Welcome

Thank you for purchasing the MEAP for Deep Learning for Natural Language Processing. This book will teach you to apply deep learning to one of most vibrant applications of current AI: the analysis of natural language. The book is addressed to anyone who needs a practical guide to current deep learning techniques, from developers to students and researchers of computational linguistics.

While rooted in linguistics, the field of computational linguistics, or Natural Language Processing (NLP, for short), borrows many tools and techniques from neighboring fields such as computer science, logic, statistics and machine learning. Much of the current research in NLP now uses deep learning. It is quite hard to keep track of the many papers and software repositories that are being produced in this fast-paced field. You may even feel you’re missing out on the cutting edge stuff. This is exactly what this book attempts to remedy: it will open up the toolbox of deep learning for NLP practitioners, outlining the pros and cons of various actual techniques and architectures. It speaks the language of NLP and is actually written by a computational linguist.

The book is written to be self-contained, but cannot cover all details of deep learning and NLP. Two excellent companion books, which can either be used as reference or primer, would be Manning’s Deep Learning with Python, and Natural Language Processing in Action. All examples in this book are written in Python and the elegant Keras library. This is a code-intensive book, with lots of examples of deep learning solutions for well-known NLP problems. The book contains references to recent literature and some of our code draws inspiration from great public domain software repositories, including the source code examples in the Keras distribution.

Deep learning is under continuous development and may seem like a volatile moving target. I hope this practical and hands-on book will help you become proficient in the current types of deep learning, and will provide you with a solid basis for keeping up with new developments.

Writing an ambitious book like this one is a daunting task, and is prone to imperfection. Your feedback is more than welcome. If you have any questions, comments, or suggestions, please share them in Manning’s Author Online forum for my book.

—Stephan Raaijmakers
brief contents

PART 1: INTRODUCTION

1  Deep learning for NLP
2  Deep learning and language: the basics

PART 2: DEEP NLP

3  Text embeddings
4  Textual similarity
5  Sequential NLP

PART 3: ADVANCED TOPICS

6  Generating text
7  Attention
8  Deep Memory-based NLP
9  Learning linguistic structure
10 Hyperparameters for Deep NLP
11 Outlook
Language comes naturally to humans, but is traditionally hard to grasp for computers. This book addresses the application of recent and cutting-edge deep learning techniques to automated language analysis. Deep learning has emerged in the last decade as the vehicle of the latest wave in AI. Results have consistently redefined the state-of-the-art for a plethora of data analysis tasks in a variety of domains. For an increasing amount of deep learning algorithms, better-than-human (human-parity or superhuman) performance has been reported: for instance, speech recognition in noisy conditions, and medical diagnosis based on images. Current deep learning-based natural language processing (NLP) outperforms all pre-existing approaches with a large margin. What exactly makes deep learning so adequate for these intricate analysis tasks, in particular language processing? This chapter presents some of the background necessary for answering this question. We will guide you through a selection of important topics in machine learning for NLP.

1.1 Overview of the book

The intended audience for this book consists of anyone working in natural language processing: computational linguists, software engineers, and machine learning or NLP students. The field of natural language processing is vast, and comprises a daunting amount of formalisms and approaches. With deep learning entering the stage, many are eager to get their feet wet, but are shied away by the highly technical nature of deep learning, and the fast pace of this field, with new approaches, software and papers emerging on a daily basis.

What you will get from this book is a thorough introduction to deep learning applied to a variety of language analysis tasks, supported by actual hands-on code. Explicitly linking the evergreens of computational linguistics, such as part-of-speech tagging, or...
query-answering, to deep learning, will help you become a proficient deep learning NLP expert. Beyond this, the book covers state of the art approaches to challenging new problems, such as explainable AI and the role of semantics.

This first chapter covers

- A short roadtrip through machine learning applied to NLP.
- A brief historical overview of deep learning.
- An introduction to vector-based representations of language.

![Figure 1.1 Chapter organization.](image)

You will find a succinct overview of NLP in Appendix 1. Let’s start off with discussing a few well-known machine learning-based NLP algorithms in some detail, illustrated with a handful of practical examples to wet your appetite. After that, we motivate the case for deep learning-based NLP.
Current natural language processing heavily relies on machine learning. Machine learning has its roots in statistics, building among others on the seminal work by Thomas Bayes and LaPlace in the 18-th and 19-th century (Bayes, LaPlace) and the least squares methods for curve approximation by Legendre in 1812 (Legendre). The field of neural computing started with the work of McCulloch and Pitts in 1943, who put forward a formal theory (and logical calculus) of neural networks (mccullochpits43). It would take until 1950 before learning machines were proposed by Alan Turing (Turing).

All machine learning algorithms that perform classification (labeling) share a single goal: to arrive at linear separability of data that is labeled with classes: labels that indicate a (usually exclusive) category a data point belongs to. Data points presented to a machine learning algorithm typically consist of vector representations of descriptive traits. These representations constitute a so-called input space. The subsequent processing, manipulation and abstraction of the input space during the learning stage of a self-learning algorithm yields a feature space. Some of this processing can be done external to the algorithm: raw data can already be converted to features as part of a pre-processing stage, which technically creates an input space consisting of features. The output space consists of class labels that separate the various data points in a dataset, based on the class boundaries. The essence of deep learning, as we will see, is to learn abstract representations in the feature space.
Training a machine learning component involves learning boundaries between classes, which may depend on quite complex functions. The burden of learning class separability can be alleviated by smart feature pre-processing. Learning the class boundaries occurs by performing implicit or explicit transformations on linearly inseparable input spaces. The following figure shows a non-linear class boundary: a line separating objects in two classes that cannot be modeled by a linear function $f(x)=ax+b$. The function corresponding to this line is a non-linear classifier. A real-world example would be a bowl of multi-colored marbles, mixed in such a way that they cannot be separated from each other by means of a straight plate (like a scoop).
We will now briefly address three types of machine learning approaches that have had major uptake in NLP:

- The single-layer perceptron and its generalization to the multilayer perceptron
- Support vector machines, and
- Memory-based learning.

While there is a lot more to the story, these three types embody respectively the *neural* or *cognitive*, *eager* and *lazy* types of machine learning. All of these approaches relate naturally to the deep learning approach to natural language analysis, which is the main topic of this book.

