId = 0;
  }
```

View of the index used to fetch the document values

Specific field to fetch the values from

Because this is an iterator, the identifier of the current document being fetched

```
  ...

  @Override
  public void reset() {
    currentId = 0;
  }

}
```

First document ID is always 0.

In the example case of the song lyrics search engine, the text of the lyrics were indexed into the `text` field. You therefore fetch the sentences and words to be used for training the word2vec model from that field.

**Listing 2.14 Reading sentences from the Lucene index**

```
Path path = Paths.get("/path/to/index");
Directory directory = FSDirectory.open(path);
IndexReader reader = DirectoryReader.open(directory);
SentenceIterator iter = new FieldValuesSentenceIterator(reader, "text");
```

Once you've set things up, you pass this new `SentenceIterator` to the word2vec implementation:

```
SentenceIterator iter = new FieldValuesSentenceIterator(reader, "text");
Word2Vec vec = new Word2Vec.Builder()
  .layerSize(100)
  .windowSize(5)
  .iterate(iter)
  .build();
vec.fit();
```

During the training phase, the `SentenceIterator` is asked to iterate over `String`s.

**Listing 2.15 For each document, passing field values to word2vec for training**

```
  @Override
  public String nextSentence() {
    if (!hasNext()) {
      return null;
    }
    try {
      Document document = reader.document(currentId,
          Collections.singleton(field));
      String sentence = document.getField(field).stringValue();
      return preProcessor != null ? preProcessor
          .preProcess(sentence) : sentence;
    } catch (IOException e) {
      throw new RuntimeException(e);
    } finally {
      currentId++;
    }
  }
```

The iterator has more sentences if the current document identifier isn't bigger than the number of documents contained in the index.

Gets the value of the text field from the current Lucene Document as a String

Gets the document with the current identifier (only the field you need is fetched)

Returns the sentence, which is eventually preprocessed if you set a preprocessor (for example, to remove unwanted characters or tokens)

Increments the document ID for the next iteration

```
  @Override
  public boolean hasNext() {
    return currentId < reader.numDocs();
  }
```

This way, word2vec can be retrained frequently on existing search engines without having to maintain the original data. The synonym expansion filter can be kept up to date as the data in the search engine is updated.

### 2.8.1 Synonyms vs. antonyms

Imagine that you have the following sentences: "I like pizza," "I hate pizza," "I like pasta," I hate pasta," I love pasta," and "I eat pasta." This would be a small set of sentences for word2vec to use to learn accurate embeddings in real life. But you can clearly see that the terms *I* on the left and *pizza* and *pasta* on the right all share verbs in between. Because word2vec learns word embeddings using similar text fragments, you may end up with similar word vectors for the verbs *like*, *hate*, *love*, and *eat*. So, word2vec may report that *love* is close to *like* and *eat* (which is fine, given that the sentences are all related to food) but also to *hate*, which is definitely not a synonym for *love*.

In some cases, this issue may not be important. Suppose you want to go out to dinner, and you're searching for a nice restaurant on the internet. You write the query "reviews of restaurants people love" in a search engine. If you get reviews about "restaurants people hate," then you'll know where *not* to go. But this is an edge case; generally, you don't want antonyms (the opposite of a synonym) to be expanded like synonyms.

Don't worry—usually, the text has enough information to tell you that although *hate* and *love* appear in similar contexts, they aren't proper synonyms. The fact that this corpus of text is only made of sentences like *I hate pizza* or *I like pasta* makes it more difficult: usually, *hate* and *like* also appear in other contexts, which helps word2vec figure out that they aren't similar. To see that, let's evaluate the nearest words of the word *nice* together with their similarity:

```
String tw = "nice";
Collection<String> wordsNearest = vec.wordsNearest(tw, 3);
System.out.println(tw + " -> " + wordsNearest);
for (String wn : wordsNearest) {
  double similarity = vec.similarity(tw, wn);
  System.out.println("sim(" + tw + "," + wn + ") : " + similarity);
  ...
}
```

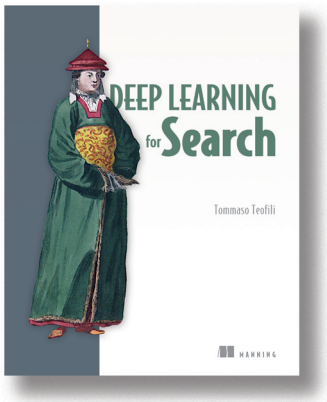
The similarity between word vectors can help you exclude nearest neighbors that aren't similar enough. A sample word2vec run over the Hot 100 Billboard dataset indicates that the nearest words of the word *nice* are *cute*, *unfair*, and *real*:

```
nice -> [cute, unfair, real]
sim(nice,cute) : 0.6139052510261536
sim(nice,unfair) : 0.5972062945365906
sim(nice,real) : 0.5814308524131775
```

*Cute* is a synonym. *Unfair* isn't an antonym but an adjective that expresses negative feelings; it's not a good result, because it's in contrast with the positive nature of *nice* and *cute*. *Real* also doesn't express the same general semantics as *nice*. To fix this, you can, for example, filter out the nearest neighbors whose similarity is less than the absolute value 0.5, or less than the highest similarity minus 0.1. You assume that the first nearest neighbor is usually good enough, as long as its similarity is greater than 0.5; once this applies, you exclude words that are too far from the nearest neighbor. In this case, filtering out words whose similarity is less than the highest nearest neighbor similarity (0.61) minus 0.1, you filter out both *unfair* and *real* (each has a similarity less than 0.60).

## 2.9 Summary

- Synonym expansion can be a handy technique to improve recall and make the users of your search engine happier.
- Common synonym-expansion techniques are based on static dictionaries and vocabularies that might require manual maintenance or are often far from the data they're used for.
- Feed-forward neural networks are the basis of many neural network architectures. In a feed-forward neural network, information flows from an input layer to an output layer; in between these two layers there may be one or more hidden layers.
- Word2vec is a feed-forward neural network-based algorithm for learning vector representations for words that can be used to find words with similar meanings—or that appear in similar contexts—so it's reasonable to use it for synonym expansion, too.
- You can either use the continuous bag of words or skip-gram architecture for word2vec. In CBOW, the target word is used as the output of the network, and the remaining words of the text fragments are used as inputs. In the skip-gram model, the target word is used as input, and the context words are outputs. Both work well, but skip-gram is usually preferred because it works better with infrequent words.
- Word2vec models can provide good results, but you need to manage word senses or parts of speech when using it for synonyms.
- In word2vec, be careful to avoid letting antonyms be used as synonyms.




Using deep learning and neural networks are the perfect way to create better search results, letting you fine tune what your search engines display, help speed up the results, and let you build a profile of your customers that let them find what they need every single time. And because deep learning systems improve the more you use them, your clients will also become happier in the bargain.

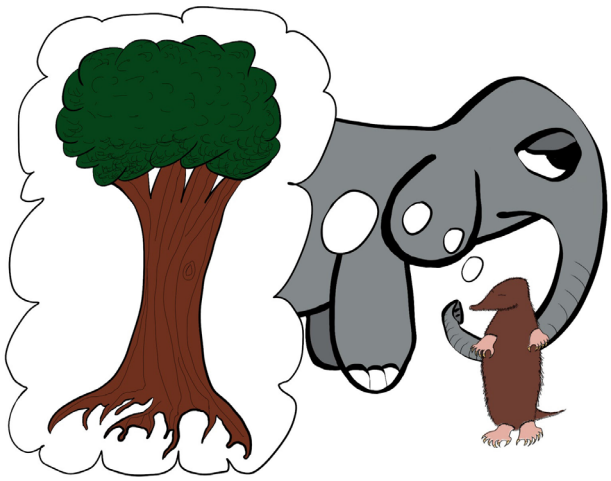
*Deep Learning for Search* teaches you how to improve the effectiveness of your search by implementing neural network-based techniques. You'll start with an overview of information retrieval principles, like indexing, searching, and ranking, as well as a fast indoctrination into deep learning. Then, you'll move through in-depth examples as you gain an understanding of how to improve typical search tasks, such as relevance, with the help of Apache Lucene and Deeplearning4j. The book wraps up with a look at advanced problems, like searching through images and translating user queries. By the time you're finished, you'll be ready to build amazing search engines that deliver the results your users need and get better as time goes on!

***What's inside:***

- Applying deep learning to search
- Generating suitable synonyms
- Accurate and relevant rankings of search results
- Searching across languages
- Content-based image search
- Search with recommendations

Written for developers comfortable with Java or a similar language. No experience with deep learning or NLP needed.

# *Neural Networks that Understand Language*



*"The elephant is like a tree, rooted firmly before climbing upwards."*

Now that you've seen how neural networks can be used to model language, it's worthwhile to build a bit more perspective on how and why they work well for language problems. The following chapter goes a bit deeper into the how and why so that you can grok how deep learning understands the structure of language. The example builds up to an interesting use of neural networks: word analogies. These are the sort of word problems that many humans and moles struggled to get right in school; but as you'll see in this chapter, we can teach a machine how to solve these problems.

Chapter 11 from *Grokking Deep Learning* by Andrew Trask.

# *Neural Networks that understand language*

## *king – man + woman ==?*

***This chapter covers***

- Natural language processing (NLP)
- Supervised NLP
- Capturing word correlation in input data
- Intro to an embedding layer
- Neural architecture
- Comparing word embeddings
- Filling in the blank
- Meaning is derived from loss
- Word analogies

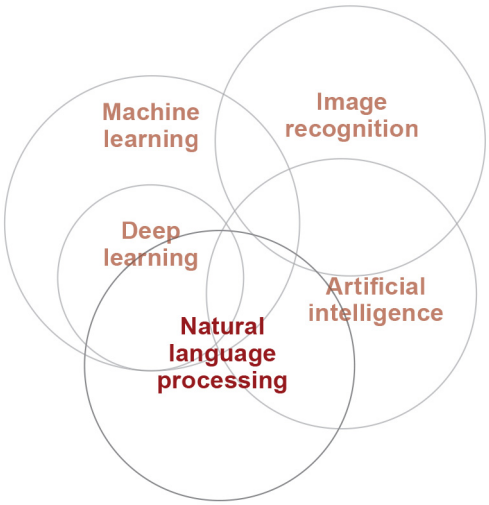
"*Man is a slow, sloppy, and brilliant thinker; computers are fast, accurate, and stupid.*"

—John Pfeiffer, in Fortune, 1961

## 11.1 What does it mean to understand language?

### What kinds of predictions do people make about language?

Up until now, we've been using neural networks to model image data. But neural networks can be used to understand a much wider variety of datasets. Exploring new datasets also teaches us a lot about neural networks in general, because different datasets often justify different styles of neural network training according the challenges hidden in the data.




We'll begin this chapter by exploring a much older field that overlaps deep learning: *natural language processing* (NLP). This field is dedicated exclusively to the automated understanding of human language (previously not using deep learning). We'll discuss the basics of deep learning's approach to this field.

## 11.2 Natural language processing (NLP)

### *NLP is divided into a collection of tasks or challenges.*

Perhaps the best way to quickly get to know NLP is to consider a few of the many challenges the NLP community seeks to solve. Here are a few types of classification problem that are common to NLP:

- Using the *characters* of a document to predict *where words start and end.*
- Using the *words* of a document to predict *where sentences start and end.*
- Using the *words in a sentence* to predict *the part of speech for each word.*
- Using *words in a sentence* to predict *where phrases start and end.*
- Using *words in a sentence* to predict *where named entity (person, place, thing) references start and end.*
- Using *sentences in a document* to predict *which pronouns refer to the same person / place / thing.*
- Using *words in a sentence* to predict the *sentiment* of a sentence.

Generally speaking, NLP tasks seek to do one of three things: label a region of text (such as part-of-speech tagging, sentiment classification, or named-entity recognition); link two or more regions of text (such as coreference, which tries to answer whether two mentions of a real-world thing are in fact referencing the same real-world thing, where the real-world thing is generally a person, place, or some other named entity); or try to fill in missing information (missing words) based on context.

Perhaps it's also apparent how machine learning and NLP are deeply intertwined. Until recently, most state-of-the-art NLP algorithms were advanced, probabilistic, non-parametric models (not deep learning). But the recent development and popularization of two major neural algorithms have swept the field of NLP: neural word embeddings and recurrent neural networks (RNNs).

In this chapter, we'll build a word-embedding algorithm and demonstrate why it increases the accuracy of NLP algorithms. In the next chapter, we'll create a recurrent neural network and demonstrate why it's so effective at predicting across sequences.

It's also worth mentioning the key role that NLP (perhaps using deep learning) plays in the advancement of artificial intelligence. AI seeks to create machines that can think and engage with the world as humans do (and beyond). NLP plays a very special role in this endeavor, because language is the bedrock of conscious logic and communication in humans. As such, methods by which machines can use and understand language form the foundation of human-like logic in machines: the foundation of thought.

## 11.3 Supervised NLP

***Words go in, and predictions come out.***

Perhaps you'll remember the following figure from chapter 2. Supervised learning is all about taking "what you know" and transforming it into "what you want to know." Up until now, "what you know" has always consisted of numbers in one way or another. But NLP uses text as input. How do you process it?


Because neural networks only map input numbers to output numbers, the first step is to convert the text into numerical form. Much as we converted the streetlight dataset, we need to convert the real-world data (in this case, text) into a *matrix* the neural network can consume. As it turns out, how we do this is extremely important!


How should we convert text to numbers? Answering that question requires some thought regarding the problem. Remember, neural networks look for correlation between their input and output layers. Thus, we want to convert text into numbers in such a way that the correlation between input and output is *most obvious* to the network. This will make for faster training and better generalization.

In order to know what input format makes input/output correlation the most obvious to the network, we need to know what the input/output dataset looks like. To explore this topic, let's take on the challenge of *topic classification*.

## 11.4 IMDB movie reviews dataset

***You can predict whether people post positive or negative reviews.***

The IMDB movie reviews dataset is a collection of review -> rating pairs that often look like the following (this is an imitation, not pulled from IMDB):

> *"This movie was terrible! The plot was dry, the acting unconvincing, and I spilled popcorn on my shirt."*
>
> —Rating: 1 (stars)

The entire dataset consists of around 50,000 of these pairs, where the input reviews are usually a few sentences and the output ratings are between 1 and 5 stars. People consider it a *sentiment dataset* because the stars are indicative of the overall sentiment of the movie review. But it should be obvious that this sentiment dataset might be very different from other sentiment datasets, such as product reviews or hospital patient reviews.

You want to train a neural network that can use the input text to make accurate predictions of the output score. To accomplish this, you must first decide how to turn the input and output datasets into matrices. Interestingly, the output dataset is a number, which perhaps makes it an easier place to start. You'll adjust the range of stars to be between 0 and 1 instead of 1 and 5, so that you can use binary `softmax`. That's all you need to do to the output. I'll show an example on the next page.

The input data, however, is a bit trickier. To begin, let's consider the raw data. It's a list of characters. This presents a few problems: not only is the input data text instead of numbers, but it's *variable-length* text. So far, neural networks always take an input of a fixed size. You'll need to overcome this.

So, the raw input won't work. The next question to ask is, "What about this data will have correlation with the output?" Representing that property might work well. For starters, I wouldn't expect any characters (in the list of characters) to have any correlation with the sentiment. You need to think about it differently.

What about the words? Several words in this dataset would have a bit of correlation. I'd bet that *terrible* and *unconvincing* have significant negative correlation with the rating. By *negative*, I mean that as they increase in frequency in any input datapoint (any review), the rating tends to decrease.

Perhaps this property is more general! Perhaps words by themselves (even out of context) would have significant correlation with sentiment. Let's explore this further.

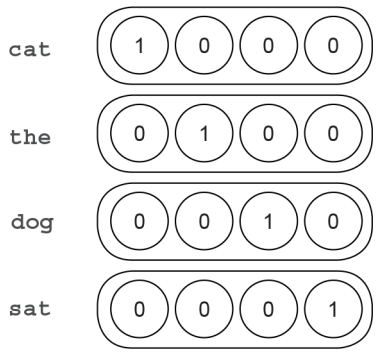
## 11.5 Capturing word correlation in input data

***Bag of words: Given a review's vocabulary, predict the sentiment.***

If you observe correlation between the vocabulary of an IMDB review and its rating, then you can proceed to the next step: creating an input matrix that represents the vocabulary of a movie review.

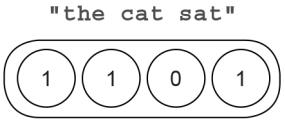
What's commonly done in this case is to create a matrix where each row (vector) corresponds to each movie review, and each column represents whether a review contains a particular word in the vocabulary. To create the vector for a review, you calculate the vocabulary of the review and then put 1 in each corresponding column for that review and 0s everywhere else. How big are these vectors? Well, if there are 2,000 words, and you need a place in each vector for each word, each vector will have 2,000 dimensions. This form of storage, called *one-hot encoding*, is the most common format for encoding binary data (the binary presence or absence of an input datapoint among a vocabulary of possible input datapoints). If the vocabulary was only four words, the one-hot encoding might look like this:

```
import numpy as np

onehots = {}
onehots['cat'] = np.array([1,0,0,0])
onehots['the'] = np.array([0,1,0,0])
onehots['dog'] = np.array([0,0,1,0])
onehots['sat'] = np.array([0,0,0,1])

sentence = ['the','cat','sat']
x = word2hot[sentence[0]] + \
    word2hot[sentence[1]] + \
    word2hot[sentence[2]]

print("Sent Encoding:" + str(x))
```




vocabulary, and this allows you to use simple vector addition to create a vector representing a subset of the total vocabulary (such as a subset corresponding to the words in a sentence).

```
Output: Sent Encoding:[1 1 0 1]
```




Note that when you create an embedding for several terms (such as "the cat sat"), you have multiple options if words occur multiple times. If the phrase was "cat cat cat," you could either sum the vector for "cat" three times (resulting in `[3,0,0,0]`) or just take the unique "cat" a single time (resulting in `[1,0,0,0]`). The latter typically works better for language.

## 11.6 Predicting movie reviews

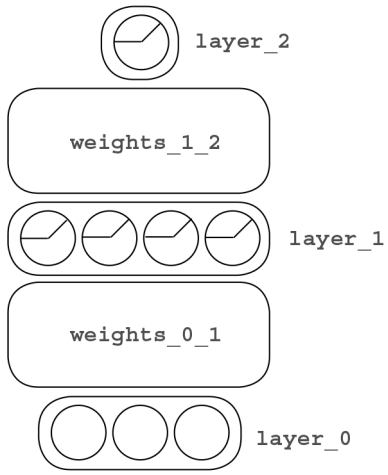
***With the encoding strategy and the previous network, you can predict sentiment.***

Using the strategy we just identified, you can build a vector for each word in the sentiment dataset and use the previous two-layer network to predict sentiment. I'll show you the code, but I strongly recommend attempting this from memory. Open a new Jupyter notebook, load in the dataset, build your one-hot vectors, and then build a neural network to predict the rating of each movie review (positive or negative). Here's how I would do the preprocessing step:

```
import sys

f = open('reviews.txt')
raw_reviews = f.readlines()
f.close()

f = open('labels.txt')
raw_labels = f.readlines()
f.close()

tokens = list(map(lambda x:set(x.split(" ")),raw_reviews))

vocab = set()
for sent in tokens:
    for word in sent:
        if(len(word)>0):
            vocab.add(word)
vocab = list(vocab)

word2index = {}
for i,word in enumerate(vocab):
    word2index[word]=i

input_dataset = list()
for sent in tokens:
    sent_indices = list()
    for word in sent:
        try:
            sent_indices.append(word2index[word])
        except:
            ""
    input_dataset.append(list(set(sent_indices)))

target_dataset = list()
for label in raw_labels:
    if label == 'positive\n':
        target_dataset.append(1)
    else:
        target_dataset.append(0)
```

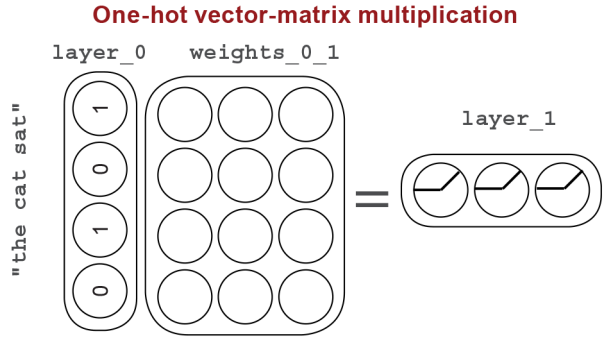
## 11.7 Intro to an embedding layer

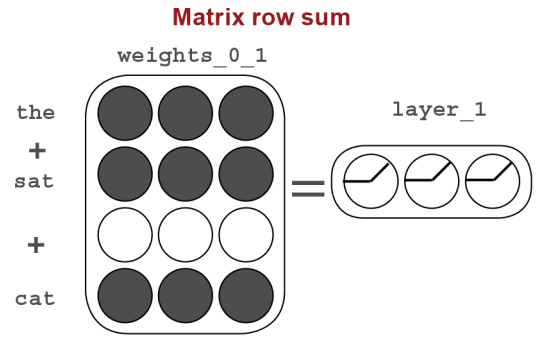
***Here's one more trick to make the network faster.***




At right is the diagram from the previous neural network, which you'll now use to predict sentiment. But before that, I want to describe the layer names. The first layer is the dataset (`layer_0`). This is followed by what's called a *linear layer* (`weights_0_1`). This is followed by a `relu` layer (`layer_1`), another linear layer (`weights_1_2`), and then the output, which is the prediction layer. As it turns out, you can take a bit of a shortcut to `layer_1` by replacing the first linear layer (`weights_0_1`) with an embedding layer.

Taking a vector of 1s and 0s is mathematically equivalent to summing several rows of a matrix. Thus, it's much more efficient to select the relevant rows of `weights_0_1` and sum them as opposed to doing a big vector-matrix multiplication. Because the sentiment vocabulary is on the order of 70,000 words, most of the vector-matrix multiplication is spent multiplying 0s in the input vector by different rows of the matrix before summing them. Selecting the rows corresponding to each word in a matrix and summing them is much more efficient.




Using this process of selecting rows and performing a sum (or average) means treating the first linear layer (`weights_0_1`) as an embedding layer. Structurally, they're identical (`layer_1` is exactly the same using either method for forward propagation). The only difference is that summing a small number of rows is much faster.

***After running the previous code, run this code.***

```
import numpy as np np.random.seed(1)

def sigmoid(x):
    return 1/(1 + np.exp(-x))

alpha, iterations = (0.01, 2)
hidden_size = 100

weights_0_1 = 0.2*np.random.random((len(vocab),hidden_size)) - 0.1
     weights_1_2 = 0.2*np.random.random((hidden_size,1)) - 0.1

correct,total = (0,0)
for iter in range(iterations):

    for i in range(len(input_dataset)-1000):

        x,y = (input_dataset[i],target_dataset[i])
        layer_1 = sigmoid(np.sum(weights_0_1[x],axis=0))
        layer_2 = sigmoid(np.dot(layer_1,weights_1_2))

        layer_2_delta = layer_2 - y
        layer_1_delta = layer_2_delta.dot(weights_1_2.T)

        weights_0_1[x] -= layer_1_delta * alpha
        weights_1_2 -= np.outer(layer_1,layer_2_delta) * alpha

        if(np.abs(layer_2_delta) < 0.5):
            correct += 1
        total += 1
        if(i % 10 == 9):
            progress = str(i/float(len(input_dataset)))
            sys.stdout.write('\rIter:'+str(iter)\
                             +' Progress:'+progress[2:4]\
                             +'.'+progress[4:6]\
                             +'% Training Accuracy:'\
                             + str(correct/float(total)) + '%')
    print()
correct,total = (0,0)
for i in range(len(input_dataset)-1000,len(input_dataset)):

    x = input_dataset[i]
    y = target_dataset[i]

    layer_1 = sigmoid(np.sum(weights_0_1[x],axis=0))
    layer_2 = sigmoid(np.dot(layer_1,weights_1_2))

    if(np.abs(layer_2 - y) < 0.5):
        correct += 1
    total += 1
print("Test Accuracy:" + str(correct / float(total)))
```

**Trains on the first 24,000 reviews**

**embed + sigmonid**

**Compares the prediction with the truth**

**linear + softmax**

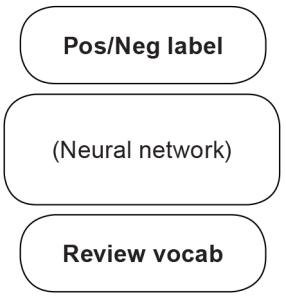
**Backpropagation**

## 11.8 Interpreting the output

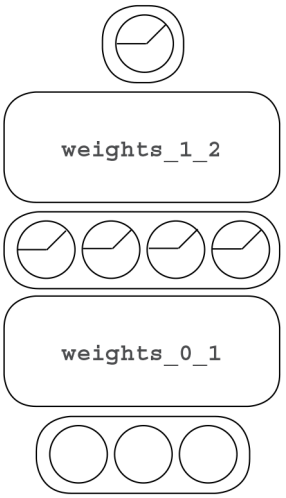
### What did the neural network learn along the way?

Here's the output of the movie reviews neural network. From one perspective, this is the same correlation summarization we've already discussed:

```
Iter:0 Progress:95.99% Training Accuracy:0.832%
Iter:1 Progress:95.99% Training Accuracy:0.8663333333333333%
Test Accuracy:0.849
```




The neural network was looking for correlation between the input datapoints and the output datapoints. But those datapoints have characteristics we're familiar with (notably those of language). Furthermore, it's extremely beneficial to consider what patterns of language would be detected by the correlation summarization, and more importantly, which ones wouldn't. After all, just because the network is able to find correlation between the input and output datasets doesn't mean it understands every useful pattern of language. Furthermore, understanding the difference between what the network (in its current configuration) is capable of learning relative to what it needs to know to properly understand language is an incredibly fruitful line of thinking. This is what researchers on the front lines of state-of-the-art research consider, and it's what we're going to consider here.




What about language did the movie reviews network learn? Let's start by considering what was presented to the network. As displayed in the diagram at top right, you presented each review's vocabulary as input and asked the network to predict one of two labels (`positive` or `negative`). Given that the correlation summarization says the network will look for correlation between the input and output datasets, at a minimum, you'd expect the network to identify words that have either a positive or negative correlation (by themselves).

This follows naturally from the correlation summarization. You present the presence or absence of a word. As such, the correlation summarization will find direct correlation between this presence/absence and each of the two labels. But this isn't the whole story.

## 11.9 Neural architecture

### *How did the choice of architecture affect what the network learned?*

We just discussed the first, most trivial type of information the neural network learned: direct correlation between the input and target datasets. This observation is largely the clean slate of neural intelligence. (If a network can't discover direct correlation between input and output data, something is probably broken.) The development of more-sophisticated architectures is based on the need to find more-complex patterns than direct correlation, and this network is no exception.

The minimal architecture needed to identify direct correlation is a two-layer network, where the network has a single weight matrix that connects directly from the input layer to the output layer. But we used a network that has a hidden layer. This begs the question, what does this hidden layer do?

Fundamentally, hidden layers are about grouping datapoints from a previous layer into $n$ groups (where $n$ is the number of neurons in the hidden layer). Each hidden neuron takes in a datapoint and answers the question, "Is this datapoint in my group?" As the hidden layer learns, it searches for useful groupings of its input. What are useful groupings?

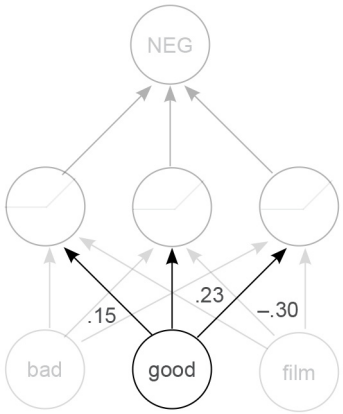
An input datapoint grouping is useful if it does two things. First, the grouping must be useful to the prediction of an output label. If it's not useful to the output prediction, *the correlation summarization will never lead the network to find the group.* This is a hugely valuable realization. Much of neural network research is about finding training data (or some other manufactured signal for the network to artificially predict) so it finds groupings that are useful for a task (such as predicting movie review stars). We'll discuss this more in a moment.

Second, a grouping is useful if it's an actual phenomenon in the data that you care about. Bad groupings just memorize the data. Good groupings pick up on phenomena that are useful linguistically. For example, when predicting whether a movie review is positive or negative, understanding the difference between "terrible" and "not terrible" is a powerful grouping. It would be great to have a neuron that turned *off* when it saw "awful" and turned *on* when it saw "not awful." This would be a powerful grouping for the next layer to use to make the final prediction. But because the input to the neural network is the vocabulary of a review, "it was great, not terrible" creates exactly the same `layer_1` value as "it was terrible, not great." For this reason, the network is very unlikely to create a hidden neuron that understands negation. Testing whether a layer is the same or different based on a certain language pattern is a great first step for knowing whether an architecture is likely to find that pattern using the correlation summarization. If you can construct two examples with an identical hidden layer, one with the pattern you find interesting and one without, the network is unlikely to find that pattern.

As you just learned, a hidden layer fundamentally groups the previous layer's data. At a granular level, each neuron classifies a datapoint as either subscribing or not subscribing to its group. At a higher level, two datapoints (movie reviews) are similar if they subscribe to many of the same groups. Finally, two inputs (words) are similar if the weights linking them to various hidden neurons (a measure of each word's group affinity) are similar. Given this knowledge, in the previous neural network, what should you observe in the weights going into the hidden neurons from the words?

### *What should you see in the weights connecting words and hidden neurons?*

Here's a hint: words that have a similar predictive power should subscribe to similar groups (hidden neuron configurations). What does this mean for the weights connecting each word to each hidden neuron?




**The three bold weights for "good" form the embedding for "good." They refl ect how much the term "good" is a member of each group (hidden neuron). Words with similar predictive power have similar word embeddings (weight values).**

Here's the answer. Words that correlate with similar labels (positive or negative) will have similar weights connecting them to various hidden neurons. This is because the neural network learns to bucket them into similar hidden neurons so that the final layer (`weights_1_2`) can make the correct positive or negative predictions. You can see this phenomenon by taking a particularly positive or negative word and searching for the other words with the most similar weight values. In other words, you can take each word and see which other words have the most similar weight values connecting them to each hidden neuron (to each group). Words that subscribe to similar groups will have similar predictive power for positive or negative labels. As such, words that subscribe to similar groups, having similar weight values, will also have similar meaning. Abstractly, in terms of neural networks, a neuron has similar meaning to other neurons in the same layer if and only if it has similar weights connecting it to the next and/or previous layers.

## 11.10 Comparing word embeddings

### *How can you visualize weight similarity?*

For each input word, you can select the list of weights proceeding out of it to the various hidden neurons by selecting the corresponding row of `weights_0_1`. Each entry in the row represents each weight proceeding from that row's word to each hidden neuron. Thus, to figure out which words are most similar to a target term, you compare each word's vector (row of the matrix) to that of the target term. The comparison of choice is called *Euclidian distance*, as shown in the following code:

```
from collections import Counter
import math

def similar(target='beautiful'):
    target_index = word2index[target]
    scores = Counter()
    for word,index in word2index.items():
        raw_difference = weights_0_1[index] - (weights_0_1[target_index])
        squared_difference = raw_difference * raw_difference
        scores[word] = -math.sqrt(sum(squared_difference))

    return scores.most_common(10)
```

This allows you to easily query for the most similar word (neuron) according to the network:

```
print(similar('beautiful'))
```

```
[('beautiful', -0.0),
 ('atmosphere', -0.70542101298),
 ('heart', -0.7339429768542354),
 ('tight', -0.7470388145765346),
 ('fascinating', -0.7549291974),
 ('expecting', -0.759886970744),
 ('beautifully', -0.7603669338),
 ('awesome', -0.76647368382398),
 ('masterpiece', -0.7708280057),
 ('outstanding', -0.7740642167)]
```

```
print(similar('terrible'))
```

```
[('terrible', -0.0),
 ('dull', -0.760788602671491),
 ('lacks', -0.76706470275372),
 ('boring', -0.7682894961694),
 ('disappointing', -0.768657),
 ('annoying', -0.78786389931),
 ('poor', -0.825784172378292),
 ('horrible', -0.83154121717),
 ('laughable', -0.8340279599),
 ('badly', -0.84165373783678)]
```

As you might expect, the most similar term to every word is itself, followed by words with similar usefulness as the target term. Again, as you might expect, because the network has only two labels (`positive` and `negative`), the input terms are grouped according to which label they tend to predict. This is a standard phenomenon of the correlation summarization. It seeks to create similar representations (`layer_1` values) within the network based on the label being predicted, so that it can predict the right label. In this case, the side effect is that the weights feeding into `layer_1` get grouped according to output label. The key takeaway is a gut instinct about this phenomenon of the correlation summarization. It consistently attempts to convince the hidden layers to be similar based on which label should be predicted.

## 11.11 What is the meaning of a neuron?

***Meaning is entirely based on the target labels being predicted.***

Note that the meanings of different words didn't totally reflect how you might group them. The term most similar to "beautiful" is "atmosphere." This is a valuable lesson. For the purposes of predicting whether a movie review is positive or negative, these words have nearly identical meaning. But in the real world, their meaning is quite different (one is an adjective and another a noun, for example).

```
print(similar('beautiful'))

[('beautiful', -0.0),
 ('atmosphere', -0.70542101298),
 ('heart', -0.7339429768542354),
 ('tight', -0.7470388145765346),
 ('fascinating', -0.7549291974),
 ('expecting', -0.759886970744),
 ('beautifully', -0.7603669338),
 ('awesome', -0.76647368382398),
 ('masterpiece', -0.7708280057),
 ('outstanding', -0.7740642167)]
```

```
print(similar('terrible'))

[('terrible', -0.0),
 ('dull', -0.760788602671491),
 ('lacks', -0.76706470275372),
 ('boring', -0.7682894961694),
 ('disappointing', -0.768657),
 ('annoying', -0.78786389931),
 ('poor', -0.825784172378292),
 ('horrible', -0.83154121717),
 ('laughable', -0.8340279599),
 ('badly', -0.84165373783678)]
```

This realization is incredibly important. The meaning (of a neuron) in the network is defined based on the target labels. Everything in the neural network is contexualized based on the correlation summarization trying to correctly make predictions. Thus, even though you and I know a great deal about these words, the neural network is entirely ignorant of all information outside the task at hand. How can you convince the network to learn more-nuanced information about neurons (in this case, word neurons)? Well, if you give it input and target data that requires a more nuanced understanding of language, it will have reason to learn more-nuanced interpretations of various terms. What should you use the neural network to predict so that it learns more interesting weight values for the word neurons?

The task you'll use to learn more-interesting weight values for the word neurons is a glorified fill-in-the blank task. Why use this? First, there's nearly infinite training data (the internet), which means nearly infinite signal for the neural network to use to learn more-nuanced information about words. Furthermore, being able to accurately fill in the blank requires at least some notion of context about the real world. For example, in the following example, is it more likely that the blank is correctly filled by the word "anvil" or "wool"? Let's see if the neural network can figure it out.

Mary had a little lamb whose __????__ was white as snow.

## 11.12 Filling in the blank

***Learn richer meanings for words by having a richer signal to learn.***

This example uses almost exactly the same neural network as the previous one, with only a few modifications. First, instead of predicting a single label given a movie review, you'll take each (five-word) phrase, remove one word (a focus term), and attempt to train a network to figure out the identity of the word you removed using the rest of the phrase. Second, you'll use a trick called *negative sampling* to make the network train a bit faster. Consider that in order to predict which term is missing, you need one label for each possible word. This would require several thousand labels, which would cause the network to train slowly. To overcome this, let's randomly ignore most of the labels for each forward propagation step (as in, pretend they don't exist). Although this may seem like a crude approximation, it's a technique that works well in practice. Here's the preprocessing code for this example:

```
import sys,random,math
from collections import Counter
import numpy as np

np.random.seed(1)
random.seed(1)
f = open('reviews.txt')
raw_reviews = f.readlines()
f.close()

tokens = list(map(lambda x:(x.split(" ")),raw_reviews))
wordcnt = Counter()
for sent in tokens:
    for word in sent:
        wordcnt[word] -= 1
vocab = list(set(map(lambda x:x[0],wordcnt.most_common())))

word2index = {}
for i,word in enumerate(vocab):
        word2index[word]=i

concatenated = list() input_dataset = list()
for sent in tokens:
    sent_indices = list()
    for word in sent:
        try:
            sent_indices.append(word2index[word])
            concatenated.append(word2index[word])
        except:
            ""
    input_dataset.append(sent_indices)
concatenated = np.array(concatenated)
random.shuffle(input_dataset)
```

