O'REILLY®

Deep Learning Cookbook

PRACTICAL RECIPES TO GET STARTED QUICKLY

Douwe Osinga

O'REILLY®

Deep Learning Cookbook

Deep learning doesn't have to be intimidating. Until recently, this machine learning method required years of study, but with frameworks such as Keras and TensorFlow, software engineers without a background in machine learning can quickly enter the field. With the recipes in this cookbook, you'll learn how to solve deep learning problems for classifying and generating text, images, and music.

Each chapter consists of several recipes needed to complete a single project, such as training a music recommender system. Author Douwe Osinga also provides a chapter with half a dozen techniques to help you if you're stuck. Examples are written in Python with code available on GitHub as a set of Python notebooks.

You'll learn how to:

- Create applications that will serve real users
- Use word embeddings to calculate text similarity
- Build a movie recommender system based on Wikipedia links
- Learn how AIs see the world by visualizing their internal state
- Build a model to suggest emojis for pieces of text
- Reuse pretrained networks to build an inverse image search service
- Compare how GANs, autoencoders, and LSTMs generate icons
- Detect music styles and index song collections

Douwe Osinga is an experienced software engineer formerly with Google, a globetrotter, and the founder of three startups. His popular software project website (*https://douweosinga.com/projects*) covers many areas of interest, including machine learning.

"This book is a great way to get started with deep learning for those who prefer practical results over theory. It helped the engineering team of my new music-tech startup Weav to get going with deep learning quickly. This book is perfect for anyone interested in learning about practical ML."

—**Lars Rasmussen**
cocreator of Google Maps

US $59.99 CAN $79.99
ISBN: 978-1-491-99584-6

Twitter: @oreillymedia
facebook.com/oreilly

Deep Learning Cookbook

Practical Recipes to Get Started Quickly

Douwe Osinga

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Deep Learning Cookbook
by Douwe Osinga

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Rachel Roumeliotis and Jeff Bleiel
Production Editor: Colleen Cole
Copyeditor: Rachel Head
Proofreader: Charles Roumeliotis
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

June 2018: First Edition

Revision History for the First Edition
2018-05-23: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491995846* for release details.

978-1-491-99584-6

[LSI]

Table of Contents

Preface

A Brief History of Deep Learning

The roots of the current deep learning boom go surprisingly far back, to the 1950s. While vague ideas of "intelligent machines" can be found further back in fiction and speculation, the 1950s and '60s saw the introduction of the first "artificial neural networks," based on a dramatically simplified model of biological neurons. Amongst these models, the Perceptron system articulated by Frank Rosenblatt garnered particular interest (and hype). Connected to a simple "camera" circuit, it could learn to distinguish different types of objects. Although the first version ran as software on an IBM computer, subsequent versions were done in pure hardware.

Interest in the multilayer perceptron (MLP) model continued through the '60s. This changed when, in 1969, Marvin Minksy and Seymour Papert published their book *Perceptrons* (MIT Press). The book contained a proof showing that linear perceptrons could not classify the behavior of a nonlinear function (XOR). Despite the limitations of the proof (nonlinear perceptron models existed at the time of the book's publication, and are even noted by the authors), its publication heralded the plummeting of funding for neural network models. Research would not recover until the 1980s, with the rise of a new generation of researchers.

The increase in computing power together with the development of the backpropagation technique (known in various forms since the '60s, but not applied in general until the '80s) prompted a resurgence of interest in neural networks. Not only did computers have the power to train larger networks, but we also had the techniques to train deeper networks efficiently. The first convolutional neural networks combined these insights with a model of visual recognition from mammalian brains, yielding for the first time networks that could efficiently recognize complex images such as handwritten digits and faces. Convolutional networks do this by applying the same "subnetwork" to different locations of the image and aggregating the results of these into higher-level features. In Chapter 12 we look at how this works in more detail.

In the ’90s and early 2000s interest in neural networks declined again as more “understandable” models like support vector machines (SVMs) and decision trees became popular. SVMs proved to be excellent classifiers for many data sources of the time, especially when coupled with human-engineered features. In computer vision, “feature engineering” became popular. This involves building feature detectors for small elements in a picture and combining them by hand into something that recognizes more complex forms. It later turned out that deep learning nets learn to recognize very similar features and learn to combine them in a very similar way. In Chapter 12 we explore some of the inner workings of these networks and visualize what they learn.

With the advent of general-purpose programming on graphics processing units (GPUs) in the late 2000s, neural network architectures were able to make great strides over the competition. GPUs contain thousands of small processors that can do trillions of operations per second in parallel. Originally developed for computer gaming, where this is needed to render complex 3D scenes in real time, it turned out that the same hardware can be used to train neural networks in parallel, achieving speed improvements of a factor of 10 or higher.

The other thing that happened was that the internet made very large training sets available. Where researchers had been training classifiers with thousands of images before, now they had access to tens if not hundreds of millions of images. Combined with larger networks, neural networks had their chance to shine. This dominance has only continued in the succeeding years, with improved techniques and applications of neural networks to areas outside of image recognition, including translation, speech recognition, and image synthesis.

Why Now?

While the boom in computational power and better techniques led to an increase in interest in neural networks, we have also seen huge strides in *usability*. In particular, deep learning frameworks like TensorFlow, Theano, and Torch allow nonexperts to construct complex neural networks to solve their own machine learning problems. This has turned a task that used to require months or years of handcoding and head-on-table-banging effort (writing efficient GPU kernels is hard!) into something that anyone can do in an afternoon (or really a few days in practice). Increased usability has greatly increased the number of researchers who can work on deep learning problems. Frameworks like Keras with an even higher level of abstraction make it possible for anyone with a working knowledge of Python and some tools to run some interesting experiments, as this book will show.

A second important factor for “why now” is that large datasets have become available for everybody. Yes, Facebook and Google might still have the upper hand with access to billions of pictures, user comments, and what have you, but datasets with millions

of items can be had from a variety of sources. In Chapter 1 we'll look at a variety of options, and throughout the book the example code for each chapter will usually show in the first recipe how to get the needed training data.

At the same time, private companies have started to produce and collect orders of magnitude more data, which has made the whole area of deep learning suddenly commercially very interesting. A model that can tell the difference between a cat and a dog is all very well, but a model that increases sales by 15% by taking all historic sales data into account can be the difference between life and death for a company.

What Do You Need to Know?

These days there is a wide choice of platforms, technologies, and programming languages for deep learning. In this book all the examples are in Python and most of the code relies on the excellent Keras framework. The example code is available on GitHub as a set of Python notebooks, one per chapter. So, having a working knowledge of the following will help:

Python
: Python 3 is preferred, but Python 2.7 should also work. We use a variety of helper libraries that all can easily be installed using pip. The code is generally straightforward so even a relative novice should be able to follow the action.

Keras
: The heavy lifting for machine learning is done almost completely by Keras. Keras is an abstraction over either TensorFlow or Theano, both deep learning frameworks. Keras makes it easy to define neural networks in a very readable way. All code is tested against TensorFlow but should also work with Theano.

NumPy, SciPy, scikit-learn
: These useful and extensive libraries are casually used in many recipes. Most of the time it should be clear what is happening from the context, but a quick read-up on them won't hurt.

Jupyter Notebook
: Notebooks are a very nice way to share code; they allow for a mixture of code, output of code, and comments, all viewable in the browser.

Each chapter has a corresponding notebook that contains working code. The code in the book often leaves out details like imports, so it is a good idea to get the code from Git and launch a local notebook. First check out the code and enter the new directory:

```
git clone https://github.com/DOsinga/deep_learning_cookbook.git
cd deep_learning_cookbook
```

Then set up a virtual environment for the project:

```
python3 -m venv venv3
source venv3/bin/activate
```

And install the dependencies:

```
pip install -r requirements.txt
```

If you have a GPU and want to use that, you'll need to uninstall `tensorflow` and install `tensorflow-gpu` instead, which you can easily do using pip:

```
pip uninstall tensorflow
pip install tensorflow-gpu
```

You'll also need to have a compatible GPU library setup, which can be a bit of a hassle.

Finally, bring up the IPython notebook server:

```
jupyter notebook
```

If everything worked, this should automatically open a web browser with an overview of the notebooks, one for each chapter. Feel free to play with the code; you can use Git to easily undo any changes you've made if you want to go back to the baseline:

```
git checkout <notebook_to_reset>.ipynb
```

The first section of every chapter lists the notebooks relevant for that chapter and the notebooks are numbered according to the chapters, so it should in general be easy to find your way around. In the notebook folder, you'll also find three other directories:

Data
: Contains data needed by the various notebooks—mostly samples of open datasets or things that would be too cumbersome to generate yourself.

Generated
: Used to store intermediate data.

Zoo
: Contains a subdirectory for each chapter that holds saved models for that chapter. If you don't have the time to actually train the models, you can still run the models by loading them from here.

How This Book Is Structured

Chapter 1 provides in-depth information about how neural networks function, where to get data from, and how to preprocess that data to make it easier to consume. Chapter 2 is about getting stuck and what to do about it. Neural nets are notoriously hard to debug and the tips and tricks in this chapter on how to make them behave will come in handy when going through the more project-oriented recipes in the rest of

the book. If you are impatient, you can skip this chapter and go back to it later when you do get stuck.

Chapters 3 through 15 are grouped around media, starting with text processing, followed by image processing, and finally music processing in Chapter 15. Each chapter describes one project split into various recipes. Typically a chapter will start with a data acquisition recipe, followed by a few recipes that build toward the goal of the chapter and a recipe on data visualization.

Chapter 16 is about using models in production. Running experiments in notebooks is great, but ultimately we want to share our results with actual users and get our models run on real servers or mobile devices. This chapter goes through the options.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

Accompanying Code

Each chapter in this book comes with one or more Python notebooks that contain the example code referred to in the chapters themselves. You can read the chapters without running the code, but it is more fun to work with the notebooks as you read. The code can be found at *https://github.com/DOsinga/deep_learning_cookbook*.

To get the example code for the recipes up and running, execute the following commands in a shell:

```
git clone https://github.com/DOsinga/deep_learning_cookbook.git
cd deep_learning_cookbook
python3 -m venv venv3
source venv3/bin/activate
pip install -r requirements.txt
jupyter notebook
```

This book is here to help you get your job done. All code in the accompanying notebooks is licensed under the permissive Apache License 2.0.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Deep Learning Cookbook* by Douwe Osinga (O'Reilly). Copyright 2018 Douwe Osinga, 978-1-491-99584-6."

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/deep-learning-cookbook*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

From academics sharing new ideas by (pre)publishing papers on *https://arxiv.org*, to hackers coding up those ideas on GitHub to public and private institutions publishing datasets for anybody to use, the world of machine learning is full of people and organizations that welcome newcomers and make it as easy to get started as it is. Open data, open source, and open access publishing—this book wouldn't be here without machine learning's culture of sharing.

What is true for the ideas presented in this book is even more true for the code in this book. Writing a machine learning model from scratch is hard, so almost all the models in the notebooks are based on code from somewhere else. This is the best way to get things done—find a model that does something similar to what you want and change it step by step, verifying at each step that things still work.

A special thanks goes out to my friend and coauthor for this book, Russell Power. Apart from helping to write this Preface, Chapter 6, and Chapter 7, he has been instrumental in checking the technical soundness of the book and the accompanying

code. Moreover, he's been an invaluable asset as a sounding board for many ideas, some of which made it into the book.

Then there is my lovely wife, who was the first line of defense when it came to proofreading chapters as they came into being. She has an uncanny ability to spot mistakes in a text that is neither in her native language nor about a subject she's previously been an expert on.

The *requirements.in* file lists the open source packages that are used in this book. A heartfelt thank you goes out to all the contributors to all of these projects. This goes doubly for Keras, since almost all the code is based on that framework and often borrows from its examples.

Example code and ideas from these packages and many blog posts contributed to this book. In particular:

Chapter 2, Getting Unstuck
: This chapter takes ideas from Slav Ivanov's blog post "37 Reasons Why Your Neural Network Is Not Working".

Chapter 3, Calculating Text Similarity Using Word Embeddings
: Thanks to Google for publishing its Word2vec model.

  Radim Řehůřek's Gensim powers this chapter, and some of the code is based on examples from this great project.

Chapter 5, Generating Text in the Style of an Example Text
: This chapter draws heavily on the great blog post "The Unreasonable Effectiveness of Recurrent Neural Networks" by Andrej Karpathy. That blog post rekindled my interest in neural networks.

  The visualization was inspired by Motoki Wu's "Visualizations of Recurrent Neural Networks".

Chapter 6, Question Matching
: This chapter was somewhat inspired by the Quora Question Pairs challenge on Kaggle.

Chapter 8, Sequence-to-Sequence Mapping
: The example code is copied from one of the Keras examples, but applied on a slightly different dataset.

Chapter 11, Detecting Multiple Images
: This chapter is based on Yann Henon's *keras_frcnn*.

Chapter 12, Image Style
: This borrows from "How Convolutional Neural Networks See the World" and of course Google's DeepDream.

Chapter 13, Generating Images with Autoencoders
: Code and ideas are based on Nicholas Normandin's Conditional Variational Autoencoder.

Chapter 14, Generating Icons Using Deep Nets
: Autoencoder training code for Keras is based on Qin Yongliang's DCGAN-Keras.

Chapter 15, Music and Deep Learning
: This was inspired by Heitor Guimarães's *gtzan.keras*.

CHAPTER 1

Tools and Techniques

In this chapter we'll take a look at common tools and techniques for deep learning. It's a good chapter to read through once to get an idea of what's what and to come back to when you need it.

We'll start out with an overview of the different types of neural networks that are covered in this book. Most of the recipes later in the book focus on getting things done and only briefly discuss how deep neural networks are architected.

We'll then discuss where to get data from. Tech giants like Facebook and Google have access to tremendous amounts of data to do their deep learning research, but there's enough data out there for us to do interesting stuff too. The recipes in this book take their data from a wide range of sources.

The next part is about preprocessing of data. This is a very important area that is often overlooked. Even if you have the right network setup and you have great data, you still need to make sure that the data you have is presented in the best way to the network. You want to make it as easy as possible for the network to learn the things it needs to learn and not get distracted by other irrelevant bits in the data.

1.1 Types of Neural Networks

Throughout this chapter and indeed the book we will talk about *networks* and *models*. Network is short for neural network and refers to a stack of connected *layers*. You feed data in on one side and transformed data comes out on the other side. Each layer implements a mathematical operation on the data flowing through it and has a set of variables that can be modified that determine the exact behavior of the layer. *Data* here refers to a *tensor*, a vector with multiple dimensions (typically two or three).

A full discussion of the different types of layers and the math behind their operations is beyond the scope of this book. The simplest type of layer, the *fully connected* layer, takes its input as a matrix, multiplies that matrix with another matrix called the *weights*, and adds a third matrix called the *bias*. Each layer is followed by an *activation* function, a mathematical function that maps the output of one layer to the input of the next layer. For example, a simple activation function called ReLU passes on all positive values, but sets negative values to zero.

Technically the term *network* refers to the architecture, the way in which the various layers are connected to each other, while a *model* is a network plus all the variables that determine the runtime behavior. Training a model modifies those variables to make the predictions fit the expected output better. In practice, though, the two terms are often used interchangeably.

The terms "deep learning" and "neural networks" in reality encompass a wide variety of models. Most of these networks will share some elements (for example, almost all classification networks will use a particular form of *loss function*). While the space of models is diverse, we can group most of them into some broad categories. Some models will use pieces from multiple categories: for example, many image classification networks have a fully connected section "head" to perform the final classification.

Fully Connected Networks

Fully connected networks were the first type of network to be researched, and dominated interest until the late 1980s. In a fully connected network, each output unit is calculated as a weighted sum of all of the inputs. The term "fully connected" arises from this behavior: every output is connected to every input. We can write this as a formula:

$$y_i = \Sigma_j W_{ij} x_j$$

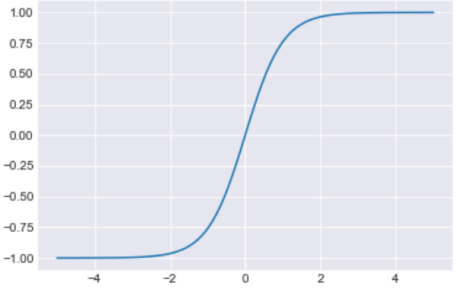
For brevity, most papers represent a fully connected network using matrix notation. In this case we are multiplying a vector of inputs with a weight matrix W to get a vector of outputs:

$$y = Wx$$

As matrix multiplication is a linear operation, a network that only contained matrix multiplies would be limited to learning linear mappings. In order to make our networks more expressive, we follow the matrix multiply with a nonlinear activation function. This can be any differentiable function, but a few are very common. The

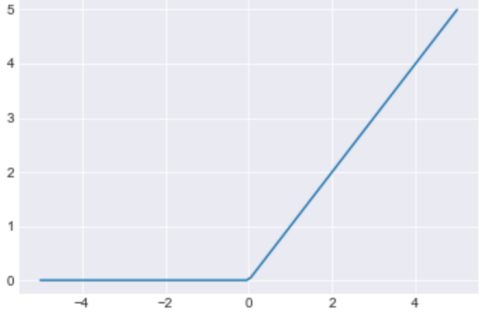
hyperbolic tangent, or *tanh*, function was until recently the dominant type of activation function, and can still be found in some models:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



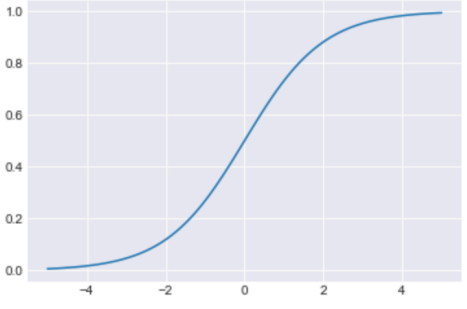
The difficulty with the tanh function is that it is very "flat" when an input is far from zero. This results in a small gradient, which means that a network can take a very long time to change behavior. Recently, other activation functions have become popular. One of the most common is the rectified linear unit, or *ReLU*, activation function:

$$relu(x) = \begin{cases} 0 & \text{if x} < 0 \\ x & \text{if x} \geq 0 \end{cases}$$



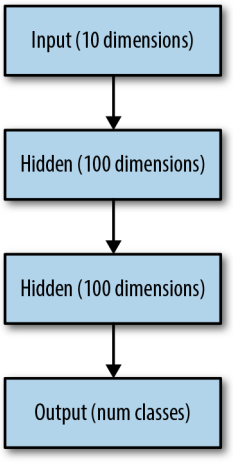
Finally, many networks use a *sigmoid* activation function in the last layer of the network. This function always outputs a value between 0 and 1. This allows the outputs to be treated as probabilities:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$



A matrix multiplication followed by the activation function is referred to as a *layer* of the network. In some networks the complete network can have over 100 layers, though fully connected networks tend to be limited to a handful. If we are solving a classification problem ("What type of cat is in this picture?"), the last layer of the network is called a *classification layer*. It will always have the same number of outputs as we have classes to choose from.