### 1.2.1 The perceptron

In 1957, the first implementation of a biologically inspired machine learning component was realized: Rosenblatt’s perceptron (Rosenblatt). This device, implemented on physical hardware, allowed processing of visual stimuli represented by a square 400 (20 by 20) array of photosensitive cells. The weights of this network were set by electromotors driving potentiometers. The learning part of this perceptron was based on a simple one-layer neural network, which effectively became the archetype of neural networks (see Figure [img-perceptron]).
Suppose you have a vector of features that describe aspects of a certain object of interest, like the words in a document, and you want to create a function from these features to a binary label (for instance, you want to decide if the document conveys a positive or a negative sentiment). The single-layer perceptron is capable of doing this. It produces a binary output \( y \) (0 or 1) from a weighted combination of input values \( x_1, \ldots, x_n \), based on a threshold \( \theta \) and a bias \( b \):

\[
f(x) = \begin{cases} 
1 & \text{if } w \cdot x + b > \theta \\
0 & \text{else}
\end{cases}
\]

**Figure 1.7 Perceptron decision function**

The weights \( w_1, \ldots, w_n \) are learned from annotated training data, consisting of input vectors labeled with output labels. The thresholded unit is called a neuron. It receives the summed and weighted input \( v \). So, assume we have a set of weights and associated inputs:

<table>
<thead>
<tr>
<th>Weight</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
</tbody>
</table>

Then their summed and weighted output would be
3*10+5*20+7*30=310

\[ v = \sum_{i=1}^{n} w_i x_i \]

**Figure 1.8 Summed and weighted perceptron input**

This simplistic network is able to learn a specific set of functions that address the class of *linearly separable problems*: problems that are separable in input space with a linear function. Usually, these are the easier problems in classification. It's quite common for data to be *heavily entangled*. Consider undoing a knot in two separate ropes. Some knots are easy and can be undone in one step. Other knots need much more steps. This is the business of machine learning algorithms: undoing the intertwining of data objects living in different classes.

For NLP, the single-layer perceptron nowadays plays a marginal role, but it underlies several derived algorithms that strive for simplicity, such as online learning (ONLINE).

A practical example of a perceptron classifier is the following. We set out to build a document classifier that categorizes raw texts as being broadly about either atheism or medical topics. The popular *20 newsgroups* dataset (20newsgroups), one of the most widely used datasets for building and evaluating document classifiers, consists of newsgroup (usenet) texts distributed over 20 hand-assigned topics. Here is what we do:

- We make a subselection for two newsgroups of interest: *alt.atheism* and *sci.med*.
- We train a simple perceptron on a *vector representation* of the documents in these two classes. A vector is nothing more than a container (an ordered list of a finite dimension) for numerical values.
- The vector representation is based on a statistical representation of words called TF.IDF, which we will discuss in the Section *representations* below. For now, just assume TF.IDF is one of those magic tricks that turn documents into vectors that can be fed to a machine learning algorithm.

Don’t worry if you don’t completely understand this code right now. It’s here to give you an idea of what the code looks like for a basic perceptron.
Listing 1.1 A simple perceptron-based document classifier.

```python
from sklearn.linear_model import Perceptron
from sklearn.datasets import fetch_20newsgroups

categories = ['alt.atheism', 'sci.med']

train = fetch_20newsgroups(subset='train', categories=categories, shuffle=True)

perceptron = Perceptron(max_iter=100)

from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer()
X_train_counts = cv.fit_transform(train.data)

from sklearn.feature_extraction.text import TfidfTransformer

tfidf_tf = TfidfTransformer()
X_train_tfidf = tfidf_tf.fit_transform(X_train_counts)

perceptron.fit(X_train_tfidf, train.target)

test_docs = ['Religion is widespread, even in modern times',
             'His kidney failed',
             'The pope is a controversial leader',
             'White blood cells fight off infections',
             'The reverend had a heart attack in church']

X_test_counts = cv.transform(test_docs)
X_test_tfidf = tfidf_tf.transform(X_test_counts)

pred = perceptron.predict(X_test_tfidf)

for doc, category in zip(test_docs, pred):
    print('%r => %s' % (doc, train.target_names[category]))
```

1. We import a basic perceptron classifier from sklearn.
2. We import a routine for fetching the 20 newsgroups dataset from sklearn.
3. We limit the categories of the dataset.
4. We obtain documents for our category selection.
5. Our perceptron is defined. It will be trained for 100 iterations.
6. The familiar CountVectorizer is fit on our training data.
7. Now, we load, fit and deploy a TF.IDF transformer from sklearn. It computes TF.IDF representations of our count vectors.
8. The perceptron is trained on the TF.IDF vectors.
9. Our test data.
10. The test data is vectorized, first to count vectors, then to TF.IDF vectors.
11. The perceptron is applied to the test documents.
12. Results are printed.

This produces the following results:

```
Religion is widespread, even in modern times => alt.atheism
```
Apparently, these few short texts can be linearly separated by a simple, weight-based algorithm. This example is a huge simplification: the topics chosen are quite distinct. In real life, linear algorithms fall short in separating topics that overlap and share similar vocabulary.

The multilayer perceptron generalizes the single-layer model of the original perceptron to a multilayer model, with at least 3 layers: an input layer, one or more hidden representational layers, and an output layer:

![Figure 1.9 A multilayer perceptron with one hidden layer (h1…hn).](image)

His kidney failed => sci.med

The pope is a controversial leader => alt.atheism

White blood cells fight off infections => sci.med

The reverend had a heart attack in church => sci.med
1.2.2 Support Vector Machines

As mentioned, machine learning algorithms that perform classification (the labeling of objects with classes) attempt to arrive at linear boundaries between data points. Recall our _linear classifier for such a linear boundary. Imagine you are seeing two objects (like an orange and an apple) on a plate, one of which is partially obscuring the other. If you close one eye, you will lose 3D stereovision, and you cannot separate the two objects in 2D. If you open both eyes, you will be able to separate the two objects. Support Vector Machines routinely perform this migration to a higher dimension, separating objects in this space. Their secret weapon: *kernels*.

![Figure 1.10 From 2D to 3D with a kernel.](image)

A support vector machine is a binary classifier that implicitly maps data in feature space to higher dimensions in which data becomes separable by a linear plane, called a `hyperplane`. This mapping is implicit, and is carried out by a `kernel function`. This is a function that transforms the original input space to an alternative representation that implicitly has a higher dimensionality, with the aim of disentangling the data and making it linearly separable.

But the migration is implicit in the sense that it takes the form of a similarity function (in the picture above) applied to two feature vectors, just computing their distance. This is cordially called the *kernel trick*, It sounds like sheer magic, but it is actually quite simple. Let’s take a look.

You should already be familiar with the dot product of two vectors. If not, please see Appendix 2 for a refresher. To recap, the standard dot product of two vectors $a$ and $b$ is the sum of the cross-product of the two vectors:

$$ a \cdot b = \sum_{i=1}^{n} a_i b_i $$
So, a dot product is just a multiplicative operation on two vectors that produces a single number.

Kernels are generalizations of this dot product between vectors: they compute the dot product between altered versions of these vectors. The nature of alteration is specified by a kernel function. Generally speaking, a kernel function takes two vectors, mixes in a constant (a kernel parameter) and adds some kernel-specific ingredients to produce a specific form of a dot product of the two vectors.

