# Neural Network Methods for Natural Language Processing

# Synthesis Lectures on Human Language Technologies

Neural Network Methods for Natural Language Processing

Yoav Goldberg

www.morganclaypool.com

ISBN: 9781627052986 paperback
ISBN: 9781627052955 ebook

DOI 10.2200/S00762ED1V01Y201703HLT037

A Publication in the Morgan & Claypool Publishers series
*SYNTHESIS LECTURES ON HUMAN LANGUAGE TECHNOLOGIES*

Lecture #37
Series Editor: Graeme Hirst, *University of Toronto*
Series ISSN
Print 1947-4040 Electronic 1947-4059

# Neural Network Methods for Natural Language Processing

Yoav Goldberg
Bar Ilan University

*SYNTHESIS LECTURES ON HUMAN LANGUAGE TECHNOLOGIES #37*

MORGAN & CLAYPOOL PUBLISHERS

## ABSTRACT

Neural networks are a family of powerful machine learning models. This book focuses on the application of neural network models to natural language data. The first half of the book (Parts I and II) covers the basics of supervised machine learning and feed-forward neural networks, the basics of working with machine learning over language data, and the use of vector-based rather than symbolic representations for words. It also covers the computation-graph abstraction, which allows to easily define and train arbitrary neural networks, and is the basis behind the design of contemporary neural network software libraries.

The second part of the book (Parts III and IV) introduces more specialized neural network architectures, including 1D convolutional neural networks, recurrent neural networks, conditioned-generation models, and attention-based models. These architectures and techniques are the driving force behind state-of-the-art algorithms for machine translation, syntactic parsing, and many other applications. Finally, we also discuss tree-shaped networks, structured prediction, and the prospects of multi-task learning.

## KEYWORDS

natural language processing, machine learning, supervised learning, deep learning, neural networks, word embeddings, recurrent neural networks, sequence to sequence models

# Contents

# Preface

Natural language processing (NLP) is a collective term referring to automatic computational processing of human languages. This includes both algorithms that take human-produced text as input, and algorithms that produce natural looking text as outputs. The need for such algorithms is ever increasing: human produce ever increasing amounts of text each year, and expect computer interfaces to communicate with them in their own language. Natural language processing is also very challenging, as human language is inherently ambiguous, ever changing, and not well defined.

Natural language is symbolic in nature, and the first attempts at processing language were symbolic: based on logic, rules, and ontologies. However, natural language is also highly ambiguous and highly variable, calling for a more statistical algorithmic approach. Indeed, the current-day dominant approaches to language processing are all based on *statistical machine learning*. For over a decade, core NLP techniques were dominated by linear modeling approaches to supervised learning, centered around algorithms such as Perceptrons, linear Support Vector Machines, and Logistic Regression, trained over very high dimensional yet very sparse feature vectors.

Around 2014, the field has started to see some success in switching from such linear models over sparse inputs to *nonlinear neural network models* over dense inputs. Some of the neural-network techniques are simple generalizations of the linear models and can be used as almost drop-in replacements for the linear classifiers. Others are more advanced, require a change of mindset, and provide new modeling opportunities. In particular, a family of approaches based on *recurrent neural networks* (RNNs) alleviates the reliance on the Markov Assumption that was prevalent in sequence models, allowing to condition on arbitrarily long sequences and produce effective feature extractors. These advances led to breakthroughs in language modeling, automatic machine translation, and various other applications.

While powerful, the neural network methods exhibit a rather strong barrier of entry, for various reasons. In this book, I attempt to provide NLP practitioners as well as newcomers with the basic background, jargon, tools, and methodologies that will allow them to understand the principles behind neural network models for language, and apply them in their own work. I also hope to provide machine learning and neural network practitioners with the background, jargon, tools, and mindset that will allow them to effectively work with language data.

Finally, I hope this book can also serve a relatively gentle (if somewhat incomplete) introduction to both NLP and machine learning for people who are newcomers to both fields.

## INTENDED READERSHIP

This book is aimed at readers with a technical background in computer science or a related field, who want to get up to speed with neural network techniques for natural language processing. While the primary audience of the book is graduate students in language processing and machine learning, I made an effort to make it useful also to established researchers in either NLP or machine learning (by including some advanced material), and to people without prior exposure to either machine learning or NLP (by covering the basics from the grounds up). This last group of people will, obviously, need to work harder.

While the book is self contained, I do assume knowledge of mathematics, in particular undergraduate level of probability, algebra, and calculus, as well as basic knowledge of algorithms and data structures. Prior exposure to machine learning is very helpful, but not required.

This book evolved out of a survey paper [Goldberg, 2016], which was greatly expanded and somewhat re-organized to provide a more comprehensive exposition, and more in-depth coverage of some topics that were left out of the survey for various reasons. This book also contains many more concrete examples of applications of neural networks to language data that do not exist in the survey. While this book is intended to be useful also for people without NLP or machine learning backgrounds, the survey paper assumes knowledge in the field. Indeed, readers who are familiar with natural language processing as practiced between roughly 2006 and 2014, with heavy reliance on machine learning and linear models, may find the journal version quicker to read and better organized for their needs. However, such readers may also appreciate reading the chapters on word embeddings (10 and 11), the chapter on conditioned generation with RNNs (17), and the chapters on structured prediction and multi-task learning (MTL) (19 and 20).

## FOCUS OF THIS BOOK

This book is intended to be self-contained, while presenting the different approaches under a unified notation and framework. However, the main purpose of the book is in introducing the neural-networks (deep-learning) machinery and its application to language data, and not in providing an in-depth coverage of the basics of machine learning theory and natural language technology. I refer the reader to external sources when these are needed.

Likewise, the book is not intended as a comprehensive resource for those who will go on and develop the next advances in neural network machinery (although it may serve as a good entry point). Rather, it is aimed at those readers who are interested in taking the existing, useful technology and applying it in useful and creative ways to their favorite language-processing problems.

Further reading For in-depth, general discussion of neural networks, the theory behind them, advanced optimization methods, and other advanced topics, the reader is referred to other existing resources. In particular, the book by Bengio et al. [2016] is highly recommended.

For a friendly yet rigorous introduction to practical machine learning, the freely available book of Daumé III [2015] is highly recommended. For more theoretical treatment of machine learning, see the freely available textbook of Shalev-Shwartz and Ben-David [2014] and the textbook of Mohri et al. [2012].