```
alpha, iterations = (0.05, 2)
hidden_size,window,negative = (50,2,5)

weights_0_1 = (np.random.rand(len(vocab),hidden_size) - 0.5) * 0.2
weights_1_2 = np.random.rand(len(vocab),hidden_size)*0

layer_2_target = np.zeros(negative+1)
layer_2_target[0] = 1

def similar(target='beautiful'):
    target_index = word2index[target]

    scores = Counter()
    for word,index in word2index.items():
        raw_difference = weights_0_1[index] - (weights_0_1[target_index])
        squared_difference = raw_difference * raw_difference
        scores[word] = -math.sqrt(sum(squared_difference))
    return scores.most_common(10)

def sigmoid(x):
    return 1/(1 + np.exp(-x))

for rev_i,review in enumerate(input_dataset * iterations):
  for target_i in range(len(review)):

    target_samples = [review[target_i]]+list(concatenated\
    [(np.random.rand(negative)*len(concatenated)).astype('int').tolist()])

    left_context = review[max(0,target_i-window):target_i]
    right_context = review[target_i+1:min(len(review),target_i+window)]

    layer_1 = np.mean(weights_0_1[left_context+right_context],axis=0)
    layer_2 = sigmoid(layer_1.dot(weights_1_2[target_samples].T))
    layer_2_delta = layer_2 - layer_2_target
    layer_1_delta = layer_2_delta.dot(weights_1_2[target_samples])

    weights_0_1[left_context+right_context] -= layer_1_delta * alpha
    weights_1_2[target_samples] -= np.outer(layer_2_delta,layer_1)*alpha

  if(rev_i % 250 == 0):
    sys.stdout.write('\rProgress:'+str(rev_i/float(len(input_dataset)
        *iterations)) + "  " + str(similar('terrible')))
  sys.stdout.write('\rProgress:'+str(rev_i/float(len(input_dataset)
        *iterations)))
print(similar('terrible'))


    Progress:0.99998 [('terrible', -0.0), ('horrible', -2.846300248788519),
    ('brilliant', -3.039932544396419), ('pathetic', -3.4868595532695967),
    ('superb', -3.6092947961276645), ('phenomenal', -3.660172529098085),
    ('masterful', -3.6856112636664564), ('marvelous', -3.9306620801551664),
```

## 11.13 Meaning is derived from loss.

With this new neural network, you can subjectively see that the word embeddings cluster in a rather different way. Where before words were clustered according to their likelihood to predict a `positive` or `negative` label, now they're clustered based on their likelihood to occur within the same phrase (sometimes regardless of sentiment).

***Predicting POS/NEG***

```
print(similar('terrible'))

[('terrible', -0.0),
 ('dull', -0.760788602671491),
 ('lacks', -0.76706470275372),
 ('boring', -0.7682894961694),
 ('disappointing', -0.768657),
 ('annoying', -0.78786389931),
 ('poor', -0.825784172378292),
 ('horrible', -0.83154121717),
 ('laughable', -0.8340279599),
 ('badly', -0.84165373783678)]

print(similar('beautiful'))

[('beautiful', -0.0),
 ('atmosphere', -0.70542101298),
 ('heart', -0.7339429768542354),
 ('tight', -0.7470388145765346),
 ('fascinating', -0.7549291974),
 ('expecting', -0.759886970744),
 ('beautifully', -0.7603669338),
 ('awesome', -0.76647368382398),
 ('masterpiece', -0.7708280057),
 ('outstanding', -0.7740642167)]
```

***Fill in the blank***

```
print(similar('terrible'))

[('terrible', -0.0),
 ('horrible', -2.79600898781),
 ('brilliant', -3.3336178881),
 ('pathetic', -3.49393193646),
 ('phenomenal', -3.773268963),
 ('masterful', -3.8376122586),
 ('superb', -3.9043150978490),
 ('bad', -3.9141673639585237),
 ('marvelous', -4.0470804427),
 ('dire', -4.178749691835959)]

print(similar('beautiful'))

[('beautiful', -0.0),
 ('lovely', -3.0145597243116),
 ('creepy', -3.1975363066322),
 ('fantastic', -3.2551041418),
 ('glamorous', -3.3050812101),
 ('spooky', -3.4881261617587),
 ('cute', -3.592955888181448),
 ('nightmarish', -3.60063813),
 ('heartwarming', -3.6348147),
 ('phenomenal', -3.645669007)]
```

The key takeaway is that, even though the network trained over the same dataset with a very similar architecture (three layers, cross entropy, `sigmoid` nonlinear), you can influence what the network learns within its weights by changing what you tell the network to predict. Even though it's looking at the same statistical information, you can target what it learns based on what you select as the input and target values. For the moment, let's call this process of choosing what you want the network to learn *intelligence targeting*.

Controlling the input/target values isn't the only way to perform intelligence targeting. You can also adjust how the network measures error, the size and types of layers it has, and the types of regularization to apply. In deep learning research, all of these techniques fall under the umbrella of constructing what's called a *loss function*.

### *Neural networks don't really learn data; they minimize the loss function.*

In chapter 4, you learned that learning is about adjusting each weight in the neural network to bring the error down to 0. In this section, I'll explain the same phenomena from a different perspective, choosing the error so the neural network learns the patterns we're interested in. Remember these lessons?

**The golden method for learning**

Adjust each weight in the correct direction and by the correct amount so `error` reduces to 0.

**The secret**

For any `input` and `goal_pred`, an exact relationship is defined between `error` and `weight`, found by combining the `prediction` and `error` formulas.

```
error = ((0.5 * weight) - 0.8) ** 2
```

Perhaps you remember this formula from the one-weight neural network. In that network, you could evaluate the error by first forward propagating (`0.5 * weight`) and then comparing to the target (0.8). I encourage you not to think about this from the perspective of two different steps (forward propagation, then error evaluation), but instead to consider the entire formula (including forward prop) to be the evaluation of an error value. This context will reveal the true cause of the different word-embedding clusterings. Even though the network and datasets were similar, the error function was fundamentally different, leading to different word clusterings within each network.

***Predicting POS/NEG***

```
print(similar('terrible'))

[('terrible', -0.0),
('dull', -0.760788602671491),
('lacks', -0.76706470275372),
('boring', -0.7682894961694),
('disappointing', -0.768657),
('annoying', -0.78786389931),
('poor', -0.825784172378292),
('horrible', -0.83154121717),
('laughable', -0.8340279599),
('badly', -0.84165373783678)]
```

***Fill in the blank***

```
print(similar('terrible'))

[('terrible', -0.0),
 ('horrible', -2.79600898781),
 ('brilliant', -3.3336178881),
 ('pathetic', -3.49393193646),
 ('phenomenal', -3.773268963),
 ('masterful', -3.8376122586),
 ('superb', -3.9043150978490),
 ('bad', -3.9141673639585237),
 ('marvelous', -4.0470804427),
 ('dire', -4.178749691835959)]
```

### *The choice of loss function determines the neural network's knowledge.*

The more formal term for an *error function* is a *loss function* or *objective function* (all three phrases are interchangeable). Considering learning to be all about minimizing a loss function (which includes forward propagation) gives a far broader perspective on how neural networks learn. Two neural networks can have identical starting weights, be trained over identical datasets, and ultimately learn very different patterns because you choose a different loss function. In the case of the two movie review neural networks, the loss function was different because you chose two different target values (positive or negative versus fill in the blank).

Different kinds of architectures, layers, regularization techniques, datasets, and non-linearities aren't really that different. These are the ways you can choose to construct a loss function. If the network isn't learning properly, the solution can often come from any of these possible categories.

For example, if a network is overfitting, you can augment the loss function by choosing simpler nonlinearities, smaller layer sizes, shallower architectures, larger datasets, or more-aggressive regularization techniques. All of these choices will have a fundamentally similar effect on the loss function and a similar consequence on the behavior of the network. They all interplay together, and over time you'll learn how changing one can affect the performance of another; but for now, the important takeaway is that learning is about constructing a loss function and then minimizing it.

Whenever you want a neural network to learn a pattern, everything you need to know to do so will be contained in the loss function. When you had only a single weight, this allowed the loss function to be simple, as you'll recall:

```
error = ((0.5 * weight) - 0.8) ** 2
```

But as you chain large numbers of complex layers together, the loss function will become more complicated (and that's OK). Just remember, if something is going wrong, the solution is in the loss function, which includes both the forward prediction and the raw error evaluation (such as mean squared error or cross entropy).

## 11.14 King – Man + Woman ~= Queen

***Word analogies are an interesting consequence of the previously built network.***

Before closing out this chapter, let's discuss what is, at the time of writing, still one of the most famous properties of neural word embeddings (word vectors like those we just created). The task of filling in the blank creates word embeddings with interesting phenomena known as *word analogies*, wherein you can take the vectors for different words and perform basic algebraic operations on them. For example, if you train the previous network on a large enough corpus, you'll be able to take the vector for `king`, subtract from it the vector for `man`, add in the vector for `woman`, and then search for the most similar vector (other than those in the query). As it turns out, the most similar vector is often the word "queen." There are even similar phenomena in the fill-in-the-blank network trained over movie reviews.

```
def analogy(positive=['terrible','good'],negative=['bad']):

    norms = np.sum(weights_0_1 * weights_0_1,axis=1)
    norms.resize(norms.shape[0],1)

    normed_weights = weights_0_1 * norms

    query_vect = np.zeros(len(weights_0_1[0]))
    for word in positive:
        query_vect += normed_weights[word2index[word]]
    for word in negative:
        query_vect -= normed_weights[word2index[word]]

    scores = Counter()
    for word,index in word2index.items():
        raw_difference = weights_0_1[index] - query_vect
        squared_difference = raw_difference * raw_difference
        scores[word] = -math.sqrt(sum(squared_difference))

    return scores.most_common(10)[1:]
```

***terrible – bad + good ~=***

```
analogy(['terrible','good'],['bad'])
 [('superb', -223.3926217861),
 ('terrific', -223.690648739),
 ('decent', -223.7045545791),
 ('fine', -223.9233021831882),
 ('worth', -224.03031703075),
 ('perfect', -224.125194533),
 ('brilliant', -224.2138041),
 ('nice', -224.244182032763),
 ('great', -224.29115420564)]
```

***elizabeth – she + he ~=***

```
analogy(['elizabeth','he'],['she'])
[('christopher', -192.7003),
 ('it', -193.3250398279812),
 ('him', -193.459063887477),
 ('this', -193.59240614759),
 ('william', -193.63049856),
 ('mr', -193.6426152274126),
 ('bruce', -193.6689279548),
 ('fred', -193.69940566948),
 ('there', -193.7189421836)]
```

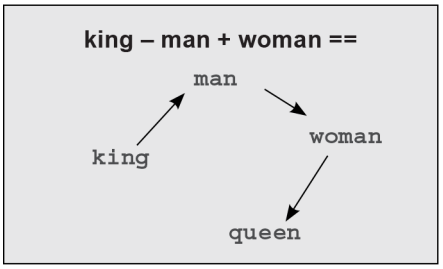
## 11.15 Word analogies

### *Linear compression of an existing property in the data*

When this property was first discovered, it created a flurry of excitement as people extrapolated many possible applications of such a technology. It's an amazing property in its own right, and it did create a veritable cottage industry around generating word embeddings of one variety or another. But the word analogy property in and of itself hasn't grown that much since then, and most of the current work in language focuses instead on recurrent architectures (which we'll get to in chapter 12).

That being said, getting a good intuition for what's going on with word embeddings as a result of a chosen loss function is extremely valuable. You've already learned that the choice of loss function can affect how words are grouped, but this word analogy phenomenon is something different. What about the new loss function causes it to happen?

If you consider a word embedding having two dimensions, it's perhaps easier to envision exactly what it means for these word analogies to work.