Let’s return to our orange and apple. The objects are described by pairs of coordinates (x,y), since the table they’re lying on is a flat XY-plane. Like other types of kernels, the so-called polynomial kernel maps lower-dimensional spaces to higher-dimensional ones. You may recall from high school math that a polynomial function produces a value using addition, subtraction, multiplication or positive exponentiation only. Like \( y=4x^2+10 \). Polynomial kernels work on two input values (vectors of numeric values) and usually a constant. They compute a result using a polynomial.

For instance, a simple quadratic kernel \( K \), using a constant \( c \), and addressing the two two-dimensional XY-vectors describing our orange and apple:

- \( x=(x_1,x_2) \)

and

- \( y=(y_1,y_2) \)

looks like:

\[
K(x,y) = \\
(c + x^T y)^2 = \\
(c^2 + x_1 y_1 + x_2 y_2)^2 = \\
(c^2 + x_1^2 y_1^2 + x_2^2 y_2^2 + 2x_1 y_1 + 2x_2 y_2 + 2x_1 y_1 x_2 y_2)
\]

Figure 1.11 Quadratic kernel.

Notice the subscripted T? That is vector transposition (see Appendix B), necessary for

Listing 1.2 Dot product in Python.

def dot_product(a, b):
    return sum([a[i]*b[i] for i in range(len(a))])
vector multiplication. What does the kernel do? It computes a product between two vectors. This product is a number expressing a relation between the two input vectors. But the tedious expansion of this kernel shows that we actually operate in a six- (and not even a three-!) dimensional space: just count the factors separated by the plus sign in the result; we have six such factors. In this case, the kernel function $K$ implicitly computes its dot product between the following vectors:

- $\langle c, x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2 \rangle$
- $\langle c, y_1^2, y_2^2, \sqrt{2}y_1, \sqrt{2}y_2, \sqrt{2}y_1y_2 \rangle$

since the result is the product of these two vectors. But it never explicitly created these vectors. The whole point of this kernel trick is that, hopefully, in the higher-dimensional space that is the playground of the kernel, things become easier separable than in the entangled input space. Kernels do not explicitly represent this space, they implicitly work in it. You can imagine that, for long vectors and large exponents of the polynomial, this kernel trick quickly becomes a practical advantage!

In the transformed space created by the kernel trick, two classes are at best separated with maximally wide boundaries (called maximum margins). The data points determining the slope of these boundaries are called support vectors.

Figure 1.12 Maximum margins of an SVM. The support vectors are the points on the dashed lines.
Learning weights that optimize the margins with the least error (as measured on some held-out test data) is the job an SVM has to solve during training. After training, the support vectors and the various weights plus biases constitute the model. New input is projected onto the support vectors, and depending on which side it lands, it receives a positive or negative label (recall that SVM’s are binary classifiers). So, SVMs throw away a lot of their training data, only keeping some of it: the support vectors. They can be called *eager* forms of machine learning.


### 1.2.3 Memory-based learning

Unlike the *eager* types of machine learning that build compact and representative models of their training data, memory-based learning (MBL, Daelemans et al) is a form of *lazy* learning: it does not compress training data into generalizations, but instead keeps all training data available in memory. During classification, the actual processing of the training takes place: input data is matched with training data by the application of similarity or distance measures. Similar to Support Vector Machines, distance functions between vectors compute similarities. But here, we work on explicit vectors, and we do not perform any dimensionality tricks.

A well-known distance function is the IB1 metric, a simplified version of which is:

```python
def IB1(a,b):
    return sum( [delta(a[i],b[i] for i in range(len(a))])

def delta(x,y):
    if x==y:
        return 0
    if x!=y:
        return 1
```

This metric computes the distance between two feature vectors on the basis of *feature value overlap*: exact similarity for symbolic (non-numerical) values. Most MBL algorithms extend these distance metrics with feature weighting (e.g. information-gain based weighting) or exemplar weighting (REF). They partition the search space for matching in sets consisting of training items with the same distance to the current test item. For instance, sets of distances $d_1, d_2, \ldots$ can be found first, after which the algorithm...
computes the most frequent class in those sets. It then votes over all classes, to determine the most probable label for the test item. The \( k \)-parameter addresses the amount of distance sets to take into account, which is why MBL often is \( k \)-nearest \textit{distances} classification rather than \( k \)-nearest \textit{neighbor} classification. MBL has interesting advantages for NLP. Keeping all original training data available for classification allows handling exceptions in language. For instance, in certain languages, morphological operations on words, such as diminutive formation, can be arranged in \textit{pockets of exceptions}: small families of subregularities. In Dutch, for instance, we encounter

- \texttt{gat} \( \rightarrow \texttt{gaatje} \) (‘small hole’)
- \texttt{pad} \( \rightarrow \texttt{paadje} \) (‘small path’)
- \texttt{blad} \( \rightarrow \texttt{blaadje} \) (‘small leaf’)

Notice the extra vowel in the diminutive form, and the diminutive suffix \texttt{-je}. We also have a subfamily of patterns like

- \texttt{kat} \( \rightarrow \texttt{katje} \) (‘small cat’)
- \texttt{rat} \( \rightarrow \texttt{ratje} \) (‘small rat’)
- \texttt{schat} \( \rightarrow \texttt{schatje} \) (‘(my) dear’)
- \texttt{schot} \( \rightarrow \texttt{schotje} \) (‘small fence’)
- \texttt{schip} \( \rightarrow \texttt{scheepje} \) (‘small ship’)
- \texttt{schaft} \( \rightarrow \texttt{schaftje} \) (‘small ship’)
- \texttt{guit} \( \rightarrow \texttt{guitje} \) (‘little maverick’)

While \texttt{schat} is phonetically quite similar to \texttt{gat} (-\texttt{ch} is pronounced similarly as -\texttt{g}-), it is inflected analogous to other words prefixed with \texttt{sch-}. But \texttt{schip} is not following this pattern.

The benefit of using a memory-based learner for highly exceptional data is that exceptions can -in principle- always be retrieved for labeling a similar exceptional input case. Memory-based learners have perfect memory: they store everything. But \textit{eager} machine learning models tend to 'compile away' these exceptions. They are after string regularities rather than exceptions. Editing away these exceptions from training data has been found detrimental for the generalization accuracy (the ability to handle new, unseen cases outside of the training data) of the resulting classifier (DaelemansForgetting). Apparently, the subregularities of pockets of exceptions are beneficial to keep. One of the challenges for deep learning, an 'eager' form of machine learning, will be to handle these subregularities in a similar vein.
Deep learning is one of the most vibrant buzz phrases of the past 5 years. Deep learning by itself is nothing new: it is, in the strictest sense, a neural network with lots of internal or hidden layers and specific filtering operations. Deep learning deploys constructive ways of dealing with large quantities of information, organized in many layers of representations. While deep learning in its essential form was invented in the sixties of the previous century, it took three decades before it was finally ready to use. In this section, we will shed some light on why this happened.

