CHAPTER 21

Natural Language Processing

They have been at a great feast of languages, and stolen the scraps.
—William Shakespeare

Natural language processing (NLP) refers to computational techniques involving language. It's a broad field, but we'll look at a few techniques, both simple and not simple.

Word Clouds

In Chapter 1, we computed word counts of users' interests. One approach to visualizing words and counts is *word clouds*, which artistically depict the words at sizes proportional to their counts.

Generally, though, data scientists don't think much of word clouds, in large part because the placement of the words doesn't mean anything other than "here's some space where I was able to fit a word."

If you ever are forced to create a word cloud, think about whether you can make the axes convey something. For example, imagine that, for each of some collection of data science–related buzzwords, you have two numbers between 0 and 100—the first representing how frequently it appears in job postings, and the second how frequently it appears on résumés:

```
data = [ ("big data", 100, 15), ("Hadoop", 95, 25), ("Python", 75, 50),
         ("R", 50, 40), ("machine learning", 80, 20), ("statistics", 20, 60),
         ("data science", 60, 70), ("analytics", 90, 3),
         ("team player", 85, 85), ("dynamic", 2, 90), ("synergies", 70, 0),
         ("actionable insights", 40, 30), ("think out of the box", 45, 10),
         ("self-starter", 30, 50), ("customer focus", 65, 15),
         ("thought leadership", 35, 35)]
```

The word cloud approach is just to arrange the words on a page in a cool-looking font (Figure 21-1).

Figure 21-1. Buzzword cloud

This looks neat but doesn't really tell us anything. A more interesting approach might be to scatter them so that horizontal position indicates posting popularity and vertical position indicates résumé popularity, which produces a visualization that conveys a few insights (Figure 21-2):

```
from matplotlib import pyplot as plt

def text_size(total: int) -> float:
    """equals 8 if total is 0, 28 if total is 200"""
    return 8 + total / 200 * 20

for word, job_popularity, resume_popularity in data:
    plt.text(job_popularity, resume_popularity, word,
             ha='center', va='center',
             size=text_size(job_popularity + resume_popularity))
plt.xlabel("Popularity on Job Postings")
plt.ylabel("Popularity on Resumes")
plt.axis([0, 100, 0, 100])
plt.xticks([])
plt.yticks([])
plt.show()
```

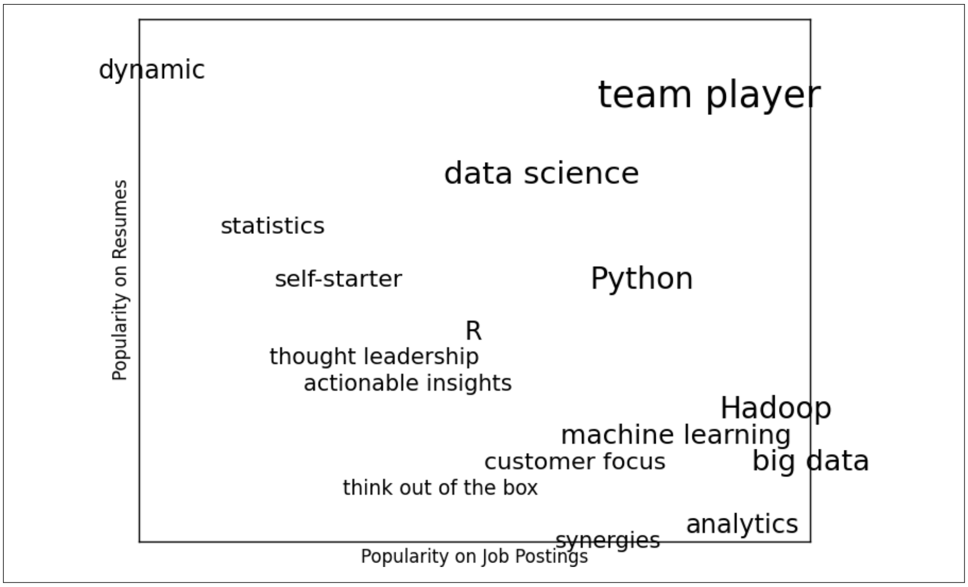

Figure 21-2. A more meaningful (if less attractive) word cloud

n-Gram Language Models

The DataSciencester VP of Search Engine Marketing wants to create thousands of web pages about data science so that your site will rank higher in search results for data science–related terms. (You attempt to explain to her that search engine algorithms are clever enough that this won't actually work, but she refuses to listen.)

Of course, she doesn't want to write thousands of web pages, nor does she want to pay a horde of "content strategists" to do so. Instead, she asks you whether you can somehow programmatically generate these web pages. To do this, we'll need some way of modeling language.

One approach is to start with a corpus of documents and learn a statistical model of language. In our case, we'll start with Mike Loukides's essay "What Is Data Science?"

As in Chapter 9, we'll use the Requests and Beautiful Soup libraries to retrieve the data. There are a couple of issues worth calling attention to.

The first is that the apostrophes in the text are actually the Unicode character `u"\u2019"`. We'll create a helper function to replace them with normal apostrophes:

```
def fix_unicode(text: str) -> str:
    return text.replace(u"\u2019", "'")
```

The second issue is that once we get the text of the web page, we'll want to split it into a sequence of words and periods (so that we can tell where sentences end). We can do this using `re.findall`:

```
import re
from bs4 import BeautifulSoup
import requests

url = "https://www.oreilly.com/ideas/what-is-data-science"
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')

content = soup.find("div", "article-body")   # find article-body div
regex = r"[\w']+|[\.]"                       # matches a word or a period

document = []

for paragraph in content("p"):
    words = re.findall(regex, fix_unicode(paragraph.text))
    document.extend(words)
```

We certainly could (and likely should) clean this data further. There is still some amount of extraneous text in the document (for example, the first word is *Section*), and we've split on midsentence periods (for example, in *Web 2.0*), and there are a handful of captions and lists sprinkled throughout. Having said that, we'll work with the document as it is.

Now that we have the text as a sequence of words, we can model language in the following way: given some starting word (say, *book*) we look at all the words that follow it in the source document. We randomly choose one of these to be the next word, and we repeat the process until we get to a period, which signifies the end of the sentence. We call this a *bigram model*, as it is determined completely by the frequencies of the bigrams (word pairs) in the original data.