```
king  = [0.6 , 0.1]
man   = [0.5 , 0.0]
woman = [0.0 , 0.8]
queen = [0.1 , 1.0]

king - man = [0.1 , 0.1]
queen - woman  = [0.1 , 0.2]
```

The relative usefulness to the final prediction between "king"/"man" and "queen"/"woman" is similar. Why? The difference between "king" and "man" leaves a vector of `royalty`. There are a bunch of male-and female-related words in one grouping, and then there's another grouping in the royal direction.

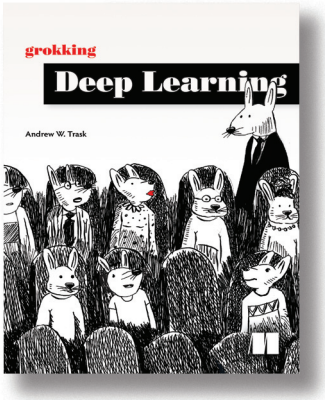
This can be traced back to the chosen loss. When the word "king" shows up in a phrase, it changes the probability of other words showing up in a certain way. It increases the probability of words related to "man" and the probability of words related to royalty. The word "queen" appearing in a phrase increases the probability of words related to "woman" and the probability of words related to royalty (as a group). Thus, because the words have this sort of Venn diagram impact on the output probability, they end up subscribing to similar combinations of groupings. Oversimplified, "king" subscribes to the male and the royal dimensions of the hidden layer, whereas "queen" subscribes to the female and royal dimensions of the hidden layer. Taking the vector for "king" and subtracting out some approximation of the male dimensions and adding in the female ones yields something close to "queen." The most important takeaway is that this is more about the properties of language than deep learning. Any linear compression of these co-occurrence statistics will behave similarly.

## 11.16 Summary

***You've learned a lot about neural word embeddings and the impact of loss on learning.***

In this chapter, we've unpacked the fundamental principles of using neural networks to study language. We started with an overview of the primary problems in natural language processing and then explored how neural networks model language at the word level using word embeddings. You also learned how the choice of loss function can change the kinds of properties that are captured by word embeddings. We finished with a discussion of perhaps the most magical of neural phenomena in this space: word analogies.

As with the other chapters, I encourage you to build the examples in this chapter from scratch. Although it may seem as though this chapter stands on its own, the lessons in loss-function creation and tuning are invaluable and will be extremely important as you tackle increasingly more complicated strategies in future chapters. Good luck!




Artificial Intelligence is one of the most exciting technologies of the century, and Deep Learning is, in many ways, the "brain" behind some of the world's smartest Artificial Intelligence systems out there.

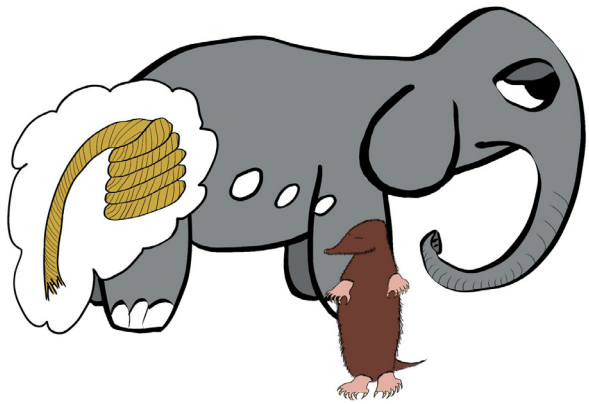
*Grokking Deep Learning* is the perfect place to begin your deep learning journey. Rather than learning the "black box" API of some library or framework, you'll understand how to build these algorithms completely from scratch. You'll understand how Deep Learning's able to learn at levels greater than humans. You'll also be able to understand the "brain" behind state-of-the-art Artificial Intelligence. Furthermore, unlike other courses that assume advanced knowledge of Calculus and use complex mathematical notation, if you're a Python hacker who passed high-school algebra, you're ready to go. And at the end, you'll even build an A.I. that learns to defeat you in a classic Atari game.

***What's inside:***

- How neural networks "learn"
- You'll build neural networks that can see and understand images
- You'll build neural networks that can translate text between languages and even write like Shakespeare
- You'll build neural networks that can learn how to play videogames

Written for readers with high school-level math and intermediate programming skills. Experience with Calculus is helpful but NOT required.

# *Sequence-to-Sequence Models for Chatbots*

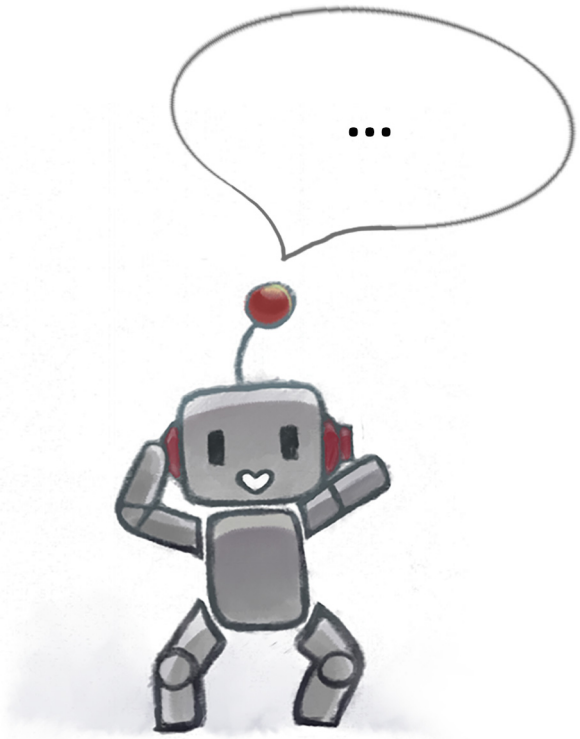


*"The elephant is nothing much more than a limp rope, frayed at the end."*

We now find ourselves at the tail-end of our exploration with more topic to cover. In this final chapter, we'll see how recursive neural networks (RNNs) can be used to model conversations. It also serves as a working introduction to one of the most popular deep learning frameworks in use today: TensorFlow. Using data from hundreds of thousands of conversations from movies, we'll build a chatbot that can give meaningful responses to users. This chatbot's a simplified version of a class of application that has recently become popular: a conversational intelligent agent. Using techniques like the ones in this book companies have built intelligent agents which can do things such as call you a cab, reorder toilet paper, and negotiate your cable bill. The applications of skills like these are endless. Before we wrap up our exploration, let's consider yet one more perspective on the elephant of a topic which is learning from language data.

Chapter 11 from *Machine Learning with TensorFlow* by Nishant Shukla.

# *Sequence-to-sequence models for chatbots*



### This chapter covers

- Examining sequence-to-sequence architecture
- Vector embedding of words
- Implementing a chatbot by using real-world data

Talking to customer service over the phone is a burden for both the customer and the company. Service providers pay a good chunk of money to hire these customer service representatives, but what if it's possible to automate most of this effort? Can we develop software to interface with customers through natural language?

The idea isn't as farfetched as you might think. Chatbots are getting a lot of hype because of unprecedented developments in natural language processing using deep-learning techniques. Perhaps, given enough training data, a chatbot could learn to navigate the most commonly addressed customer problems through natural conversations. If the chatbot were truly efficient, it could not only save the company money by eliminating the need to hire representatives, but even accelerate the customer's search for an answer.

In this chapter, you'll build a chatbot by feeding a neural network thousands of examples of input and output sentences. Your training dataset is a pair of English utterances; for example, if you ask, "How are you?" the chatbot should respond, "Fine, thank you."

> **NOTE** In this chapter, we're thinking of *sequences* and *sentences* as interchangeable concepts. In our implementation, a sentence will be a sequence of letters. Another common approach is to represent a sentence as a sequence of words.

In effect, the algorithm will try to produce an intelligent natural language response to each natural language query. You'll be implementing a neural network that uses two primary concepts taught in previous chapters: multiclass classification and recurrent neural networks (RNNs).

## 11.1 Building on classification and RNNs

Remember, *classification* is a machine-learning approach to predict the category of an input data item. Furthermore, multiclass classification allows for more than two classes. You saw in chapter 4 how to implement such an algorithm in TensorFlow. Specifically, the cost function between the model's prediction (a sequence of numbers) and the ground truth (a one-hot vector) tries to find the distance between two sequences by using the cross-entropy loss.

**NOTE** A one-hot vector is like an all-zero vector, except one of the dimensions has a value of 1.

In this case, implementing a chatbot, you'll use a variant of the cross-entropy loss to measure the difference between two sequences: the model's response (which is a sequence) against the ground truth (which is also a sequence).

**EXERCISE 11.1**

In TensorFlow, you can use the cross-entropy loss function to measure the similarity between a one-hot vector, such as (1, 0, 0), and a neural network's output, such as (2.34, 0.1, 0.3). On the other hand, English sentences aren't numeric vectors. How can you use the cross-entropy loss to measure the similarity between English sentences?

**ANSWER**

A crude approach would be to represent each sentence as a vector by counting the frequency of each word within the sentence. Then compare the vectors to see how closely they match up.

You may recall that RNNs are a neural network design for incorporating not only input from the current time step, but also state information from previous inputs. Chapter 10 covered these in great detail, and they'll be used again in this chapter. RNNs represent input and output as time-series data, which is exactly what you need to represent sequences.

A naïve idea is to use an out-of-the-box RNN to implement a chatbot. Let's see why this is a bad approach. The input and output of the RNN are natural language sentences, so the inputs ($x_t$, $x_{t-1}$, $x_{t-2}$, …) and outputs ($y_t$, $y_{t-1}$, $y_{t-2}$, …) can be sequences of words. The problem in using an RNN to model conversations is that the RNN produces an output result immediately. If your input is a sequence of words (*How, are, you*), the first output word will depend on only the first input word. The output sequence item $y_t$ of the RNN couldn't look ahead to future parts of the input sentence to make a decision; it would be limited by knowledge of only previous input sequences ($x_t$, $x_{t-1}$, $x_{t-2}$, … ). The naïve RNN model tries to come up with a response to the user's query before they've finished asking it, which can lead to incorrect results.

Instead, you'll end up using two RNNs: one for the input sentence and the other for the output sequence. After the input sequence is finished being processed by the first RNN, it'll send the hidden state to the second RNN to process the output sentence. You can see the two RNNs labeled Encoder and Decoder in figure 11.1.

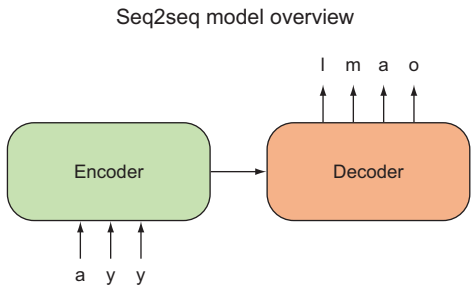



**Figure 11.1** Here's a high-level view of your neural network model. The input *ayy* is passed into the encoder RNN, and the decoder RNN is expected to respond with *lmao*. These are just toy examples for your chatbot, but you could imagine more-complicated pairs of sentences for the input and output.

We're bringing concepts of multiclass classification and RNNs from previous chapters into designing a neural network that learns to map an input sequence to an output sequence. The RNNs provide a way of encoding the input sentence, passing a summarized state vector to the decoder, and then decoding it to a response sentence. To measure the cost between the model's response and the ground truth, we look to the function used in multiclass classification, the cross-entropy loss, for inspiration.

This architecture is called a *sequence-to-sequence (seq2seq) neural network architecture*. The training data you use will be thousands of pairs of sentences mined from movie scripts. The algorithm will observe these dialogue examples and eventually learn to form responses to arbitrary queries you might ask it.

**EXERCISE 11.2**

What other industries could benefit from a chatbot?

**ANSWER**

One example is a conversation partner for young students as an educational tool to teach various subjects such as English, math, and even computer science.

By the end of the chapter, you'll have your own chatbot that can respond somewhat intelligently to your queries. It won't be perfect, because this model always responds the same way for the same input query.

Suppose, for example, that you're traveling to a foreign country without any ability to speak the language. A clever salesman hands you a book, claiming it's all you need to respond to sentences in the foreign language. You're supposed to use it like a dictionary. When someone says a phrase in the foreign language, you can look it up, and the book will have the response written out for you to read aloud: "If someone says *Hello*, you say *Hi*."

Sure, it might be a practical lookup table for small talk, but can a lookup table get you the correct response for arbitrary dialogue? Of course not! Consider looking up

the question "Are you hungry?" The answer to that question is stamped in the book and will never change.

The lookup table is missing state information, which is a key component in dialogue. In your seq2seq model, you'll suffer from a similar issue; but it's a good start! Believe it or not, as of 2017, hierarchical state representation for intelligent dialogue still isn't the norm; many chatbots start out with these seq2seq models.

## 11.2 Seq2seq architecture

The seq2seq model attempts to learn a neural network that predicts an output sequence from an input sequence. Sequences are a little different from traditional vectors, because a sequence implies an ordering of events.

Time is an intuitive way to order events: we usually end up alluding to words related to time, such as *temporal*, *time series*, *past*, and *future*. For example, we like to say that RNNs propagate information to *future time* steps. Or, RNNs capture *temporal dependencies*.

**NOTE** RNNs are covered in detail in chapter 10.

The seq2seq model is implemented using multiple RNNs. A single RNN cell is depicted in figure 11.2; it serves as the building block for the rest of the seq2seq model architecture.

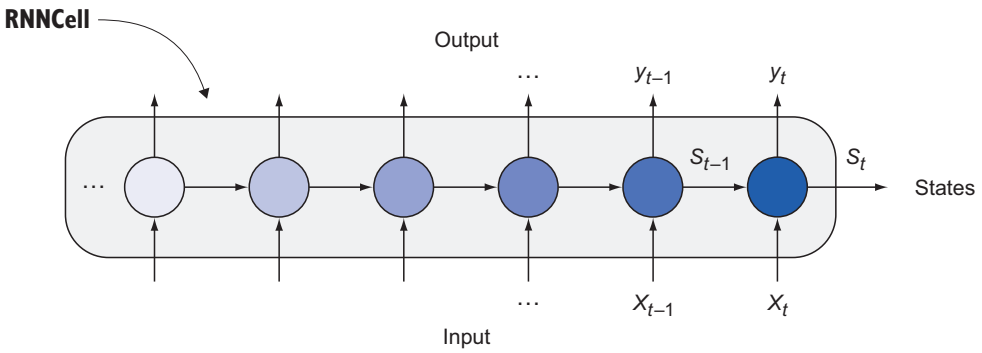



**Figure 11.2 The input, output, and states of an RNN. You can ignore the intricacies of exactly how an RNN is implemented. All that matters is the formatting of your input and output.**

First, you'll learn how to stack RNNs on top of each other to improve the model's complexity. Then you'll learn how to pipe the hidden state of one RNN to another RNN, so that you can have an "encoder" and "decoder" network. As you'll begin to see, it's fairly easy to start using RNNs.

After that, you'll get an introduction to converting natural language sentences into a sequence of vectors. After all, RNNs understand only numeric data, so you'll absolutely need this conversion process. Because a *sequence* is another way of saying "a list of tensors," you need to make sure you can convert your data accordingly. For example, a sentence is a sequence of words, but words aren't tensors. The process of converting words to tensors or, more commonly, vectors is called *embedding*.

Last, you'll put all these concepts together to implement the seq2seq model on real-world data. The data will come from thousands of conversations from movie scripts.

You can hit the ground running with the following code listing. Open a new Python file, and start copying listing 11.1 to set up constants and placeholders. You'll define the shape of the placeholder to be `[None, seq_size, input_dim]`, where `None` means the size is dynamic because the batch size may change, `seq_size` is the length of the sequence, and `input_dim` is the dimension of each sequence item.

**Listing 11.1 Setting up constants and placeholders**