But first: which problem does deep learning actually solve for NLP? Deep learning can be seen as a very effective statistical technique for working with (very) many parameters. Deep learning is able to handle effectively millions of parameters, each one encoding an aspect of input data. Layers in deep learning act as transformations that -step by step- accommodate input data with the labels we assign to that data; they *disentangle* the spaghetti of input data in such a way that labels can be assigned more easily. The fact that we can stack many such declutter steps on top of each other is a major *forte* of deep learning. For language, deep learning provides two types of advantages:

- The repeated application of data decluttering steps proves good for NLP, but this is not specific for language; it applies to virtual every modality deep learning is applied to.
- Deep learning has facilities for handling sequential information, with memory operators and buffers. This is quite important for language and in this respect, deep learning is a
form of *stateful* machine learning, as opposed to the usually *stateless* other types of machine learning models. These models also usually perform just a single disentanglement step (like Support Vector Machines).

![Figure 1.14 A timeline of neural learning. As we will point out, the gist of Deep Learning was actually invented in the 1960s.](https://www.slideshare.net/deview/251-implementing-deep-learning-using-cu-dnn)

Central to deep learning is the learning of hierarchical representations of data. Under a vertical interpretation of a multi-layer neural network, every 'lower' layer feeds into a 'higher' layer. Layers can be seen as complex functions processing a set of inputs and weights. These weights encode the importance of the information stored in the network. Networks receive their inputs in a dedicated input layer, and process that input layer-by-layer, sending it 'upwards' into the network.

Output layers, finally, produce an outcome: the label the model assigns to its input. All layers except input and output layers are 'hidden layers' as they cannot be readily observed. As mentioned, hidden layers in neural networks disentangle linearly inseparable input data layer-by-layer, step-by-step.

During training, weights are estimated and fine-tuned between 'neurons', which are the basic processing units of a neural network. Every layer keeps a record of the weights for the neurons that feed into that layer. Estimating weights is the essential business of neural networks.

Since the layers in a neural network are hierarchically organized (they are stacked), the representations they produce can be interpreted as hierarchical representations as well: going from specific (close to the input layer) to more abstract (close to the output layer). In general, it is hard to come up with human-understandable interpretations of these representations.
Yet, in the field of image analysis, the weights associated with the layers of deep networks have been shown to encode interpretable concepts. They can be visualized, and the lower layers appear to encode pixels, where higher layers represent edges, corners and finally even concepts like facial objects (see Figure 1).

In 1965, presumably the first reference to such hierarchical representation learning was published by Ivakhnenko and Lapa. Their paper describes a group method of data handling (GMDH): a method for producing outputs by layers feeding into each other, based on the following formula:

$$Y(x_1, \ldots, x_n) = a_0 + \sum_{i=1}^{n} a_i x_i + \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} x_i x_j + \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} a_{ijk} x_i x_j x_k + \ldots$$

This scary looking formula describes nothing but a function $Y$ that computes a complex sum of weighted combinations (groups) of input values, with every factor $a$ being a weight or coefficient. Notice that we have a one-dimensional weight or coefficient matrix $a_1$, a two-dimensional one $a_{ij}$, a three-dimensional one $a_{ijk}$, and so on. The weight matrices are usually referred to as 'polynomial models'. The coefficients can be learned from training data through simple methods such as least squares (wikipedia:ls). In a picture:
In 1971, the first "deep" 8-layer network based on GMDH was presented [ivakhnenko1971].

Yet, this memorable fact did not coincide with the official launch of deep learning. The simplistic methods for weight tuning by GMDH were not scalable to large scale training. It took a while before more scalable weight tuning methods like backpropagation came into being: around 1970 (see Section 2.4 of Deep learning with Python for an introduction to backpropagation).

But even with backpropagation, neural networks suffered from various practical problems, including the notorious vanishing gradient problem. This problem arises during training a network. The crucial ingredient of backpropagation is the stepwise minimization of the error function of a neural network, by taking partial derivatives of the error function of the network, differentiating for all weights, and moving stepwise towards its minimum. Gradients are a generalization of the one-variable partial derivative of a function. Setting a partial derivative to zero finds a local maximum or minimum of a function. If this function is an error function that computes the error a network makes for predicting certain output based on weights, we can look for weight adjustments that push the error function ever closer to its minimum. This procedure is called gradient descent, and it’s the driving force behind backpropagation. Backpropagation has built up an impressive track record, and underlies the majority of neural network results.

However, for deep and complex networks, with millions of weights, weight adjustments
can easily become too tiny to be useful: they just vanish, and gradient descent is no longer effective. Backpropagation deploys the so-called 'chain rule' from calculus for computing the weight adaptations per layer. The chain rule is an algorithm for computing derivatives of functions that are applied to functions. Essentially, this is what happens when we apply activation functions to the output of layers (which themselves apply activation functions to the output of the layers that feed into them, and so on).

As mentioned, gradient descent uses composed derivatives by working across layers and their respective activation functions, and makes according weight updates that move the network closer to perfection. Now, activation functions with gradients in intervals capped by small numbers (like between 0 and 1) result in weight adaptations that are small by nature, and repeated multiplication of these small numbers with the chain rule leads to thinning, and, eventually, evaporation of values. This means that the weight adaptations (which are computed from the top-most layer of the network, just 'under' the output layer) never reach the layers close to the input layer, which, subsequently do not get "trained".

So, how did deep learning escape from this conundrum? There are at least a couple of solutions that have alleviated the problem. The most prominent one is the use of a feature selection/data reconstruction cycle, as put forward by Restricted Boltzmann Machines (RBM’s). RBM’s are complete networks that learn probability distributions from data. They can be stacked on top of each other, as layers, where every layer is a separate RBM sending its hidden layer data as input to the next layer, and not through inter-layer connections between hidden layers. This set-up allows for layer-wise training of networks, and eliminates much of the vanishing gradient problem, since gradients don’t need to travel far down the network: they are confined to separate layers.

In addition to all this, a new type of activation function has become popular in Deep Learning: the rectified linear unit ReLU. A ReLU is a very simple non-linear function that computes a maximum of two values, one of which is the input to a neuron.

Specifically

\[ \text{ReLU}(x) = \max(0, x) \]

So, \( \text{ReLU}(x) \) just returns which one is bigger: zero or \( x \). It eliminates all values for \( x \) below zero.

Now, if we apply this function to every neuron in our network, only the ones with positive values promote their values:
• \( y = \text{ReLU}(\sum_i (\text{weight}_i * \text{input}_i) + \text{bias}) \)

This formula expresses the effect of ReLU applied to a sum (\( \sum \)) of inputs multiplied by weights, augmented with a bias term. Figure 3 shows how this works.