What about a starting word? We can just pick randomly from words that *follow* a period. To start, let's precompute the possible word transitions. Recall that `zip` stops when any of its inputs is done, so that `zip(document, document[1:])` gives us precisely the pairs of consecutive elements of `document`:

```
from collections import defaultdict

transitions = defaultdict(list)
for prev, current in zip(document, document[1:]):
    transitions[prev].append(current)
```

Now we're ready to generate sentences:

```
def generate_using_bigrams() -> str:
    current = "."   # this means the next word will start a sentence
    result = []
    while True:
        next_word_candidates = transitions[current]    # bigrams (current, _)
        current = random.choice(next_word_candidates)  # choose one at random
        result.append(current)                         # append it to results
        if current == ".": return " ".join(result)     # if "." we're done
```

The sentences it produces are gibberish, but they're the kind of gibberish you might put on your website if you were trying to sound data-sciencey. For example:

> If you may know which are you want to data sort the data feeds web friend someone on trending topics as the data in Hadoop is the data science requires a book demonstrates why visualizations are but we do massive correlations across many commercial disk drives in Python language and creates more tractable form making connections then use and uses it to solve a data.
>
> —Bigram Model

We can make the sentences less gibberishy by looking at *trigrams*, triplets of consecutive words. (More generally, you might look at *n-grams* consisting of *n* consecutive words, but three will be plenty for us.) Now the transitions will depend on the previous *two* words:

```
trigram_transitions = defaultdict(list)
starts = []

for prev, current, next in zip(document, document[1:], document[2:]):

    if prev == ".":              # if the previous "word" was a period
        starts.append(current)   # then this is a start word

    trigram_transitions[(prev, current)].append(next)
```

Notice that now we have to track the starting words separately. We can generate sentences in pretty much the same way:

```
def generate_using_trigrams() -> str:
    current = random.choice(starts)   # choose a random starting word
    prev = "."                        # and precede it with a '.'
    result = [current]
    while True:
        next_word_candidates = trigram_transitions[(prev, current)]
        next_word = random.choice(next_word_candidates)

        prev, current = current, next_word
        result.append(current)

        if current == ".":
            return " ".join(result)
```

This produces better sentences like:

> In hindsight MapReduce seems like an epidemic and if so does that give us new insights into how economies work That's not a question we could even have asked a few years there has been instrumented.

—Trigram Model

Of course, they sound better because at each step the generation process has fewer choices, and at many steps only a single choice. This means that we frequently generate sentences (or at least long phrases) that were seen verbatim in the original data. Having more data would help; it would also work better if we collected *n*-grams from multiple essays about data science.

Grammars

A different approach to modeling language is with *grammars*, rules for generating acceptable sentences. In elementary school, you probably learned about parts of speech and how to combine them. For example, if you had a really bad English teacher, you might say that a sentence necessarily consists of a *noun* followed by a *verb*. If you then have a list of nouns and verbs, you can generate sentences according to the rule.

We'll define a slightly more complicated grammar:

```
from typing import List, Dict

# Type alias to refer to grammars later
Grammar = Dict[str, List[str]]

grammar = {
    "_S"  : ["_NP _VP"],
    "_NP" : ["_N",
             "_A _NP _P _A _N"],
    "_VP" : ["_V",
             "_V _NP"],
    "_N"  : ["data science", "Python", "regression"],
    "_A"  : ["big", "linear", "logistic"],
    "_P"  : ["about", "near"],
    "_V"  : ["learns", "trains", "tests", "is"]
}
```

I made up the convention that names starting with underscores refer to *rules* that need further expanding, and that other names are *terminals* that don't need further processing.

So, for example, `"_S"` is the "sentence" rule, which produces an `"_NP"` ("noun phrase") rule followed by a `"_VP"` ("verb phrase") rule.

The verb phrase rule can produce either the `"_V"` ("verb") rule, or the verb rule followed by the noun phrase rule.

Notice that the `"_NP"` rule contains itself in one of its productions. Grammars can be recursive, which allows even finite grammars like this to generate infinitely many different sentences.

How do we generate sentences from this grammar? We'll start with a list containing the sentence rule `["_S"]`. And then we'll repeatedly expand each rule by replacing it with a randomly chosen one of its productions. We stop when we have a list consisting solely of terminals.

For example, one such progression might look like:

```
['_S']
['_NP','_VP']
['_N','_VP']
['Python','_VP']
['Python','_V','_NP']
['Python','trains','_NP']
['Python','trains','_A','_NP','_P','_A','_N']
['Python','trains','logistic','_NP','_P','_A','_N']
['Python','trains','logistic','_N','_P','_A','_N']
['Python','trains','logistic','data science','_P','_A','_N']
['Python','trains','logistic','data science','about','_A', '_N']
['Python','trains','logistic','data science','about','logistic','_N']
['Python','trains','logistic','data science','about','logistic','Python']
```

How do we implement this? Well, to start, we'll create a simple helper function to identify terminals:

```
def is_terminal(token: str) -> bool:
    return token[0] != "_"
```

Next we need to write a function to turn a list of tokens into a sentence. We'll look for the first nonterminal token. If we can't find one, that means we have a completed sentence and we're done.

If we do find a nonterminal, then we randomly choose one of its productions. If that production is a terminal (i.e., a word), we simply replace the token with it. Otherwise, it's a sequence of space-separated nonterminal tokens that we need to `split` and then splice into the current tokens. Either way, we repeat the process on the new set of tokens.

Putting it all together, we get:

```
def expand(grammar: Grammar, tokens: List[str]) -> List[str]:
    for i, token in enumerate(tokens):
        # If this is a terminal token, skip it.
        if is_terminal(token): continue

        # Otherwise, it's a nonterminal token,
        # so we need to choose a replacement at random.
        replacement = random.choice(grammar[token])

        if is_terminal(replacement):
            tokens[i] = replacement
        else:
            # Replacement could be, e.g., "_NP _VP", so we need to
```

```
                # split it on spaces and splice it in.
                tokens = tokens[:i] + replacement.split() + tokens[(i+1):]

            # Now call expand on the new list of tokens.
            return expand(grammar, tokens)

        # If we get here, we had all terminals and are done.
        return tokens
```

And now we can start generating sentences:

```
    def generate_sentence(grammar: Grammar) -> List[str]:
        return expand(grammar, ["_S"])
```

Try changing the grammar—add more words, add more rules, add your own parts of speech—until you're ready to generate as many web pages as your company needs.

Grammars are actually more interesting when they're used in the other direction. Given a sentence, we can use a grammar to *parse* the sentence. This then allows us to identify subjects and verbs and helps us make sense of the sentence.

Using data science to generate text is a neat trick; using it to *understand* text is more magical. (See "For Further Exploration" on page 301 for libraries that you could use for this.)

An Aside: Gibbs Sampling

Generating samples from some distributions is easy. We can get uniform random variables with:

```
    random.random()
```

and normal random variables with:

```
    inverse_normal_cdf(random.random())
```

But some distributions are harder to sample from. *Gibbs sampling* is a technique for generating samples from multidimensional distributions when we only know some of the conditional distributions.