Layers in the middle of the network are called *hidden layers*, and the individual outputs from a hidden layer are sometimes referred to as *hidden units*. The term "hidden" comes from the fact that these units are not directly visible from the outside as inputs or outputs for our model. The number of outputs in these layers depends on the model:

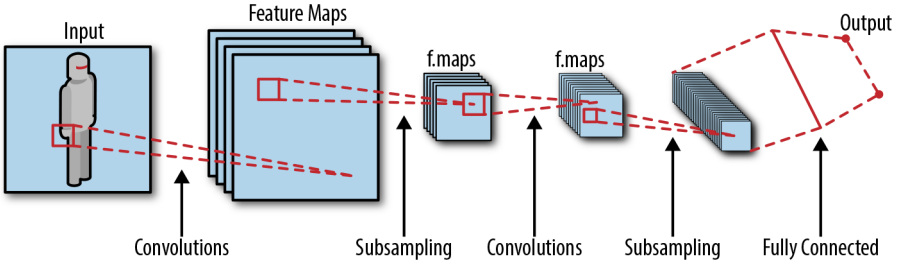


While there are some rules of thumb about how to choose the number and size of hidden layers, there is no general policy for choosing the best setup other than trial and error.

Convolutional Networks

Early research used fully connected networks to try to solve a wide variety of problems. But when our input is images, fully connected networks can be a poor choice. Images are very large: a single 256×256-pixel image (a common resolution for classification) has 256×256×3 inputs (3 colors for each pixel). If this model has a single hidden layer with 1,000 hidden units, then this layer will have almost 200 million parameters (learnable values)! Since image models require quite a few layers to perform well at classification, if we implemented them just using fully connected layers we would end up with billions of parameters.

With so many parameters, it would be almost impossible for us to avoid *overfitting* our model (overfitting is described in detail in the next chapter; it refers to when a network fails to generalize, but just memorizes outcomes). *Convolutional neural networks* (CNNs) provide a way for us to train superhuman image classifiers using far fewer parameters. They do this by mimicking how animals and humans see:



The fundamental operation in a CNN is a *convolution*. Instead of applying a function to an entire input image, a convolution scans across a small window of the image at a time. At each location it applies a *kernel* (typically a matrix multiplication followed by an activation function, just like in a fully connected network). Individual kernels are often referred to as *filters*. The result of applying the kernel to the entire image is a new, possibly smaller image. For example, a common filter shape is (3, 3). If we were to apply 32 of these filters to our input image, we would need 3 * 3 * 3 (input colors) * 32 = 864 parameters—that's a big savings over a fully connected network!

Subsampling

This operation saves on the number of parameters, but now we have a different problem. Each layer in the network can only "look" at a 3×3 layer of the image at a time: if this is the case, how can we possibly recognize objects that take up the entire image?

To handle this, a typical convolution network uses *subsampling* to reduce the size of the image as it passes through the network. Two common mechanisms are used for subsampling:

Strided convolutions
: In a strided convolution, we simply skip one or more pixels while sliding our convolution filter across the image. This results in a smaller size image. For example, if our input image was 256×256, and we skip every other pixel, then our output image will be 128×128 (we are ignoring the issue of padding at the edges of the image for simplicity). This type of strided downsampling is commonly found in generator networks (see "Adversarial Networks and Autoencoders" on page 9).

Pooling
: Instead of skipping over pixels during convolution, many networks use *pooling layers* to shrink their inputs. A pooling layer is actually another form of convolution, but instead of multiplying our input by a matrix, we apply a pooling operator. Typically pooling uses the *max* or *average* operator. Max pooling takes the largest value from each *channel* (color) over the region it is scanning. Average pooling instead averages all of the values over the region. (It can be thought of as a simple type of blurring of the input.)

One way to think about subsampling is as a way to increase the abstraction level of what the network is doing. On the lowest level, our convolutions detect small, local features. There are many features that are not very deep. With each pooling step, we increase the abstraction level; the number of features is reduced, but the depth of each feature increases. This process is continued until we end up with very few features with a high level of abstraction that can be used for prediction.

Prediction

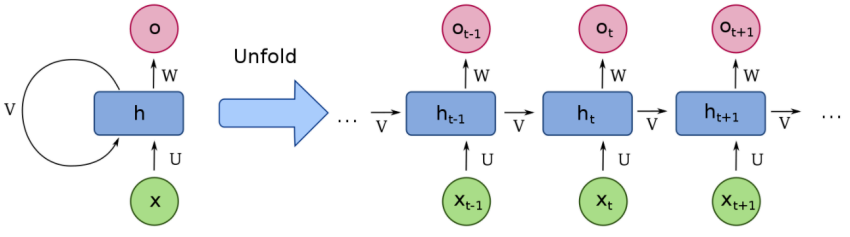
After stacking a number of convolutional and pooling layers together, CNNs use one or two fully connected layers at the head of the network to output a prediction.

Recurrent Networks

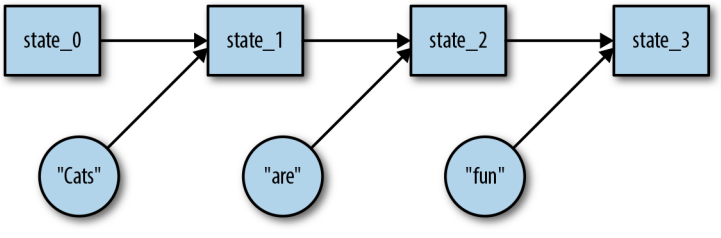
Recurrent neural networks (RNNs) are similar in concept to CNNs but are structurally very different. Recurrent networks are frequently applied when we have a sequential input. These inputs are commonly found when working with text or voice processing. Instead of processing a single example completely (as we might use a CNN for an image), with sequential problems we can process only a portion of the problem at a time. For example, let's consider building a network that writes Shakespearean plays for us. Our input would naturally be the existing plays by Shakespeare:

```
Lear. Attend the lords of France and Burgundy, Gloucester.
Glou. I shall, my liege.
```

What we want the network to learn to do is to predict the next word of the play for us. To do so, it needs to "remember" the text that it has seen so far. Recurrent networks give us a mechanism to do this. They also allow us to build models that naturally work across inputs of varying lengths (sentences or chunks of speech, for example). The most basic form of an RNN looks like this:



Conceptually, you can think of this RNN as a very deep fully connected network that we have "unrolled." In this conceptual model, each layer of the network takes two inputs instead of the one we are used to:



Recall that in our original fully connected network, we had a matrix multiplication operation like:

$$y = Wx$$

The simplest way to add our second input to this operation is to just concatenate it to our hidden state:

$$hidden_i = W\{hidden_{i-1}|x\}$$

where in this case the "|" stands for concatenate. As with our fully connected network, we can apply an activation function to the output of our matrix multiplication to obtain our new state:

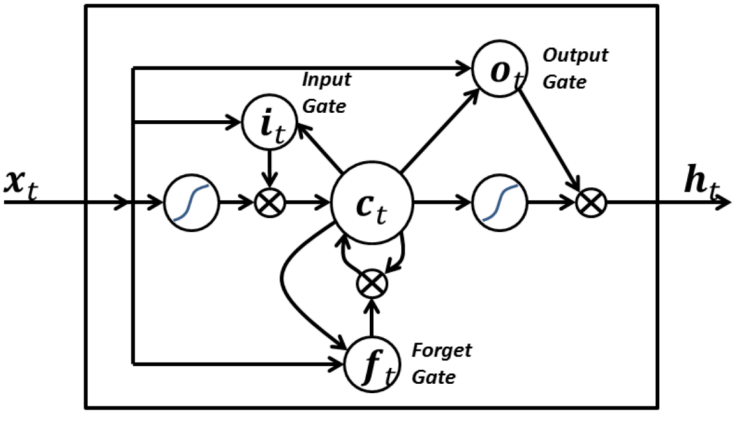
$$hidden_i = f(W\{hidden_{i-1}|x\})$$

With this interpretation of our RNN, we also can easily understand how it can be trained: we simply treat the RNN as we would an unrolled fully connected network and train it normally. This is referred to in literature as *backpropagation through time* (BPTT). If we have very long inputs, it is common to split them into smaller-sized pieces and train each piece independently. While this does not work for every problem, it is generally safe and is a widely used technique.

Vanishing gradients and LSTMs

Our naive RNN unfortunately tends to perform more poorly than we would like for long input sequences. This is because its structure makes it likely to encounter the "vanishing gradients" problem. Vanishing gradients result from the fact that our unrolled network is very deep. Each time we go through an activation function, there's a chance it will result in a small gradient getting passed through (for instance, ReLU activation functions have a zero gradient for any input < 0). Once this happens for a single unit, no more training can be passed down further through the network via that unit. This results in an ever-sparser training signal as we go down. The observed result is extremely slow or nonexistent learning of the network.

To combat this, researchers developed an alternative mechanism for building RNNs. The basic model of unrolling our state over time is kept, but instead of doing a simple matrix multiply followed by the activation function, we have a more complex way of passing our state forward (source: Wikipedia):

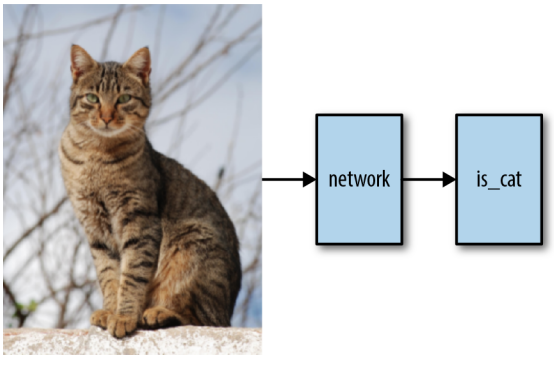


A *long short-term memory network* (LSTM) replaces our single matrix multiplication with four, and introduces the idea of *gates* that are multiplied with a vector. The key behavior that enables an LSTM to learn more effectively than vanilla RNNs is that

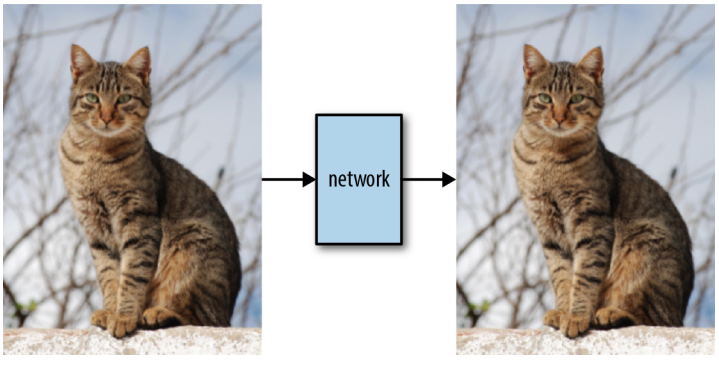
there is always a path from the final prediction to any layer that preserves gradients. The details of how it accomplishes this are beyond the scope of this chapter, but several excellent tutorials exist on the web.

Adversarial Networks and Autoencoders

Adversarial networks and autoencoders do not introduce new structural components, like the networks we've talked about so far. Instead, they use the structure most appropriate to the problem: an adversarial network or autoencoder for images will use convolutions, for example. Where they differ is in how they are trained. Most normal networks are trained to predict an output (is this a cat?) from an input (a picture):



Autoencoders are instead trained to output back the image they are presented:



Why would we want to do this? If the hidden layers in the middle of our network contain a representation of the input image that has (significantly) less information than the original image yet from which the original image can be reconstructed, then this results in a form of compression: we can take any image and represent it just by

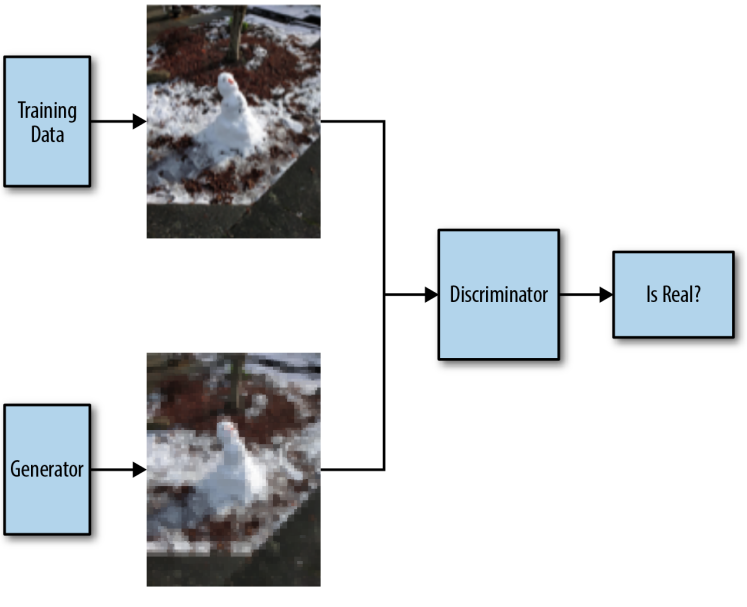
the values from the hidden layer. One way to think about this is that we take the original image and use the network to project it into an abstract space. Each point in that space can then be converted back into an image.

Autoencoders have been successfully applied to small images, but the mechanism for training them does not scale up to larger problems. The space in the middle from which the images are drawn is in practice not "dense" enough, and many of the points don't actually represent coherent images.

We'll seen an example of an autoencoder network in Chapter 13.

Adversarial networks are a more recent model that can actually generate realistic images. They work by splitting the problem into two parts: a generator network and a discriminator network. The generator network takes a small random seed and produces a picture (or text). The discriminator network tries to determine if an input image is "real" or if it came from the generator network.

When we train our adversarial model, we train both of these networks at the same time:



We sample some images from our generator network and feed them through our discriminator network. The generator network is rewarded for producing images that can fool the discriminator. The discriminator network also has to correctly recognize real images (it can't just always say an image is a fake). By making the networks compete against each other, this procedure can result in a generator network that pro-

duces high-quality natural images. Chapter 14 shows how we can use generative adversarial networks to generate icons.

Conclusion

There are a great many ways to architect a network, and the choice obviously is mostly driven by the purpose of the network. Designing a new type of network is firmly in the research realm, and even reimplementing a type of network described in a paper is hard. In practice the easiest thing to do is to find an example that does something in the direction of what you want and change it step by step until it really does what you want.

1.2 Acquiring Data

One of the key reasons why deep learning has taken off in recent years is the dramatic increase in the availability of data. Twenty years ago networks were trained with thousands of images; these days companies like Facebook and Google work with billions of images.

Having access to all the information from their users no doubt gives these and other internet giants a natural advantage in the deep learning field. However, there are many data sources easily accessible on the internet that, with a little massaging, can fit many training purposes. In this section, we'll discuss the most important ones. For each, we'll look into how to acquire the data, what popular libraries are available to help with parsing, and what typical use cases are. I'll also refer you to any recipes that use this data source.

Wikipedia

Not only does the English Wikipedia comprise more than 5 million articles, but Wikipedia is also available in hundreds of languages, albeit with widely different levels of depth and quality. The basic wiki idea only supports links as a way to encode structure, but over time Wikipedia has gone beyond this.

Category pages link to pages that share a property or a subject, and since Wikipedia pages link back to their categories, we can effectively use them as tags. Categories can be very simple, like "Cats," but sometimes encode information in their names that effectively assigns (key, value) pairs to a page, like "Mammals described in 1758." The category hierarchy, like much on Wikipedia, is fairly ad hoc, though. Moreover, recursive categories can only be traced by walking up the tree.

Templates were originally designed as segments of wiki markup that are meant to be copied automatically ("transcluded") into a page. You add them by putting the template's name in `{{double braces}}`. This made it possible to keep the layout of differ-

ent pages in sync—for example, all city pages have an info box with properties like population, location, and flag that are rendered consistently across pages.

These templates have parameters (like the population) and can be seen as a way to embed structured data into a Wikipedia page. In Chapter 4 we use this to extract a set of movies that we then use to train a movie recommender system.

Wikidata

Wikidata is Wikipedia's structured data cousin. It is lesser known and also less complete, but even more ambitious. It is intended to provide a common source of data that can be used by anyone under a public domain license. As such, it makes for an excellent source of freely available data.

All Wikidata is stored as triplets of the form (subject, predicate, object). All subjects and predicates have their own entries in Wikidata that list all predicates that exist for them. Objects can be Wikidata entries or literals such as strings, numbers, or dates. This structure takes inspiration from early ideas around the semantic web.

Wikidata has its own query language that looks like SQL with some interesting extensions. For example:

```
SELECT ?item ?itemLabel ?pic
WHERE
{
        ?item wdt:P31 wd:Q146 .
        OPTIONAL {
                ?item wdt:P18 ?pic
        }
        SERVICE wikibase:label {
          bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en"
        }
}
```

will select a series of cats and their pictures. Anything that starts with a question mark is a variable. `wdt:P31`, or property 31, means "is an instance of," and `wd:Q146` is the class of house cats. So the fourth line stores in `item` anything that is an instance of cats. The `OPTIONAL { .. }` clause then tries to look up pictures for the item and the last magic line tries to find a label for the item using the auto-language feature or, failing that, English.

In Chapter 10 we use a combination of Wikidata and Wikipedia to acquire canonical images for categories to use as a basis for a reverse image search engine.

OpenStreetMap

OpenStreetMap is like Wikipedia, but for maps. Whereas with Wikipedia the idea is that if everybody in the world put down everything they knew in a wiki, we'd have the

best encyclopedia possible, OpenStreetMap (OSM) is based on the idea that if everybody put the roads they knew in a wiki, we'd have the best mapping system possible. Remarkably, both of these ideas have worked out quite well.

While the coverage of OSM is rather uneven, ranging from areas that are barely covered to places where it rivals or exceeds what can be found on Google Maps, the sheer amount of data and the fact that it is all freely available makes it a great resource for all types of projects that are of a geographical nature.