For a strong introduction to NLP, see the book of Jurafsky and Martin [2008]. The information retrieval book by Manning et al. [2008] also contains relevant information for working with language data.

Finally, for getting up-to-speed with linguistic background, the book of Bender [2013] in this series provides a concise but comprehensive coverage, directed at computationally minded readers. The first chapters of the introductory grammar book by Sag et al. [2003] are also worth reading.

As of this writing, the progress of research in neural networks and Deep Learning is very fast paced. The state-of-the-art is a moving target, and I cannot hope to stay up-to-date with the latest-and-greatest. The focus is thus with covering the more established and robust techniques, that were proven to work well in several occasions, as well as selected techniques that are not yet fully functional but that I find to be established and/or promising enough for inclusion.

Yoav Goldberg
March 2017

# Acknowledgments

This book grew out of a survey paper I've written on the topic [Goldberg, 2016], which in turn grew out of my frustration with the lack organized and clear material on the intersection of deep learning and natural language processing, as I was trying to learn it and teach it to my students and collaborators. I am thus indebted to the numerous people who commented on the survey paper (in its various forms, from initial drafts to post-publication comments), as well as to the people who commented on various stages of the book's draft. Some commented in person, some over email, and some in random conversations on Twitter. The book was also influenced by people who did not comment on it per-se (indeed, some never read it) but discussed topics related to it. Some are deep learning experts, some are NLP experts, some are both, and others were trying to learn both topics. Some (few) contributed through very detailed comments, others by discussing small details, others in between. But each of them influenced the final form of the book. They are, in alphabetical order: Yoav Artzi, Yonatan Aumann, Jason Baldridge, Miguel Ballesteros, Mohit Bansal, Marco Baroni, Tal Baumel, Sam Bowman, Jordan Boyd-Graber, Chris Brockett, Ming-Wei Chang, David Chiang, Kyunghyun Cho, Grzegorz Chrupala, Alexander Clark, Raphael Cohen, Ryan Cotterell, Hal Daumé III, Nicholas Dronen, Chris Dyer, Jacob Eisenstein, Jason Eisner, Michael Elhadad, Yad Faeq, Manaal Faruqui, Amir Globerson, Fréderic Godin, Edward Grefenstette, Matthew Honnibal, Dirk Hovy, Moshe Koppel, Angeliki Lazaridou, Tal Linzen, Thang Luong, Chris Manning, Stephen Merity, Paul Michel, Margaret Mitchell, Piero Molino, Graham Neubig, Joakim Nivre, Brendan O'Connor, Nikos Pappas, Fernando Pereira, Barbara Plank, Ana-Maria Popescu, Delip Rao, Tim Rocktäschel, Dan Roth, Alexander Rush, Naomi Saphra, Djamé Seddah, Erel Segal-Halevi, Avi Shmidman, Shaltiel Shmidman, Noah Smith, Anders Søgaard, Abe Stanway, Emma Strubell, Sandeep Subramanian, Liling Tan, Reut Tsarfaty, Peter Turney, Tim Vieira, Oriol Vinyals, Andreas Vlachos, Wenpeng Yin, and Torsten Zesch.

The list excludes, of course, the very many researchers I've communicated with through their academic writings on the topic.

The book also benefited a lot from—and was shaped by—my interaction with the Natural Language Processing Group at Bar-Ilan University (and its soft extensions): Yossi Adi, Roee Aharoni, Oded Avraham, Ido Dagan, Jessica Ficler, Jacob Goldberger, Hila Gonen, Joseph Keshet, Eliyahu Kiperwasser, Ron Konigsberg, Omer Levy, Oren Melamud, Gabriel Stanovsky, Ori Shapira, Micah Shlain, Vered Shwartz, Hillel Taub-Tabib, and Rachel Wities. Most of them belong in both lists, but I tried to keep things short.

The anonymous reviewers of the book and the survey paper—while unnamed (and sometimes annoying)—provided a solid set of comments, suggestions, and corrections, which I can safely say dramatically improved many aspects of the final product. Thanks, whoever you are!

And thanks also to Graeme Hirst, Michael Morgan, Samantha Draper, and C.L. Tondo for orchestrating the effort.

As usual, all mistakes are of course my own. Do let me know if you find any, though, and be listed in the next edition if one is ever made.

Finally, I would like to thank my wife, Noa, who was patient and supportive when I disappeared into writing sprees, my parents Esther and Avner and brother Nadav who were in many cases more excited about the idea of me writing a book than I was, and the staff at The Streets Cafe (King George branch) and Shne'or Cafe who kept me well fed and served me drinks throughout the writing process, with only very minimal distractions.

Yoav Goldberg
March 2017

CHAPTER 1

# Introduction

## 1.1 THE CHALLENGES OF NATURAL LANGUAGE PROCESSING

Natural language processing (NLP) is the field of designing methods and algorithms that take as input or produce as output unstructured, natural language data. Human language is highly ambiguous (consider the sentence *I ate pizza with friends*, and compare it to *I ate pizza with olives*), and also highly variable (the core message of *I ate pizza with friends* can also be expressed as *friends and I shared some pizza*). It is also ever changing and evolving. People are great at producing language and understanding language, and are capable of expressing, perceiving, and interpreting very elaborate and nuanced meanings. At the same time, while we humans are great *users* of language, we are also very poor at formally understanding and describing the rules that *govern* language.

Understanding and producing language using computers is thus highly challenging. Indeed, the best known set of methods for dealing with language data are using *supervised machine learning* algorithms, that attempt to infer usage patterns and regularities from a set of pre-annotated input and output pairs. Consider, for example, the task of classifying a document into one of four categories: Sports, Politics, Gossip, and Economy. Obviously, the words in the documents provide very strong hints, but which words provide what hints? Writing up rules for this task is rather challenging. However, readers can easily categorize a document into its topic, and then, based on a few hundred human-categorized examples in each category, let a supervised machine learning algorithm come up with the patterns of word usage that help categorize the documents. Machine learning methods excel at problem domains where a good set of rules is very hard to define but annotating the expected output for a given input is relatively simple.

Besides the challenges of dealing with ambiguous and variable inputs in a system with ill-defined and unspecified set of rules, natural language exhibits an additional set of properties that make it even more challenging for computational approaches, including machine learning: it is *discrete*, *compositional*, and *sparse*.