For example, imagine rolling two dice. Let *x* be the value of the first die and *y* be the sum of the dice, and imagine you wanted to generate lots of (*x*, *y*) pairs. In this case it's easy to generate the samples directly:

```
    from typing import Tuple
    import random

    def roll_a_die() -> int:
        return random.choice([1, 2, 3, 4, 5, 6])

    def direct_sample() -> Tuple[int, int]:
        d1 = roll_a_die()
        d2 = roll_a_die()
        return d1, d1 + d2
```

But imagine that you only knew the conditional distributions. The distribution of *y* conditional on *x* is easy—if you know the value of *x*, *y* is equally likely to be $x + 1$, $x + 2$, $x + 3$, $x + 4$, $x + 5$, or $x + 6$:

```
def random_y_given_x(x: int) -> int:
    """equally likely to be x + 1, x + 2, ... , x + 6"""
    return x + roll_a_die()
```

The other direction is more complicated. For example, if you know that *y* is 2, then necessarily *x* is 1 (since the only way two dice can sum to 2 is if both of them are 1). If you know *y* is 3, then *x* is equally likely to be 1 or 2. Similarly, if *y* is 11, then *x* has to be either 5 or 6:

```
def random_x_given_y(y: int) -> int:
    if y <= 7:
        # if the total is 7 or less, the first die is equally likely to be
        # 1, 2, ..., (total - 1)
        return random.randrange(1, y)
    else:
        # if the total is 7 or more, the first die is equally likely to be
        # (total - 6), (total - 5), ..., 6
        return random.randrange(y - 6, 7)
```

The way Gibbs sampling works is that we start with any (valid) values for *x* and *y* and then repeatedly alternate replacing *x* with a random value picked conditional on *y* and replacing *y* with a random value picked conditional on *x*. After a number of iterations, the resulting values of *x* and *y* will represent a sample from the unconditional joint distribution:

```
def gibbs_sample(num_iters: int = 100) -> Tuple[int, int]:
    x, y = 1, 2 # doesn't really matter
    for _ in range(num_iters):
        x = random_x_given_y(y)
        y = random_y_given_x(x)
    return x, y
```

You can check that this gives similar results to the direct sample:

```
def compare_distributions(num_samples: int = 1000) -> Dict[int, List[int]]:
    counts = defaultdict(lambda: [0, 0])
    for _ in range(num_samples):
        counts[gibbs_sample()][0] += 1
        counts[direct_sample()][1] += 1
    return counts
```

We'll use this technique in the next section.

Topic Modeling

When we built our "Data Scientists You May Know" recommender in Chapter 1, we simply looked for exact matches in people's stated interests.

A more sophisticated approach to understanding our users' interests might try to identify the *topics* that underlie those interests. A technique called *latent Dirichlet allocation* (LDA) is commonly used to identify common topics in a set of documents. We'll apply it to documents that consist of each user's interests.

LDA has some similarities to the Naive Bayes classifier we built in Chapter 13, in that it assumes a probabilistic model for documents. We'll gloss over the hairier mathematical details, but for our purposes the model assumes that:

- There is some fixed number K of topics.
- There is a random variable that assigns each topic an associated probability distribution over words. You should think of this distribution as the probability of seeing word w given topic k.
- There is another random variable that assigns each document a probability distribution over topics. You should think of this distribution as the mixture of topics in document d.
- Each word in a document was generated by first randomly picking a topic (from the document's distribution of topics) and then randomly picking a word (from the topic's distribution of words).

In particular, we have a collection of `documents`, each of which is a `list` of words. And we have a corresponding collection of `document_topics` that assigns a topic (here a number between 0 and $K - 1$) to each word in each document.

So, the fifth word in the fourth document is:

```
documents[3][4]
```

and the topic from which that word was chosen is:

```
document_topics[3][4]
```

This very explicitly defines each document's distribution over topics, and it implicitly defines each topic's distribution over words.

We can estimate the likelihood that topic 1 produces a certain word by comparing how many times topic 1 produces that word with how many times topic 1 produces *any* word. (Similarly, when we built a spam filter in Chapter 13, we compared how many times each word appeared in spams with the total number of words appearing in spams.)

Although these topics are just numbers, we can give them descriptive names by looking at the words on which they put the heaviest weight. We just have to somehow generate the `document_topics`. This is where Gibbs sampling comes into play.

We start by assigning every word in every document a topic completely at random. Now we go through each document one word at a time. For that word and document, we construct weights for each topic that depend on the (current) distribution of topics in that document and the (current) distribution of words for that topic. We then use those weights to sample a new topic for that word. If we iterate this process many times, we will end up with a joint sample from the topic–word distribution and the document–topic distribution.

To start with, we'll need a function to randomly choose an index based on an arbitrary set of weights:

```python
def sample_from(weights: List[float]) -> int:
    """returns i with probability weights[i] / sum(weights)"""
    total = sum(weights)
    rnd = total * random.random()      # uniform between 0 and total
    for i, w in enumerate(weights):
        rnd -= w                       # return the smallest i such that
        if rnd <= 0: return i          # weights[0] + ... + weights[i] >= rnd
```

For instance, if you give it weights `[1, 1, 3]`, then one-fifth of the time it will return 0, one-fifth of the time it will return 1, and three-fifths of the time it will return 2. Let's write a test:

```python
from collections import Counter

# Draw 1000 times and count
draws = Counter(sample_from([0.1, 0.1, 0.8]) for _ in range(1000))
assert 10 < draws[0] < 190   # should be ~10%, this is a really loose test
assert 10 < draws[1] < 190   # should be ~10%, this is a really loose test
assert 650 < draws[2] < 950  # should be ~80%, this is a really loose test
assert draws[0] + draws[1] + draws[2] == 1000
```

Our documents are our users' interests, which look like:

```python
documents = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

And we'll try to find:

```
K = 4
```

topics. In order to calculate the sampling weights, we'll need to keep track of several counts. Let's first create the data structures for them.

- How many times each topic is assigned to each document:

  ```
  # a list of Counters, one for each document
  document_topic_counts = [Counter() for _ in documents]
  ```

- How many times each word is assigned to each topic:

  ```
  # a list of Counters, one for each topic
  topic_word_counts = [Counter() for _ in range(K)]
  ```

- The total number of words assigned to each topic:

  ```
  # a list of numbers, one for each topic
  topic_counts = [0 for _ in range(K)]
  ```

- The total number of words contained in each document:

  ```
  # a list of numbers, one for each document
  document_lengths = [len(document) for document in documents]
  ```

- The number of distinct words:

  ```
  distinct_words = set(word for document in documents for word in document)
  W = len(distinct_words)
  ```

- And the number of documents:

  ```
  D = len(documents)
  ```

Once we populate these, we can find, for example, the number of words in `documents[3]` associated with topic 1 as follows:

```
document_topic_counts[3][1]
```

And we can find the number of times *nlp* is associated with topic 2 as follows:

```
topic_word_counts[2]["nlp"]
```

Now we're ready to define our conditional probability functions. As in Chapter 13, each has a smoothing term that ensures every topic has a nonzero chance of being chosen in any document and that every word has a nonzero chance of being chosen for any topic:

```
def p_topic_given_document(topic: int, d: int, alpha: float = 0.1) -> float:
    """
```

```
    The fraction of words in document 'd'
    that are assigned to 'topic' (plus some smoothing)
    """
    return ((document_topic_counts[d][topic] + alpha) /
            (document_lengths[d] + K * alpha))

def p_word_given_topic(word: str, topic: int, beta: float = 0.1) -> float:
    """
    The fraction of words assigned to 'topic'
    that equal 'word' (plus some smoothing)
    """
    return ((topic_word_counts[topic][word] + beta) /
            (topic_counts[topic] + W * beta))
```

We'll use these to create the weights for updating topics:

```
def topic_weight(d: int, word: str, k: int) -> float:
    """
    Given a document and a word in that document,
    return the weight for the kth topic
    """
    return p_word_given_topic(word, k) * p_topic_given_document(k, d)

def choose_new_topic(d: int, word: str) -> int:
    return sample_from([topic_weight(d, word, k)
                        for k in range(K)])
```

There are solid mathematical reasons why `topic_weight` is defined the way it is, but their details would lead us too far afield. Hopefully it makes at least intuitive sense that—given a word and its document—the likelihood of any topic choice depends on both how likely that topic is for the document and how likely that word is for the topic.

This is all the machinery we need. We start by assigning every word to a random topic and populating our counters appropriately:

```
random.seed(0)
document_topics = [[random.randrange(K) for word in document]
                   for document in documents]

for d in range(D):
    for word, topic in zip(documents[d], document_topics[d]):
        document_topic_counts[d][topic] += 1
        topic_word_counts[topic][word] += 1
        topic_counts[topic] += 1
```

Our goal is to get a joint sample of the topics–word distribution and the documents–topic distribution. We do this using a form of Gibbs sampling that uses the conditional probabilities defined previously:

```
import tqdm

for iter in tqdm.trange(1000):
    for d in range(D):
        for i, (word, topic) in enumerate(zip(documents[d],
                                              document_topics[d])):

            # remove this word / topic from the counts
            # so that it doesn't influence the weights
            document_topic_counts[d][topic] -= 1
            topic_word_counts[topic][word] -= 1
            topic_counts[topic] -= 1
            document_lengths[d] -= 1

            # choose a new topic based on the weights
            new_topic = choose_new_topic(d, word)
            document_topics[d][i] = new_topic

            # and now add it back to the counts
            document_topic_counts[d][new_topic] += 1
            topic_word_counts[new_topic][word] += 1
            topic_counts[new_topic] += 1
            document_lengths[d] += 1
```

What are the topics? They're just numbers 0, 1, 2, and 3. If we want names for them, we have to do that ourselves. Let's look at the five most heavily weighted words for each (Table 21-1):

```
for k, word_counts in enumerate(topic_word_counts):
    for word, count in word_counts.most_common():
        if count > 0:
            print(k, word, count)
```

Table 21-1. Most common words per topic

Topic 0	Topic 1	Topic 2	Topic 3
Java	R	HBase	regression
Big Data	statistics	Postgres	libsvm
Hadoop	Python	MongoDB	scikit-learn
deep learning	probability	Cassandra	machine learning
artificial intelligence	pandas	NoSQL	neural networks

Based on these I'd probably assign topic names:

```
topic_names = ["Big Data and programming languages",
               "Python and statistics",
               "databases",
               "machine learning"]
```

at which point we can see how the model assigns topics to each user's interests:

```
for document, topic_counts in zip(documents, document_topic_counts):
    print(document)
    for topic, count in topic_counts.most_common():
        if count > 0:
            print(topic_names[topic], count)
    print()
```

which gives:

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Big Data and programming languages 4 databases 3
['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
databases 5
['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
Python and statistics 5 machine learning 1
```

and so on. Given the "ands" we needed in some of our topic names, it's possible we should use more topics, although most likely we don't have enough data to successfully learn them.

Word Vectors

A lot of recent advances in NLP involve deep learning. In the rest of this chapter we'll look at a couple of them using the machinery we developed in Chapter 19.

One important innovation involves representing words as low-dimensional vectors. These vectors can be compared, added together, fed into machine learning models, or anything else you want to do with them. They usually have nice properties; for example, similar words tend to have similar vectors. That is, typically the word vector for *big* is pretty close to the word vector for *large*, so that a model operating on word vectors can (to some degree) handle things like synonymy for free.

Frequently the vectors will exhibit delightful arithmetic properties as well. For instance, in some such models if you take the vector for *king*, subtract the vector for *man*, and add the vector for *woman*, you will end up with a vector that's very close to the vector for *queen*. It can be interesting to ponder what this means about what the word vectors actually "learn," although we won't spend time on that here.

Coming up with such vectors for a large vocabulary of words is a difficult undertaking, so typically we'll *learn* them from a corpus of text. There are a couple of different schemes, but at a high level the task typically looks something like this:

1. Get a bunch of text.
2. Create a dataset where the goal is to predict a word given nearby words (or alternatively, to predict nearby words given a word).
3. Train a neural net to do well on this task.
4. Take the internal states of the trained neural net as the word vectors.

In particular, because the task is to predict a word given nearby words, words that occur in similar contexts (and hence have similar nearby words) should have similar internal states and therefore similar word vectors.

Here we'll measure "similarity" using *cosine similarity*, which is a number between –1 and 1 that measures the degree to which two vectors point in the same direction:

```
from scratch.linear_algebra import dot, Vector
import math

def cosine_similarity(v1: Vector, v2: Vector) -> float:
    return dot(v1, v2) / math.sqrt(dot(v1, v1) * dot(v2, v2))

assert cosine_similarity([1., 1, 1], [2., 2, 2]) == 1, "same direction"
assert cosine_similarity([-1., -1], [2., 2]) == -1,    "opposite direction"
assert cosine_similarity([1., 0], [0., 1]) == 0,       "orthogonal"
```

Let's learn some word vectors to see how this works.