OSM is downloadable for free in a binary format or a huge XML file. The whole world is tens of gigabytes, but there are a number of locations on the internet where we can find OSM dumps per country or region if we want to start smaller.

The binary and XML formats both have the same structure: a map is made out of a series of *nodes* that each have a *latitude* and a *longitude*, followed by a series of *ways* that combine previously defined nodes into larger structures. Finally, there are *relations* that combine anything that was seen before (nodes, ways, or relations) into superstructures.

Nodes are used to represents points on the maps, including individual features, as well as to define the shapes of ways. Ways are used for simple shapes, like buildings and road segments. Finally, relations are used for anything that contains more than one shape or very big things like coastlines or borders.

Later in the book, we'll look at a model that takes in satellite images and rendered maps and tries to learn to recognize roads automatically. The actual data used for those recipes is not specifically from OSM, but it is the sort of thing that OSM is used for in deep learning. The "Images to OSM" project, for example, shows how to train a network to learn to extract shapes of sports fields from satellite images to improve OSM itself.

Twitter

As a social network Twitter might have trouble competing with the much bigger Facebook, but as a source for text to train deep learning models, it is much better. Twitter's API is nicely rounded and allows for all kinds of apps. To the budding machine learning hacker though, the streaming API is possibly the most interesting.

The so-called Firehose API offered by Twitter streams all tweets directly to a client. As one can imagine, this is a rather large amount of data. On top of that, Twitter charges serious money for this. It is less known that the free Twitter API offers a sampled version of the Firehose API. This API returns only 1% of all tweets, but that is plenty for many text processing applications.

Tweets are limited in size and come with a set of interesting metainformation like the author, a timestamp, sometimes a location, and of course tags, images, and URLs. In

Chapter 7 we look at using this API to build a classifier to predict emojis based on a bit of text. We tap into the streaming API and keep only the tweets that contain exactly one emoji. It takes a few hours to get a decent training set, but if you have access to a computer with a stable internet connection, letting it run for a few days shouldn't be an issue.

Twitter is a popular source of data for experiments in sentiment analysis, which arguably predicting emojis is a variation of, but models aimed at language detection, location disambiguation, and named entity recognition have all been trained successfully on Twitter data too.

Project Gutenberg

Long before Google Books—in fact, long before Google and even the World Wide Web, back in 1971, Project Gutenberg launched with the aim to digitize all books. It contains the full text of over 50,000 works, not just novels, poetry, short stories, and drama, but also cookbooks, reference works, and issues of periodicals. Most of the works are in the public domain and they can all be freely downloaded from the website.

This is a massive amount of text in a convenient format, and if you don't mind that most of the texts are a little older (since they are no longer in copyright) it's a very good source of data for experiments in text processing. In Chapter 5 we use Project Gutenberg to get a copy of Shakespeare's collected works as a basis to generate more Shakespeare-like texts. All it takes is this one-liner if you have the Python library available:

```
shakespeare = strip_headers(load_etext(100))
```

The material available via Project Gutenberg is mostly in English, although a small amount of works are available in other languages. The project started out as pure ASCII but has since evolved to support a number of character encodings, so if you download a non-English text, you need to make sure that you have the right encoding—not everything in the world is UTF-8 yet. In Chapter 8 we extract all dialogue from a set of books retrieved from Project Gutenberg and then train a chatbot to mimic those conversations.

Flickr

Flickr is a photo sharing site that has been in operation since 2004. It originally started as a side project for a massively multiplayer online game called *Game Neverending*. When the game failed to become a business on its own, the company's founders realized that the photo sharing part of the company was taking off and so they executed what is called a *pivot*, completely changing the main focus of the company. Flickr was sold to Yahoo a year later.

Among the many, many photo sharing sites out there, Flickr stands out as a useful source of images for deep learning experiments for a few reasons.

One is that Flickr has been at this for a long time and has collected a set of billions of images. This might pale in comparison to the number of images that people upload to Facebook in a single month, but since users upload photos to Flickr that they are proud of for public consumption, Flickr images are on average of higher quality and of more general interest.

A second reason is licensing. Users on Flickr pick a license for their photos, and many pick some form of Creative Commons licensing that allows for reuse of some kind without asking permission. While you typically don't need this if you run a bunch of photos through your latest nifty algorithm and are only interested in the end results, it is quite essential if your project ultimately needs to republish the original or modified images. Flickr makes this possible.

The last and possibly most important advantage that Flickr has over most of its competitors is the API. Just like Twitter's, it is a well-thought-out, REST-style API that makes it easy to do anything you can do with the site in an automatic fashion. And just like with Twitter there are good Python bindings for the API, which makes it even easier to start experimenting. All you need is the right library and a Flickr API key.

The main features of the API relevant for this book are searching for images and fetching of images. The search is quite versatile and mimics most of the search options of the main website, although some advanced filters are unfortunately missing. Fetching images can be done for a large variety of sizes. It is often useful to get started more quickly with smaller versions of the images first and scale up later.

In Chapter 9 we use the Flickr API to fetch two sets of images, one with dogs and one with cats, and train a classifier to learn the difference between the two.

The Internet Archive

The Internet Archive has a stated mission of providing "universal access to all knowledge." The project is probably most famous for its Wayback Machine, a web interface that lets users look at web pages over time. It contains over 300 billion captures dating all the way back to 2001 in what the project calls a three-dimensional web index.

But the Internet Archive is far bigger than the Wayback Machine and comprises a ragtag assortment of documents, media, and datasets covering everything from books out of copyright to NASA images to cover art for CDs to audio and video material. These are all really worth browsing through and often inspire new projects on the spot.

One interesting example is a set of all Reddit comments up to 2015 with over 50 million entries. This started out as a project of a Reddit user who just patiently used the Reddit API to download all of them and then announced that on Reddit. When the question came up of where to host it, the Internet Archive turned out to be a good option (though the same data can be found on Google's BigQuery for even more immediate analysis).

An example we use in this book is the set of Stack Exchange questions. Stack Exchange has always been licensed under a Creative Commons license, so nothing would stop us from downloading these sets ourselves, but getting them from the Internet Archive is so much easier. In this book we use this dataset to train a model to match questions with answers (see Chapter 6).

Crawling

If you need anything specific for your project, chances are that the data you are after is not accessible through a public API. And even if there is a public API, it might be rate limited to the point of being useless. Historic results for your favorite sports are hard to come by. Your local newspaper might have an online archive, but probably no API or data dump. Instagram has a nice API, but the recent changes to the terms of service make it hard to use it to acquire a large set of training data.

In these cases, you can always resort to scraping, or, if you want to sound more respectable, crawling. In the simplest scenario you just want to get a copy of a website on your local system and you have no prior knowledge about the structure of that website or the format of the URLs. In that case you just start with the root of the website, fetch the web content of it, extract all links from that web content, and do the same for each of those links until you find no more new links. This is how Google does it too, be it at a larger scale. Scrapy is a useful framework for this sort of thing.

Sometimes there is an obvious hierarchy, like a travel website with pages for countries, regions in those countries, cities in those regions, and finally attractions in those cities. In that case it might be more useful to write a more targeted scraper that successively works its way through the various layers of hierarchy until it has all the attractions.

Other times there is an internal API to take advantage of. Many content-oriented websites will load the overall layout and then use a JSON call back to the web server to get the actual data and insert this on the fly into the template. This makes it easy to support infinite scrolling and search. The JSON returned from the server is often easy to make sense of, as are the parameters passed to the server. The Chrome extension Request Maker shows all requests that a page makes and is a good way to see if anything useful goes over the line.

Then there are the websites that don't want to be crawled. Google might have built an empire on scraping the world, but many of its services very cleverly detect signs of scraping and will block you and possibly anybody making requests from your IP address until you do a captcha. You can play with rate limiting and user agents, but at some point you might have to resort to scraping using a browser.

WebDriver, a framework developed for testing websites by instrumenting a browser, can be very helpful in these situations. The fetching of the pages is done with your choice of browser, so to the web server everything seems as real as it can get. You can then "click" on links using your control script to go to the next page and inspect the results. Consider sprinkling the code with delays to make it seem like a human is exploring the site and you should be good to go.

The code in Chapter 10 uses crawling techniques to fetch images from Wikipedia. There is a URL scheme to go from a Wikipedia ID to the corresponding image, but it doesn't always pan out. In that case we fetch the page that contains the image and follow the link graph until we get to the actual image.

Other Options

There are many ways to get data. The ProgrammableWeb lists more than 18,000 public APIs (though some of those are in a state of disrepair). Here are three that are worth highlighting:

Common Crawl
: Crawling one site is doable if the site is not very big. But what if you want to crawl all of the major pages of the internet? The Common Crawl runs a monthly crawl fetching around 2 billion web pages each time in an easy-to-process format. AWS has this as a public dataset, so if you happen to run on that platform that's an easy way to run jobs on the web at large.

Facebook
: Over the years the Facebook API has shifted subtly from being a really useful resource to build applications on top of Facebook's data to a resource to build applications that make Facebook's data better. While this is understandable from Facebook's perspective, as a data prospector one often wonders about the data it could make public. Still, the Facebook API is a useful resource—especially the Places API in situations where OSM is just too unevenly edited.

US government
: The US government on all levels publishes a huge amount of data, and all of it is freely accessible. For example, the census data has detailed information about the US population, while Data.gov has a portal with many different datasets all over the spectrum. On top of that, individual states and cities have their own resources worth looking at.

1.3 Preprocessing Data

Deep neural networks are remarkably good at finding patterns in data that can help in learning to predict the labels for the data. This also means that we have to be careful with the data we give them; any pattern in the data that is not relevant for our problem can make the network learn the wrong thing. By preprocessing data the right way we can make sure that we make things as easy as possible for our networks.

Getting a Balanced Training Set

An apocryphal story relates how the US Army once trained a neural network to discriminate between camouflaged tanks and plain forest—a useful skill when automatically analyzing satellite data. At first sight they did everything right. On one day they flew a plane over a forest with camouflaged tanks in it and took pictures, and on another day they did the same when there were no tanks, making sure to photograph scenes that were similar but not quite the same. They split the data up into training and test sets and let the network train.

The network trained well and started to get good results. However, when the researchers sent it out to be tested in the wild, people thought it was a joke. The predictions seemed utterly random. After some digging, it turned out that the input data had a problem. All the pictures containing tanks had been taken on a sunny day, while the pictures with just forest happened to have been taken on a cloudy day. So while the researchers thought their network had learned to discriminate between tanks and nontanks, they really had trained a network to observe the weather.

Preprocessing data is all about making sure the network picks up on the signals we want it to pick up on and is not distracted by things that don't matter. The first step here is to make sure that we actually have the right input data. Ideally the data should resemble as closely as possible the real-world situation.

Making sure that the signal in the data is the signal we are trying to learn seems obvious, but it is easy to get this wrong. Getting data is hard, and every source has its own peculiarities.

There are a few things we can do when we find our input data is tainted. The best thing is, of course, to rebalance the data. So in the tanks versus forest example, we would try to get pictures for both scenarios in all types of weather. (When you think about it, even if all the original pictures had been taken in sunny weather, the training set would still have been suboptimal—a balanced set would contain weather conditions of all types.)

A second option is to just throw out some data to make the set more balanced. Maybe there were some pictures of tanks taken on cloudy days after all, but not enough—so we could throw out some of the sunny pictures. This obviously cuts down the size of

the training set, however, and might not be an option. (Data augmentation, discussed in "Preprocessing of Images" on page 22, could help.)

A third option is to try to fix the input data, say by using a photo filter that makes the weather conditions appear more similar. This is tricky though, and can easily lead to other or even more artifacts that the network might detect.

Creating Data Batches

Neural networks consume data in batches (sets of input/output pairs). It is important to make sure that these batches are properly randomized. Imagine we have a set of pictures, the first half all depicting cats and the second half dogs. Without shuffling, it would be impossible for the network to learn anything from this dataset: almost all batches would either contain only cats or only dogs. If we use Keras and if we have our data entirely in memory, this is easily accomplished using the `fit` method since it will do the shuffling for us:

```
char_cnn_model.fit(training_data, training_labels, epochs=20, batch_size=128)
```

This will randomly create batches with a size of 128 from the `training_data` and `training_labels` sets. Keras takes care of the proper randomizing. As long as we have our data in memory, this is usually the way to go.

In some circumstances we might want to call `fit` with one batch at a time, in which case we do need to make sure things are properly shuffled. `numpy.random.shuffle` will do just fine, though we have to take care to shuffle the data and the labels in unison.

We don't always have all the data in memory, though. Sometimes the data would be too big or needs to be processed on the fly and isn't available in the ideal format. In those situations we use `fit_generator`:

```
char_cnn_model.fit_generator(
    data_generator(train_tweets, batch_size=BATCH_SIZE),
    epochs=20
)
```

Here, `data_generator` is a generator that yields batches of data. The generator has to make sure that the data is properly randomized. If the data is read from a file, shuffling is not really an option. If the file comes from an SSD and the records are all the same size, we can shuffle by seeking randomly inside of the file. If this is not the case and the file has some sort of sorting, we can increase randomness by having multiple file handles in the same file, all at different locations.

When setting up a generator that produces batches on the fly, we also need to pay attention to keep things properly randomized. For example, in Chapter 4 we build a

movie recommender system by training on Wikipedia articles, using as the unit of training links from the movie page to some other page. The easiest way to generate these (FromPage, ToPage) pairs would be to randomly pick a FromPage and then randomly pick a ToPage from all the links found on FromPage.

This works, of course, but it will select links from pages with fewer links on them more often than it should. A FromPage with one link on it has the same chance of being picked in the first step as a page with a hundred links. In the second step, though, that one link is certain to be picked, while any of the links from the page with a hundred links has only a small chance of selection.

Training, Testing, and Validation Data

After we've set up our clean, normalized data and before the actual training phase, we need to split the data up in a training set, a test set, and possibly a validation set. As with many things, the reason we do this has to do with overfitting. Networks will almost always memorize a little bit of the training data rather than learn generalizations. By separating a small amount of the data into a test set that we don't use for training, we can measure to what extent this is happening; after each epoch we measure accuracy over both the training and the test set, and as long as the two numbers don't diverge too much, we're fine.

If we have our data in memory we can use `train_test_split` from `sklearn` to neatly split our data into training and test sets:

```
data_train, data_test, label_train, label_test = train_test_split(
    data, labels, test_size=0.33, random_state=42)
```

This will create a test set containing 33% of the data. The `random_state` variable is used for the random seed, which guarantees that if we run the same program twice, we get the same results.

When feeding our network using a generator, we need to do the splitting ourselves. One general though not very efficient approach would be to use something like:

```
def train_or_test(gen, train=True):
    for i, x in enumerate(gen):
        if (i % 4 == 0) != train:
            yield x
```

When `train` is `False` this yields every fourth element coming from the generator `gen`. When it is `True` it yields the rest.

Sometimes a third set is split off from the training data, called the *validation set*. There is some confusion in the naming here; when there are only two sets the test set is sometimes also called the validation set (or holdout set). In a scenario where we have training, validation, and test sets, the validation set is used to measure perfor-

mance while tuning the model. The test set is meant to be used only when all tuning is done and no more changes are going to be made to the code.

The reason to keep this third set is to stop us from manually overfitting. A complex neural network can have a very large number of tuning options or hyperparameters. Finding the right values for these hyperparameters is an optimization problem that can also suffer from overfitting. We keep adjusting those parameters until the performance on the validation set no longer increases. By having a test set that was not used during tuning, we can make sure that we didn't inadvertently optimize our hyper parameters for the validation set.

Preprocessing of Text

A lot of neural networking problems involve text processing. Preprocessing the input texts in these situations involves mapping the input text to a vector or matrix that we can feed into a network.

Typically, the first step is to break up the text into units. There are two common ways to do this: on a character or a word basis.

Breaking up a text into a stream of single characters is straightforward and gives us a predictable number of different tokens. If all our text is in one phoneme-based script, the number of different tokens is quite restricted.

Breaking up a text into words is a more complicated tokenizing strategy, especially in scripts that don't indicate the beginning and ending of words. Moreover, there is no obvious upper limit to the number of different tokens that we'll end up with. A number of text processing toolkits have a "tokenize" function that usually also allows for the removal of accents and optionally converts all tokens to lowercase.

A process called *stemming*, where we convert each word to its root form (by dropping any grammar-related modifications), can help, especially for languages that are more grammar-heavy than English. In Chapter 8 we'll encounter a subword tokenizing strategy that breaks up complicated words into subtokens thereby guaranteeing a specific upper limit on the number of different tokens.

Once we have our text split up into tokens, we need to vectorize it. The simplest way of doing this is called *one-hot encoding*. Here, we assign to each unique token an integer i from 0 to the number of tokens and then represent each token as a vector containing only 0s, except for the ith entry, which contains a 1. In Python code this would be:

```
idx_to_token = list(set(tokens))
token_to_idx = {token: idx for idx, token in enumerate(idx_to_token)}
one_hot = lambda token: [1 if i == token_to_idx[token] else 0
                         for i in range(len(idx_to_token))]
encoded = np.asarray([one_hot(token) for token in tokens])
```

This should leave us with a large two-dimensional array ready for consumption.

One-hot encoding works when we process text at a character level. It also works for word-level processing, though for texts with large vocabularies it can get unwieldy. There are two popular encoding strategies that work around this.

The first one is to treat a document as a "bag of words." Here, we don't care about the order of the words, just whether a certain word is present. We can then represent a document as a vector with an entry for each unique token. In the simplest scheme we just put a 1 if the word is present in that document and a 0 if not.

Since the top 100 most frequently occurring words in English make up about half of all texts, they are not very useful for text classifying tasks; almost all documents will contain them, so having those in our vectors doesn't really help much. A common strategy is to just drop them from our bag of words so the network can focus on the words that do make a difference.

Term frequency–inverse document frequency, or tf–idf, is a more sophisticated version of this. Instead of storing a 1 if a token is present in a document, we store the relative frequency of the term in the document compared to how often the term occurs throughout the entire corpus of documents. The intuition here is that it is more meaningful for a less common token to appear in a document than a token that appears all the time. Scikit-learn comes with methods to calculate this automatically.

A second way to handle word-level encoding is by way of embeddings. Chapter 3 is all about embeddings and offers a good way to understand how they work. With embeddings we associate a vector of a certain size—typically with a length of 50 to 300—with each token. When we feed in a document represented as a sequence of token IDs, an embedding layer will automatically look up the corresponding embedding vectors and output a two-dimensional array.