Figure 1.17 ReLU operations at work on the final hidden layer output of a neural net.

The ReLU function is differentiable almost everywhere but for exactly 0, and its derivative is quite simple:

- \( \text{ReLU}'(x) = 1 \) if \( x > 0 \) and 0 else

which has beneficiary effects on the speed and scalability of the network computations during backpropagation.

A traditional activation function is the sigmoid function:

- \( \text{sigmoid}(x) = 1/(1+e^{-x}) \)

To witness the dramatic effect the choice of an activation has on the performance of your neural network, let’s try out an overly deep network on a small snippet of sentiment data.

Scenario: You want to train a deep network on a sentiment labeling task. The task consists of labeling texts with sentiment labels: 1 for positive sentiment, and 0 for negative. You are unsure about which activation function you should choose. Can you find out experimentally the best option?
Our processing pipeline is as follows:

![Diagram of the processing pipeline]

Processing sentiment-labeled documents with a deep MLP.

**Figure 1.18 Labeling sentiment with a deep multilayer perceptron (MLP).**

Our data, taken from PAngLeePlotTokSUBJ looks like this: a set of sentences, labeled with either a zero (negative sentiment) or a one (positive sentiment)

<table>
<thead>
<tr>
<th>Review</th>
<th>Sentiment score</th>
</tr>
</thead>
<tbody>
<tr>
<td>smart and alert, thirteen conversations about one thing is a small gem</td>
<td>1</td>
</tr>
<tr>
<td>color, musical bounce and warm seas lapping on island shores and just enough science to send you home thinking</td>
<td>1</td>
</tr>
<tr>
<td>it is not a mass-market entertainment but an uncompromising attempt by one artist to think about another</td>
<td>1</td>
</tr>
<tr>
<td>a love affair with a veterinarian who is a non-practicing jew</td>
<td>1</td>
</tr>
<tr>
<td>initially reluctant to help, daniel's conscience eventually gets the better of him</td>
<td>0</td>
</tr>
<tr>
<td>his plans go awry, however, when his older brother, keith, returns from prison</td>
<td>0</td>
</tr>
<tr>
<td>inspired and motivated, the kids save the day, showing bravery and nobility</td>
<td>0</td>
</tr>
</tbody>
</table>

In raw format, our data looks like this. It is tab-delimited data, with a header containing the names of two columns:

```
"text"<tab>"label"
smart and alert, thirteen conversations about one thing is a small gem <tab>1
...
```

Our model is a ten-layer deep network, listed below. The model is trained on 90% of our training data while keeping 10% separate for validation purposes.
### Listing 1.4 A 10-layer deep MLP sentiment classifier with sigmoid activation functions.

```python
from keras.models import Sequential
from keras.utils import np_utils
from keras.preprocessing.text import Tokenizer
from keras.layers.core import Dense, Activation

import pandas as pd
import sys

data = pd.read_csv(sys.argv[1], sep='\t')
docs = data['text']
tokenizer = Tokenizer()
tokenizer.fit_on_texts(docs)

X_train = tokenizer.texts_to_matrix(docs, mode='binary')
y_train = np_utils.to_categorical(data['label'])

input_dim = X_train.shape[1]
b_classes = y_train.shape[1]

model = Sequential()
model.add(Dense(128, input_dim=input_dim))
model.add(Activation('sigmoid'))
model.add(Dense(128))
model.add(Activation('sigmoid'))
model.add(Dense(128))
model.add(Activation('sigmoid'))
model.add(Dense(128))
model.add(Activation('sigmoid'))
model.add(Dense(128))
model.add(Activation('sigmoid'))
model.add(Dense(128))
model.add(Activation('sigmoid'))
model.add(Dense(128))
model.add(Activation('sigmoid'))
model.add(Dense(128))
model.add(Activation('sigmoid'))
model.add(Dense(128))
model.add(Activation('sigmoid'))
model.add(Dense(b_classes))
model.add(Activation('softmax'))
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

print("Training...")
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.1,
          shuffle=False, verbose=2)
```

1. The pandas library has many handy functions for processing comma-separated (csv) and tab-delimited (tsv) data.
2. Our data is be tab-delimited: sentences separated by tabs from their labels. We read the data into a 'dataframe' (a primitive of pandas), and extract the "text" field (the column labeled with "text" in our data) as our document set.
3. The numpy library contains a function for
Keras has a Tokenizer facility for converting text into numerical vectors, consisting of unique integers referring to the original words.

We apply the Tokenizer to our documents.

Next, we generate vector representations of our documents. Together, they form a matrix. The tokenizer builds up a lexicon mapping words to integers, and generates binary vectors of a fixed dimension, with a '1' for every word in the input document, counting from zero. So, a '1' on position '3' means that word '2' in the lexicon is in the document. An example is presented below.

We infer the input size (dimension) from our vectorized data, and the number of classes.

The network contains 10 Dense layers (standard, fully-connected layers), and deploys sigmoid activation functions that pass the incoming connections into each neuron through the sigmoid function.

The output layer is a dense layer with as many neurons as the number of classes.

The softmax activation function generates output probabilities.

We compile the model.

The model is fitted on our data, using 10% of the training data as held-out data for testing.

Here is an example of a Keras Tokenizer vectorizing text data:

```python
>>> docs = ['smart and alert, thirteen conversations about one thing is a small gem', 'not very smart movie']
>>> tok=Tokenizer()
>>> tok.fit_on_texts(docs)
>>> tok.texts_to_matrix(doc, mode='binary')
array([[0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
>>> tok.word_index
{'and': 1, 'a': 9, 'about': 5, 'very': 15, 'not': 14, 'conversations': 4, 'is': 8, 'one': 6, 'mart': 13, 'thing': 7, 'thirteen': 3, 'movie': 16, 'small': 10, 'alert': 2, 'gem': 11, 'smart': 12}
```

On our data, the model produces the following output:

```
Train on 1800 samples, validate on 200 samples
Epoch 1/10
2s - loss: 0.7079 - acc: 0.5078 - val_loss: 0.6937 - val_acc: 0.5200
Epoch 2/10
1s - loss: 0.6983 - acc: 0.5144 - val_loss: 0.6938 - val_acc: 0.5200
Epoch 3/10
1s - loss: 0.6984 - acc: 0.5100 - val_loss: 0.6955 - val_acc: 0.5200
Epoch 4/10
1s - loss: 0.6988 - acc: 0.5000 - val_loss: 0.6979 - val_acc: 0.5200
Epoch 5/10
1s - loss: 0.6994 - acc: 0.4922 - val_loss: 0.6994 - val_acc: 0.5200
Epoch 6/10
1s - loss: 0.6999 - acc: 0.4989 - val_loss: 0.6986 - val_acc: 0.5200
Epoch 7/10
1s - loss: 0.6999 - acc: 0.4978 - val_loss: 0.6966 - val_acc: 0.5200
```
For an introduction to Keras, refer to Appendix 3. The network doesn’t seem to learn at all: its validation accuracy (the accuracy it attains on a held out test portion of its training data, during training) does not increase. You can see this in the above listing: the val_acc value remains fixed at 0.52 throughout training. Further, the accuracy of the classifier (acc) as computed on its own training data, fluctuates around 50%, and doesn’t seem to get any better.