Language is symbolic and discrete. The basic elements of written language are characters. Characters form words that in turn denote objects, concepts, events, actions, and ideas. Both characters and words are discrete symbols: words such as "hamburger" or "pizza" each evoke in us a certain mental representations, but they are also distinct symbols, whose meaning is external to them and left to be interpreted in our heads. There is no inherent relation between "hamburger" and "pizza" that can be inferred from the symbols themselves, or from the individual letters they are made of. Compare that to concepts such as color, prevalent in machine vision, or acoustic signals: these concepts are continuous, allowing, for example, to move from a colorful image to a gray-scale one using a simple mathematical operation, or to compare two different colors based on inherent properties such as hue and intensity. This cannot be easily done with words—there is no simple operation that will allow us to move from the word "red" to the word "pink" without using a large lookup table or a dictionary.

Language is also compositional: letters form words, and words form phrases and sentences. The meaning of a phrase can be larger than the meaning of the individual words that comprise it, and follows a set of intricate rules. In order to interpret a text, we thus need to work beyond the level of letters and words, and look at long sequences of words such as sentences, or even complete documents.

The combination of the above properties leads to *data sparseness*. The way in which words (discrete symbols) can be combined to form meanings is practically infinite. The number of possible valid sentences is tremendous: we could never hope to enumerate all of them. Open a random book, and the vast majority of sentences within it you have not seen or heard before. Moreover, it is likely that many sequences of four-words that appear in the book are also novel to you. If you were to look at a newspaper from just 10 years ago, or imagine one 10 years in the future, many of the words, in particular names of persons, brands, and corporations, but also slang words and technical terms, will be novel as well. There is no clear way of generalizing from one sentence to another, or defining the similarity between sentences, that does not depend on their meaning—which is unobserved to us. This is very challenging when we come to learn from examples: even with a huge example set we are very likely to observe events that never occurred in the example set, and that are very different than all the examples that did occur in it.

## 1.2 NEURAL NETWORKS AND DEEP LEARNING

Deep learning is a branch of machine learning. It is a re-branded name for neural networks—a family of learning techniques that was historically inspired by the way computation works in the brain, and which can be characterized as learning of parameterized differentiable mathematical functions.[1] The name deep-learning stems from the fact that many layers of these differentiable function are often chained together.

While all of machine learning can be characterized as learning to make predictions based on past observations, deep learning approaches work by learning to not only predict but also to *correctly represent* the data, such that it is suitable for prediction. Given a large set of desired input-output mapping, deep learning approaches work by feeding the data into a network that produces successive transformations of the input data until a final transformation predicts the output. The transformations produced by the network are learned from the given input-output mappings, such that each transformation makes it easier to relate the data to the desired label.

[1]In this book we take the mathematical view rather than the brain-inspired view.

While the human designer is in charge of designing the network architecture and training regime, providing the network with a proper set of input-output examples, and encoding the input data in a suitable way, a lot of the heavy-lifting of learning the correct representation is performed automatically by the network, supported by the network's architecture.

## 1.3 DEEP LEARNING IN NLP

Neural networks provide a powerful learning machinery that is very appealing for use in natural language problems. A major component in neural networks for language is the use of an *embedding layer*, a mapping of discrete symbols to continuous vectors in a relatively low dimensional space. When embedding words, they transform from being isolated distinct symbols into mathematical objects that can be operated on. In particular, distance between vectors can be equated to distance between words, making it easier to generalize the behavior from one word to another. This representation of words as vectors is learned by the network as part of the training process. Going up the hierarchy, the network also learns to combine word vectors in a way that is useful for prediction. This capability alleviates to some extent the discreteness and data-sparsity problems.

There are two major kinds of neural network architectures, that can be combined in various ways: feed-forward networks and recurrent/recursive networks.

Feed-forward networks, in particular multi-layer perceptrons (MLPs), allow to work with fixed sized inputs, or with variable length inputs in which we can disregard the order of the elements. When feeding the network with a set of input components, it learns to combine them in a meaningful way. MLPs can be used whenever a linear model was previously used. The nonlinearity of the network, as well as the ability to easily integrate pre-trained word embeddings, often lead to superior classification accuracy.

Convolutional feed-forward networks are specialized architectures that excel at extracting local patterns in the data: they are fed arbitrarily sized inputs, and are capable of extracting meaningful local patterns that are sensitive to word order, regardless of where they appear in the input. These work very well for identifying indicative phrases or idioms of up to a fixed length in long sentences or documents.

Recurrent neural networks (RNNs) are specialized models for sequential data. These are network components that take as input a sequence of items, and produce a fixed size vector that summarizes that sequence. As "summarizing a sequence" means different things for different tasks (i.e., the information needed to answer a question about the sentiment of a sentence is different from the information needed to answer a question about its grammaticality), recurrent networks are rarely used as standalone component, and their power is in being trainable components that can be fed into other network components, and trained to work in tandem with them. For example, the output of a recurrent network can be fed into a feed-forward network that will try to predict some value. The recurrent network is used as an input-transformer that is trained to produce informative representations for the feed-forward network that will operate on top of it. Recurrent networks are very impressive models for sequences, and are arguably the most exciting offer of neural networks for language processing. They allow abandoning the *markov assumption* that was prevalent in NLP for decades, and designing models that can condition on entire sentences, while taking word order into account when it is needed, and not suffering much from statistical estimation problems stemming from data sparsity. This capability leads to impressive gains in *language-modeling*, the task of predicting the probability of the next word in a sequence (or, equivalently, the probability of a sequence), which is a cornerstone of many NLP applications. Recursive networks extend recurrent networks from sequences to trees.

Many of the problems in natural language are *structured*, requiring the production of complex output structures such as sequences or trees, and neural network models can accommodate that need as well, either by adapting known structured-prediction algorithms for linear models, or by using novel architectures such as sequence-to-sequence (encoder-decoder) models, which we refer to in this book as conditioned-generation models. Such models are at the heart of state-of-the-art machine translation.

Finally, many language prediction tasks are related to each other, in the sense that knowing to perform one of them will help in learning to perform the others. In addition, while we may have a shortage of *supervised* (labeled) training data, we have ample supply of raw text (unlabeled data). Can we learn from related tasks and un-annotated data? Neural network approaches provide exciting opportunities for both MTL (learning from related problems) and semi-supervised learning (learning from external, unannotated data).