The embedding layer will learn the right weights for each term, just like any layer in a neural network. This often takes a lot of learning, both in terms of processing and the required amount of data. A nice aspect of embeddings, though, is that there are pretrained sets available for download and we can seed our embedding layer with these. Chapter 7 has a good example of this approach.

Preprocessing of Images

Deep neural networks have turned out to be very effective when it comes to working with images, for anything from detecting cats in videos to applying the style of different artists to selfies. As with text, though, it is essential to properly preprocess the input images.

The first step is normalization. Many networks only operate on a specific size, so the first step is to resize/crop the images to that target size. Both center cropping and

direct resizing are often used, though sometimes a combination works better in order to preserve more of the image while keeping resize distortion somewhat in check.

To normalize the colors, for each pixel we usually subtract the mean value and divide by the standard deviation. This makes sure that all values on average center around 0 and that the nearly 70% of all values are within the comfortable [–1, 1] range. A new development here is the use of *batch normalization*; rather than normalizing all data beforehand, this subtracts the mean of the batch and divides by the standard deviation. This leads to better results and can just be made part of the network.

Data augmentation is a strategy to increase the amount of training data by adding variations of our training images. If we add to our training data versions of our images flipped horizontally, in a way we double our training data—a mirrored cat is still a cat. Looking at this in another way, what we are doing is telling our network that flips can be ignored. If all our cat pictures have the cat looking in one direction, our network might learn that that is part of catness; adding flips undoes that.

Keras has a handy `ImageDataGenerator` class that you can configure to produce all kinds of image variations, including rotations, translations, color adjustments, and magnification. You can then use that as a data generator for the `fit_generator` method on your model:

```
datagen = ImageDataGenerator(
    rotation_range=20,
    horizontal_flip=True)

model.fit_generator(datagen.flow(x_train, y_train, batch_size=32),
                    steps_per_epoch=len(x_train) / 32, epochs=epochs)
```

Conclusion

Preprocessing of data is an important step before training a deep learning model. A common thread in all of this is that we want it to be as easy as possible for networks to learn the right thing and not be confused by irrelevant features of the input. Getting a balanced training set, creating randomized training batches, and the various ways to normalize the data are all a big part of this.

CHAPTER 2

Getting Unstuck

Deep learning models are often treated as a black box; we pour data in at one end and an answer comes out at the other without us having to care much about how our network learns. While it is true that deep neural nets can be remarkably good at distilling a signal out of complex input data, the flip side of treating these networks as black boxes is that it isn't always clear what to do when things get stuck.

A common theme among the techniques we discuss here is that we want the network to *generalize* rather than to *memorize*. It is worth pondering the question of why neural networks generalize at all. Some of the models described in this book and used in production contain millions of parameters that would allow the network to memorize inputs with very many examples. If everything goes well, though, it doesn't do this, but rather develops generalized rules about its input.

If things don't go well, you can try the techniques described in this chapter. We'll start out by looking at how we know that we're stuck. We'll then look at various ways in which we can preprocess our input data to make it easier for the network to work with.

2.1 Determining That You Are Stuck

Problem

How do you know when your network is stuck?

Solution

Look at various metrics while the network trains.

The most common signs that things are not well with a neural network are that the network is not learning anything or that it is learning the wrong thing. When we set up the network, we specify the *loss function*. This determines what the network is trying to optimize for. During training the loss is continuously printed. If this value doesn't go down after a few iterations, we're in trouble. The network is not learning anything measured by its own notion of progress.

A second metric that comes in handy is *accuracy*. This shows the percentage of the inputs for which the network is predicting the right answer. As the loss goes down, the accuracy should go up. If accuracy does not go up even though the loss is decreasing, then our network is learning something, but not the thing we were hoping for. Accuracy can take a while, though, to pick up. A complex visual network will take a long time before it gets any labels right while still learning, so take this into account before giving up prematurely.

The third thing to look for, and this is probably the most common way to get stuck, is *overfitting*. With overfitting we see our loss decrease and our accuracy increase, but the accuracy we see over our testing set doesn't keep up. Assuming we have a testing set and have added this to the metrics to track, we can see this each time an epoch finishes. Typically the testing accuracy at first increases with the accuracy of the training set, but then a gap appears, and oftentimes the testing accuracy even starts to drop while the training accuracy keeps increasing.

What's happening here is that our network is learning a direct mapping between the inputs and the expected outputs, rather than learning generalizations. As long as it sees an input it has seen before, everything looks cool. But confronted with a sample from the test set, which it hasn't seen during training, it starts to fail.

Discussion

Paying attention to the metrics that are displayed during training is a good way to keep track of the progress of the learning process. The three metrics we discussed here are the most important, but frameworks like Keras offer many more and the option to build them yourselves.

2.2 Solving Runtime Errors

Problem

What should you do when your network complains about incompatible shapes?

Solution

Look at the network structure and experiment with different numbers.

Keras is a great abstraction over hairier frameworks like TensorFlow or Theano, but like any abstraction, this comes at a cost. When all is well our clearly defined model runs happily on top of TensorFlow or Theano. When it doesn't, though, we get an error from the depths of the underlying framework. These errors are hard to make sense of without understanding the intricacies of those frameworks—which is what we wanted to avoid in the first place by using Keras.

There are two things that can help and don't require us to go on a deep dive. The first is to print the structure of our network. Let's say we have a simple model that takes in five variables and classifies into eight categories:

```
data_in = Input(name='input', shape=(5,))
fc = Dense(12, activation='relu')(data_in)
data_out = Dense(8, activation='sigmoid')(fc)
model = Model(inputs=[data_in], outputs=[data_out])
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

We can now inspect the model with:

```
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
input (InputLayer)           (None, 5)                 0
_________________________________________________________________
dense_5 (Dense)              (None, 12)                72
_________________________________________________________________
dense_6 (Dense)              (None, 8)                 104
=================================================================
Total params: 176
Trainable params: 176
Non-trainable params: 0
```

Now if we get a runtime error about an incompatible shape, of the feared form:

```
InvalidArgumentError: Incompatible shapes: X vs. Y
```

we know something internal must be wrong that isn't easy to track down using the stack trace. There are some other things to try, though.

First, take a look at whether any of the shapes are either `X` or `Y`. If so, that's probably where the problem is. Knowing that is half the work—which of course still leaves the other half. The other thing to pay attention to is the names of the layers. Often they come back in the error message, sometimes in a mangled form. Keras auto-assigns names to anonymous layers, so looking at the summary is useful in this respect too. If needed we can assign our own names, like with the input layer in the example shown here.

If we can't find the shape or the name that the runtime error is mentioning, we can try something else before having to dive in (or post on StackOverflow): use different numbers.

Neural networks contain loads of hyperparameters, like the sizes of the various layers. These are usually picked because they seem reasonable, given other networks that do similar things. But their actual value is somewhat arbitrary. In our example, does the hidden layer really need 12 units? Would 11 do a lot worse, and would 13 lead to overfitting?

Probably not. We tend to pick numbers that feel nice, often powers of two. So if you are stuck on a runtime error, change these numbers and see what it does to the error message. If the error message remains the same, the variable that you changed has nothing to do with it. Once it starts changing, though, you know you've reached something related.

This can be subtle. For example, some networks require that all batches have the same size. If your data isn't divisible by the batch size, your last batch will be too small and you'll get an error like:

```
Incompatible shapes: [X,784] vs. [Y,784]
```

Here `X` would be the batch size and `Y` the size of your last incomplete batch. You might recognize `X` as your batch size, but `Y` is hard to place. But if you change the batch size, `Y` also changes, which provides a hint as to where to look.

Discussion

Understanding errors reported by the framework that is abstracted away by Keras is fundamentally tricky. The abstraction breaks, and we suddenly see the internals of the machinery. The techniques from this recipe allow you to postpone looking into those details by spotting shapes and names in the errors and, failing that, experimenting with numbers and seeing what changes.

2.3 Checking Intermediate Results

Problem

Your network quickly gets to a promising level of accuracy but refuses to go beyond that.

Solution

Check whether it hasn't gotten stuck at an obvious local maximum.

One situation in which this can happen is when one label is far more common than any others, and your network quickly learns that always predicting this outcome gives decent results. It is not hard to verify that this is happening; just feed the network a sample of inputs and look at the outputs. If all outputs are the same, you are stuck this way.

Some of the following recipes in this chapter offer suggestions for how to fix this. Alternatively, you could play with the distribution of the data. If 95% of your examples are dogs and only 5% cats, the network might not see enough cats. By artificially changing the distribution to, say, 65%/35%, you make it a little easier for the network.

This is, of course, not without its own risks. The network might now have more of a chance to learn about cats, but it will also learn the wrong base distribution, or prior. This means that in case of doubt the network will now be more likely to pick "cat" as the answer, even though, all things being equal, "dog" is more likely.

Discussion

Looking at the distribution of output labels of a network for a small sample of inputs is an easy way to get an idea of what is actually being done, yet it is often overlooked. Playing with the distribution is a way to try to get the network unstuck if it focuses on just the top answer, but you should probably consider other techniques too.

There are other things to look out for in the output when a network isn't converging quickly; the occurrence of NaNs is an indication of exploding gradients, and if the outputs of your network seem to be clipped and can't seem to reach the right values, you might have an incorrect activation function on your final layer.

2.4 Picking the Right Activation Function (for Your Final Layer)

Problem

How do you pick the right activation function for your final layer when things are off?

Solution

Make sure that the activation function corresponds with the intention of the network.

A good way to get started with deep learning is to find an example online somewhere and modify it step by step until it does what you want it to do. However, if the intention of the example network is different from what your goal, you might have to change the activation function of the final layer. Let's take a look at some common choices.

The softmax activation function makes sure that the sum of the output vector is exactly 1. This is an appropriate activation function for networks that output exactly one label for a given input (for example, an image classifier). The output vector will then represent the probability distribution—if the entry for "cat" in the output vector is .65, then the network thinks that it sees a cat with 65% certainty. Softmax only works when there is one answer. When multiple answers are possible, give the sigmoid activation a try.

A linear activation function is appropriate for regression problems when we need to predict a numeric value given an input. An example would be to predict a movie rating given a series of movie reviews. The linear activation function will take the values of the previous layer and multiply them with a set of weights such that it best fits the expected output. Just as it is a good idea to normalize the input data into a [–1, 1] range or thereabouts, it often helps to do the same for outputs. So, if our movie ratings are between 0 and 5, we'd subtract 2.5 and divide by the same when creating the training data.

If the network outputs an image, make sure that the activation function you use is in line with how you normalize the pixels. The standard normalization of deducting the mean pixel value and dividing by the standard deviation results in values that center around 0, so it won't work with sigmoid, and since 30% of the values will fall outside the range [–1, 1] tanh won't be a good fit either. You can still use these, but you'd have to change the normalization applied to your output.

Depending on what you know about the output distribution, it might be useful to do something even more fancy. Movie ratings, for example, tend to center around 3.7 or so, so using that as the center could well yield better results. When the actual distribution is skewed such that values around the average are much more likely than outliers, using a tanh activation function can be appropriate. This squashes any value into a [–1, 1] range. By mapping the expected outputs to the same range, keeping the expected distribution in mind, we can mimic any shape of our output data.

Discussion

Picking the right output activation function is crucial, but in most cases not difficult. If your output represents a probability distribution with one possible outcome, softmax is for you; otherwise, you need to experiment.

You also need to make sure that the loss function works with the activation function of the final layer. The loss function steers the training of the network by calculating how "wrong" a prediction is, given an expected value. We saw that a softmax activation function is the right choice when a network does multilabel predictions; in that case you probably want to go with a categorical loss function like Keras's `categorical_crossentropy`.

2.5 Regularization and Dropout

Problem

Once you have detected your network is overfitting, what can you do about it?

Solution

Restrict what the network can do by using regularization and dropout.

A neural network with enough parameters can fit any input/output mapping by memorizing. Accuracy seems great while training, but of course the network fails to perform very well on data it hasn't seen before and so does poorly on the test data or indeed in production. The network is overfitting.

One obvious way to stop the network from overfitting is to reduce the number of parameters that we have by decreasing the number of layers or making each layer smaller. But this of course also reduces the expressive power of our network. Regularization and dropout offer us something in between by restricting the expressive power of our network in a way that doesn't hurt the ability to learn (too much).

With regularization we add penalties to *extreme* values for parameters. The intuition here is that in order to fit an arbitrary input/output mapping, a network would need arbitrary parameters, while learned parameters tend to be in a normal range. So, making it harder to get to those arbitrary parameters should keep the network on the path of learning rather than memorizing.

Application in Keras is straightforward:

```
dense = Dense(128,
              activation='relu',
              kernel_regularizer=regularizers.l2(0.01))(flatten)
```

Regularizers can be applied to the weights of the kernel or the bias of the layer, or to the output of the layer. Which one and what penalty to use is mostly a matter of trial and error. 0.01 seems like a popular starting value.

Dropout is a similar technique, but more radical. Rather than keeping the weights of neurons in check, we randomly ignore a percentage of all neurons during training.

Similar to regularization, this makes it harder for a network to memorize input/output pairs, since it can't rely on specific neurons working during training. This nudges the network into learning general, robust features rather than one-off, specific ones to cover one training instance.

In Keras dropout is applied to a layer using the `Dropout` (pseudo)layer:

```
max_pool_1x = MaxPooling1D(window)(conv_1x)
dropout_1x = Dropout(0.3)(max_pool_1x)
```

This applies a dropout of 30% to the max-pooling layer, ignoring 30% of its neurons during training.

When doing inference, dropout is not applied. All things being equal this would increase the output of the layer by over 40%, so the framework automatically scales these outputs back.

Discussion

As you make your network more expressive, its tendency to overfit or memorize its inputs rather than learn general features will increase. Both regularization and dropout can play a role to reduce this effect. Both work by reducing the freedom of the network to develop arbitrary features, by punishing extreme values (regularization) or by ignoring the contribution of a percentage of the neurons in a layer (dropout).

An interesting alternative way to look at how networks with dropout work is to consider that if we have *N* neurons and we randomly switch a certain percentage of the neurons off, we really have created a generator that can create a very large variety of different but related networks. During training these different networks all learn the task at hand, but at evaluation time they all run in parallel and their average opinion is taken. So even if some of them start overfitting, chances are that this is drowned out in the aggregate vote.

2.6 Network Structure, Batch Size, and Learning Rate

Problem

How do you find the best network structure, batch size, and learning rate for a given problem?

Solution

Start small and work your way up.

Once we've identified the sort of network we'll need to solve a specific problem, we still have to make a number of implementation decisions. The more important among those are decisions about the network structure, the learning rate, and the batch size.

Let's start with the network structure. How many layers will we have? How big will each of those layers be? A decent strategy is to start with the smallest sizes that could possibly work. Being all enthusiastic about the "deep" in deep learning, there is a certain temptation to start with many layers. But typically if a one- or two-layer network doesn't perform at all, chances are that adding more layers isn't going to really help.

Continuing with the size of each individual layer, larger layers can learn more, but they also take longer and have more space to hide problems. As with the number of layers, start small and expand from there. If you suspect that the expressive power of a smaller network will be insufficient to make any sense of your data, consider simplifying your data; start with a small network that only distinguishes between the two most popular labels and then gradually increase the complexity of both the data and the network.

The batch size is the number of samples we feed into the network before adjusting the weights. The larger the batch size, the longer it takes to finish one, but the more accurate the gradient is. In order to get results quickly, it is advisable to start with a smallish batch size—32 seems to work well.

The learning rate determines how much we'll change the weights in our network in the direction of the derived gradient. The higher the rate, the quicker we move through the landscapes. Too big a rate, though, and we risk skipping over the good bits and we start thrashing. When we take into account that a smaller batch size leads to a less accurate gradient, it stands to reason that we should combine a small batch size with a smaller learning rate. So, the suggestion here is again to start out small and, when things work, experiment with larger batch rates and higher learning rates.

Training on GPUs impacts this assessment. GPUs efficiently run steps in parallel, so there is no real reason to pick a batch size that is so small that it leaves part of the GPU idle. What batch size that is depends of course on the network, but as long as the time per batch doesn't increase by much when you increase the batch size, you're still on the right side of things. A second consideration when running on GPUs is memory. When a batch no longer fits in the memory of the GPU things start to fail and you'll start to see out of memory messages.

Discussion

Network structure, batch size, and learning rate are some of the important hyper parameters that impact the performance of networks but have little to do with the actual strategy. For all of these a reasonable strategy is to start small (but big enough that things still work) and go bigger step by step, observing that the network still performs.

As we increase the number of layers and the size of each layer, we'll start to see symptoms of overfitting at some point (training and testing accuracy start to diverge, for example). That might be a good time to look at regularization and dropout.

CHAPTER 3

Calculating Text Similarity Using Word Embeddings

Before we get started, this is the first chapter with actual code in it. Chances are you skipped straight to here, and who would blame you? To follow the recipes it really helps though if you have the accompanying code up and running. You can easily do this by executing the following commands in a shell:

```
git clone \
  https://github.com/DOsinga/deep_learning_cookbook.git
cd deep_learning_cookbook
python3 -m venv venv3
source venv3/bin/activate
pip install -r requirements.txt
jupyter notebook
```

You can find a more detailed explanation in "What Do You Need to Know?" on page 9.

In this chapter we'll look at word embeddings and how they can help us to calculate the similarities between pieces of text. Word embeddings are a powerful technique used in natural language processing to represent words as vectors in an *n*-dimensional space. The interesting thing about this space is that words that have similar meanings will appear close to each other.

The main model we'll use here is a version of Google's Word2vec. This is not a deep neural model. In fact, it is no more than a big lookup table from word to vector and therefore hardly a model at all. The Word2vec embeddings are produced as a side effect of training a network to predict a word from context for sentences taken from Google News. Moreover, it is possibly the best-known example of an embedding, and embeddings are an important concept in deep learning.

Once you start looking for them, high-dimensional spaces with semantic properties start popping up everywhere in deep learning. We can build a movie recommender by projecting movies into a high-dimensional space (Chapter 4) or create a map of handwritten digits using only two dimensions (Chapter 13). Image recognition networks project images into a space such that similar images are near to each other (Chapter 10).

In the current chapter we'll focus on just word embeddings. We'll start with using a pretrained word embedding model to calculate word similarities, then show some interesting Word2vec math. We'll then explore how to visualize these high-dimensional spaces.

Next, we'll take a look at how we can exploit the semantic properties of word embeddings like Word2vec for domain-specific ranking. We'll treat the words and their embeddings as the entities they represent, with some interesting results. We'll start with finding entity classes in Word2vec embeddings—in this case, countries. We'll then show how to rank terms against these countries and how to visualize these results on a map.

Word embeddings are a powerful way to map words to vectors and have many uses. They are often used as a preprocessing step for text.

There are two Python notebooks associated with this chapter:

```
03.1 Using pretrained word embeddings
03.2 Domain specific ranking using word2vec cosine distance
```

3.1 Using Pretrained Word Embeddings to Find Word Similarity

Problem

You need to find out whether two words are similar but not equal, for example when you're verifying user input and you don't want to require the user to exactly enter the expected word.

Solution

You can use a pretrained word embedding model. We'll use `gensim` in this example, a useful library in general for topic modeling in Python.

The first step is to acquire a pretrained model. There are a number of pretrained models available for download on the internet, but we'll go with the Google News one. It has embeddings for 3 million words and was trained on roughly 100 billion words taken from the Google News archives. Downloading it will take a while, so we'll cache the file locally:

```
MODEL = 'GoogleNews-vectors-negative300.bin'
path = get_file(MODEL + '.gz',
    'https://s3.amazonaws.com/dl4j-distribution/%s.gz' % MODEL)
unzipped = os.path.join('generated', MODEL)
if not os.path.isfile(unzipped):
    with open(unzipped, 'wb') as fout:
        zcat = subprocess.Popen(['zcat'],
                                stdin=open(path),
                                stdout=fout
                               )
        zcat.wait()
```

```
Downloading data from GoogleNews-vectors-negative300.bin.gz
1647050752/1647046227 [==============================] - 71s 0us/step
```

Now that we have the model downloaded, we can load it into memory. The model is quite big and this will take around 5 GB of RAM:

```
model = gensim.models.KeyedVectors.load_word2vec_format(MODEL, binary=True)
```

Once the model has finished loading, we can use it to find similar words:

```
model.most_similar(positive=['espresso'])
```

```
[(u'cappuccino', 0.6888186931610107),
 (u'mocha', 0.6686209440231323),
 (u'coffee', 0.6616827249526978),
 (u'latte', 0.6536752581596375),
 (u'caramel_macchiato', 0.6491267681121826),
 (u'ristretto', 0.6485546827316284),
 (u'espressos', 0.6438628435134888),
 (u'macchiato', 0.6428250074386597),
 (u'chai_latte', 0.6308028697967529),
 (u'espresso_cappuccino', 0.6280542612075806)]
```

Discussion

Word embeddings associate an *n*-dimensional vector with each word in the vocabulary in such a way that similar words are near each other. Finding similar words is a mere nearest-neighbor search, for which there are efficient algorithms even in high-dimensional spaces.

Simplifying things somewhat, the Word2vec embeddings are obtained by training a neural network to predict a word from its context. So, we ask the network to predict which word it should pick for X in a series of fragments; for example, "the cafe served a X that really woke me up."

This way words that can be inserted into similar patterns will get vectors that are close to each other. We don't care about the actual task, just about the assigned weights, which we will get as a side effect of training this network.

Later in this book we'll see how word embeddings can also be used to feed words into a neural network. It is much more feasible to feed a 300-dimensional embedding vector into a network than a 3-million-dimensional one that is one-hot encoded. Moreover, a network fed with pretrained word embeddings doesn't have to learn the relationships between the words, but can start with the real task at hand immediately.

3.2 Word2vec Math

Problem

How can you automatically answer questions of the form "A is to B as C is to what?"

Solution

Use the semantic properties of the Word2vec model. The `gensim` library makes this rather straightforward:

```
def A_is_to_B_as_C_is_to(a, b, c, topn=1):
  a, b, c = map(lambda x:x if type(x) == list else [x], (a, b, c))
  res = model.most_similar(positive=b + c, negative=a, topn=topn)
  if len(res):
    if topn == 1:
      return res[0][0]
    return [x[0] for x in res]
  return None
```

We can now apply this to arbitrary words—for example, to find what relates to "king" the way "son" relates to "daughter":

```
A_is_to_B_as_C_is_to('man', 'woman', 'king')
```

```
u'queen'
```

We can also use this approach to look up the capitals of selected countries:

```
for country in 'Italy', 'France', 'India', 'China':
    print('%s is the capital of %s' %
          (A_is_to_B_as_C_is_to('Germany', 'Berlin', country), country))
```

```
Rome is the capital of Italy
Paris is the capital of France
Delhi is the capital of India
Beijing is the capital of China
```

or to find the main products of companies (note the # placeholder for any number used in these embeddings):

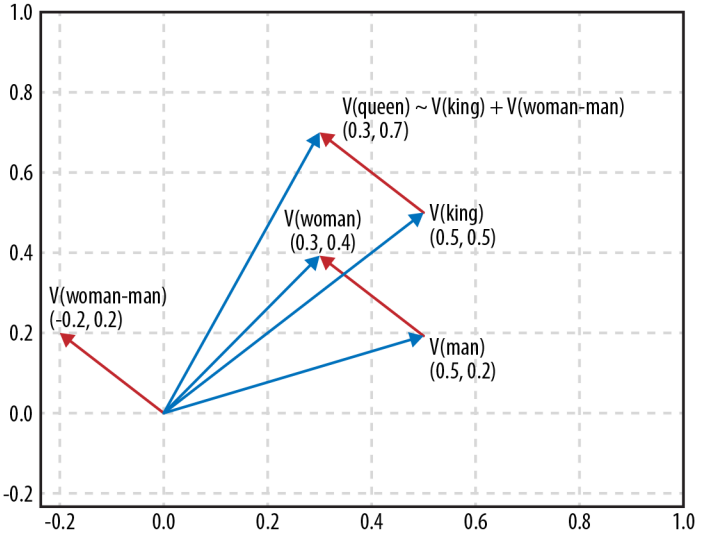
```
for company in 'Google', 'IBM', 'Boeing', 'Microsoft', 'Samsung':
  products = A_is_to_B_as_C_is_to(
    ['Starbucks', 'Apple'], ['Starbucks_coffee', 'iPhone'], company, topn=3)
```

```
    print('%s -> %s' %
          (company, ', '.join(products)))
```

```
Google -> personalized_homepage, app, Gmail
IBM -> DB2, WebSphere_Portal, Tamino_XML_Server
Boeing -> Dreamliner, airframe, aircraft
Microsoft -> Windows_Mobile, SyncMate, Windows
Samsung -> MM_A###, handset, Samsung_SCH_B###
```

Discussion

As we saw in the previous step, the vectors associated with the words encode the meaning of the words—words that are similar to each other have vectors that are close to each other. It turns out that the difference between word vectors also encodes the difference between words, so if we take the vector for the word "son" and deduct the vector for the word "daughter" we end up with a difference that can be interpreted as "going from male to female." If we add this difference to the vector for the word "king" we end up near the vector for the word "queen":



The `most_similar` method takes one or more positive words and one or more negative words. It looks up the corresponding vectors, then deducts the negative from the positive and returns the words that have vectors nearest to the resulting vector.

So in order to answer the question "A is to B as C is to?" we want to deduct A from B and then add C, or call `most_similar` with `positive = [B, C]` and `negative = [A]`. The example `A_is_to_B_as_C_is_to` adds two small features to this behavior. If we request only one example, it will return a single item, rather than a list with one item. Similarly, we can return either lists or single items for A, B, and C.

Being able to provide lists turned out to be useful in the product example. We asked for three products per company, which makes it more important to get the vector exactly right than if we only asked for one. By providing "Starbucks" and "Apple," we get a more exact vector for the concept of "is a product of."

3.3 Visualizing Word Embeddings

Problem

You want to get some insight into how word embeddings partition a set of objects.

Solution

A 300-dimensional space is hard to browse, but luckily we can use an algorithm called *t-distributed stochastic neighbor embedding* (t-SNE) to fold a higher-dimensional space into something more comprehensible, like two dimensions.

Let's say we want to look at how three sets of terms are partitioned. We'll pick countries, sports, and drinks:

```
beverages = ['espresso', 'beer', 'vodka', 'wine', 'cola', 'tea']
countries = ['Italy', 'Germany', 'Russia', 'France', 'USA', 'India']
sports = ['soccer', 'handball', 'hockey', 'cycling', 'basketball', 'cricket']

items = beverages + countries + sports
```

Now let's look up their vectors:

```
item_vectors = [(item, model[item])
                    for item in items
                    if item in model]
```

We can now use t-SNE to find the clusters in the 300-dimensional space:

```
vectors = np.asarray([x[1] for x in item_vectors])
lengths = np.linalg.norm(vectors, axis=1)
norm_vectors = (vectors.T / lengths).T
tsne = TSNE(n_components=2, perplexity=10,
            verbose=2).fit_transform(norm_vectors)
```

Let's use matplotlib to show the results in a nice scatter plot:

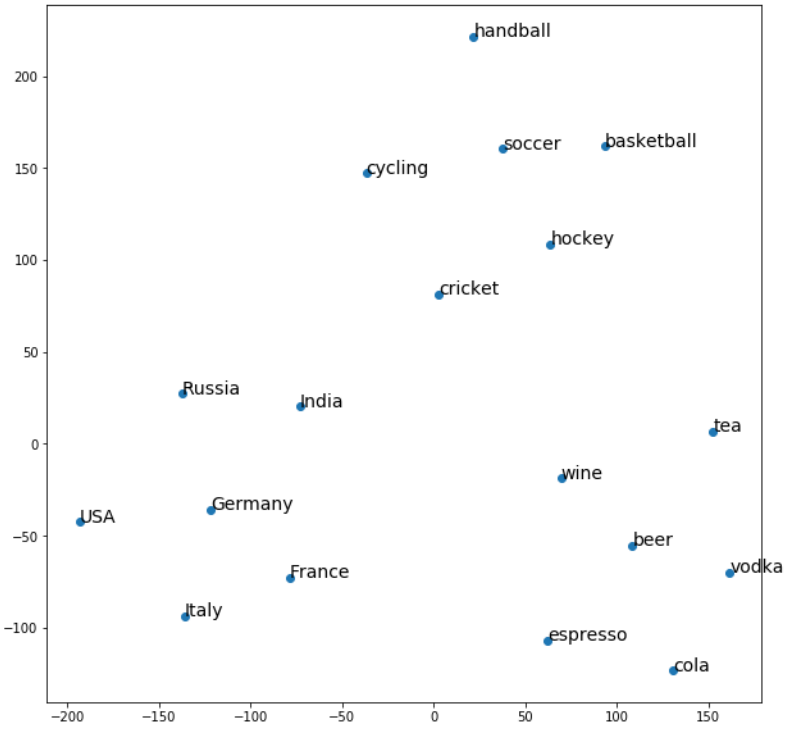
```
x=tsne[:,0]
y=tsne[:,1]

fig, ax = plt.subplots()
ax.scatter(x, y)

for item, x1, y1 in zip(item_vectors, x, y):
    ax.annotate(item[0], (x1, y1))

plt.show()
```

The result is:



Discussion

t-SNE is a clever algorithm; you give it a set of points in a high-dimensional space, and it iteratively tries to find the best projection onto a lower-dimensional space (usually a plane) that maintains the distances between the points as well as possible. It is therefore very suitable for visualizing higher dimensions like (word) embeddings.

For more complex situations, the `perplexity` parameter is something to play around with. This variable loosely determines the balance between local accuracy and overall accuracy. Setting it to a low value creates small clusters that are locally accurate; setting it higher leads to more local distortions, but with better overall clusters.

3.4 Finding Entity Classes in Embeddings

Problem

In high-dimensional spaces there are often subspaces that contain only entities of one class. How do you find those spaces?

Solution

Apply a support vector machine (SVM) on a set of examples and counterexamples. For example, let's find the countries in the Word2vec space. We'll start by loading up the model again and exploring things similar to a country, Germany:

```
model = gensim.models.KeyedVectors.load_word2vec_format(MODEL, binary=True)
model.most_similar(positive=['Germany'])
```

```
[(u'Austria', 0.7461062073707581),
 (u'German', 0.7178748846054077),
 (u'Germans', 0.6628648042678833),
 (u'Switzerland', 0.6506867408752441),
 (u'Hungary', 0.6504981517791748),
 (u'Germnay', 0.649348258972168),
 (u'Netherlands', 0.6437495946884155),
 (u'Cologne', 0.6430779099464417)]
```

As you can see there are a number of countries nearby, but words like "German" and the names of German cities also show up in the list. We could try to construct a vector that best represents the concept of "country" by adding up the vectors of many countries rather than just using Germany, but that only goes so far. The concept of country in the embedding space isn't a point, it is a shape. What we need is a real classifier.

Support vector machines have proven effective for classification tasks like this. Scikit-learn has an easy-to-deploy solution. The first step is to build a training set. For this recipe getting positive examples is not hard since there are only so many countries:

```
positive = ['Chile', 'Mauritius', 'Barbados', 'Ukraine', 'Israel',
  'Rwanda', 'Venezuela', 'Lithuania', 'Costa_Rica', 'Romania',
  'Senegal', 'Canada', 'Malaysia', 'South_Korea', 'Australia',
  'Tunisia', 'Armenia', 'China', 'Czech_Republic', 'Guinea',
  'Gambia', 'Gabon', 'Italy', 'Montenegro', 'Guyana', 'Nicaragua',
  'French_Guiana', 'Serbia', 'Uruguay', 'Ethiopia', 'Samoa',
  'Antarctica', 'Suriname', 'Finland', 'Bermuda', 'Cuba', 'Oman',
  'Azerbaijan', 'Papua', 'France', 'Tanzania', 'Germany' … ]
```

Having more positive examples is of course better, but for this example using 40–50 will give us a good idea of how the solution works.

We also need some negative examples. We sample these directly from the general vocabulary of the Word2vec model. We could get unlucky and draw a country and put it in the negative examples, but given the fact that we have 3 million words in the model and there are less than 200 countries in the world, we'd have to be very unlucky indeed:

```
negative = random.sample(model.vocab.keys(), 5000)
negative[:4]
```

```
[u'Denys_Arcand_Les_Invasions',
 u'2B_refill',
```

```
 u'strained_vocal_chords',
 u'Manifa']
```

Now we'll create a labeled training set based on the positive and negative examples. We'll use `1` as the label for something being a country, and `0` for it not being a country. We'll follow the convention of storing the training data in a variable `X` and the labels in a variable `y`:

```
labelled = [(p, 1) for p in positive] + [(n, 0) for n in negative]
random.shuffle(labelled)
X = np.asarray([model[w] for w, l in labelled])
y = np.asarray([l for w, l in labelled])
```

Let's train the model. We'll set aside a fraction of the data to evaluate how we are doing:

```
TRAINING_FRACTION = 0.7
cut_off = int(TRAINING_FRACTION * len(labelled))
clf = svm.SVC(kernel='linear')
clf.fit(X[:cut_off], y[:cut_off])
```

The training should happen almost instantaneously even on a not very powerful computer since our dataset is relatively small. We can have a peek at how we are doing by looking at how many times the model has the right prediction for the bits of the eval set:

```
res = clf.predict(X[cut_off:])

missed = [country for (pred, truth, country) in
          zip(res, y[cut_off:], labelled[cut_off:]) if pred != truth]
100 - 100 * float(len(missed)) / len(res), missed
```

The results you get will depend a bit on the positive countries selected and which negative samples you happened to draw. I mostly get a list of countries that it missed —typically because the country name also means something else, like Jordan, but there are also some genuine misses in there. The precision comes out at 99.9% or so.

We can now run the classifier over all of the words to extract the countries:

```
res = []
for word, pred in zip(model.index2word, all_predictions):
  if pred:
    res.append(word)
    if len(res) == 150:
      break
random.sample(res, 10)
```

```
[u'Myanmar',
 u'countries',
 u'Sri_Lanka',
 u'Israelis',
 u'Australia',
 u'Pyongyang',
```

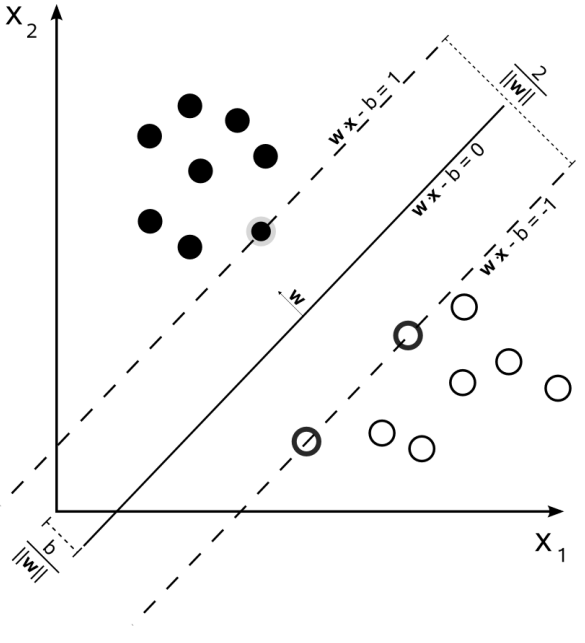
```
    u'New_Hampshire',
    u'Italy',
    u'China',
    u'Philippine']
```

The results are pretty good, though not perfect. The word "countries" itself, for example, is classified as a country, as are entities like continents or US states.

Discussion

Support vector machines are effective tools when it comes to finding classes within a higher-dimensional space like word embeddings. They work by trying to find hyperplanes that separate the positive examples from the negative examples.

Countries in Word2vec are all somewhat near to each other since they share a semantic aspect. SVMs help us find the cloud of countries and come up with boundaries. The following diagram visualizes this in two dimensions:



SVMs can be used for all kinds of ad hoc classifiers in machine learning since they are effective even if the number of dimensions is greater than the number of samples, like in this case. The 300 dimensions could allow the model to overfit the data, but because the SVM tries to find a simple model to fit the data, we can still generalize from a dataset as small as a few dozen examples.

The results achieved are pretty good, though it is worth noting that in a situation where we have 3 million negative examples, 99.7% precision would still give us 9,000 false positives, drowning out the actual countries.

3.5 Calculating Semantic Distances Inside a Class

Problem

How do you find the most relevant items from a class for a given criterion?

Solution

Given a class, for example *countries*, we can rank the members of that class against a criterion, by looking at the relative distances:

```
country_to_idx = {country['name']: idx for idx, country in enumerate(countries)}
country_vecs = np.asarray([model[c['name']] for c in countries])
country_vecs.shape
```