Now, let’s compare this to a network with exactly the same structure, but now equipped with ReLU activation functions:

```
Listing 1.5 A deep MLP sentiment classifier with ReLU activation functions.
...
model = Sequential()
model.add(Dense(128, input_dim=input_dim))
model.add(Activation('relu'))
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dense(128))
...
```

This produces

```
Epoch 1/10
2s - loss: 0.6042 - acc: 0.6128 - val_loss: 0.3713 - val_acc: 0.8350
Epoch 2/10
1s - loss: 0.1335 - acc: 0.9478 - val_loss: 0.5356 - val_acc: 0.8250
Epoch 3/10
1s - loss: 0.0073 - acc: 0.9983 - val_loss: 0.9263 - val_acc: 0.8500
Epoch 4/10
1s - loss: 1.3958e-05 - acc: 1.0000 - val_loss: 0.9707 - val_acc: 0.8550
Epoch 5/10
1s - loss: 6.7025e-06 - acc: 1.0000 - val_loss: 1.0057 - val_acc: 0.8550
Epoch 6/10
1s - loss: 4.2353e-06 - acc: 1.0000 - val_loss: 1.0420 - val_acc: 0.8550
Epoch 7/10
1s - loss: 2.8474e-06 - acc: 1.0000 - val_loss: 1.0798 - val_acc: 0.8500
Epoch 8/10
1s - loss: 2.0100e-06 - acc: 1.0000 - val_loss: 1.1124 - val_acc: 0.8500
Epoch 9/10
1s - loss: 1.4673e-06 - acc: 1.0000 - val_loss: 1.1427 - val_acc: 0.8500
Epoch 10/10
1s - loss: 1.1042e-06 - acc: 1.0000 - val_loss: 1.1698 - val_acc: 0.8500
```

Much better! We now obtain a score of 85% on the 10% held-out portion of our data during training, and even a 100% accuracy score on our training data (which actually
may mean we’re overfitting out model). (Check that the results of the ReLU network are consistently better than those of the sigmoid network by running both networks a number of times.)

In the next chapter, we will explore the technology of deep learning further. Deep learning needs data in vectorized formats. How do we turn textual data into such representations?

1.4 Vector representations of language

Most machine learning algorithms work with vectors: fixed-size containers (sequences) of usually numerical values. In this section, we address procedures for generating these vectors from text. From a mathematical point of view, these vectors correspond to points in multi-dimensional spaces. Machine learning is all about measuring distances between objects (points) in these spaces, which, for typical machine learning applications like text mining, are very high-dimensional, and, as such, escape our human intuition in terms of geometry. In order to convert texts to vectors, quite a number of approaches are possible. There are roughly two types of vector representations of language:

- Representational vectors
- Operational vectors

As mentioned, we will focus mainly on representational vectors in this section. These are
vector representations that are not learned from data.

1.4.1 Representational vectors

Representational vectors represent texts by describing them across a number of human-interpretable feature dimensions. For text, trivially, representing words by characters is the most simple form of such a vector. An example would be 10-dimensional character-valued vectors:

- vector-

But usually, we deploy more elaborate representations. Consider again Dutch diminutive formation in lexical morphology. Dutch uses diminutive suffixes

- `hospitaal` → `hospitaal+tje` ('small hospital')
- `woning` → `wonin+kje` ('small house')

These suffixes are conditioned on a fixed number of phonetic (sound), morphological (word formation) and orthographic (spelling) features of the suffix (ending) of a word, plus similar features of the preceding context. An imaginable feature vector with 12 dimensions (separated below with commas) plus a class label for the example above would be:

- +,h,O,s,-,p,i,=-,-,t,a,l,T

with the "+,-" indicating phonetic stress or no stress on the syllables `hos`, `pi`, and `taal`, and the other features denoting phonemic representations of the characters in the word. The T label indicates the type of diminutive suffix (here: `-tje`).

In our representation, words that are shorter than 12 characters obtain dummy values for absent dimensions:

- =,=,=,=,=,=,=,=,=,=,=,v,A,xt,J ('vacht', fur)

and words that exceed 12 characters become truncated:

- +,G,e,t,-,m,@,=-,-,n,i,t,J ('…geetmenietje', from 'vergeetmenietje' (forget-me-not))

The bag-of-words representations that we discussed above are also representational in the sense that every dimension can be interpreted as representing a clear feature dimension: the presence (binary-valued, yes/no) of a certain word in an index lexicon.
As an example, the following sentences

- Natural language is hard for computers.
- Computers are capable of learning natural language.

can be represented by binary vectors as follows. First, a lexicon consisting of the set of different words is created, each word linked to a unique identifier.

**Table 1.2 A simple lexicon pairing words with identifiers.**

<table>
<thead>
<tr>
<th>Word</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>are</td>
<td>1</td>
</tr>
<tr>
<td>capable</td>
<td>2</td>
</tr>
<tr>
<td>computers</td>
<td>3</td>
</tr>
<tr>
<td>for</td>
<td>4</td>
</tr>
<tr>
<td>hard</td>
<td>5</td>
</tr>
<tr>
<td>is</td>
<td>6</td>
</tr>
<tr>
<td>language</td>
<td>7</td>
</tr>
<tr>
<td>learning</td>
<td>8</td>
</tr>
<tr>
<td>natural</td>
<td>9</td>
</tr>
<tr>
<td>of</td>
<td>10</td>
</tr>
</tbody>
</table>

Based on this lexicon, we index the two sentences like this:

<table>
<thead>
<tr>
<th>Word</th>
<th>are</th>
<th>capable</th>
<th>computer</th>
<th>for</th>
<th>hard</th>
<th>is</th>
<th>language</th>
<th>learning</th>
<th>natural</th>
<th>of</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

and

<table>
<thead>
<tr>
<th>Word</th>
<th>are</th>
<th>capable</th>
<th>computer</th>
<th>for</th>
<th>hard</th>
<th>is</th>
<th>language</th>
<th>learning</th>
<th>natural</th>
<th>of</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

with every 0 or 1 at a certain position $i$ indicating the absence or presence of the $i$-th word in the lexicon in the sentence.