To start with, we'll need a toy dataset. The commonly used word vectors are typically derived from training on millions or even billions of words. As our toy library can't cope with that much data, we'll create an artificial dataset with some structure to it:

```
colors = ["red", "green", "blue", "yellow", "black", ""]
nouns = ["bed", "car", "boat", "cat"]
verbs = ["is", "was", "seems"]
adverbs = ["very", "quite", "extremely", ""]
adjectives = ["slow", "fast", "soft", "hard"]

def make_sentence() -> str:
    return " ".join([
        "The",
        random.choice(colors),
        random.choice(nouns),
        random.choice(verbs),
        random.choice(adverbs),
        random.choice(adjectives),
        "."
    ])

NUM_SENTENCES = 50

random.seed(0)
sentences = [make_sentence() for _ in range(NUM_SENTENCES)]
```

This will generate lots of sentences with similar structure but different words; for example, "The green boat seems quite slow." Given this setup, the colors will mostly appear in "similar" contexts, as will the nouns, and so on. So if we do a good job of assigning word vectors, the colors should get similar vectors, and so on.

In practical usage, you'd probably have a corpus of millions of sentences, in which case you'd get "enough" context from the sentences as they are. Here, with only 50 sentences, we have to make them somewhat artificial.

As mentioned earlier, we'll want to one-hot-encode our words, which means we'll need to convert them to IDs. We'll introduce a `Vocabulary` class to keep track of this mapping:

```
from scratch.deep_learning import Tensor

class Vocabulary:
    def __init__(self, words: List[str] = None) -> None:
        self.w2i: Dict[str, int] = {}  # mapping word -> word_id
        self.i2w: Dict[int, str] = {}  # mapping word_id -> word

        for word in (words or []):     # If words were provided,
            self.add(word)             # add them.

    @property
    def size(self) -> int:
        """how many words are in the vocabulary"""
        return len(self.w2i)

    def add(self, word: str) -> None:
        if word not in self.w2i:        # If the word is new to us:
            word_id = len(self.w2i)     # Find the next id.
            self.w2i[word] = word_id    # Add to the word -> word_id map.
            self.i2w[word_id] = word    # Add to the word_id -> word map.

    def get_id(self, word: str) -> int:
        """return the id of the word (or None)"""
        return self.w2i.get(word)

    def get_word(self, word_id: int) -> str:
        """return the word with the given id (or None)"""
        return self.i2w.get(word_id)

    def one_hot_encode(self, word: str) -> Tensor:
        word_id = self.get_id(word)
        assert word_id is not None, f"unknown word {word}"

        return [1.0 if i == word_id else 0.0 for i in range(self.size)]
```

These are all things we could do manually, but it's handy to have it in a class. We should probably test it:

```
vocab = Vocabulary(["a", "b", "c"])
assert vocab.size == 3,              "there are 3 words in the vocab"
assert vocab.get_id("b") == 1,       "b should have word_id 1"
assert vocab.one_hot_encode("b") == [0, 1, 0]
assert vocab.get_id("z") is None,    "z is not in the vocab"
assert vocab.get_word(2) == "c",     "word_id 2 should be c"
vocab.add("z")
assert vocab.size == 4,              "now there are 4 words in the vocab"
assert vocab.get_id("z") == 3,       "now z should have id 3"
assert vocab.one_hot_encode("z") == [0, 0, 0, 1]
```

We should also write simple helper functions to save and load a vocabulary, just as we have for our deep learning models:

```
import json

def save_vocab(vocab: Vocabulary, filename: str) -> None:
    with open(filename, 'w') as f:
        json.dump(vocab.w2i, f)       # Only need to save w2i

def load_vocab(filename: str) -> Vocabulary:
    vocab = Vocabulary()
    with open(filename) as f:
        # Load w2i and generate i2w from it
        vocab.w2i = json.load(f)
        vocab.i2w = {id: word for word, id in vocab.w2i.items()}
    return vocab
```

We'll be using a word vector model called *skip-gram* that takes as input a word and generates probabilities for what words are likely to be seen near it. We will feed it training pairs `(word, nearby_word)` and try to minimize the `SoftmaxCrossEntropy` loss.

Another common model, *continuous bag-of-words* (CBOW), takes the nearby words as the inputs and tries to predict the original word.

Let's design our neural network. At its heart will be an *embedding* layer that takes as input a word ID and returns a word vector. Under the covers we can just use a lookup table for this.

We'll then pass the word vector to a `Linear` layer with the same number of outputs as we have words in our vocabulary. As before, we'll use `softmax` to convert these outputs to probabilities over nearby words. As we use gradient descent to train the model, we will be updating the vectors in the lookup table. Once we've finished training, that lookup table gives us our word vectors.

Let's create that embedding layer. In practice we might want to embed things other than words, so we'll construct a more general `Embedding` layer. (Later we'll write a `TextEmbedding` subclass that's specifically for word vectors.)

In its constructor we'll provide the number and dimension of our embedding vectors, so it can create the embeddings (which will be standard random normals, initially):

```
from typing import Iterable
from scratch.deep_learning import Layer, Tensor, random_tensor, zeros_like

class Embedding(Layer):
    def __init__(self, num_embeddings: int, embedding_dim: int) -> None:
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim

        # One vector of size embedding_dim for each desired embedding
        self.embeddings = random_tensor(num_embeddings, embedding_dim)
        self.grad = zeros_like(self.embeddings)

        # Save last input id
        self.last_input_id = None
```

In our case we'll only be embedding one word at a time. However, in other models we might want to embed a sequence of words and get back a sequence of word vectors. (For example, if we wanted to train the CBOW model described earlier.) So an alternative design would take sequences of word IDs. We'll stick with one at a time, to make things simpler.

```
    def forward(self, input_id: int) -> Tensor:
        """Just select the embedding vector corresponding to the input id"""
        self.input_id = input_id    # remember for use in backpropagation

        return self.embeddings[input_id]
```

For the backward pass we'll get a gradient corresponding to the chosen embedding vector, and we'll need to construct the corresponding gradient for `self.embeddings`, which is zero for every embedding other than the chosen one:

```
    def backward(self, gradient: Tensor) -> None:
        # Zero out the gradient corresponding to the last input.
        # This is way cheaper than creating a new all-zero tensor each time.
        if self.last_input_id is not None:
            zero_row = [0 for _ in range(self.embedding_dim)]
            self.grad[self.last_input_id] = zero_row

        self.last_input_id = self.input_id
        self.grad[self.input_id] = gradient
```

Because we have parameters and gradients, we need to override those methods:

```
    def params(self) -> Iterable[Tensor]:
        return [self.embeddings]

    def grads(self) -> Iterable[Tensor]:
        return [self.grad]
```

As mentioned earlier, we'll want a subclass specifically for word vectors. In that case our number of embeddings is determined by our vocabulary, so let's just pass that in instead:

```
class TextEmbedding(Embedding):
    def __init__(self, vocab: Vocabulary, embedding_dim: int) -> None:
        # Call the superclass constructor
        super().__init__(vocab.size, embedding_dim)

        # And hang onto the vocab
        self.vocab = vocab
```