### 1.3.1 SUCCESS STORIES

Fully connected feed-forward neural networks (MLPs) can, for the most part, be used as a drop-in replacement wherever a linear learner is used. This includes binary and multi-class classification problems, as well as more complex structured prediction problems. The nonlinearity of the network, as well as the ability to easily integrate pre-trained word embeddings, often lead to superior classification accuracy. A series of works[2] managed to obtain improved syntactic parsing results by simply replacing the linear model of a parser with a fully connected feed-forward network. Straightforward applications of a feed-forward network as a classifier replacement (usually coupled with the use of pre-trained word vectors) provide benefits for many language tasks, including the very well basic task of language modeling[3] CCG supertagging,[4] dialog state tracking,[5] and pre-ordering for statistical machine translation.[6] Iyyer et al. [2015] demonstrate that multi-layer feed-forward networks can provide competitive results on sentiment classification and factoid question answering. Zhou et al. [2015] and Andor et al. [2016] integrate them in a beam-search structured-prediction system, achieving stellar accuracies on syntactic parsing, sequence tagging and other tasks.

[2][Chen and Manning, 2014, Durrett and Klein, 2015, Pei et al., 2015, Weiss et al., 2015]
[3]See Chapter 9, as well as Bengio et al. [2003], Vaswani et al. [2013].
[4][Lewis and Steedman, 2014]
[5][Henderson et al., 2013]
[6][de Gispert et al., 2015]

Networks with convolutional and pooling layers are useful for classification tasks in which we expect to find strong local clues regarding class membership, but these clues can appear in different places in the input. For example, in a document classification task, a single key phrase (or an ngram) can help in determining the topic of the document [Johnson and Zhang, 2015]. We would like to learn that certain sequences of words are good indicators of the topic, and do not necessarily care where they appear in the document. Convolutional and pooling layers allow the model to learn to find such local indicators, regardless of their position. Convolutional and pooling architecture show promising results on many tasks, including document classification,[7] short-text categorization,[8] sentiment classification,[9] relation-type classification between entities,[10] event detection,[11] paraphrase identification,[12] semantic role labeling,[13] question answering,[14] predicting box-office revenues of movies based on critic reviews,[15] modeling text interestingness,[16] and modeling the relation between character-sequences and part-of-speech tags.[17]

In natural language we often work with structured data of arbitrary sizes, such as sequences and trees. We would like to be able to capture regularities in such structures, or to model similarities between such structures. Recurrent and recursive architectures allow working with sequences and trees while preserving a lot of the structural information. Recurrent networks [Elman, 1990] are designed to model sequences, while recursive networks [Goller and Küchler, 1996] are generalizations of recurrent networks that can handle trees. Recurrent models have been shown to produce very strong results for language modeling,[18] as well as for sequence tagging,[19] machine translation,[20] parsing,[21] and many other tasks including noisy text normalization,[22] dialog state tracking,[23] response generation,[24] and modeling the relation between character sequences and part-of-speech tags.[25]

[7][Johnson and Zhang, 2015]
[8][Wang et al., 2015a]
[9][Kalchbrenner et al., 2014, Kim, 2014]
[10][dos Santos et al., 2015, Zeng et al., 2014]
[11][Chen et al., 2015, Nguyen and Grishman, 2015]
[12][Yin and Schütze, 2015]
[13][Collobert et al., 2011]
[14][Dong et al., 2015]
[15][Bitvai and Cohn, 2015]
[16][Gao et al., 2014]
[17][dos Santos and Zadrozny, 2014]
[18]Some notable works are Adel et al. [2013], Auli and Gao [2014], Auli et al. [2013], Duh et al. [2013], Jozefowicz et al. [2016], Mikolov [2012], Mikolov et al. [2010, 2011].
[19][Irsoy and Cardie, 2014, Ling et al., 2015b, Xu et al., 2015]
[20][Cho et al., 2014b, Sundermeyer et al., 2014, Sutskever et al., 2014, Tamura et al., 2014]
[21][Dyer et al., 2015, Kiperwasser and Goldberg, 2016b, Watanabe and Sumita, 2015]
[22][Chrupala, 2014]
[23][Mrkšić et al., 2015]
[24][Kannan et al., 2016, Sordoni et al., 2015]
[25][Ling et al., 2015b]

Recursive models were shown to produce state-of-the-art or near state-of-the-art results for constituency[26] and dependency[27] parse re-ranking, discourse parsing,[28] semantic relation classification,[29] political ideology detection based on parse trees,[30] sentiment classification,[31] target-dependent sentiment classification,[32] and question answering.[33]

## 1.4 COVERAGE AND ORGANIZATION

The book consists of four parts. Part I introduces the basic learning machinery we'll be using throughout the book: supervised learning, MLPs, gradient-based training, and the computation-graph abstraction for implementing and training neural networks. Part II connects the machinery introduced in the first part with language data. It introduces the main sources of information that are available when working with language data, and explains how to integrate them with the neural networks machinery. It also discusses word-embedding algorithms and the distributional hypothesis, and feed-forward approaches to language modeling. Part III deals with specialized architectures and their applications to language data: 1D convolutional networks for working with ngrams, and RNNs for modeling sequences and stacks. RNNs are the main innovation of the application of neural networks to language data, and most of Part III is devoted to them, including the powerful conditioned-generation framework they facilitate, and attention-based models. Part IV is a collection of various advanced topics: recursive networks for modeling trees, structured prediction models, and multi-task learning.

Part I, covering the basics of neural networks, consists of four chapters. Chapter 2 introduces the basic concepts of supervised machine learning, parameterized functions, linear and log-linear models, regularization and loss functions, training as optimization, and gradient-based training methods. It starts from the ground up, and provides the needed material for the following chapters. Readers familiar with basic learning theory and gradient-based learning may consider skipping this chapter. Chapter 3 spells out the major limitation of linear models, motivates the need for nonlinear models, and lays the ground and motivation for multi-layer neural networks. Chapter 4 introduces feed-forward neural networks and the MLPs. It discusses the definition of multi-layer networks, their theoretical power, and common subcomponents such as nonlinearities and loss functions. Chapter 5 deals with neural network training. It introduces the computation-graph abstraction that allows for automatic gradient computations for arbitrary networks (the back-propagation algorithm), and provides several important tips and tricks for effective network training.