```
(184, 300)
```

We can now, as before, extract the vectors for the countries into a `numpy` array that lines up with the countries:

```
countries = list(country_to_cc.keys())
country_vecs = np.asarray([model[c] for c in countries])
```

A quick sanity check to see which countries are most like Canada:

```
dists = np.dot(country_vecs, country_vecs[country_to_idx['Canada']])
for idx in reversed(np.argsort(dists)[-8:]):
    print(countries[idx], dists[idx])
```

```
Canada 7.5440245
New_Zealand 3.9619699
Finland 3.9392405
Puerto_Rico 3.838145
Jamaica 3.8102934
Sweden 3.8042784
Slovakia 3.7038736
Australia 3.6711009
```

The Caribbean countries are somewhat surprising and a lot of the news about Canada must be related to hockey, given the appearance of Slovakia and Finland in the list, but otherwise it doesn't look unreasonable.

Let's switch gears and do some ranking for an arbitrary term over the set of countries. For each country we'll calculate the distance between the name of the country and the term we want to rank against. Countries that are "closer" to the term are more relevant for the term:

```
def rank_countries(term, topn=10, field='name'):
    if not term in model:
        return []
    vec = model[term]
    dists = np.dot(country_vecs, vec)
    return [(countries[idx][field], float(dists[idx]))
            for idx in reversed(np.argsort(dists)[-topn:])]
```

For example:

```
rank_countries('cricket')
```

```
[('Sri_Lanka', 5.92276668548584),
 ('Zimbabwe', 5.336524486541748),
 ('Bangladesh', 5.192488670349121),
 ('Pakistan', 4.948408126831055),
 ('Guyana', 3.9162840843200684),
 ('Barbados', 3.757995128631592),
 ('India', 3.7504401206970215),
 ('South_Africa', 3.6561498641967773),
 ('New_Zealand', 3.642028331756592),
 ('Fiji', 3.608567714691162)]
```

Since the Word2vec model we are using was trained on Google News, the ranker will return countries that are mostly known for the given term in the recent news. India might be more often mentioned for cricket, but as long as it is also covered for other things, Sri Lanka can still win.

Discussion

In spaces where we have members of different classes projected into the same dimensions, we can use the cross-class distances as a measure of affinity. Word2vec doesn't quite represent a conceptual space (the word "Jordan" can refer to the river, the country, or a person), but it is good enough to nicely rank countries on relevance for various concepts.

A similar approach is often taken when building recommender systems. For the Netflix challenge, for example, a popular strategy was to use user ratings for movies as a way to project users and movies into a shared space. Movies that are close to a user are then expected to be rated highly by that user.

In situations where we have two spaces that are not the same, we can still use this trick if we can calculate the projection matrix to go from one space to the other. This is possible if we have enough candidates whose positions we know in both spaces.

3.6 Visualizing Country Data on a Map

Problem

How can you visalize country rankings from an experiment on a map?

Solution

GeoPandas is a perfect tool to visualize numerical data on top of a map.

This nifty library combines the power of Pandas with geographical primitives and comes preloaded with a few maps. Let's load up the world:

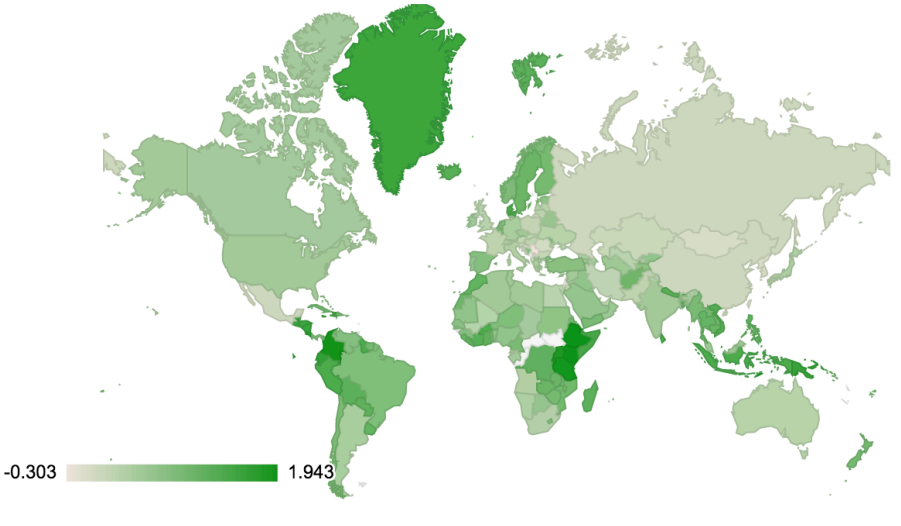
```
world = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
world.head()
```

This shows us some basic information about a set of countries. We can add a column to the `world` object based on our `rank_countries` function:

```
def map_term(term):
    d = {k.upper(): v for k, v in rank_countries(term,
                                                 topn=0,
                                                 field='cc3')}
    world[term] = world['iso_a3'].map(d)
    world[term] /= world[term].max()
    world.dropna().plot(term, cmap='OrRd')

map_term('coffee')
```

This draws, for example, the map for coffee quite nicely, highlighting the coffee consuming countries and the coffee producing countries.



Discussion

Visualizing data is an important technique for machine learning. Being able to look at the data, whether it is the input or the result of some algorithm, allows us to quickly spot anomalies. Do people in Greenland really drink that much coffee? Or are we seeing an artifact because of "Greenlandic coffee" (a variation on Irish coffee)? And those countries in the middle of Africa—do they really neither drink nor produce coffee? Or do we just have no data on them because they don't occur in our embeddings?

GeoPandas is the perfect tool to analyze geographically coded information and builds on the general data capabilities of Pandas, which we'll see more of in Chapter 6.

CHAPTER 4

Building a Recommender System Based on Outgoing Wikipedia Links

Recommender systems are traditionally trained on previously collected ratings from users. We want to predict ratings from users, so starting with historical ratings feels like a natural fit. However, this requires us to have a substantial set of ratings before we can get going and it doesn't allow us to do a good job on new items for which we don't have ratings yet. Moreover, we deliberately ignore the metainformation that we have on items.

In this chapter you'll explore how to build a simple movie recommender system based solely on outgoing Wikipedia links. You'll start by extracting a training set from Wikipedia and then train embeddings based on these links. You'll then implement a simple support vector machine classifier to give recommendations. Finally, you'll explore how you can use your newly trained embeddings to predict review scores for the movies.

The code in this chapter can be found in these notebooks:

```
04.1 Collect movie data from Wikipedia
04.2 Build a recommender system based on outgoing Wikipedia links
```

4.1 Collecting the Data

Problem

You want to obtain a dataset for training for a specific domain, like movies.

Solution

Parse a Wikipedia dump and extract only the pages that are movies.

The code in this recipe shows how to fetch and extract training data from Wikipedia, which is a very useful skill. However, downloading and processing a full dump takes a rather long time. The *data* directory of the notebook folder contains the top 10,000 movies pre-extracted that we'll use in the rest of the chapter, so you don't need to run the steps in this recipe.

Let's start by downloading a recent dump from Wikipedia. You can easily do this using your favorite browser, and if you don't need the very latest version, you should probably pick a nearby mirror. But you can also do it programmatically. Here's how to get the latest dump pages:

```
index = requests.get('https://dumps.wikimedia.org/backup-index.html').text
soup_index = BeautifulSoup(index, 'html.parser')
dumps = [a['href'] for a in soup_index.find_all('a')
             if a.has_attr('href') and a.text[:-1].isdigit()]
```

We'll now go through the dumps and find the newest one that has actually finished processing:

```
for dump_url in sorted(dumps, reverse=True):
    print(dump_url)
    dump_html = index = requests.get(
        'https://dumps.wikimedia.org/enwiki/' + dump_url).text
    soup_dump = BeautifulSoup(dump_html, 'html.parser')
    pages_xml = [a['href'] for a in soup_dump.find_all('a')
                 if a.has_attr('href')
                 and a['href'].endswith('-pages-articles.xml.bz2')]
    if pages_xml:
        break
    time.sleep(0.8)
```

Note the sleep to stay under the rate limiting of Wikipedia. Now let's fetch the dump:

```
wikipedia_dump = pages_xml[0].rsplit('/')[-1]
url = url = 'https://dumps.wikimedia.org/' + pages_xml[0]
path = get_file(wikipedia_dump, url)
path
```

The dump we retrieved is a bz2-compressed XML file. We'll use `sax` to parse the Wikipedia XML. We're interested in the `<title>` and the `<page>` tags so our `ContentHandler` looks like this:

```
class WikiXmlHandler(xml.sax.handler.ContentHandler):
    def __init__(self):
        xml.sax.handler.ContentHandler.__init__(self)
        self._buffer = None
        self._values = {}
        self._movies = []
        self._curent_tag = None
```

```
    def characters(self, content):
        if self._curent_tag:
            self._buffer.append(content)

    def startElement(self, name, attrs):
        if name in ('title', 'text'):
            self._curent_tag = name
            self._buffer = []

    def endElement(self, name):
        if name == self._curent_tag:
            self._values[name] = ' '.join(self._buffer)

        if name == 'page':
            movie = process_article(**self._values)
            if movie:
                self._movies.append(movie)
```

For each `<page>` tag this collects the contents of the title and of the text into the `self._values` dictionary and calls `process_article` with the collected values.

Although Wikipedia started out as a hyperlinked text-based encyclopedia, over the years it has developed into a more structured data dump. One way this is done is by having pages link back to so-called *category pages*. These links function as tags. The page for the film *One Flew Over the Cuckoo's Nest* links to the category page "1975 films," so we know it is a movie from 1975. Unfortunately, there is no such thing as a category page for just movies. Fortunately, there is a better way: Wikipedia templates.

Templates started out as a way to make sure that pages that contain similar information have that information rendered in the same way. The "infobox" template is very useful for data processing. Not only does it contain a list of key/value pairs applicable to the subject of the page, but it also has a type. One of the types is "film," which makes the task of extracting all movies a lot easier.

For each movie we want to extract the name, the outgoing links and, just because we can, the properties stored in the infobox. The aptly named `mwparserfromhell` does a decent job of parsing Wikipedia:

```
def process_article(title, text):
    rotten = [(re.findall('\d\d?\d?%', p),
        re.findall('\d\.\d\/\d+|$', p), p.lower().find('rotten tomatoes'))
        for p in text.split('\n\n')]
    rating = next(((perc[0], rating[0]) for perc, rating, idx in rotten
        if len(perc) == 1 and idx > -1), (None, None))
    wikicode = mwparserfromhell.parse(text)
    film = next((template for template in wikicode.filter_templates()
                 if template.name.strip().lower() == 'infobox film'),
                None)
    if film:
        properties = {param.name.strip_code().strip():
```

```
                    param.value.strip_code().strip()
                    for param in film.params
                    if param.value.strip_code().strip()
                   }
        links = [x.title.strip_code().strip()
                 for x in wikicode.filter_wikilinks()]
        return (title, properties, links) + rating
```

We can now feed the bzipped dump into the parser:

```
parser = xml.sax.make_parser()
handler = WikiXmlHandler()
parser.setContentHandler(handler)
for line in subprocess.Popen(['bzcat'],
                             stdin=open(path),
                             stdout=subprocess.PIPE).stdout:
  try:
    parser.feed(line)
  except StopIteration:
    break
```

Finally, let's save the results so next time we need the data, we don't have to process for hours:

```
with open('wp_movies.ndjson', 'wt') as fout:
  for movie in handler._movies:
    fout.write(json.dumps(movie) + '\n')
```

Discussion

Wikipedia is not only a great resource to answer questions about almost any area of human knowledge; it also is the starting point for many deep learning experiments. Knowing how to parse the dumps and extract the relevant bits is a skill useful for many projects.

At 13 GB the dumps are sizeable downloads. Parsing the Wikipedia markup language comes with its own challenges: the language has grown organically over the years and doesn't seem to have a strong underlying design. But with today's fast connections and some great open source libraries to help with the parsing, it has all become quite doable.

In some situations the Wikipedia API might be more appropriate. This REST interface to Wikipedia allows you to search and query in a number of powerful ways and only fetch the articles that you need. Getting all the movies that way would take a long time given the rate limiting, but for smaller domains it is an option.

If you end up parsing Wikipedia for many projects, it might be worth it to first import the dump into a database like Postgres so you can query the dataset directly.

4.2 Training Movie Embeddings

Problem

How can you use link data between entities to produce suggestions like "If you liked this, you might also be interested in that"?

Solution

Train embeddings using some metainformation as connectors. This recipe builds on the previous one by using the movies and links extracted there. To make the dataset a bit smaller and less noisy, we'll work with only the top 10,000 movies determined by popularity on Wikipedia.

We'll treat the outgoing links as the connectors. The intuition here is that movies that link to the same page are similar. They might have the same director or be of the same genre. As the model trains, it learns not only which movies are similar, but also which links are similar. This way it can generalize and discover that a link to the year 1978 has a similar meaning as a link to 1979, which in turn helps with movie similarity.

We'll start by counting the outgoing links as a quick way to see whether what we have is reasonable:

```
link_counts = Counter()
for movie in movies:
    link_counts.update(movie[2])
link_counts.most_common(3)
```

```
[(u'Rotten Tomatoes', 9393),
 (u'Category:English-language films', 5882),
 (u'Category:American films', 5867)]
```

Our model's task is to determine whether a certain link can be found on the Wikipedia page of a movie, so we need to feed it labeled examples of matches and nonmatches. We'll keep only links that occur at least three times and build a list of all valid (link, movie) pairs, which we'll store for quick lookups later. We keep the same handy as a set for quick lookups later:

```
top_links = [link for link, c in link_counts.items() if c >= 3]
link_to_idx = {link: idx for idx, link in enumerate(top_links)}
movie_to_idx = {movie[0]: idx for idx, movie in enumerate(movies)}
pairs = []
for movie in movies:
    pairs.extend((link_to_idx[link], movie_to_idx[movie[0]])
                 for link in movie[2] if link in link_to_idx)
pairs_set = set(pairs)
```

We are now ready to introduce our model. Schematically, we take both the `link_id` and the `movie_id` as a number and feed those into their respective embedding layers. The embedding layer will allocate a vector of `embedding_size` for each possible input. We then set the dot product of these two vectors to be the output of our model. The model will learn weights such that this dot product will be close to the label. These weights will then project movies and links into a space such that movies that are similar end up in a similar location:

```
def movie_embedding_model(embedding_size=30):
    link = Input(name='link', shape=(1,))
    movie = Input(name='movie', shape=(1,))
    link_embedding = Embedding(name='link_embedding',
        input_dim=len(top_links), output_dim=embedding_size)(link)
    movie_embedding = Embedding(name='movie_embedding',
        input_dim=len(movie_to_idx), output_dim=embedding_size)(movie)
    dot = Dot(name='dot_product', normalize=True, axes=2)(
        [link_embedding, movie_embedding])
    merged = Reshape((1,))(dot)
    model = Model(inputs=[link, movie], outputs=[merged])
    model.compile(optimizer='nadam', loss='mse')
    return model

model = movie_embedding_model()
```

We'll feed the model using a generator. The generator yields batches of data made up of positive and negative examples.

We sample the positive samples from the pairs array and then fill it up with negative examples. The negative examples are randomly picked and we make sure they are not in the `pairs_set`. We then return the data in a format that our network expects, an input/output tuple:

```
def batchifier(pairs, positive_samples=50, negative_ratio=5):
    batch_size = positive_samples * (1 + negative_ratio)
    batch = np.zeros((batch_size, 3))
    while True:
        for idx, (link_id, movie_id) in enumerate(
                random.sample(pairs, positive_samples)):
            batch[idx, :] = (link_id, movie_id, 1)
        idx = positive_samples
        while idx < batch_size:
            movie_id = random.randrange(len(movie_to_idx))
            link_id = random.randrange(len(top_links))
            if not (link_id, movie_id) in pairs_set:
                batch[idx, :] = (link_id, movie_id, -1)
                idx += 1
        np.random.shuffle(batch)
        yield {'link': batch[:, 0], 'movie': batch[:, 1]}, batch[:, 2]
```

Time to train the model:

```
positive_samples_per_batch=512

model.fit_generator(
    batchifier(pairs,
               positive_samples=positive_samples_per_batch,
               negative_ratio=10),
    epochs=25,
    steps_per_epoch=len(pairs) // positive_samples_per_batch,
    verbose=2
)
```

Training times will depend on your hardware, but if you start with the 10,000 movie dataset they should be fairly short, even on a laptop without GPU acceleration.

We can now extract the movie embeddings from our model by accessing the weights of the `movie_embedding` layer. We normalize them so we can use the dot product as an approximation of the cosine similarity:

```
movie = model.get_layer('movie_embedding')
movie_weights = movie.get_weights()[0]
lens = np.linalg.norm(movie_weights, axis=1)
normalized = (movie_weights.T / lens).T
```

Now let's see if the embeddings make some sense:

```
def neighbors(movie):
    dists = np.dot(normalized, normalized[movie_to_idx[movie]])
    closest = np.argsort(dists)[-10:]
    for c in reversed(closest):
        print(c, movies[c][0], dists[c])

neighbors('Rogue One')
```

```
29 Rogue One 0.9999999
3349 Star Wars: The Force Awakens 0.9722805
101 Prometheus (2012 film) 0.9653338
140 Star Trek Into Darkness 0.9635347
22 Jurassic World 0.962336
25 Star Wars sequel trilogy 0.95218825
659 Rise of the Planet of the Apes 0.9516557
62 Fantastic Beasts and Where to Find Them (film) 0.94662267
42 The Avengers (2012 film) 0.94634
37 Avatar (2009 film) 0.9460137
```

Discussion

Embeddings are a useful technique, and not just for words. In this recipe we've trained a simple network and produced embeddings for movies with reasonable results. This technique can be applied any time we have a way to connect items. In

this case we used the outgoing Wikipedia links, but we could also use incoming links or the words that appear on the page.

The model we trained here is extremely simple. All we do is ask it to come up with an embedding space such that the combination of the vector for the movie and the vector for the link can be used to predict whether or not they will co-occur. This forces the network to project movies into a space such that similar movies end up in a similar location. We can use this space to find similar movies.

In the Word2vec model we use the context of a word to predict the word. In the example of this recipe we don't use the context of the link. For outgoing links it doesn't seem like a particularly useful signal, but if we were using incoming links, it might have made sense. Pages linking to movies do this in a certain order, and we could use the context of the links to improve our embedding.

Alternatively, we could use the actual Word2vec code and run it over any of the pages that link to movies, but keep the links to movies as special tokens. This would then create a mixed movie and word embedding space.

4.3 Building a Movie Recommender

Problem

How can you build a recommender system based on embeddings?

Solution

Use a support vector machine to separate the positively ranked items from the negatively ranked items.

The previous recipe let us cluster movies and make suggestions like "If you liked *Rogue One*, you should also check out *Interstellar*." In a typical recommender system we want to show suggestions based on a series of movies that the user has rated. As we did in Chapter 3, we can use an SVM to do just this. Let's take the best and worst movies according to *Rolling Stone* from 2015 and pretend they are user ratings:

```
best = ['Star Wars: The Force Awakens', 'The Martian (film)',
        'Tangerine (film)', 'Straight Outta Compton (film)',
        'Brooklyn (film)', 'Carol (film)', 'Spotlight (film)']
worst = ['American Ultra', 'The Cobbler (2014 film)',
         'Entourage (film)', 'Fantastic Four (2015 film)',
         'Get Hard', 'Hot Pursuit (2015 film)', 'Mortdecai (film)',
         'Serena (2014 film)', 'Vacation (2015 film)']
y = np.asarray([1 for _ in best] + [0 for _ in worst])
X = np.asarray([normalized_movies[movie_to_idx[movie]]
                for movie in best + worst])
```

Constructing and training a simple SVM classifier based on this is easy:

```
clf = svm.SVC(kernel='linear')
clf.fit(X, y)
```

We can now run the new classifier over all the movies in our dataset and print the best five and the worst five:

```
estimated_movie_ratings = clf.decision_function(normalized_movies)
best = np.argsort(estimated_movie_ratings)
print('best:')
for c in reversed(best[-5:]):
    print(c, movies[c][0], estimated_movie_ratings[c])

print('worst:')
for c in best[:5]:
    print(c, movies[c][0], estimated_movie_ratings[c])
```

```
best:
(6870, u'Goodbye to Language', 1.24075226186855)
(6048, u'The Apu Trilogy', 1.2011876298842317)
(481, u'The Devil Wears Prada (film)', 1.1759994747169913)
(307, u'Les Mis\xe9rables (2012 film)', 1.1646775074857494)
(2106, u'A Separation', 1.1483743944891462)
worst:
(7889, u'The Comebacks', -1.5175929012505527)
(8837, u'The Santa Clause (film series)', -1.4651252650867073)
(2518, u'The Hot Chick', -1.464982008376793)
(6285, u'Employee of the Month (2006 film)', -1.4620595013243951)
(7339, u'Club Dread', -1.4593221506016203)
```

Discussion

As we saw in the previous chapter, we can use support vector machines to efficiently construct a classifier that distinguishes between two classes. In this case, we have it distinguish between good movies and bad movies based on the embeddings that we have previously learned.

Since an SVM finds one or more hyperplanes that separate the "good" examples from the "bad" examples, we can use this as the personalization function—the movies that are the furthest from the separating hyperplane and on the right side are the movies that should be liked best.

4.4 Predicting Simple Movie Properties

Problem

You want to predict simple movie properties, like Rotten Tomatoes ratings.

Solution

Use a linear regression model on the learned vectors of the embedding model to predict movie properties.

Let's try this for Rotten Tomatoes ratings. Luckily they are already present in our data in `movie[-2]` as a string of the form *N%*:

```
rotten_y = np.asarray([float(movie[-2][:-1]) / 100
                       for movie in movies if movie[-2]])
rotten_X = np.asarray([normalized_movies[movie_to_idx[movie[0]]]
                       for movie in movies if movie[-2]])
```

This should get us data for about half our movies. Let's train on the first 80%:

```
TRAINING_CUT_OFF = int(len(rotten_X) * 0.8)
regr = LinearRegression()
regr.fit(rotten_X[:TRAINING_CUT_OFF], rotten_y[:TRAINING_CUT_OFF])
```

Now let's see how we're doing on the last 20%:

```
error = (regr.predict(rotten_X[TRAINING_CUT_OFF:]) -
         rotten_y[TRAINING_CUT_OFF:])
'mean square error %2.2f' % np.mean(error ** 2)
```

```
mean square error 0.06
```

That looks really impressive! But while it is a testament to how effective linear regression can be, there is an issue with our data that makes predicting the Rotten Tomatoes score easier: we've been training on the top 10,000 movies, and while popular movies aren't always better, on average they do get better ratings.

We can get an idea of how well we're doing by comparing our predictions with just always predicting the average score:

```
error = (np.mean(rotten_y[:TRAINING_CUT_OFF]) - rotten_y[TRAINING_CUT_OFF:])
'mean square error %2.2f' % np.mean(error ** 2)
```

```
'mean square error 0.09'
```

Our model does perform quite a bit better, but the underlying data makes it easy to produce a reasonable result.

Discussion

Complex problems often need complex solutions, and deep learning can definitely give us those. However, starting with the simplest thing that could possibly work is often a good approach. It gets us started quickly and gives us an idea of whether we're looking in the right direction: if the simple model doesn't produce any useful results at all it's not that likely that a complex model will help, whereas if the simple model does work there's a good chance that a more complex model can help us achieve better results.

Linear regression models are as simple as they come. The model tries to find a set of factors such that the linear combination of these factors and our vectors approach the target value as closely as possible. One nice aspect of these models compared to most machine learning models is that we can actually see what the contribution of each of the factors is.

CHAPTER 5

Generating Text in the Style of an Example Text

In this chapter we'll look at how we can use recurrent neural networks (RNNs) to generate text in the style of a body of text. This makes for fun demos. People have used this type of network to generate anything from names of babies to descriptions of colors. These demos are a good way to get comfortable with recurrent networks. RNNs have their practical uses too—later in the book we'll use them to train a chatbot and build a recommender system for music based on harvested playlists, and RNNs have been used in production to track objects in video.

The recurrent neural network is a type of neural network that is helpful when working with time or sequences. We'll first look at Project Gutenberg as a source of free books and download the collected works of William Shakespeare using some simple code. Next, we'll use an RNN to produce texts that seem Shakespearean (if you don't pay too much attention) by training the network on downloaded text. We'll then repeat the trick on Python code, and see how to vary the output. Finally, since Python code has a predictable structure, we can look at which neurons fire on which bits of code and visualize the workings of our RNN.

The code for this chapter can be found in the following Python notebook:

```
05.1 Generating Text in the Style of an Example Text
```

5.1 Acquiring the Text of Public Domain Books

Problem

You want to download the full text of some public domain books to use to train your model.

Solution

Use the Python API for Project Gutenberg.

Project Gutenberg contains the complete texts of over 50,000 books. There is a handy Python API available to browse and download these books. We can download any book if we know the ID:

```
shakespeare = load_etext(100)
shakespeare = strip_headers(shakespeare)
```

We can get a book's ID either by browsing the website and extracting it from the book's URL or by querying *http://www.gutenberg.org/* by author or title. Before we can query, though, we need to populate the metainformation cache. This will create a local database of all books available. It takes a bit of time, but only needs to be done once:

```
cache = get_metadata_cache()
cache.populate()
```

We can now discover all works by Shakespeare:

```
for text_id in get_etexts('author', 'Shakespeare, William'):
    print(text_id, list(get_metadata('title', text_id))[0])
```

Discussion

Project Gutenberg is a volunteer project to digitize books. It focuses on making available the most important books in English that are out of copyright in the United States, though it also has books in other languages. It was started in 1971, long before the invention of the World Wide Web by Michael Hart.

Any work published in the US before 1923 is in the public domain, so most books found in the Gutenberg collection are older than that. This means that the language can be somewhat dated, but for natural language processing the collection remains an unrivalled source of training data. Going through the Python API not only makes access easy but also respects the restrictions that the site puts up for automatic downloading of texts.

5.2 Generating Shakespeare-Like Texts

Problem

How do you generate text in a specific style?

Solution

Use a character-level RNN.

Let's start by acquiring Shakespeare's collected works. We'll drop the poems, so we're left with a more consistent set of just the plays. The poems happen to be collected in the first entry:

```
shakespeare = strip_headers(load_etext(100))
plays = shakespeare.split('\nTHE END\n', 1)[-1]
```

We're going to feed the text in character by character and we'll one-hot encode each character—that is, every character will be encoded as a vector containing all 0s and one 1. For this, we need to know which characters we're going to encounter:

```
chars = list(sorted(set(plays)))
char_to_idx = {ch: idx for idx, ch in enumerate(chars)}
```

Let's create our model that will take a sequence of characters and predict a sequence of characters. We'll feed the sequence into a number of LSTM layers that do the work. The `TimeDistributed` layer lets our model output a sequence again:

```
def char_rnn_model(num_chars, num_layers, num_nodes=512, dropout=0.1):
    input = Input(shape=(None, num_chars), name='input')
    prev = input
    for i in range(num_layers):
        prev = LSTM(num_nodes, return_sequences=True)(prev)
    dense = TimeDistributed(Dense(num_chars, name='dense',
                                  activation='softmax'))(prev)
    model = Model(inputs=[input], outputs=[dense])
    optimizer = RMSprop(lr=0.01)
    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizer, metrics=['accuracy'])
    return model
```

We are going to feed in random fragments from the plays to the network, so a generator seems appropriate. The generator will yield blocks of pairs of sequences, where the sequences of the pairs are just one character apart:

```
def data_generator(all_text, num_chars, batch_size):
    X = np.zeros((batch_size, CHUNK_SIZE, num_chars))
    y = np.zeros((batch_size, CHUNK_SIZE, num_chars))
    while True:
        for row in range(batch_size):
            idx = random.randrange(len(all_text) - CHUNK_SIZE - 1)
            chunk = np.zeros((CHUNK_SIZE + 1, num_chars))
            for i in range(CHUNK_SIZE + 1):
                chunk[i, char_to_idx[all_text[idx + i]]] = 1
            X[row, :, :] = chunk[:CHUNK_SIZE]
            y[row, :, :] = chunk[1:]
        yield X, y
```

Now we'll train the model. We'll set `steps_per_epoch` such that each character should have a decent chance to be seen by the network:

```
model.fit_generator(
    data_generator(plays, len(chars), batch_size=256),
```

```
        epochs=10,
        steps_per_epoch=2 * len(plays) / (256 * CHUNK_SIZE),
        verbose=2
    )
```

After training we can generate some output. We pick a random fragment from the plays and let the model guess what the next character is. We then add the next character to the fragment and repeat until we've reached the required number of characters:

```
def generate_output(model, start_index=None, diversity=1.0, amount=400):
    if start_index is None:
        start_index = random.randint(0, len(plays) - CHUNK_SIZE - 1)
    fragment = plays[start_index: start_index + CHUNK_SIZE]
    generated = fragment
    for i in range(amount):
        x = np.zeros((1, CHUNK_SIZE, len(chars)))
        for t, char in enumerate(fragment):
            x[0, t, char_to_idx[char]] = 1.
        preds = model.predict(x, verbose=0)[0]
        preds = np.asarray(preds[len(generated) - 1])
        next_index = np.argmax(preds)
        next_char = chars[next_index]

        generated += next_char
        fragment = fragment[1:] + next_char
    return generated

for line in generate_output(model).split('\n'):
    print(line)
```

After 10 epochs we should see some text that reminds us of Shakespeare, but we need around 30 for it to start to look like it could fool a casual reader that is not paying too close attention:

```
FOURTH CITIZEN. They were all the summer hearts.
  The King is a virtuous mistress.
CLEOPATRA. I do not know what I have seen him damn'd in no man
  That we have spoken with the season of the world,
  And therefore I will not speak with you.
  I have a son of Greece, and my son
  That we have seen the sea, the seasons of the world
  I will not stay the like offence.

OLIVIA. If it be aught and servants, and something
  have not been a great deal of state)) of the world, I will not stay
  the forest was the fair and not by the way.
SECOND LORD. I will not serve your hour.
FIRST SOLDIER. Here is a princely son, and the world
  in a place where the world is all along.
SECOND LORD. I will not see thee this:
  He hath a heart of men to men may strike and starve.
  I have a son of Greece, whom they say,
  The whiteneth made him like a deadly hand
```

```
And make the seasons of the world,
And then the seasons and a fine hands are parted
To the present winter's parts of this deed.
The manner of the world shall not be a man.
The King hath sent for thee.
The world is slain.
```

It's somewhat suspicious that both Cleopatra and the Second Lord have a son of Greece, but the present winter and the world being slain are appropriately *Game of Thrones*.

Discussion

In this recipe we saw how we can use RNNs to generate text in a certain style. The results are quite convincing, especially given the fact that the model predicts on a character-by-character level. Thanks to the LSTM architecture, the network is capable of learning relationships that span quite large sequences—not just words, but sentences, and even the basic structure of the layout of Shakespeare's plays.

Even though the example shown here isn't very practical, RNNs can be. Any time we want a network to learn a sequence of items, an RNN is probably a good choice.

Other toy apps people have built using this technique have generated baby names, names for paint colors, and even recipes.

More practical RNNs can be used to predict the next character a user is going to type for a smartphone keyboard app, or predict the next move in a chess game when trained on a set of openings. This type of network has also been used to predict sequences like weather patterns or even stock market prices.

Recurrent networks are quite fickle, though. Seemingly small changes to the network architecture can lead to a situation where they no longer converge because of the so-called *exploding gradient problem*. Sometimes during training, after making progress for a number of epochs, the network seems to collapse and starts forgetting what it learns. As always, it is best to start with something simple that works and add complexity step by step, while keeping track of what was changed.

For a slightly more in-depth discussion of RNNs, see Chapter 1.

5.3 Writing Code Using RNNs

Problem

How can you generate Python code using a neural network?

Solution

Train a recurrent neural network over the Python code that comes with the Python distribution that runs your scripts.

We can in fact use pretty much the same model as in the previous recipe for this task. As is often the case with deep learning, the key thing is to get the data. Python ships with the source code of many modules. Since they are stored in the directory where the *random.py* module sits, we can collect them using:

```
def find_python(rootdir):
    matches = []
    for root, dirnames, filenames in os.walk(rootdir):
        for fn in filenames:
            if fn.endswith('.py'):
                matches.append(os.path.join(root, fn))

    return matches
srcs = find_python(random.__file__.rsplit('/', 1)[0])
```

We could then read in all these source files and concatenate them into one document and start generating new snippets, just as we did with the Shakespearean text in the previous recipe. This works reasonably well, but when generating snippets, it becomes clear that a good chunk of Python source code is actually English. English appears both in the form of comments and the contents of strings. We want our model to learn Python, not English!

Stripping out the comments is easy enough:

```
COMMENT_RE = re.compile('#.*')
src = COMMENT_RE.sub('', src)
```

Removing the contents of strings is slightly more involved. Some strings contain useful patterns, rather than English. As a rough rule, we're going to replace any bit of text that has more than six letters and at least one space with `"MSG"`:

```
def replacer(value):
    if ' ' in value and sum(1 for ch in value if ch.isalpha()) > 6:
        return 'MSG'
    return value
```

Finding the occurrences of string literals can be done concisely with a regular expression. Regular expressions are rather slow though, and we're running them over a sizeable amount of code. In this case it's better to just scan the strings:

```
def replace_literals(st):
    res = []
    start_text = start_quote = i = 0
    quote = ''
    while i < len(st):
        if quote:
```

```
            if st[i: i + len(quote)] == quote:
                quote = ''
                start_text = i
                res.append(replacer(st[start_quote: i]))
        elif st[i] in '"\'':
            quote = st[i]
            if i < len(st) - 2 and st[i + 1] == st[i + 2] == quote:
                quote = 3 * quote
            start_quote = i + len(quote)
            res.append(st[start_text: start_quote])
        if st[i] == '\n' and len(quote) == 1:
            start_text = i
            res.append(quote)
            quote = ''
        if st[i] == '\\':
            i += 1
        i += 1
    return ''.join(res) + st[start_text:]
```

Even cleaned up this way, we end up with megabytes of pure Python code. We can now train the model as before, but on Python code rather than on plays. After 30 epochs or so, we should have something workable and can generate code.

Discussion

Generating Python code is no different from writing a Shakespearean-style play—at least for a neural network. We've seen that cleaning up the input data is an important aspect of data processing for neural networks. In this case we made sure to remove most traces of English from the source code. This way the network can focus on learning Python and not be distracted by also having to allocate neurons to learning English.

We could further regularize the input. For example, we could pipe all the source code first through a "pretty printer" so that it would all have the same layout and our network could focus on learning that, rather than the diversity found in the current code. One step further would be to tokenize the Python code using the built-in tokenizer, and then let the network learn this parsed version and use `untokenize` to generate the code.

5.4 Controlling the Temperature of the Output

Problem

You want to control the variability of the generated code.

Solution

Use the predictions as a probability distribution, rather than picking the highest value.

In the Shakespeare example, we picked the character in the predictions that had the highest score. This approach results in the output that is the best liked by the model. The drawback is that we get the same output for every start. Since we picked a random start sequence from the actual Shakespearean texts that didn't matter much. But if we want to generate Python functions, it would be nice to always start in the same way—let's say with `/ndef`—and look at various solutions.

The predictions of our network are the result of a softmax activation function and can therefore be seen as a probability distribution. So, rather than picking the maximum value, we can let `numpy.random.multinomial` give us an answer. `multinomial` runs n experiments and takes the probability of how likely the outcomes are. By running it with $n = 1$, we get what we want.

At this point we can introduce the notion of temperature in how we draw the outcomes. The idea is that the higher the temperature is, the more random the outcomes are, while lower temperatures are closer to the pure deterministic outcomes we saw earlier. We do this by scaling the logs of the predictions accordingly and then applying the softmax function again to get back to probabilities. Putting this all together we get:

```
def generate_code(model, start_with='\ndef ',
                  end_with='\n\n', diversity=1.0):
    generated = start_with
    yield generated
    for i in range(2000):
        x = np.zeros((1, len(generated), len(chars)))
        for t, char in enumerate(generated):
            x[0, t, char_to_idx[char]] = 1.
        preds = model.predict(x, verbose=0)[0]

        preds = np.asarray(preds[len(generated) - 1]).astype('float64')
        preds = np.log(preds) / diversity
        exp_preds = np.exp(preds)
        preds = exp_preds / np.sum(exp_preds)
        probas = np.random.multinomial(1, preds, 1)
        next_index = np.argmax(probas)
        next_char = chars[next_index]
        yield next_char

        generated += next_char
        if generated.endswith(end_with):
            break
```

We're finally ready to have some fun. At `diversity=1.0` the following code is produced. Note how the model generated our `"MSG"` placeholder and, apart from confusing `val` and `value`, almost got us running code:

```
    def _calculate_ratio(val):
        """MSG"""
        if value and value[0] != '0':
            raise errors.HeaderParseError(
                "MSG".format(Storable))
        return value
```

Discussion

Using the output of the softmax activation function as a probability distribution allows us to get a variety of results that correspond to what the model "intends." An added bonus is that it allows us to introduce the notion of temperature, so we can control how "random" the output is. In Chapter 13 we'll look at how *variational autoencoders* use a similar technique to control the randomness of what is generated.

The generated Python code can certainly pass for the real thing if we don't pay attention to the details. One way to improve the results further would be to call the `compile` function on the generated code and only keep code that compiles. That way we can make sure that it is at least syntactically correct. A slight variation of that approach would be to not start over on a syntax error, but just drop the line where the error occurs and everything that follows and try again.

5.5 Visualizing Recurrent Network Activations

Problem

How can you gain insight into what a recurrent network is doing?

Solution

Extract the activations from the neurons while they process text. Since we're going to visualize the neurons, it makes sense to reduce their number. This will degrade the performance of the model a bit, but makes things simpler:

```
flat_model = char_rnn_model(len(py_chars), num_layers=1, num_nodes=512)
```

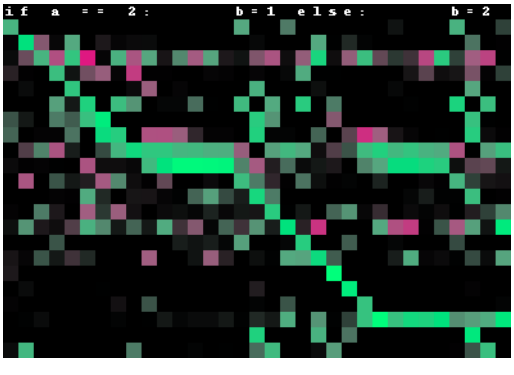
This model is a bit simpler and gets us slightly less accurate results, but it is good enough for visualizations. Keras has a handy method called `function` that allows us to specify an input and an output layer and will then run whatever part of the network is needed to convert from one to the other. The following method provides the network with a bit of text (a sequence of characters) and gets the activations for a specific layer back:

```
def activations(model, code):
    x = np.zeros((1, len(code), len(py_char_to_idx)))
    for t, char in enumerate(code):
        x[0, t, py_char_to_idx[char]] = 1.
    output = model.get_layer('lstm_3').output
    f = K.function([model.input], [output])
    return f([x])[0][0]
```

Now the question is which neurons to look at. Even our simplified model has 512 neurons. Activations in an LSTM are between –1 and 1, so a simple way to find interesting neurons is to just pick the highest value corresponding to each character. `np.argmax(act, axis=1)` will get us that. We can visualize those neurons using:

```
img = np.full((len(neurons) + 1, len(code), 3), 128)
scores = (act[:, neurons].T + 1) / 2
img[1:, :, 0] = 255 * (1 - scores)
img[1:, :, 1] = 255 * scores
```

This will produce a small bitmap. After we enlarge the bitmap and plot the code on top, we get:



This looks interesting. The top neuron seems to keep track of where new statements start. The one with the green bars keeps track of spaces, but only in as far as they are used for indentation. The last-but-one neuron seems to fire when there is an = sign, but not when there is a ==, suggesting the network learned the difference between assignment and equality.

Discussion

Deep learning models can be very effective, but their results are notoriously hard to explain. We more or less understand the mechanics of the training and inference, but it is often difficult to explain a concrete result, other than pointing to the actual calculations. Visualizing activations is one way of making what the network learned a little clearer.

Looking at the neurons with the highest activation for each character quickly gets us a set of neurons that might be of interest. Alternatively, we could explicitly try to look for neurons that fire in specific circumstances, for example inside brackets.

Once we have a specific neuron that looks interesting, we can use the same coloring technique to highlight larger chunks of code.

CHAPTER 6

Question Matching

We've now seen a few examples of how we can construct and use word embeddings to compare terms with one another. It's natural to ask how we can extend this idea to larger blocks of text. Can we create semantic embeddings of entire sentences or paragraphs? In this chapter, we'll try to do just that: we're going to use data from Stack Exchange to build embeddings for entire questions; we can then use those embeddings to find similar documents or questions.

We'll start out by downloading and parsing our training data from the Internet Archive. Then we'll briefly explore how Pandas can be helpful for analyzing data. We let Keras do the heavy lifting when it comes to featurizing our data and building a model for the task at hand. We then look into how to feed this model from a Pandas `DataFrame` and how we can run it to draw conclusions.

The code for this chapter can be found in the following notebook:

```
06.1 Question matching
```

6.1 Acquiring Data from Stack Exchange

Problem

You need to access a large set of questions to kick-start your training.

Solution

Use the Internet Archive to retrieve a dump of questions.

A Stack Exchange data dump is freely available on the Internet Archive, which hosts a number of interesting datasets (as well as striving to provide an archive of the entire

web). The data is laid out with one ZIP file for each area on Stack Exchange (e.g., travel, sci-fi, etc.). Let's download the file for the travel section:

```
xml_7z = utils.get_file(
    fname='travel.stackexchange.com.7z',
    origin=('https://ia800107.us.archive.org/27/'
            'items/stackexchange/travel.stackexchange.com.7z'),
)
```

While the input is technically an XML file, the structure is simple enough that we can get away with just reading individual lines and splitting out the fields. This is a bit brittle, of course. We will limit ourselves to processing 1 million records from the dataset; this keeps our memory usage from blowing up and should be enough data for us to work with. We'll save the processed data as a JSON file so we won't have to do the processing again the next time around:

```
def extract_stackexchange(filename, limit=1000000):
    json_file = filename + 'limit=%s.json' % limit

    rows = []
    for i, line in enumerate(os.popen('7z x -so "%s" Posts.xml'
                                      % filename)):
        line = str(line)
        if not line.startswith('  <row'):
            continue

        if i % 1000 == 0:
            print('\r%05d/%05d' % (i, limit), end='', flush=True)

        parts = line[6:-5].split('"')
        record = {}
        for i in range(0, len(parts), 2):
            k = parts[i].replace('=', '').strip()
            v = parts[i+1].strip()
            record[k] = v
        rows.append(record)

        if len(rows) > limit:
            break

    with open(json_file, 'w') as fout:
        json.dump(rows, fout)

    return rows


rows = download_stackexchange()
```

Discussion

The Stack Exchange datasets is a great source for question/answer pairs that comes with a nice reuse license. As long as you give attribution you can use it in pretty much any way you want. Converting the zipped XML into the more easily consumable JSON format is a good preprocessing step.

6.2 Exploring Data Using Pandas

Problem

How do you quickly explore a large dataset so you can make sure it contains what you expect?

Solution

Use Python's Pandas.

Pandas is a powerful framework for data processing in Python. In some ways it is comparable to a spreadsheet; the data is stored in rows and columns and we can quickly filter, convert, and aggregate on the records. Let's start by converting our rows of Python dictionaries into a `DataFrame`. Pandas tries to "guess" the types of some columns. We'll coerce the columns we care about into the right format:

```
df = pd.DataFrame.from_records(rows)
df = df.set_index('Id', drop=False)
df['Title'] = df['Title'].fillna('').astype('str')
df['Tags'] = df['Tags'].fillna('').astype('str')
df['Body'] = df['Body'].fillna('').astype('str')
df['Id'] = df['Id'].astype('int')
df['PostTypeId'] = df['PostTypeId'].astype('int')
```

With `df.head` we can now see what's going on in our database.

We can also use Pandas to take a quick look at popular questions in our data:

```
list(df[df['ViewCount'] > 2500000]['Title'])
```

```
['How to horizontally center a &lt;div&gt; in another &lt;div&gt;?',
 'What is the best comment in source code you have ever encountered?',
 'How do I generate random integers within a specific range in Java?',
 'How to redirect to another webpage in JavaScript/jQuery?',
 'How can I get query string values in JavaScript?',
 'How to check whether a checkbox is checked in jQuery?',
 'How do I undo the last commit(s) in Git?',
 'Iterate through a HashMap',
 'Get selected value in dropdown list using JavaScript?',
 'How do I declare and initialize an array in Java?']
```

As you might expect, the most popular questions are general questions about frequently used languages.

Discussion

Pandas is a great tool for many types of data analysis, whether you just want to have a casual look at the data or you want to do in-depth analysis. It can be tempting to try to leverage Pandas for many tasks, but unfortunately the Pandas interface is not at all regular and for complex operations the performance can be significantly worse than using a real database. Lookups in Pandas are significantly more expensive than using a Python dictionary, so be careful!

6.3 Using Keras to Featurize Text

Problem

How do you quickly create feature vectors from text?

Solution

Use the `Tokenizer` class from Keras.

Before we can feed text into a model, we need to convert it into feature vectors. A common way to do this is to assign an integer to each of the top *N* words in a text and then replace each word by its integer. Keras makes this really straightforward:

```
from keras.preprocessing.text import Tokenizer
VOCAB_SIZE = 50000

tokenizer = Tokenizer(num_words=VOCAB_SIZE)
tokenizer.fit_on_texts(df['Body'] + ' ' + df['Title'])
```

Now let's tokenize the titles and bodies of our whole dataset:

```
df['title_tokens'] = tokenizer.texts_to_sequences(df['Title'])
df['body_tokens'] = tokenizer.texts_to_sequences(df['Body'])
```

Discussion

Converting text to a series of numbers by using a tokenizer is one of the classic ways of making text consumable by a neural network. In the previous chapter we converted text on a per-character basis. Character-based models take as input individual characters (removing the need for a tokenizer). The trade-off is in how long it takes to train the model: because you're forcing the model to learn how to tokenize and stem words, you need more training data and more time.

One of the drawbacks of processing texts on a per-word basis is the fact that there is no practical upper limit to the number of different words that can appear in the texts, especially if we have to handle typos and errors. In this recipe we only pay attention to words that appear in the top 50,000 by count, which is one way around this problem.

6.4 Building a Question/Answer Model

Problem

How do you calculate embeddings for questions?

Solution

Train a model to predict whether a question and an answer from the Stack Exchange dataset match.

Whenever we construct a model, the first question we should ask is: "What is our objective?" That is, what is the model going to try to classify?

Ideally we'd have a list of "similar questions to this one," which we could use to train our model. Unfortunately, it would be very expensive to acquire such a dataset! Instead, we'll rely on a surrogate objective: let's see if we can train our model to, given a question, distinguish between the matching answer and an answer from a random question. This will force the model to learn a good representation of titles and bodies.

We start off our model by defining our inputs. In this case we have two inputs, the title (question) and body (answer):

```
title = layers.Input(shape=(None,), dtype='int32', name='title')
body = layers.Input(shape=(None,), dtype='int32', name='body')
```

Both are of varying length, so we have to pad them. The data for each field will be a list of integers, one for each word in the title or the body.

Now we want to define a shared set of layers that both inputs will be passed through. We're first going to construct an embedding for the inputs, then mask out the invalid values, and add all of the words' values together:

```
 embedding = layers.Embedding(
        mask_zero=True,
        input_dim=vocab_size,
        output_dim=embedding_size
    )

mask = layers.Masking(mask_value=0)
def _combine_sum(v):
    return K.sum(v, axis=2)
```

```
sum_layer = layers.Lambda(_combine_sum)
```

Here, we've specified a `vocab_size` (how many words are in our vocabulary) and an `embedding_size` (how wide our embedding of each word should be; the GoogleNews vectors are 300 dimensions, for example).

Now let's apply these layers to our word inputs:

```
title_sum = sum_layer(mask(embedding(title)))
body_sum = sum_layer(mask(embedding(body)))
```

Now that we have a single vector for our title and body, we can compare them to each other with a cosine distance, just like we did in Recipe 4.2. In Keras, that is expressed via the `dot` layer:

```
sim = layers.dot([title_sum, word_sum], normalize=True, axes=1)
```

Finally, we can define our model. It takes the title and the body in and outputs the similarity between the two:

```
sim_model = models.Model(inputs=[title,body], outputs=[sim])
sim_model.compile(loss='mse', optimizer='rmsprop')
```

Discussion

The model we've built here learns to match questions and answers but really the only freedom we give it is to change the embeddings of the words such that the sums of the embeddings of the title and the body match. This should get us embeddings for questions such that questions that are similar will have similar embeddings, because similar questions will have similar answers.

Our training model is compiled with two parameters telling Keras how to improve the model:

The loss function
: This tells the system how "wrong" a given answer is. For example, if we told the network that `title_a` and `body_a` should output 1.0, but the network predicts 0.8, how bad of an error is that? This becomes a more complex problem when we have multiple outputs, but we'll cover that later. For this model, we're going to use *mean squared error*. For the previous example, this means we would penalize the model by (1.0–0.8) ** 2, or 0.04. This loss will be propagated back through the model and improve the embeddings each time the model sees an example.

The optimizer
: There are many ways that loss can be used to improve our model. These are called *optimization strategies*, or *optimizers*. Fortunately, Keras comes with a number of reliable optimizers built in, so we won't have to worry much about

this: we can just pick a suitable one. In this case, we're using the `rmsprop` optimizer, which tends to perform very well across a wide range of problems.

6.5 Training a Model with Pandas

Problem

How do you train a model on data contained in Pandas?

Solution

Build a data generator that leverages the filter and sample features of Pandas.

As in the previous recipe, we are going to train our model to distinguish between a question title and the correct answer (body) versus the answer to another random question. We can write that out as a generator that iterates over our dataset. It will output a 1 for the correct question title and body and a 0 for a random title and body:

```
def data_generator(batch_size, negative_samples=1):
    questions = df[df['PostTypeId'] == 1]
    all_q_ids = list(questions.index)

    batch_x_a = []
    batch_x_b = []
    batch_y = []

    def _add(x_a, x_b, y):
        batch_x_a.append(x_a[:MAX_DOC_LEN])
        batch_x_b.append(x_b[:MAX_DOC_LEN])
        batch_y.append(y)

    while True:
        questions = questions.sample(frac=1.0)

        for i, q in questions.iterrows():
            _add(q['title_tokens'], q['body_tokens'], 1)

            negative_q = random.sample(all_q_ids, negative_samples)
            for nq_id in negative_q:
                _add(q['title_tokens'],
                     df.at[nq_id, 'body_tokens'], 0)

            if len(batch_y) >= batch_size:
                yield ({
                    'title': pad_sequences(batch_x_a, maxlen=None),
                    'body': pad_sequences(batch_x_b, maxlen=None),
                }, np.asarray(batch_y))

                batch_x_a = []
```

```
            batch_x_b = []
            batch_y = []
```

The only complication here is the batching of the data. This is not strictly necessary, but extremely important for performance. All deep learning models are optimized to work on chunks of data at a time. The best batch size to use depends on the problem you're working on. Using larger batches means your model sees more data for each update and therefore can more accurately update its weights, but on the flip side it can't update as often. Bigger batch sizes also take more memory. It's best to start small and keep doubling the batch size until the results no longer improve.

Now let's train the model:

```
sim_model.fit_generator(
    data_generator(batch_size=128),
    epochs=10,
    steps_per_epoch=1000
)
```

We'll train it for 10,000 steps, divided into 10 epochs of 1,000 steps each. Each step will process 128 documents, so our network will end up seeing 1.28M training examples. If you have a GPU, you'll be surprised how quickly this runs!

6.6 Checking Similarities

Problem

You'd like to use Keras to predict values by using the weights of another network.

Solution

Construct a second model that uses different input and output layers from the original network, but shares some of the other layers.

Our `sim_model` has been trained and as part of that learned how to go from a title to a `title_sum`, which is really what we are after. The model that just does that is:

```
embedding_model = models.Model(inputs=[title], outputs=[title_sum])
```

We can now use the "embedding" model to compute a representation for each question in our dataset. Let's wrap this up in a class for easy reuse:

```
questions = df[df['PostTypeId'] == 1]['Title'].reset_index(drop=True)
question_tokens = pad_sequences(tokenizer.texts_to_sequences(questions))

class EmbeddingWrapper(object):
    def __init__(self, model):
        self._questions = questions
        self._idx_to_question = {i:s for (i, s) in enumerate(questions)}
        self._weights = model.predict({'title': question_tokens},
```

```
                                        verbose=1, batch_size=1024)
        self._model = model
        self._norm = np.sqrt(np.sum(self._weights * self._weights
                                    + 1e-5, axis=1))

    def nearest(self, question, n=10):
        tokens = tokenizer.texts_to_sequences([sentence])
        q_embedding = self._model.predict(np.asarray(tokens))[0]
        q_norm= np.sqrt(np.dot(q_embedding, q_embedding))
        dist = np.dot(self._weights, q_embedding) / (q_norm * self._norm)

        top_idx = np.argsort(dist)[-n:]
        return pd.DataFrame.from_records([
            {'question': self._r[i], 'similarity': float(dist[i])}
            for i in top_idx
        ])
```

And now we can use it:

```
lookup = EmbeddingWrapper(model=sum_embedding_trained)
lookup.nearest('Python Postgres object relational model')
```

This produces the following results:

Similarity	Question
0.892392	working with django and sqlalchemy but backend…
0.893417	Python ORM that auto-generates/updates tables …
0.893883	Dynamic Table Creation and ORM mapping in SqlA…
0.896096	SQLAlchemy with count, group_by and order_by u…
0.897706	SQLAlchemy: Scan huge tables using ORM?
0.902693	Efficiently updating database using SQLAlchemy…
0.911446	What are some good Python ORM solutions?
0.922449	python orm
0.924316	Python libraries to construct classes from a r…
0.930865	python ORM allowing for table creation and bul…

In a very short training time, our network managed to figure out that "SQL," "query," and "INSERT" are all related to Postgres!

Discussion

In this recipe we saw how we can use part of a network to predict the values we're after, even if the overall network was trained to predict something else. The functional API of Keras provides a nice separation between the layers, how they are connected, and which combination of input and output layers forms a model.

As we'll see later in this book, this gives us a lot of flexibility. We can take a pre-trained network and use one of the middle layers as an output layer, or we can take one of those middle layers and add some new layers (see Chapter 9). We can even run the network backwards (see Chapter 12).