The other built-in methods will all work as is, but we'll add a couple more methods specific to working with text. For example, we'd like to be able to retrieve the vector for a given word. (This is not part of the `Layer` interface, but we are always free to add extra methods to specific layers as we like.)

```
    def __getitem__(self, word: str) -> Tensor:
        word_id = self.vocab.get_id(word)
        if word_id is not None:
            return self.embeddings[word_id]
        else:
            return None
```

This dunder method will allow us to retrieve word vectors using indexing:

```
word_vector = embedding["black"]
```

And we'd also like the embedding layer to tell us the closest words to a given word:

```
    def closest(self, word: str, n: int = 5) -> List[Tuple[float, str]]:
        """Returns the n closest words based on cosine similarity"""
        vector = self[word]

        # Compute pairs (similarity, other_word), and sort most similar first
        scores = [(cosine_similarity(vector, self.embeddings[i]), other_word)
                  for other_word, i in self.vocab.w2i.items()]
        scores.sort(reverse=True)

        return scores[:n]
```

Our embedding layer just outputs vectors, which we can feed into a `Linear` layer.

Now we're ready to assemble our training data. For each input word, we'll choose as target words the two words to its left and the two words to its right.

Let's start by lowercasing the sentences and splitting them into words:

```
import re

# This is not a great regex, but it works on our data.
tokenized_sentences = [re.findall("[a-z]+|[.]", sentence.lower())
                       for sentence in sentences]
```

at which point we can construct a vocabulary:

```
# Create a vocabulary (that is, a mapping word -> word_id) based on our text.
vocab = Vocabulary(word
                   for sentence_words in tokenized_sentences
                   for word in sentence_words)
```

And now we can create training data:

```
from scratch.deep_learning import Tensor, one_hot_encode

inputs: List[int] = []
targets: List[Tensor] = []

for sentence in tokenized_sentences:
    for i, word in enumerate(sentence):          # For each word
        for j in [i - 2, i - 1, i + 1, i + 2]:   # take the nearby locations
            if 0 <= j < len(sentence):           # that aren't out of bounds
                nearby_word = sentence[j]        # and get those words.

                # Add an input that's the original word_id
                inputs.append(vocab.get_id(word))

                # Add a target that's the one-hot-encoded nearby word
                targets.append(vocab.one_hot_encode(nearby_word))
```

With the machinery we've built up, it's now easy to create our model:

```
from scratch.deep_learning import Sequential, Linear

random.seed(0)
EMBEDDING_DIM = 5  # seems like a good size

# Define the embedding layer separately, so we can reference it.
embedding = TextEmbedding(vocab=vocab, embedding_dim=EMBEDDING_DIM)

model = Sequential([
    # Given a word (as a vector of word_ids), look up its embedding.
    embedding,
    # And use a linear layer to compute scores for "nearby words."
    Linear(input_dim=EMBEDDING_DIM, output_dim=vocab.size)
])
```

Using the machinery from Chapter 19, it's easy to train our model:

```
from scratch.deep_learning import SoftmaxCrossEntropy, Momentum, GradientDescent

loss = SoftmaxCrossEntropy()
optimizer = GradientDescent(learning_rate=0.01)

for epoch in range(100):
    epoch_loss = 0.0
    for input, target in zip(inputs, targets):
        predicted = model.forward(input)
```

```
            epoch_loss += loss.loss(predicted, target)
            gradient = loss.gradient(predicted, target)
            model.backward(gradient)
            optimizer.step(model)
        print(epoch, epoch_loss)           # Print the loss
        print(embedding.closest("black"))  # and also a few nearest words
        print(embedding.closest("slow"))   # so we can see what's being
        print(embedding.closest("car"))    # learned.
```

As you watch this train, you can see the colors getting closer to each other, the adjectives getting closer to each other, and the nouns getting closer to each other.

Once the model is trained, it's fun to explore the most similar words:

```
pairs = [(cosine_similarity(embedding[w1], embedding[w2]), w1, w2)
         for w1 in vocab.w2i
         for w2 in vocab.w2i
         if w1 < w2]
pairs.sort(reverse=True)
print(pairs[:5])
```

which (for me) results in:

```
[(0.9980283554864815, 'boat', 'car'),
 (0.9975147744587706, 'bed', 'cat'),
 (0.9953153441218054, 'seems', 'was'),
 (0.9927107440377975, 'extremely', 'quite'),
 (0.9836183658415987, 'bed', 'car')]
```

(Obviously *bed* and *cat* are not really similar, but in our training sentences they appear to be, and that's what the model is capturing.)

We can also extract the first two principal components and plot them:

```
from scratch.working_with_data import pca, transform
import matplotlib.pyplot as plt

# Extract the first two principal components and transform the word vectors
components = pca(embedding.embeddings, 2)
transformed = transform(embedding.embeddings, components)

# Scatter the points (and make them white so they're "invisible")
fig, ax = plt.subplots()
ax.scatter(*zip(*transformed), marker='.', color='w')

# Add annotations for each word at its transformed location
for word, idx in vocab.w2i.items():
    ax.annotate(word, transformed[idx])

# And hide the axes
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)

plt.show()
```

which shows that similar words are indeed clustering together (Figure 21-3):

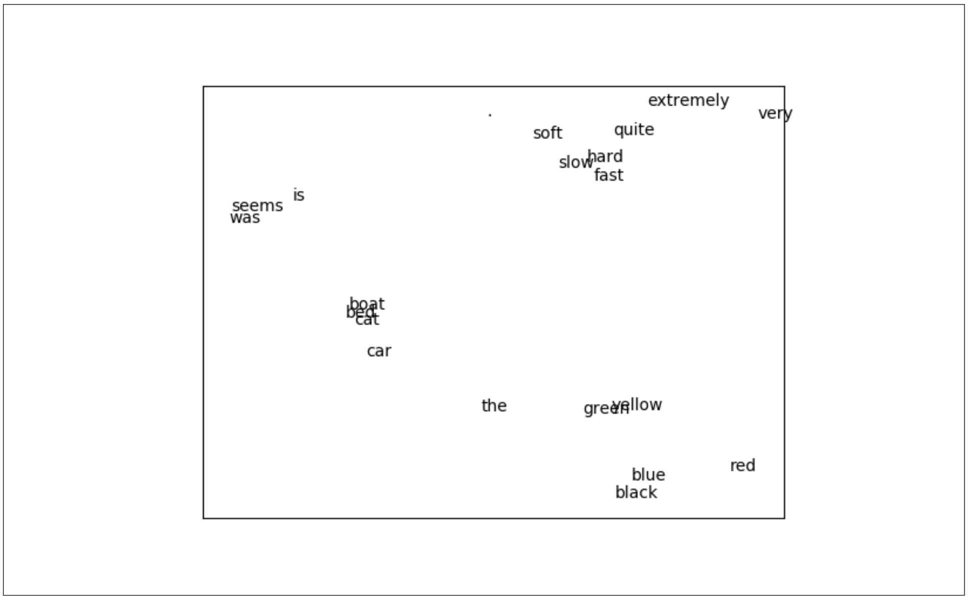

Figure 21-3. Word vectors

If you're interested, it's not hard to train CBOW word vectors. You'll have to do a little work. First, you'll need to modify the `Embedding` layer so that it takes as input a *list* of IDs and outputs a *list* of embedding vectors. Then you'll have to create a new layer (`Sum`?) that takes a list of vectors and returns their sum.