[26][Socher et al., 2013a]
[27][Le and Zuidema, 2014, Zhu et al., 2015a]
[28][Li et al., 2014]
[29][Hashimoto et al., 2013, Liu et al., 2015]
[30][Iyyer et al., 2014b]
[31][Hermann and Blunsom, 2013, Socher et al., 2013b]
[32][Dong et al., 2014]
[33][Iyyer et al., 2014a]

Part II introducing language data, consists of seven chapters. Chapter 6 presents a typology of common language-processing problems, and discusses the available sources of information (features) available for us when using language data. Chapter 7 provides concrete case studies, showing how the features described in the previous chapter are used for various natural language tasks. Readers familiar with language processing can skip these two chapters. Chapter 8 connects the material of Chapters 6 and 7 with neural networks, and discusses the various ways of encoding language-based features as inputs for neural networks. Chapter 9 introduces the language modeling task, and the feed-forward neural language model architecture. This also paves the way for discussing pre-trained word embeddings in the following chapters. Chapter 10 discusses distributed and distributional approaches to word-meaning representations. It introduces the word-context matrix approach to distributional semantics, as well as neural language-modeling inspired word-embedding algorithms, such as GloVe and Word2Vec, and discusses the connection between them and the distributional methods. Chapter 11 deals with using word embeddings outside of the context of neural networks. Finally, Chapter 12 presents a case study of a task-specific feed-forward network that is tailored for the Natural Language Inference task.

Part III introducing the specialized convolutional and recurrent architectures, consists of five chapters. Chapter 13 deals with convolutional networks, that are specialized at learning informative ngram patterns. The alternative hash-kernel technique is also discussed. The rest of this part, Chapters 14–17, is devoted to RNNs. Chapter 14 describes the RNN abstraction for modeling sequences and stacks. Chapter 15 describes concrete instantiations of RNNs, including the Simple RNN (also known as Elman RNNs) and gated architectures such as the Long Short-term Memory (LSTM) and the Gated Recurrent Unit (GRU). Chapter 16 provides examples of modeling with the RNN abstraction, showing their use within concrete applications. Finally, Chapter 17 introduces the conditioned-generation framework, which is the main modeling technique behind state-of-the-art machine translation, as well as unsupervised sentence modeling and many other innovative applications.

Part IV is a mix of advanced and non-core topics, and consists of three chapters. Chapter 18 introduces tree-structured recursive networks for modeling trees. While very appealing, this family of models is still in research stage, and is yet to show a convincing success story. Nonetheless, it is an important family of models to know for researchers who aim to push modeling techniques beyond the state-of-the-art. Readers who are mostly interested in mature and robust techniques can safely skip this chapter. Chapter 19 deals with structured prediction. It is a rather technical chapter. Readers who are particularly interested in structured prediction, or who are already familiar with structured prediction techniques for linear models or for language processing, will likely appreciate the material. Others may rather safely skip it. Finally, Chapter 20 presents multi-task and semi-supervised learning. Neural networks provide ample opportunities for multi-task and semi-supervised learning. These are important techniques, that are still at the research stage. However, the existing techniques are relatively easy to implement, and do provide real gains. The chapter is not technically challenging, and is recommended to all readers.

**Dependencies** For the most part, chapters, depend on the chapters that precede them. An exception are the first two chapters of Part II, which do not depend on material in previous chapters and can be read in any order. Some chapters and sections can be skipped without impacting the understanding of other concepts or material. These include Section 10.4 and Chapter 11 that deal with the details of word embedding algorithms and the use of word embeddings outside of neural networks; Chapter 12, describing a specific architecture for attacking the Stanford Natural Language Inference (SNLI) dataset; and Chapter 13 describing convolutional networks. Within the sequence on recurrent networks, Chapter 15, dealing with the details of specific architectures, can also be relatively safely skipped. The chapters in Part IV are for the most part independent of each other, and can be either skipped or read in any order.

## 1.5 WHAT'S NOT COVERED

The focus is on applications of neural networks to language processing tasks. However, some subareas of language processing with neural networks were deliberately left out of scope of this book. Specifically, I focus on processing written language, and do not cover working with speech data or acoustic signals. Within written language, I remain relatively close to the lower level, relatively well-defined tasks, and do not cover areas such as dialog systems, document summarization, or question answering, which I consider to be vastly open problems. While the described techniques can be used to achieve progress on these tasks, I do not provide examples or explicitly discuss these tasks directly. Semantic parsing is similarly out of scope. Multi-modal applications, connecting language data with other modalities such as vision or databases are only very briefly mentioned. Finally, the discussion is mostly English-centric, and languages with richer morphological systems and fewer computational resources are only very briefly discussed.

**Some important basics** are also not discussed. Specifically, two crucial aspects of good work in language processing are *proper evaluation* and *data annotation*. Both of these topics are left outside the scope of this book, but the reader should be aware of their existence.

*Proper evaluation* includes the choice of the right metrics for evaluating performance on a given task, best practices, fair comparison with other work, performing error analysis, and assessing statistical significance.

*Data annotation* is the bread-and-butter of NLP systems. Without data, we cannot train supervised models. As researchers, we very often just use "standard" annotated data produced by someone else. It is still important to know the source of the data, and consider the implications resulting from its creation process. Data annotation is a very vast topic, including proper formulation of the annotation task; developing the annotation guidelines; deciding on the source of annotated data, its coverage and class proportions, good train-test splits; and working with annotators, consolidating decisions, validating quality of annotators and annotation, and various similar topics.

## 1.6 A NOTE ON TERMINOLOGY

The word "feature" is used to refer to a concrete, linguistic input such as a word, a suffix, or a part-of-speech tag. For example, in a first-order part-of-speech tagger, the features might be "current word, previous word, next word, previous part of speech." The term "input vector" is used to refer to the actual input that is fed to the neural network classifier. Similarly, "input vector entry" refers to a specific value of the input. This is in contrast to a lot of the neural networks literature in which the word "feature" is overloaded between the two uses, and is used primarily to refer to an input-vector entry.