```
import tensorflow as tf

input_dim = 1
seq_size = 6

input_placeholder = tf.placeholder(dtype=tf.float32,
                                   shape=[None, seq_size, input_dim])
```

All you need is TensorFlow.

Dimension of each sequence element

Maximum length of sequence

To generate an RNN cell like the one in figure 11.2, TensorFlow provides a helpful `LSTMCell` class. Listing 11.2 shows how to use it and extract the outputs and states from the cell. Just for convenience, the listing defines a helper function called `make_cell` to set up the LSTM RNN cell. Remember, just defining a cell isn't enough: you also need to call `tf.nn.dynamic_rnn` on it to set up the network.

**Listing 11.2 Making a simple RNN cell**

```
def make_cell(state_dim):
    return tf.contrib.rnn.LSTMCell(state_dim)

with tf.variable_scope("first_cell") as scope:
    cell = make_cell(state_dim=10)
    outputs, states = tf.nn.dynamic_rnn(cell,
                                        input_placeholder,
                                        dtype=tf.float32)
```

Check out the tf.contrib.rnn documentation for other types of cells, such as GRU.

There will be two generated results: outputs and states.

This is the input sequence to the RNN.

You might remember from previous chapters that you can improve a neural network's complexity by adding more and more hidden layers. More layers means more parameters, and that likely means the model can represent more functions; it's more flexible.

You know what? You can stack cells on top of each other. Nothing is stopping you. Doing so makes the model more complex, so perhaps this two-layered RNN model will perform better because it's more expressive. Figure 11.3 shows two cells stacked together.

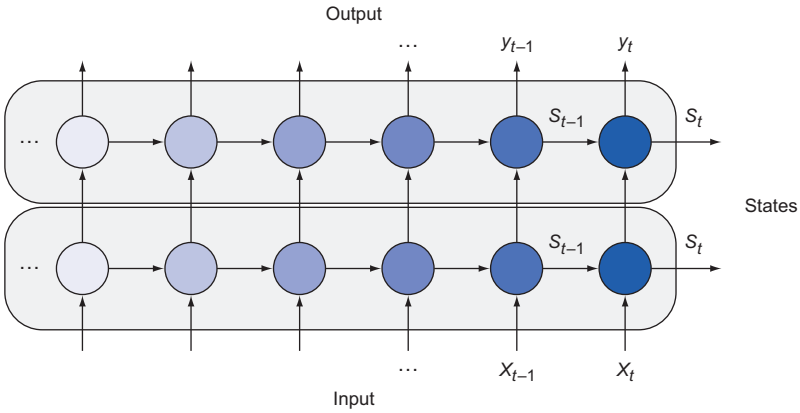



**Figure 11.3** **You can stack RNN cells to form a more complicated architecture.**

**WARNING** The more flexible the model, the more likely that it'll overfit the training data.

In TensorFlow, you can intuitively implement this two-layered RNN network. First, you create a new variable scope for the second cell. To stack RNNs together, you can pipe the output of the first cell to the input of the second cell. The following listing shows how to do exactly this.

**Listing 11.3 Stacking two RNN cells**

```
with tf.variable_scope("second_cell") as scope:
    cell2 = make_cell(state_dim=10)
    outputs2, states2 = tf.nn.dynamic_rnn(cell2,
                                          outputs,
                                          dtype=tf.float32)
```

**Defining a variable scope helps avoid runtime errors due to variable reuse.**

**Input to this cell will be the other cell's output.**

What if you wanted four layers of RNNs? Or 10? For example, figure 11.4 shows four RNN cells stacked atop each other.

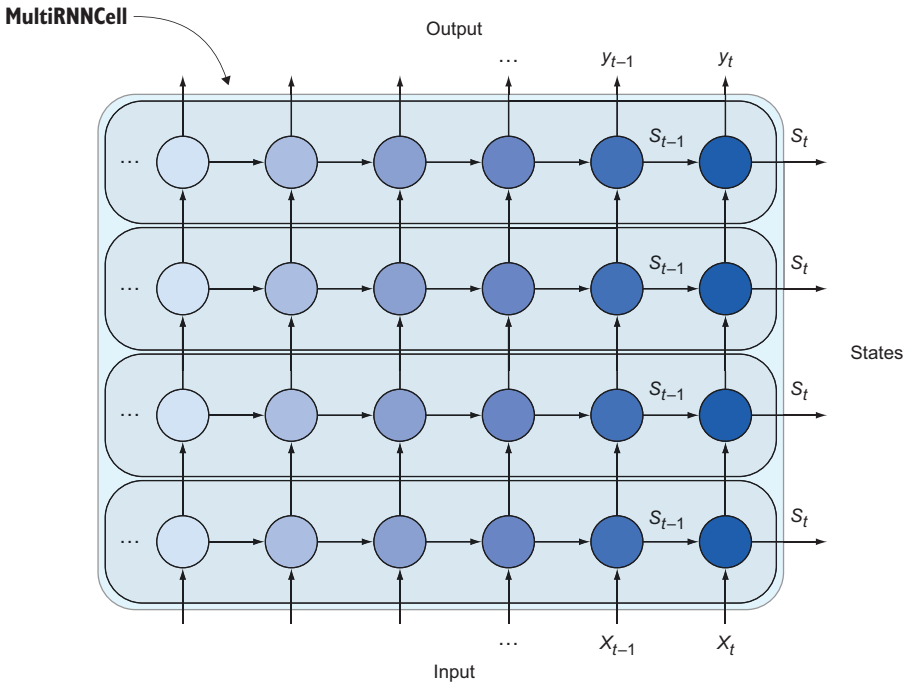



**Figure 11.4** **TensorFlow lets you stack as many RNN cells as you want.**

A useful shortcut for stacking cells that the TensorFlow library supplies is called `MultiRNNCell`. The following listing shows how to use this helper function to build arbitrarily large RNN cells.

**Listing 11.4 Using `MultiRNNCell` to stack multiple cells**

```
def make_multi_cell(state_dim, num_layers):
    cells = [make_cell(state_dim) for _ in range(num_layers)]
    return tf.contrib.rnn.MultiRNNCell(cells)

multi_cell = make_multi_cell(state_dim=10, num_layers=4)
outputs4, states4 = tf.nn.dynamic_rnn(multi_cell,
                                      input_placeholder,
                                      dtype=tf.float32)
```

**The for-loop syntax is the preferred way to construct a list of RNN cells.**

So far, you've grown RNNs vertically by piping outputs of one cell to the inputs of another. In the seq2seq model, you'll want one RNN cell to process the input sen-

tence, and another RNN cell to process the output sentence. To communicate between the two cells, you can also connect RNNs horizontally by connecting states from cell to cell, as shown in figure 11.5.

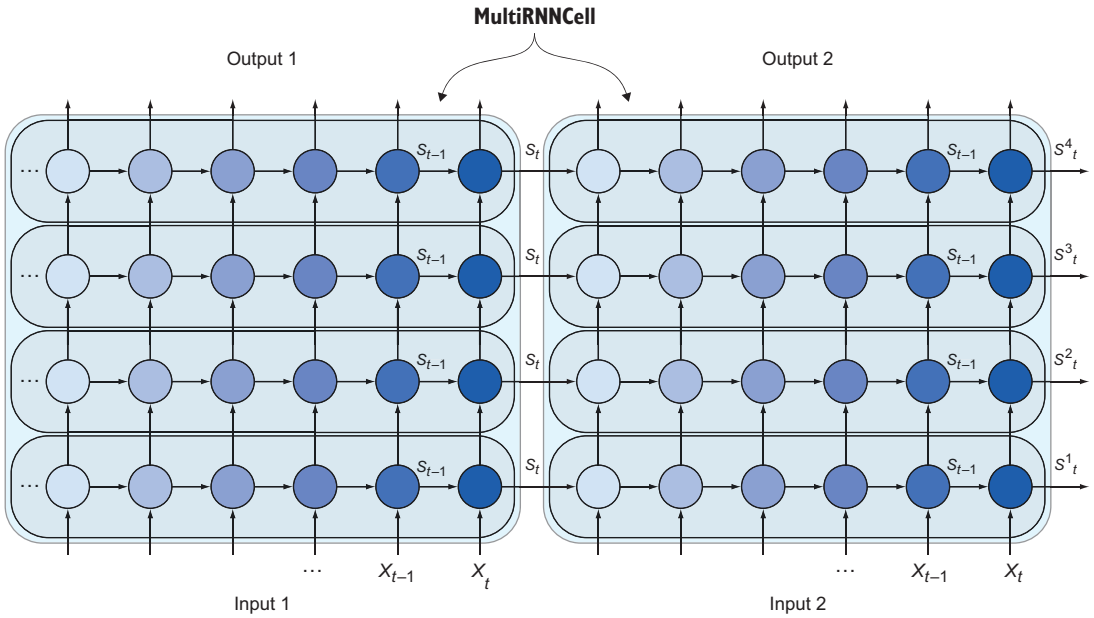



**Figure 11.5** **You can use the last states of the first cell as the next cell's initial state. This model can learn mapping from an input sequence to an output sequence. The model is called *seq2seq*.**

You've stacked RNN cells vertically and connected them horizontally, vastly increasing the number of parameters in the network! Is this utter blasphemy? Yes. You've built a monolithic architecture by composing RNNs every which way. But there's a method to this madness, because this insane neural network architecture is the backbone of the seq2seq model.

As you can see in figure 11.5, the seq2seq model appears to have two input sequences and two output sequences. But only input 1 will be used for the input sentence, and only output 2 will be used for the output sentence.

You may be wondering what to do with the other two sequences. Strangely enough, the output 1 sequence is entirely unused by the seq2seq model. And, as you'll see, the input 2 sequence is crafted using some of output 2 data, in a feedback loop.

Your training data for designing a chatbot will be pairs of input and output sentences, so you'll need to better understand how to embed words into a tensor. The next section covers how to do so in TensorFlow.

**EXERCISE 11.3**

Sentences may be represented by a sequence of characters or words, but can you think of other sequential representations of sentences?

**ANSWER**

Phrases and grammatical information (verbs, nouns, and so forth) could both be used. More frequently, real applications use *natural language processing* (NLP) lookups to standardize word forms, spellings, and meanings. One example of a library that does this translation is *fastText* from Facebook (https://github.com/facebookresearch/fastText).

## 11.3 Vector representation of symbols

Words and letters are symbols, and converting symbols to numeric values is easy in TensorFlow. For example, let's say you have four words in your vocabulary: word$_0$: *the*; word$_1$: *fight*; word$_2$: *wind*; and word$_3$: *like*.

Let's say you want to find the embeddings for the sentence "Fight the wind." The symbol *fight* is located at index 1 of the lookup table, *the* at index 0, and *wind* at index 2. If you want to find the embedding of the word *fight*, you have to refer to its index, which is 1, and consult the lookup table at index 1 to identify the embedding value. In our first example, each word is associated with a number, as shown in figure 11.6.

| Word | Number |
|---|---|
| the | 17 |
| fight | 22 |
| wind | 35 |
| like | 51 |

**Figure 11.6 A mapping from symbols to scalars**

The following listing shows how to define such a mapping between symbols and numeric values using TensorFlow code.

**Listing 11.5 Defining a lookup table of scalars**