A simple Python example for computing such binary bag-of-words representations is listed in [bow1](https://github.com/nltk/nltk/blob/master/nltk/data/tests/unit/test_tokenization/test_bow.py). This example creates a count vector for documents in a tab-separated text file, with a structure like:

```
text label
"Natural language is hard for computers." 0
"Computers are capable of learning natural language." 1
```

where the labels are the different classes the different texts reside in. Every document becomes represented with a vector of fixed size, with at every position a count for a specific word bound to that position. The algorithm fits a dictionary function called a `CountVectorizer` (a construct native to the SciKit library `sklearn`, see 2) to the documents,
capped to the 1,000 most recent words. Effectively, these 1,000 words form the lexicon
by which texts are indexed, leading to vectors of maximum size 1,000 (if the number of
different words in the underlying documents is \(N<1,000\), the vector size will be \(N\)).

### Listing 1.6 Bag-of-words

```python
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

trainingdata = pd.read_csv("train.tsv", header=0, encoding='utf-8', delimiter="\t")

cv = CountVectorizer(analyzer = "word",   
    tokenizer = None,    
    preprocessor = None, 
    stop_words = None,   
    max_features = 1000)

docvec=cv.fit_transform(trainingdata["text"]).toarray()

print docvec

print cv.vocabulary_
```

1. We import the pandas library, for reading our tab-separated file. We will use
   pandas a lot in this book.
2. From sklearn, we import the CountVectorizer.
3. We have pandas read our input data into a data frame. This data frame is
   essentially a hash, and our column labeled with "text" can be easily retrieved from
   it.
4. We define a CountVectorizer, with no special operations regaring tokenization or
   handling stopwords. The lexicon size is set to 1,000, meaning we have at most
   vectors of 1,000 elements long. In our case, they’re much shorter!
5. We ‘fit’ the vectorizer on our data, meaning it induces a lexicon from it. This
   lexicon contains mappings of words to vector positions. Every word gets a unique
   position in the result vectors. We transform (vectorize) our documents in one go.
6. We print the vectorized documents.
7. We print the lexicon of the CountVectorizer.

This produces

```
[ [ 0 0 1 1 1 1 1 0 1 0 ]
 [ 1 1 1 0 0 0 1 1 1 ]
]
(u'hard': 4, u'natural': 8, u'for': 3, u'language': 6, u'capable': 1, u'of': 9, u'is': 5,
 u'computers': 2, u'are': 0, u'learning': 7)
```

with two vectorized documents, and a dictionary linking positions (starting at zero) in
each vector to words. For instance, word position zero is for 'are'; only the second
document has a '1' on this position. (If there would be two instances of 'are' in the second
document, we would have a value of 2 here.)

One-hot vector representations of words consist of a sparsely populated \( N \)-dimensional vector, with \( N \) the size of a lexicon. Only one of the dimensions has value 1, the dimension corresponding to a given word. For instance, given a lexicon of 100,000 words, every word in a document is represented by an unwieldy 100,000-dimensional vector with just one digit 'on'.

1.4.2 Operational vectors

Operational vector representations reflect a derived representation of data, as produced by some algorithm. Typically, these vectors are not human-interpretable as they are produced by irreversible computations producing numerical vectors.

An example is the \( \text{tf.idf} \) vector representation. With this representation, words are weighted with numerical scores consisting of a product of term frequency and inverse document frequency. These scores express a degree of salience: lower weighted words like stopwords are less salient (or 'special').

Document representations based on \( \text{tf.idf} \) weighting help machine learning algorithms to zoom in on important words. Virtually every machine learning computes (often implicitly) similarities between vectors, and stressing important dimensions (like salient words) while downplaying others (like stopwords) can contribute to fine-grained similarity estimates.

The term frequency quantity expresses the frequency of a given word in the document to be represented:

\[
\text{tf}(w \mid d) = | \{ w \in d \} |
\]

This simply defines the term frequency of a word \( w \) given a document \( d \) as the amount of times the word \( w \) occurs in \( d \).

The inverse document frequency describes the frequency of the word in other documents in a given document collection \( D \) consisting of documents \( d \):

\[
\text{idf}(w \mid d, D) = \log \frac{|D|}{|\{ d \in D : w \in d \}|}
\]

The product of these two quantities, \( \text{tf.idf} \), produces a number that balances the frequency of a word with the number of documents it appears in:
\[ tf.idf(w \mid d, D) = tf(w \mid d) \times idf(w \mid d, D) \]

Whenever the latter number is high (meaning a given word is quite common), the \( idf \) quantity will approach zero, since \( \log(1) = 0 \) and the ratio of the log will reach 1 when \( d \) occurs in all documents in the collection \( D \). The \( idf \) factor thus effectively suppresses the contribution of the frequency of the word to the \( tf.idf \) weight. This implies that stop words like \( \text{the, is, what} \) usually (depending on the document collection \( D \)) will have low \( tf.idf \) scores, whereas peculiar words will typically produce high scores.

Binary bag-of-words vectors can be augmented with frequencies:

\[ BOW(d = w_1, \ldots, w_n \mid L) = \vec{v}[i] \rightarrow tf(w_i \mid d) \text{ if } w_i \in L \]

or with \( tf.idf \):

\[ BOW(d = w_1, \ldots, w_n \mid L, D) = \vec{v}[i] \rightarrow tf.idf(w_i \mid d) \text{ if } w_i \in L \]

Another type of operational vectors for texts is produced by neural networks: neural word embeddings.

**NEURAL WORD EMBEDDINGS**

Neural word embeddings are easily one of the most important inventions in text mining of the last decade. These embeddings, commonly known as \textit{word2vec} models (Mikolov), generate operational vector representations of words. Embeddings are produced by neural networks that predict words given a certain context or, vice versa, a context given a certain word. The inputs and outputs to these networks usually consist of one-hot vectors. For instance, the latter variant has the following structure:
with input vectors representing a single word of dimension $V$, a hidden unit of dimension $N$ and a composed output layer of dimension $M$, with subvectors corresponding to predicted context words. Two weight matrices $W$ and $W'$ encode the learned weights. Once the network is trained on large amounts of words mapping to contexts, the hidden layer serves as a vector representation of each input word. We will discuss this in much more detail in Chapter 3.

During training, the network fine-tunes hidden layer representations that can represent words in both a compact and semantically meaningful way: two words that share similar contexts are associated with vectors that are relatively similar. This is a form of distributional semantic similarity. Neural networks that create word embeddings are by no means deep: they are in fact shallow, typically consisting of one hidden layer and one input layer. Word embeddings have become very popular as input to deep neural networks, however.

*Let’s assume we are building a sentiment classifier. The classifier needs to work with fixed size vectors, and every vector represents a short document, say a tweet. *

We are able to generate neural word embeddings for every word in our tweets by using a freely available large set of 300-dimensional vectors produced by Google on a one billion
words corpus of news texts. One way of combining the separate word embeddings is by averaging: we sum all \(N\)-dimensional vectors, again producing an \(N\)-dimensional result vector, and divide every component of that vector by \(K\), with \(K\) being the number of words in our tweet. So, the tweet is represented with an average word embedding, which, geometrically, corresponds to the centroid of the vector space spanned by the various word embeddings vectors of its words.