## 1.7 MATHEMATICAL NOTATION

We use bold uppercase letters to represent matrices ($\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{Z}$), and bold lowercase letters to represent vectors ($\boldsymbol{b}$). When there are series of related matrices and vectors (for example, where each matrix corresponds to a different layer in the network), superscript indices are used ($\boldsymbol{W}^1$, $\boldsymbol{W}^2$). For the rare cases in which we want indicate the power of a matrix or a vector, a pair of brackets is added around the item to be exponentiated: $(\boldsymbol{W})^2, (\boldsymbol{W}^3)^2$. We use [] as the index operator of vectors and matrices: $\boldsymbol{b}_{[i]}$ is the $i$th element of vector $\boldsymbol{b}$, and $\boldsymbol{W}_{[i,j]}$ is the element in the $i$th column and $j$th row of matrix $\boldsymbol{W}$. When unambiguous, we sometimes adopt the more standard mathematical notation and use $b_i$ to indicate the $i$th element of vector $\boldsymbol{b}$, and similarly $w_{i,j}$ for elements of a matrix $\boldsymbol{W}$. We use $\cdot$ to denote the dot-product operator: $\boldsymbol{w} \cdot \boldsymbol{v} = \sum_i w_i v_i = \sum_i \boldsymbol{w}_{[i]} \boldsymbol{v}_{[i]}$. We use $\boldsymbol{x}_{1:n}$ to indicate a sequence of vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$, and similarly $x_{1:n}$ is the sequence of items $x_1, \ldots, x_n$. We use $\boldsymbol{x}_{n:1}$ to indicate the reverse sequence. $\boldsymbol{x}_{1:n}[i] = \boldsymbol{x}_i$, $\boldsymbol{x}_{n:1}[i] = \boldsymbol{x}_{n-i+1}$. We use $[\boldsymbol{v}_1; \boldsymbol{v}_2]$ to denote vector concatenation.

While somewhat unorthodox, **unless otherwise stated, vectors are assumed to be row vectors.** The choice to use row vectors, which are right multiplied by matrices ($\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}$), is somewhat non standard—a lot of the neural networks literature use column vectors that are left multiplied by matrices ($\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$). We trust the reader to be able to adapt to the column vectors notation when reading the literature.[34]

[34]The choice to use the row vectors notation was inspired by the following benefits: it matches the way input vectors and network diagrams are often drawn in the literature; it makes the hierarchical/layered structure of the network more transparent and puts the input as the left-most variable rather than being nested; it results in fully connected layer dimensions being $d_{in} \times d_{out}$ rather than $d_{out} \times d_{in}$; and it maps better to the way networks are implemented in code using matrix libraries such as numpy.

PART I

# Supervised Classification and Feed-forward Neural Networks

CHAPTER 2

# Learning Basics and Linear Models

Neural networks, the topic of this book, are a class of supervised machine learning algorithms.

This chapter provides a quick introduction to supervised machine learning terminology and practices, and introduces linear and log-linear models for binary and multi-class classification.

The chapter also sets the stage and notation for later chapters. Readers who are familiar with linear models can skip ahead to the next chapters, but may also benefit from reading Sections 2.4 and 2.5.

Supervised machine learning theory and linear models are very large topics, and this chapter is far from being comprehensive. For a more complete treatment the reader is referred to texts such as Daumé III [2015], Shalev-Shwartz and Ben-David [2014], and Mohri et al. [2012].

## 2.1 SUPERVISED LEARNING AND PARAMETERIZED FUNCTIONS

The essence of supervised machine learning is the creation of mechanisms that can look at examples and produce generalizations. More concretely, rather than designing an algorithm to perform a task ("distinguish spam from non-spam email"), we design an algorithm whose input is a set of labeled examples ("This pile of emails are spam. This other pile of emails are not spam."), and its output is a function (or a program) that receives an instance (an email) and produces the desired label (spam or not-spam). It is expected that the resulting function will produce correct label predictions also for instances it has not seen during training.

As searching over the set of all possible programs (or all possible functions) is a very hard (and rather ill-defined) problem, we often restrict ourselves to search over specific families of functions, e.g., the space of all linear functions with $d_{in}$ inputs and $d_{out}$ outputs, or the space of all decision trees over $d_{in}$ variables. Such families of functions are called *hypothesis classes*. By restricting ourselves to a specific hypothesis class, we are injecting the learner with *inductive bias*—a set of assumptions about the form of the desired solution, as well as facilitating efficient procedures for searching for the solution. For a broad and readable overview of the main families of learning algorithms and the assumptions behind them, see the book by Domingos [2015].

The hypothesis class also determines what can and cannot be represented by the learner. One common hypothesis class is that of high-dimensional linear function, i.e., functions of the form:[1]

$$f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b} \tag{2.1}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}} \quad \boldsymbol{W} \in \mathbb{R}^{d_{in} \times d_{out}} \quad \boldsymbol{b} \in \mathbb{R}^{d_{out}}.$$

Here, the vector $\boldsymbol{x}$ is the *input* to the function, while the matrix $\boldsymbol{W}$ and the vector $\boldsymbol{b}$ are the *parameters*. The goal of the learner is to set the values of the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ such that the function behaves as intended on a collection of input values $\boldsymbol{x}_{1:k} = \boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ and the corresponding desired outputs $\boldsymbol{y}_{1:k} = \boldsymbol{y}_i, \ldots, \boldsymbol{y}_k$. The task of searching over the space of functions is thus reduced to one of searching over the space of parameters. It is common to refer to parameters of the function as $\Theta$. For the linear model case, $\Theta = \boldsymbol{W}, \boldsymbol{b}$. In some cases we want the notation to make the parameterization explicit, in which case we include the parameters in the function's definition: $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b}$.

As we will see in the coming chapters, the hypothesis class of linear functions is rather restricted, and there are many functions that it cannot represent (indeed, it is limited to *linear* relations). In contrast, *feed-forward neural networks with hidden layers*, to be discussed in Chapter 4, are also parameterized functions, but constitute a very strong hypothesis class—they are *universal approximators*, capable of representing any Borel-measurable function.[2] However, while restricted, linear models have several desired properties: they are easy and efficient to train, they often result in convex optimization objectives, the trained models are somewhat interpretable, and they are often very effective in practice. Linear and log-linear models were the dominant approaches in statistical NLP for over a decade. Moreover, they serve as the basic building blocks for the more powerful nonlinear feed-forward networks which will be discussed in later chapters.