Each word represents a training example where the input is the word IDs for the surrounding words, and the target is the one-hot encoding of the word itself.

The modified `Embedding` layer turns the surrounding words into a list of vectors, the new `Sum` layer collapses the list of vectors down to a single vector, and then a `Linear` layer can produce scores that can be `softmax`ed to get a distribution representing "most likely words, given this context."

I found the CBOW model harder to train than the skip-gram one, but I encourage you to try it out.

Recurrent Neural Networks

The word vectors we developed in the previous section are often used as the inputs to neural networks. One challenge to doing this is that sentences have varying lengths: you could think of a 3-word sentence as a `[3, embedding_dim]` tensor and a 10-word sentence as a `[10, embedding_dim]` tensor. In order to, say, pass them to a `Linear` layer, we need to do something about that first variable-length dimension.

One option is to use a `Sum` layer (or a variant that takes the average); however, the *order* of the words in a sentence is usually important to its meaning. To take a common example, "dog bites man" and "man bites dog" are two very different stories!

Another way of handling this is using *recurrent neural networks* (RNNs), which have a *hidden state* they maintain between inputs. In the simplest case, each input is combined with the current hidden state to produce an output, which is then used as the new hidden state. This allows such networks to "remember" (in a sense) the inputs they've seen, and to build up to a final output that depends on all the inputs and their order.

We'll create pretty much the simplest possible RNN layer, which will accept a single input (corresponding to, e.g., a single word in a sentence, or a single character in a word), and which will maintain its hidden state between calls.

Recall that our `Linear` layer had some weights, `w`, and a bias, `b`. It took a vector `input` and produced a different vector as `output` using the logic:

```
output[o] = dot(w[o], input) + b[o]
```

Here we'll want to incorporate our hidden state, so we'll have *two* sets of weights—one to apply to the `input` and one to apply to the previous `hidden` state:

```
output[o] = dot(w[o], input) + dot(u[o], hidden) + b[o]
```

Next, we'll use the `output` vector as the new value of `hidden`. This isn't a huge change, but it will allow our networks to do wonderful things.

```
from scratch.deep_learning import tensor_apply, tanh

class SimpleRnn(Layer):
    """Just about the simplest possible recurrent layer."""
    def __init__(self, input_dim: int, hidden_dim: int) -> None:
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        self.w = random_tensor(hidden_dim, input_dim, init='xavier')
        self.u = random_tensor(hidden_dim, hidden_dim, init='xavier')
        self.b = random_tensor(hidden_dim)

        self.reset_hidden_state()

    def reset_hidden_state(self) -> None:
        self.hidden = [0 for _ in range(self.hidden_dim)]
```

You can see that we start out the hidden state as a vector of 0s, and we provide a function that people using the network can call to reset the hidden state.

Given this setup, the `forward` function is reasonably straightforward (at least, it is if you remember and understand how our `Linear` layer worked):

```
    def forward(self, input: Tensor) -> Tensor:
        self.input = input              # Save both input and previous
        self.prev_hidden = self.hidden  # hidden state to use in backprop.

        a = [(dot(self.w[h], input) +           # weights @ input
              dot(self.u[h], self.hidden) +     # weights @ hidden
              self.b[h])                        # bias
             for h in range(self.hidden_dim)]

        self.hidden = tensor_apply(tanh, a)  # Apply tanh activation
        return self.hidden                   # and return the result.
```

The `backward` pass is similar to the one in our `Linear` layer, except that it needs to compute an additional set of gradients for the `u` weights:

```
    def backward(self, gradient: Tensor):
        # Backpropagate through the tanh
        a_grad = [gradient[h] * (1 - self.hidden[h] ** 2)
                  for h in range(self.hidden_dim)]

        # b has the same gradient as a
        self.b_grad = a_grad

        # Each w[h][i] is multiplied by input[i] and added to a[h],
        # so each w_grad[h][i] = a_grad[h] * input[i]
        self.w_grad = [[a_grad[h] * self.input[i]
                        for i in range(self.input_dim)]
                       for h in range(self.hidden_dim)]

        # Each u[h][h2] is multiplied by hidden[h2] and added to a[h],
        # so each u_grad[h][h2] = a_grad[h] * prev_hidden[h2]
        self.u_grad = [[a_grad[h] * self.prev_hidden[h2]
                        for h2 in range(self.hidden_dim)]
                       for h in range(self.hidden_dim)]

        # Each input[i] is multiplied by every w[h][i] and added to a[h],
        # so each input_grad[i] = sum(a_grad[h] * w[h][i] for h in ...)
        return [sum(a_grad[h] * self.w[h][i] for h in range(self.hidden_dim))
                for i in range(self.input_dim)]
```

And finally we need to override the `params` and `grads` methods:

```
    def params(self) -> Iterable[Tensor]:
        return [self.w, self.u, self.b]

    def grads(self) -> Iterable[Tensor]:
        return [self.w_grad, self.u_grad, self.b_grad]
```

This "simple" RNN is so simple that you probably shouldn't use it in practice.

Our `SimpleRnn` has a couple of undesirable features. One is that its entire hidden state is used to update the input every time you call it. The other is that the entire hidden state is overwritten every time you call it. Both of these make it difficult to train; in particular, they make it difficult for the model to learn long-range dependencies.

For this reason, almost no one uses this kind of simple RNN. Instead, they use more complicated variants like the LSTM ("long short-term memory") or the GRU ("gated recurrent unit"), which have many more parameters and use parameterized "gates" that allow only some of the state to be updated (and only some of the state to be used) at each timestep.

There is nothing particularly *difficult* about these variants; however, they involve a great deal more code, which would not be (in my opinion) correspondingly more edifying to read. The code for this chapter on GitHub includes an LSTM implementation. I encourage you to check it out, but it's somewhat tedious and so we won't discuss it further here.

One other quirk of our implementation is that it takes only one "step" at a time and requires us to manually reset the hidden state. A more practical RNN implementation might accept sequences of inputs, set its hidden state to 0s at the beginning of each sequence, and produce sequences of outputs. Ours could certainly be modified to behave this way; again, this would require more code and complexity for little gain in understanding.