The process is depicted in the following diagram:

![Diagram](image)

**Figure 1.21** Averaged word2vec vectors as document representations.

The following code snippet does exactly all that. It starts with defining an array of tweets, in this case a few tweets by Donald Trump. Subsequently, using the *gensim* open source library (see the Appendix), it loads the Google news-based word2vec model storing 3—dimensional word vectors (it’s huge, so this takes a while). After that, it creates a vector container, to which it adds an averaged word2vec vector for every tweet. These vectors could serve as training data by linking them to adequate labels such as topics or sentiments.
Listing 1.7 Average word2vec vectors

```python
import gensim
import numpy as np

# A list of tweets
tweets=["With the great vote on Cutting Taxes, this could be a big day for the Stock
Market - and YOU","Putting Pelosi/Schumer Liberal Puppet Jones into office in
Alabama would hurt our great Republican Agenda of low on taxes, tough on crime,
strong on military and borders...& so much more. Look at your 401-k’s since
Election. Highest Stock Market EVER! Jobs are roaring back!",
...

# Load Google model of news vectors, 300 dimensions.
model = gensim.models.Word2Vec.load_word2vec_format('GoogleNews-vectors-negative300.bin',
binary=True)

# Create a result vector
vectA=[]

for tweet in tweets:
    vect=np.zeros(300)
    n=0
    for word in tweet.split(" 
    if word in model.wv:
        vect=np.add(vect, model.wv[word])
        n+=1
    vect=np.divide(vect,n)
    vectA.append(vect)

return vectA
```

1. A list of tweets.
2. We load a pre-trained word2vec model from Google, consisting of 300-dimensional vectors derived from news data.
3. We create a result vector, consisting of zeros.
4. We declare a counter for the number of words in a tweet we find in our word2vec model
5. To the result vector, we add the word2vec vectors of the words in the tweet, if those words are in the word2vec model. The gensim method `wv` defined on word2vec models performs that check.
6. We normalize the vector by dividing the aggregated word2vec contributions by the number of words that were found in the model.

This produces normalized word2vec vectors for tweets. It turns out these representations are adequate for performing all kinds of analyses, like sentiment classification. More on that later; in Chapter 3, we will actually build word2vec embeddings ourselves rather than using pre-built embeddings.
1.5 Vector sanitization

Either representational or operational, vectors can be sanitized or optimized by a number of post-processing procedures. We will discuss two of them: dimensionality reduction by hashing, and normalization.

1.5.1 The Hashing trick

Large vectors are unwieldy to handle, they take up lots of memory, and may consist of many irrelevant, sparsely populated dimensions. Imagine indexing documents on the basis of a very large lexicon, say 100,000 words. This would, even for the shortest text, lead to a vector representation of 100,000 dimensions. Representing every binary digit (0,1) with one bit, every vector claims 12,500 bytes, or 12 kilobyte. Now imagine your training data for, say, a topic classifier, contains 100,000 labeled example documents. This produces roughly a 1.2 gigabyte dataset, which is a hefty size, even for current hardware. As it turns out, there is a handy option available for reducing dimensionality through algorithmic processing: the hashing trick, also known as feature hashing.

The hashing trick is an algorithm for reducing the dimensionality of large feature vectors, by applying a hashing function to the features in the vectors. The hashing function maps every feature to an index, and the algorithm updates the information at those indices only.

Listing 1.8 Feature hashing.

```python
def feat_hash(featureV, vecSize):
    outputV = numpy.array(vecSize)
    for f in range(0, len(featureV)):
        if featureV[f] == 1:
            dim = hash_function(InverseLexicon[f])
            outputV[dim % vecSize] += 1
    return outputV
```

In this code snippet, assume we have an inverted lexicon `InverseLexicon`, which maps an integer to a word (rather than mapping words to indices). This inverse lexicon restores for the positive indices in the binary-valued input vector `featureV` a hash value given some hashing function. The value is capped to fall in the range `0...vecSize` (with `vecSize` the desired size of the output vector). In this way, the indexing is performed by the hashing function: similar input values will lead to similar numerical indices being incremented. The amount of similarity is catered for by the specific hash function chosen. Collisions may occur, where different features share the same indices, due to the use of the `modulo` function:

- 5 mod 4 = 1
- 9 mod 4 = 1
but these can be handled graciously with an extra hashing function (see e.g. en.wikipedia.org/wiki/Feature_hashing).

### 1.5.2 Vector normalization

Vectors represent quantities in vector space. It is intuitive to think about vectors as arrows with both magnitude (the length of an arrow) and direction (the angle of an arrow with respect to the origin of the space they live in):

![Vectors in 2D space](image)

**Figure 1.22 Vectors represented in 2D space.**

The magnitude of a vector $v$ can be computed using Pythagoras' theorem:

$$||v|| = \sqrt{\sum_{i} v_i^2}$$

Dividing every component of the vector $v$ by its magnitude forces the magnitude of the vector to be exactly 1:

Vectors containing numeric information (like the tf-idf vectors we came across) can be *normalized*. This means that they are squeezed into a subspace, reducing the variance across their dimensions. In the following picture, all vectors have been normalized to a magnitude of 1:
In this picture, all vectors A, B, and C have been normalized by dividing all components of every vector by their magnitude. For a vector \( v \), its normalized version can be expressed as:

\[
\hat{v} = \frac{v}{\|v\|}
\]

Any such normalized vector is called a unit vector. Why bother with normalization? Because it forces vectors to lie in the same data range, aiding any machine learning algorithm that is sensitive to outlier data.

### 1.6 Wrapping up

In this chapter, you learned about

- Different forms of NLP through a global tour addressing examples from 'deep linguistics' and 'shallow linguistics'.
- Machine learning methods relevant for NLP, such as kernel machines, memory-based learning, perceptrons and statistical approaches to NLP. We implemented a perceptron performing topic analysis.
- Vector representations of language for machine learning purposes, including word2vec models. You learned how to create simple binary vector representations of texts.

Now that we’ve gotten all of that out of our way, it is time we actually turn to deep learning. The next chapter discusses the fundamental architectures of deep learning for
NLP. We will return to the topic of vector representations in Chapter 3, where we discuss operational vector representations (embeddings) of texts.

In this book, we will sometimes draw pragmatic inspiration or re-use code snippets from public domain open source code. Specific sources of inspiration include:

- The Keras source code base, containing many examples addressing natural language processing
- The code accompanying the companion book *Deep learning with Python* by Manning.
- Popular open source websites like adventuresinmachinelearning.com, machinelearningmastery.com, blogs like karpathy.github.io and communities like StackOverflow.

Since open source coding is much like an *echo chamber*, with code being routinely re-used and shared, it is often impossible to trace back original contributions.

### 1.7 References