## 2.2 TRAIN, TEST, AND VALIDATION SETS

Before delving into the details of linear models, let's reconsider the general setup of the machine learning problem. We are faced with a dataset of $k$ input examples $\boldsymbol{x}_{1:k}$ and their corresponding gold labels $\boldsymbol{y}_{1:k}$, and our goal is to produce a function $f(\boldsymbol{x})$ that correctly maps inputs $\boldsymbol{x}$ to outputs $\hat{\boldsymbol{y}}$, as evidenced by the training set. How do we know that the produced function $f()$ is indeed a good one? One could run the training examples $\boldsymbol{x}_{1:k}$ through $f()$, record the answers $\hat{\boldsymbol{y}}_{1:k}$, compare them to the expected labels $\boldsymbol{y}_{1:k}$, and measure the accuracy. However, this process will not be very informative—our main concern is the ability of $f()$ to generalize well to unseen examples. A function $f()$ that is implemented as a lookup table, that is, looking for the input $\boldsymbol{x}$ in its memory and returning the corresponding value $\boldsymbol{y}$ for instances is has seen and a random value otherwise, will get a perfect score on this test, yet is clearly not a good classification function as it has zero generalization ability. We rather have a function $f()$ that gets some of the training examples wrong, providing that it will get unseen examples correctly.

[1]As discussed in Section 1.7. This book takes a somewhat un-orthodox approach and assumes vectors are *row vectors* rather than column vectors.
[2]See further discussion in Section 4.3.

**Leave-one out** We must assess the trained function's accuracy on instances it has not seen during training. One solution is to perform *leave-one-out cross-validation*: train $k$ functions $f_{1:k}$, each time leaving out a different input example $\boldsymbol{x}_i$, and evaluating the resulting function $f_i()$ on its ability to predict $\boldsymbol{x}_i$. Then train another function $f()$ on the entire trainings set $\boldsymbol{x}_{1:k}$. Assuming that the training set is a representative sample of the population, this percentage of functions $f_i()$ that produced correct prediction on the left-out samples is a good approximation of the accuracy of $f()$ on new inputs. However, this process is very costly in terms of computation time, and is used only in cases where the number of annotated examples $k$ is very small (less than a hundred or so). In language processing tasks, we very often encounter training sets with well over $10^5$ examples.

**Held-out set** A more efficient solution in terms of computation time is to split the training set into two subsets, say in a 80%/20% split, train a model on the larger subset (the *training set*), and test its accuracy on the smaller subset (the *held-out set*). This will give us a reasonable estimate on the accuracy of the trained function, or at least allow us to compare the quality of different trained models. However, it is somewhat wasteful in terms training samples. One could then re-train a model on the entire set. However, as the model is trained on substantially more data, the error estimates of the model trained on less data may not be accurate. This is generally a good problem to have, as more training data is likely to result in better rather than worse predictors.[3]

Some care must be taken when performing the split—in general it is better to shuffle the examples prior to splitting them, to ensure a balanced distribution of examples between the training and held-out sets (for example, you want the proportion of gold labels in the two sets to be similar). However, sometimes a random split is not a good option: consider the case where your input are news articles collected over several months, and your model is expected to provide predictions for new stories. Here, a random split will over-estimate the model's quality: the training and held-out examples will be from the same time period, and hence on more similar stories, which will not be the case in practice. In such cases, you want to ensure that the training set has older news stories and the held-out set newer ones—to be as similar as possible to how the trained model will be used in practice.

**A three-way split** The split into train and held-out sets works well if you train a single model and wants to assess its quality. However, in practice you often train several models, compare their quality, and select the best one. Here, the two-way split approach is insufficient—selecting the best model according to the held-out set's accuracy will result in an overly optimistic estimate of the model's quality. You don't know if the chosen settings of the final classifier are good in general, or are just good for the particular examples in the held-out sets. The problem will be even worse if you perform error analysis based on the held-out set, and change the features or the architecture of the model based on the observed errors. You don't know if your improvements based on the held-

[3]Note, however, that some setting in the training procedure, in particular the learning rate and regularization weight may be sensitive to the training set size, and tuning them based on some data and then re-training a model with the same settings on larger data may produce sub-optimal results.

out sets will carry over to new instances. The accepted methodology is to use a three-way split of the data into train, validation (also called *development*), and test sets. This gives you two held-out sets: a *validation set* (also called *development set*), and a *test set*. All the experiments, tweaks, error analysis, and model selection should be performed based on the validation set. Then, a single run of the final model over the test set will give a good estimate of its expected quality on unseen examples. It is important to keep the test set as pristine as possible, running as few experiments as possible on it. Some even advocate that you should not even look at the examples in the test set, so as to not bias the way you design your model.

## 2.3 LINEAR MODELS

Now that we have established some methodology, we return to describe linear models for binary and multi-class classification.

### 2.3.1 BINARY CLASSIFICATION

In binary classification problems we have a single output, and thus use a restricted version of Equation (2.1) in which $d_{out} = 1$, making $\boldsymbol{w}$ a vector and $b$ a scalar.

$$f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{w} + b. \tag{2.2}$$

The range of the linear function in Equation (2.2) is $[-\infty, +\infty]$. In order to use it for binary classification, it is common to pass the output of $f(\boldsymbol{x})$ through the $sign$ function, mapping negative values to $-1$ (the negative class) and non-negative values to $+1$ (the positive class).

Consider the task of predicting which of two neighborhoods an apartment is located at, based on the apartment's price and size. Figure 2.1 shows a 2D plot of some apartments, where the $x$-axis denotes the monthly rent price in USD, while the $y$-axis is the size in square feet. The blue circles are for Dupont Circle, DC and the green crosses are in Fairfax, VA. It is evident from the plot that we can separate the two neighborhoods using a straight line—apartments in Dupont Circle tend to be more expensive than apartments in Fairfax of the same size.[4] The dataset is *linearly separable*: the two classes can be separated by a straight line.

Each data-point (an apartment) can be represented as a 2-dimensional (2D) vector $\boldsymbol{x}$ where $\boldsymbol{x}_{[0]}$ is the apartment's size and $\boldsymbol{x}_{[1]}$ is its price. We then get the following linear model:

$$\begin{aligned}\hat{y} = \text{sign}(f(\boldsymbol{x})) &= \text{sign}(\boldsymbol{x} \cdot \boldsymbol{w} + b) \\ &= \text{sign}(\text{size} \times w_1 + \text{price} \times w_2 + b),\end{aligned}$$

where $\cdot$ is the dot-product operation, $b$ and $\boldsymbol{w} = [w_1, w_2]$ are free parameters, and we predict Fairfax if $\hat{y} \geq 0$ and Dupont Circle otherwise. The goal of learning is setting the values of $w_1$,

[4]Note that looking at either size or price alone would not allow us to cleanly separate the two groups.