```
embeddings_0d = tf.constant([17, 22, 35, 51])
```

Or maybe the words are associated with vectors, as shown in figure 11.7. This is often the preferred method of representing words. You can find a thorough tutorial on vector representation of words in the official TensorFlow docs: http://mng.bz/35M8.

| Word | Vector |
|---|---|
| the | [1, 0, 0, 0] |
| fight | [0, 1, 0, 0] |
| wind | [0, 0, 1, 0] |
| like | [0, 0, 0, 1] |

**Figure 11.7 A mapping from symbols to vectors**

You can implement the mapping between words and vectors in TensorFlow, as shown in the following listing.

**Listing 11.6 Defining a lookup table of 4D vectors**

```
embeddings_4d = tf.constant([[1, 0, 0, 0],
                             [0, 1, 0, 0],
                             [0, 0, 1, 0],
                             [0, 0, 0, 1]])
```

This may sound over the top, but you can represent a symbol by a tensor of any rank you want, not just numbers (rank 0) or vectors (rank 1). In figure 11.8, you're mapping symbols to tensors of rank 2.

| Word | Tensor |
|---|---|
| the | [[1, 0], [0, 0]] |
| fight | [[0, 1], [0, 0]] |
| wind | [[0, 0], [1, 0]] |
| like | [[0, 0], [0, 1]] |

**Figure 11.8 A mapping from symbols to tensors**

The following listing shows how to implement this mapping of words to tensors in TensorFlow.

**Listing 11.7 Defining a lookup table of tensors**

```
embeddings_2x2d = tf.constant([[[1, 0], [0, 0]],
                               [[0, 1], [0, 0]],
                               [[0, 0], [1, 0]],
                               [[0, 0], [0, 1]]])
```

The `embedding_lookup` function provided by TensorFlow is an optimized way to access embeddings by indices, as shown in the following listing.

**Listing 11.8 Looking up the embeddings**

```
ids = tf.constant([1, 0, 2])
lookup_0d = sess.run(tf.nn.embedding_lookup(embeddings_0d, ids))
print(lookup_0d)

lookup_4d = sess.run(tf.nn.embedding_lookup(embeddings_4d, ids))
print(lookup_4d)

lookup_2x2d = sess.run(tf.nn.embedding_lookup(embeddings_2x2d, ids))
print(lookup_2x2d)
```

**Embeddings lookup corresponding to the words *fight*, *the*, and *wind***

In reality, the embedding matrix isn't something you ever have to hardcode. These listings are for you to understand the ins and outs of the `embedding_lookup` function in TensorFlow, because you'll be using it heavily soon. The embedding lookup table will be learned automatically over time by training the neural network. You start by defining a random, normally distributed lookup table. Then, TensorFlow's optimizer will adjust the matrix values to minimize the cost.

**EXERCISE 11.4**

Follow the official TensorFlow word2vec tutorial to get more familiar with embeddings: www.tensorflow.org/tutorials/word2vec.

**ANSWER**

This tutorial will teach you to visualize the embeddings using TensorBoard.

## 11.4 Putting it all together

The first step in using natural language input in a neural network is to decide on a mapping between symbols and integer indices. Two common ways to represent sentences is by a sequence of *letters* or a sequence of *words*. Let's say, for simplicity, that you're dealing with sequences of letters, so you'll need to build a mapping between characters and integer indices.

> **NOTE** The official code repository is available at the book's website (www.manning.com/books/machine-learning-with-tensorflow) and on GitHub (http://mng.bz/EB5A). From there, you can get the code running without needing to copy and paste from the book.

The following listing shows how to build mappings between integers and characters. If you feed this function a list of strings, it'll produce two dictionaries, representing the mappings.

**Listing 11.9 Extracting character vocab**

```
def extract_character_vocab(data):
    special_symbols = ['<PAD>', '<UNK>', '<GO>',  '<EOS>']
    set_symbols = set([character for line in data for character in line])
    all_symbols = special_symbols + list(set_symbols)
    int_to_symbol = {word_i: word
                     for word_i, word in enumerate(all_symbols)}
    symbol_to_int = {word: word_i
                     for word_i, word in int_to_symbol.items()}

    return int_to_symbol, symbol_to_int

input_sentences = ['hello stranger', 'bye bye']
output_sentences = ['hiya', 'later alligator']

input_int_to_symbol, input_symbol_to_int =
    extract_character_vocab(input_sentences)

output_int_to_symbol, output_symbol_to_int =
    extract_character_vocab(output_sentences
```

List of input sentences for training

List of corresponding output sentences for training

Next, you'll define all your hyperparameters and constants in listing 11.10. These are usually values you can tune by hand through trial and error. Typically, greater values for the number of dimensions or layers result in a more complex model, which is rewarding if you have big data, fast processing power, and lots of time.

**Listing 11.10 Defining hyperparameters**

```
NUM_EPOCS = 300
RNN_STATE_DIM = 512
RNN_NUM_LAYERS = 2
ENCODER_EMBEDDING_DIM = DECODER_EMBEDDING_DIM = 64

BATCH_SIZE = int(32)
LEARNING_RATE = 0.0003

INPUT_NUM_VOCAB = len(input_symbol_to_int)
OUTPUT_NUM_VOCAB = len(output_symbol_to_int)
```

Number of epochs

RNN's hidden dimension size

RNN's number of stacked cells

Embedding dimension of sequence elements for the encoder and decoder

Batch size

It's possible to have different vocabularies between the encoder and decoder.

Let's list all placeholders next. As you can see in listing 11.11, the placeholders nicely organize the input and output sequences necessary to train the network. You'll have to track both the sequences and their lengths. For the decoder part, you'll also need to compute the maximum sequence length. The `None` value in the shape of these

placeholders means the tensor may take on an arbitrary size in that dimension. For example, the batch size may vary in each run. But for simplicity, you'll keep the batch size the same at all times.

**Listing 11.11 Listing placeholders**

```
# Encoder placeholders
encoder_input_seq = tf.placeholder(
    tf.int32,
    [None, None],
    name='encoder_input_seq'
)

encoder_seq_len = tf.placeholder(
    tf.int32,
    (None,),
    name='encoder_seq_len'
)

# Decoder placeholders
decoder_output_seq = tf.placeholder(
    tf.int32,
    [None, None],
    name='decoder_output_seq'
)

decoder_seq_len = tf.placeholder(
    tf.int32,
    (None,),
    name='decoder_seq_len'
)

max_decoder_seq_len = tf.reduce_max(
    decoder_seq_len,
    name='max_decoder_seq_len'
)
```

Sequence of integers for the encoder's input

Shape is batch-size × sequence length

Lengths of sequences in a batch

Shape is dynamic because the length of a sequence can change

Sequence of integers for the decoder's output

Shape is batch-size × sequence length

Lengths of sequences in a batch

Shape is dynamic because the length of a sequence can change

Maximum length of a decoder sequence in a batch

Let's define helper functions to construct RNN cells. These functions, shown in the following listing, should appear familiar to you from the previous section.

**Listing 11.12 Helper functions to build RNN cells**

```
def make_cell(state_dim):
    lstm_initializer = tf.random_uniform_initializer(-0.1, 0.1)
    return tf.contrib.rnn.LSTMCell(state_dim, initializer=lstm_initializer)

def make_multi_cell(state_dim, num_layers):
    cells = [make_cell(state_dim) for _ in range(num_layers)]
    return tf.contrib.rnn.MultiRNNCell(cells)
```

You'll build the encoder and decoder RNN cells by using the helper functions you've just defined. As a reminder, we've copied the seq2seq model for you in figure 11.9, to visualize the encoder and decoder RNNs.

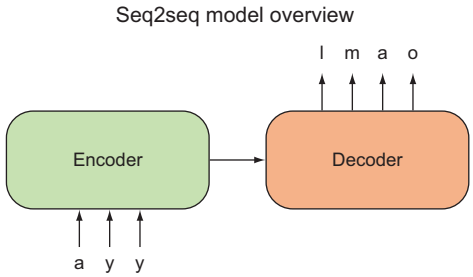



**Figure 11.9 The seq2seq model learns a transformation between an input sequence to an output sequence by using an encoder RNN and a decoder RNN.**

Let's talk about the encoder cell part first, because in listing 11.13 you'll build the encoder cell. The produced states of the encoder RNN will be stored in a variable called `encoder_state`. RNNs also produce an output sequence, but you don't need access to that in a standard seq2seq model, so you can ignore it or delete it.

It's also typical to convert letters or words in a vector representation, often called *embedding*. TensorFlow provides a handy function called `embed_sequence` that can help you embed the integer representation of symbols. Figure 11.10 shows how the encoder input accepts numeric values from a lookup table. You can see it in action at the beginning of listing 11.13.

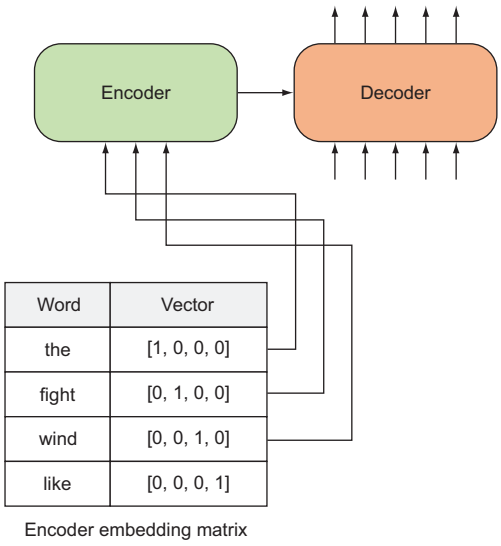



**Figure 11.10 The RNNs accept only sequences of numeric values as input or output, so you'll convert your symbols to vectors. In this case, the symbols are words, such as *the*, *fight*, *wind*, and *like*. Their corresponding vectors are associated in the embedding matrix.**

**Listing 11.13 Encoder embedding and cell**

```
# Encoder embedding

encoder_input_embedded = tf.contrib.layers.embed_sequence(
    encoder_input_seq,
    INPUT_NUM_VOCAB,
    ENCODER_EMBEDDING_DIM
)