Example: Using a Character-Level RNN

The newly hired VP of Branding did not come up with the name *DataSciencester* himself, and (accordingly) he suspects that a better name might lead to more success for the company. He asks you to use data science to suggest candidates for replacement.

One "cute" application of RNNs involves using *characters* (rather than words) as their inputs, training them to learn the subtle language patterns in some dataset, and then using them to generate fictional instances from that dataset.

For example, you could train an RNN on the names of alternative bands, use the trained model to generate new names for fake alternative bands, and then hand-select the funniest ones and share them on Twitter. Hilarity!

Having seen this trick enough times to no longer consider it clever, you decide to give it a shot.

After some digging, you find that the startup accelerator Y Combinator has published a list of its top 100 (actually 101) most successful startups, which seems like a good starting point. Checking the page, you find that the company names all live inside `<b class="h4">` tags, which means it's easy to use your web scraping skills to retrieve them:

```
from bs4 import BeautifulSoup
import requests

url = "https://www.ycombinator.com/topcompanies/"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')

# We get the companies twice, so use a set comprehension to deduplicate.
companies = list({b.text
                  for b in soup("b")
                  if "h4" in b.get("class", ())})
assert len(companies) == 101
```

As always, the page may change (or vanish), in which case this code won't work. If so, you can use your newly learned data science skills to fix it or just get the list from the book's GitHub site.

So what is our plan? We'll train a model to predict the next character of a name, given the current character *and* a hidden state representing all the characters we've seen so far.

As usual, we'll actually predict a probability distribution over characters and train our model to minimize the `SoftmaxCrossEntropy` loss.

Once our model is trained, we can use it to generate some probabilities, randomly sample a character according to those probabilities, and then feed that character as its next input. This will allow us to *generate* company names using the learned weights.

To start with, we should build a `Vocabulary` from the characters in the names:

```
vocab = Vocabulary([c for company in companies for c in company])
```

In addition, we'll use special tokens to signify the start and end of a company name. This allows the model to learn which characters should *begin* a company name and also to learn when a company name is *finished*.

We'll just use the regex characters for start and end, which (luckily) don't appear in our list of companies:

```
START = "^"
STOP = "$"

# We need to add them to the vocabulary too.
```

```
vocab.add(START)
vocab.add(STOP)
```

For our model, we'll one-hot-encode each character, pass it through two `SimpleRnn`s, and then use a `Linear` layer to generate the scores for each possible next character:

```
HIDDEN_DIM = 32  # You should experiment with different sizes!

rnn1 =  SimpleRnn(input_dim=vocab.size, hidden_dim=HIDDEN_DIM)
rnn2 =  SimpleRnn(input_dim=HIDDEN_DIM, hidden_dim=HIDDEN_DIM)
linear = Linear(input_dim=HIDDEN_DIM, output_dim=vocab.size)

model = Sequential([
    rnn1,
    rnn2,
    linear
])
```

Imagine for the moment that we've trained this model. Let's write the function that uses it to generate new company names, using the `sample_from` function from "Topic Modeling" on page 282:

```
from scratch.deep_learning import softmax

def generate(seed: str = START, max_len: int = 50) -> str:
    rnn1.reset_hidden_state()  # Reset both hidden states
    rnn2.reset_hidden_state()
    output = [seed]            # Start the output with the specified seed

    # Keep going until we produce the STOP character or reach the max length
    while output[-1] != STOP and len(output) < max_len:
        # Use the last character as the input
        input = vocab.one_hot_encode(output[-1])

        # Generate scores using the model
        predicted = model.forward(input)

        # Convert them to probabilities and draw a random char_id
        probabilities = softmax(predicted)
        next_char_id = sample_from(probabilities)

        # Add the corresponding char to our output
        output.append(vocab.get_word(next_char_id))

    # Get rid of START and END characters and return the word
    return ''.join(output[1:-1])
```

At long last, we're ready to train our character-level RNN. It will take a while!

```
loss = SoftmaxCrossEntropy()
optimizer = Momentum(learning_rate=0.01, momentum=0.9)

for epoch in range(300):
    random.shuffle(companies)  # Train in a different order each epoch.
    epoch_loss = 0             # Track the loss.
    for company in tqdm.tqdm(companies):
        rnn1.reset_hidden_state()  # Reset both hidden states.
        rnn2.reset_hidden_state()
        company = START + company + STOP   # Add START and STOP characters.

        # The rest is just our usual training loop, except that the inputs
        # and target are the one-hot-encoded previous and next characters.
        for prev, next in zip(company, company[1:]):
            input = vocab.one_hot_encode(prev)
            target = vocab.one_hot_encode(next)
            predicted = model.forward(input)
            epoch_loss += loss.loss(predicted, target)
            gradient = loss.gradient(predicted, target)
            model.backward(gradient)
            optimizer.step(model)

    # Each epoch, print the loss and also generate a name.
    print(epoch, epoch_loss, generate())

    # Turn down the learning rate for the last 100 epochs.
    # There's no principled reason for this, but it seems to work.
    if epoch == 200:
        optimizer.lr *= 0.1
```

After training, the model generates some actual names from the list (which isn't surprising, since the model has a fair amount of capacity and not a lot of training data), as well as names that are only slightly different from training names (Scripe, Loinbare, Pozium), names that seem genuinely creative (Benuus, Cletpo, Equite, Vivest), and names that are garbage-y but still sort of word-like (SFitreasy, Sint ocanelp, GliyOx, Doorboronelhav).

Unfortunately, like most character-level-RNN outputs, these are only mildly clever, and the VP of Branding ends up unable to use them.

If I up the hidden dimension to 64, I get a lot more names verbatim from the list; if I drop it to 8, I get mostly garbage. The vocabulary and final weights for all these model sizes are available on the book's GitHub site, and you can use `load_weights` and `load_vocab` to use them yourself.

As mentioned previously, the GitHub code for this chapter also contains an implementation for an LSTM, which you should feel free to swap in as a replacement for the `SimpleRnn`s in our company name model.

For Further Exploration

- NLTK is a popular library of NLP tools for Python. It has its own entire book, which is available to read online.

- gensim is a Python library for topic modeling, which is a better bet than our from-scratch model.
- spaCy is a library for "Industrial Strength Natural Language Processing in Python" and is also quite popular.
- Andrej Karpathy has a famous blog post, "The Unreasonable Effectiveness of Recurrent Neural Networks", that's very much worth reading.
- My day job involves building AllenNLP, a Python library for doing NLP research. (At least, as of the time this book went to press, it did.) The library is quite beyond the scope of this book, but you might still find it interesting, and it has a cool interactive demo of many state-of-the-art NLP models.