# Encoder output

encoder_multi_cell = make_multi_cell(RNN_STATE_DIM, RNN_NUM_LAYERS)

encoder_output, encoder_state = tf.nn.dynamic_rnn(
    encoder_multi_cell,
    encoder_input_embedded,
    sequence_length=encoder_seq_len,
    dtype=tf.float32
)

del(encoder_output)
```

Input seq of numbers (row indices)

Rows of embedding matrix

Columns of embedding matrix

You don't need to hold on to that value.

The decoder RNN's output is a sequence of numeric values representing a natural language sentence and a special symbol to represent that the sequence has ended. You'll label this end-of-sequence symbol as <EOS>. Figure 11.11 illustrates this process. The input sequence to the decoder RNN will look similar to the decoder's output sequence, except instead of having the <EOS> (end of sequence) special symbol at the end of each sentence, it will have a <GO> special symbol at the front. That way, after the decoder reads its input from left to right, it starts out with no extra information about the answer, making it a robust model.

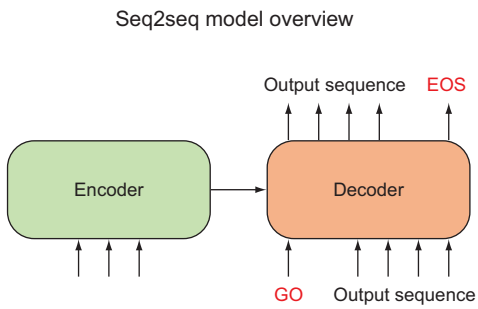



**Figure 11.11 The decoder's input is prefixed with a special <GO> symbol, whereas the output is suffixed by a special <EOS> symbol.**

Listing 11.14 shows how to correctly perform these slicing and concatenating operations. The newly constructed sequence for the decoder's input will be called `decoder_input_seq`. You'll use TensorFlow's `tf.concat` operation to glue together

matrices. In the listing, you define a `go_prefixes` matrix, which will be a column vector containing only the `<GO>` symbol.

**Listing 11.14 Preparing input sequences to the decoder**

```
decoder_raw_seq = decoder_output_seq[:, :-1]
go_prefixes = tf.fill([BATCH_SIZE, 1], output_symbol_to_int['<GO>'])
decoder_input_seq = tf.concat([go_prefixes, decoder_raw_seq], 1)
```

Crops the matrix by ignoring the very last column

Creates a column vector of <GO> symbols

Concatenates the <GO> vector to the beginning of the cropped matrix

Now let's construct the decoder cell. As shown in listing 11.15, you'll first embed the decoder sequence of integers into a sequence of vectors, called `decoder_input_embedded`.

The embedded version of the input sequence will be fed to the decoder's RNN, so go ahead and create the decoder RNN cell. One more thing: you'll need a layer to map the output of the decoder to a one-hot representation of the vocabulary, which you call `output_layer`. The process of setting up the decoder starts out to be similar to that with the encoder.

**Listing 11.15 Decoder embedding and cell**

```
decoder_embedding = tf.Variable(tf.random_uniform([OUTPUT_NUM_VOCAB,
                                                   DECODER_EMBEDDING_DIM]))
decoder_input_embedded = tf.nn.embedding_lookup(decoder_embedding,
                                                decoder_input_seq)

decoder_multi_cell = make_multi_cell(RNN_STATE_DIM, RNN_NUM_LAYERS)

output_layer_kernel_initializer =
    tf.truncated_normal_initializer(mean=0.0, stddev=0.1)
output_layer = Dense(
    OUTPUT_NUM_VOCAB,
    kernel_initializer = output_layer_kernel_initializer
)
```

Okay, here's where things get weird. You have two ways to retrieve the decoder's output: during training and during inference. The training decoder will be used only during training, whereas the inference decoder will be used for testing on never-before-seen data.

The reason for having two ways to obtain an output sequence is that during training, you have the ground-truth data available, so you can use information about the known output to help speed the learning process. But during inference, you have no

ground-truth output labels, so you must resort to making inferences by using only the input sequence.

The following listing implements the training decoder. You'll feed `decoder_input_seq` into the decoder's input, using `TrainingHelper`. This helper op manages the input to the decoder RNN for you.

**Listing 11.16 Decoder output (training)**

```
with tf.variable_scope("decode"):

    training_helper = tf.contrib.seq2seq.TrainingHelper(
        inputs=decoder_input_embedded,
        sequence_length=decoder_seq_len,
        time_major=False
    )

    training_decoder = tf.contrib.seq2seq.BasicDecoder(
        decoder_multi_cell,
        training_helper,
        encoder_state,
        output_layer
    )

    training_decoder_output_seq, _, _ = tf.contrib.seq2seq.dynamic_decode(
        training_decoder,
        impute_finished=True,
        maximum_iterations=max_decoder_seq_len
    )
```

If you care to obtain output from the seq2seq model on test data, you no longer have access to `decoder_input_seq`. Why? Well, the decoder input sequence is derived from the decoder output sequence, which is available only with the training dataset.

The following listing implements the decoder output op for the inference case. Here again, you'll use a helper op to feed the decoder an input sequence.

**Listing 11.17 Decoder output (inference)**

```
with tf.variable_scope("decode", reuse=True):
    start_tokens = tf.tile(
        tf.constant([output_symbol_to_int['<GO>']],
                    dtype=tf.int32),
        [BATCH_SIZE],
        name='start_tokens')

    inference_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(
        embedding=decoder_embedding,
        start_tokens=start_tokens,
        end_token=output_symbol_to_int['<EOS>']
    )
```

**Helper for the inference process**

```
inference_decoder = tf.contrib.seq2seq.BasicDecoder(
    decoder_multi_cell,
    inference_helper,
    encoder_state,
    output_layer
)
```

Basic decoder

Performs dynamic decoding using the decoder

```
inference_decoder_output_seq, _, _ = tf.contrib.seq2seq.dynamic_decode(
    inference_decoder,
    impute_finished=True,
    maximum_iterations=max_decoder_seq_len
)
```

Compute the cost using TensorFlow's `sequence_loss` method. You'll need access to the inferred decoder output sequence and the ground-truth output sequence. The following listing defines the cost function in code.

**Listing 11.18 Cost function**

Renames the tensors for your convenience

```
training_logits =
    tf.identity(training_decoder_output_seq.rnn_output, name='logits')
inference_logits =
    tf.identity(inference_decoder_output_seq.sample_id, name='predictions')
```

```
masks = tf.sequence_mask(
    decoder_seq_len,
    max_decoder_seq_len,
    dtype=tf.float32,
    name='masks'
)
```

Creates the weights for sequence_loss

```
cost = tf.contrib.seq2seq.sequence_loss(
    training_logits,
    decoder_output_seq,
    masks
)
```

Uses TensorFlow's built-in sequence loss function

Last, let's call an optimizer to minimize the cost. But you'll do one trick you might have never seen before. In deep networks like this one, you need to limit extreme gradient change to ensure that the gradient doesn't change too dramatically, a technique called *gradient clipping*. Listing 11.19 shows you how to do so.

**EXERCISE 11.5**

Try the seq2seq model without gradient clipping to experience the difference.

**ANSWER**

You'll notice that without gradient clipping, sometimes the network adjusts the gradients too much, causing numerical instabilities.

**Listing 11.19 Calling an optimizer**

```
optimizer = tf.train.AdamOptimizer(LEARNING_RATE)

gradients = optimizer.compute_gradients(cost)
capped_gradients = [(tf.clip_by_value(grad, -5., 5.), var)
                    for grad, var in gradients if grad is not None]
train_op = optimizer.apply_gradients(capped_gradients)
```

Gradient clipping

That concludes the seq2seq model implementation. In general, the model is ready to be trained after you've set up the optimizer, as in the previous listing. You can create a session and run `train_op` with batches of training data to learn the parameters of the model.

Oh, right, you need training data from someplace! How can you obtain thousands of pairs of input and output sentences? Fear not—the next section covers exactly that.

## 11.5 Gathering dialogue data

The Cornell Movie Dialogues corpus (http://mng.bz/W28O) is a dataset of more than 220,000 conversations from more than 600 movies. You can download the zip file from the official web page.

> **WARNING** Because there's a huge amount of data, you can expect the training algorithm to take a long time. If your TensorFlow library is configured to use only the CPU, it might take an entire day to train. On a GPU, training this network may take 30 minutes to an hour.

An example of a small snippet of back-and-forth conversation between two people (A and B) is the following:

*A: They do not!*

*B: They do too!*

*A: Fine.*

Because the goal of the chatbot is to produce intelligent output for every possible input utterance, you'll structure your training data based on contingent pairs of conversation. In the example, the dialogue generates the following pairs of input and output sentences:

- "They do not!" → "They do too!"
- "They do too!" → "Fine."

For your convenience, we've already processed the data and made it available for you online. You can find it at www.manning.com/books/machine-learning-with-tensorflow or http://mng.bz/wWo0. After completing the download, you can run the following listing, which uses the `load_sentences` helper function from the GitHub repo under the `Concept03_seq2seq.ipynb` Jupyter Notebook.

**Listing 11.20 Training the model**

```
input_sentences = load_sentences('data/words_input.txt')
output_sentences = load_sentences('data/words_output.txt')

input_seq = [
    [input_symbol_to_int.get(symbol, input_symbol_to_int['<UNK>'])
        for symbol in line]
    for line in input_sentences
]

output_seq = [
    [output_symbol_to_int.get(symbol, output_symbol_to_int['<UNK>'])
        for symbol in line] + [output_symbol_to_int['<EOS>']]
    for line in output_sentences
]

sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()

for epoch in range(NUM_EPOCS + 1):

    for batch_idx in range(len(input_sentences) // BATCH_SIZE):

        input_data, output_data = get_batches(input_sentences,
                                              output_sentences,
                                              batch_idx)

        input_batch, input_lenghts = input_data[batch_idx]
        output_batch, output_lengths = output_data[batch_idx]

        _, cost_val = sess.run(
            [train_op, cost],
            feed_dict={
                encoder_input_seq: input_batch,
                encoder_seq_len: input_lengths,
                decoder_output_seq: output_batch,
                decoder_seq_len: output_lengths
            }
        )

saver.save(sess, 'model.ckpt')
sess.close()
```

Loads the input sentences as a list of strings

Loads the corresponding output sentences the same way

Loops through the letters

Loops through the lines of text

Appends the EOS symbol to the end of the output data

Loops through the lines

Loops through the epochs

It's a good idea to save the learned parameters.

Loops by the number of batches

Gets input and output pairs for the current batch

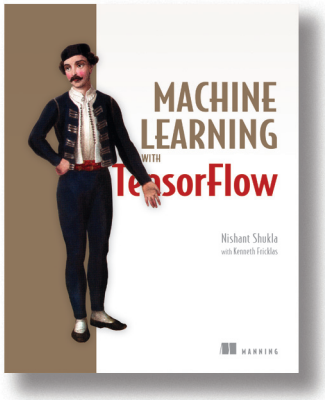
Runs the optimizer on the current batch

Because you saved the model parameters to a file, you can easily load it onto another program and query the network for responses to new input. Run the `inference_logits` op to obtain the chatbot response.

## 11.6 Summary

In this chapter, you built a real-world example of a seq2seq network, putting to work all the TensorFlow knowledge you learned in the previous chapters:

- You built a seq2seq neural network by putting to work all the TensorFlow knowledge you've acquired from the book so far.
- You learned how to embed natural language in TensorFlow.
- You used RNNs as a building block for a more interesting model.
- After training the model on examples of dialogue from movie scripts, you were able to treat the algorithm like a chatbot, inferring natural language responses from natural language input.




*Machine Learning with TensorFlow* teaches machine learning algorithms and how to implement solutions with TensorFlow. You'll start with an overview of machine learning concepts. Next, you'll learn the essentials needed to begin using TensorFlow before moving on to specific machine learning problems and solutions. With lots of diagrams, code examples, and exercises, this tutorial teaches you cutting-edge machine learning algorithms and techniques to solve them. Each chapter zooms into a prominent example of machine learning, such as classification, regression, anomaly detection, clustering, and neural networks. Cover them all to master the basics, or cater to your needs by skipping around. By the end of this book, you'll be able to solve classification, clustering, regression, and prediction problems in the real world.

***What's inside:***

- Formulating machine learning frameworks for real-world problems
- Understanding machine learning problems
- Solving problems with TensorFlow
- Visualizing algorithms with TensorBoards
- Using well-studied neural network architectures
- Reusing provided code for your own applications

This book’s for programmers who have some experience with Python and linear algebra concepts like vectors and matrices. No experience with machine learning’s necessary.

# index

## Symbols



## A



## B



## C



## D

## E



## F



## G



## H



## I

## J



## K



## L



## M



## N

## O



## P



## Q



## R



## S

## T



## V



## W

## Z


manning