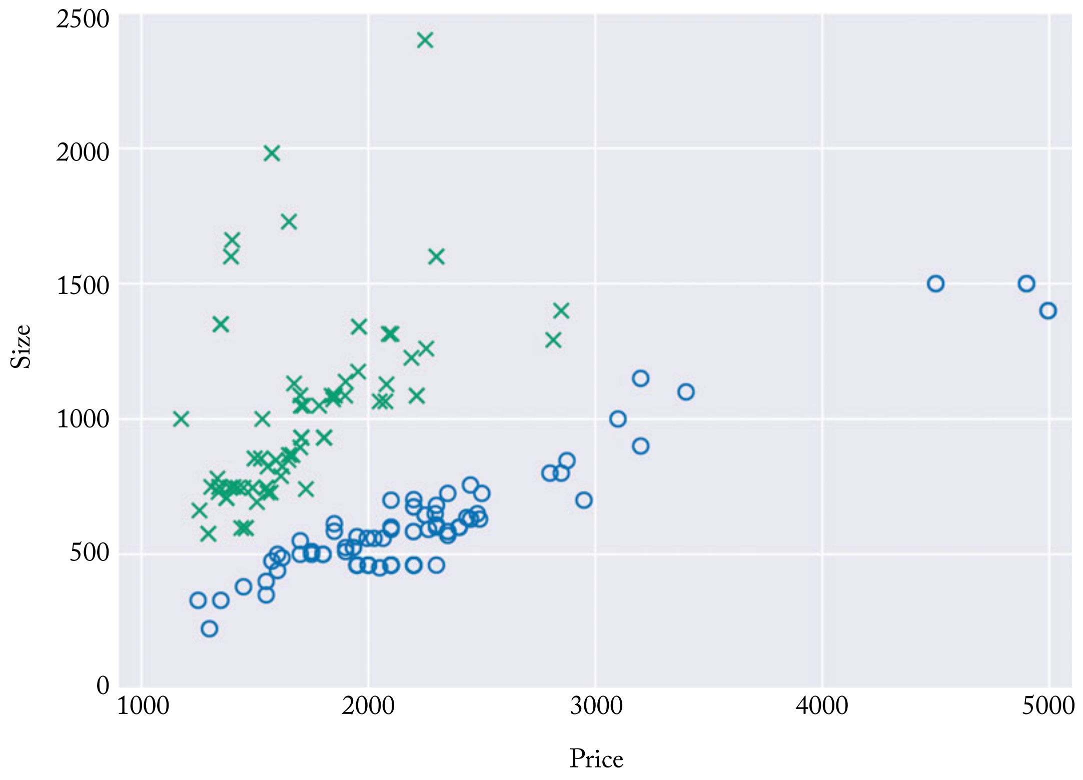


**Figure 2.1:** Housing data: rent price in USD vs. size in square ft. Data source: Craigslist ads, collected from June 7–15, 2015.

$w_2$, and $b$ such that the predictions are correct for all data-points we observe.[5] We will discuss learning in Section 2.7 but for now consider that we expect the learning procedure to set a high value to $w_1$ and a low value to $w_2$. Once the model is trained, we can classify new data-points by feeding them into this equation.

It is sometimes not possible to separate the data-points using a straight line (or, in higher dimensions, a linear hyperplane)—such datasets are said to be *nonlinearly separable*, and are beyond the hypothesis class of linear classifiers. The solution would be to either move to a higher dimension (add more features), move to a richer hypothesis class, or allow for some mis-classification.[6]

[5]Geometrically, for a given $\boldsymbol{w}$ the points $\boldsymbol{x} \cdot \boldsymbol{w} + b = 0$ define a *hyperplane* (which in two dimensions corresponds to a line) that separates the space into two regions. The goal of learning is then finding a hyperplane such that the classification induced by it is correct.

[6]Misclassifying some of the examples is sometimes a good idea. For example, if we have reason to believe some of the data-points are *outliers*—examples that belong to one class, but are labeled by mistake as belonging to the other class.

Feature Representations In the example above, each data-point was a pair of size and price measurements. Each of these properties is considered a *feature* by which we classify the data-point. This is very convenient, but in most cases the data-points are not given to us directly as lists of features, but as real-world objects. For example, in the apartments example we may be given a list of apartments to classify. We then need to make a concious decision and select the measurable properties of the apartments that we believe will be useful features for the classification task at hand. Here, it proved effective to focus on the price and the size. We could also look at additional properties, such as the number of rooms, the height of the ceiling, the type of floor, the geo-location coordinates, and so on. After deciding on a set of features, we create a *feature extraction* function that maps a real world object (i.e., an apartment) to a vector of measurable quantities (price and size) which can be used as inputs to our models. The choice of the features is crucial to the success of the classification accuracy, and is driven by the informativeness of the features, and their availability to us (the geo-location coordinates are much better predictors of the neighborhood than the price and size, but perhaps we only observe listings of past transactions, and do not have access to the geo-location information). When we have two features, it is easy to plot the data and see the underlying structures. However, as we see in the next example, we often use many more than just two features, making plotting and precise reasoning impractical.

A central part in the design of linear models, which we mostly gloss over in this text, is the design of the feature function (so called *feature engineering*). One of the promises of deep learning is that it vastly simplifies the feature-engineering process by allowing the model designer to specify a small set of core, basic, or "natural" features, and letting the trainable neural network architecture combine them into more meaningful higher-level features, or *representations*. However, one still needs to specify a suitable set of core features, and tie them to a suitable architecture. We discuss common features for textual data in Chapters 6 and 7.

We usually have many more than two features. Moving to a language setup, consider the task of distinguishing documents written in English from documents written in German. It turns out that letter frequencies make for quite good predictors (features) for this task. Even more informative are counts of letter *bigrams*, i.e., pairs of consecutive letters.[7] Assuming we have an alphabet of 28 letters (a–z, space, and a special symbol for all other characters including digits, punctuations, etc.) we represent a document as a $28 \times 28$ dimensional vector $\boldsymbol{x} \in \mathbb{R}^{784}$, where each entry $\boldsymbol{x}_{[i]}$ represents a count of a particular letter combination in the document, normalized by the document's length. For example, denoting by $x_{ab}$ the entry of $\boldsymbol{x}$ corresponding to the

[7]While one may think that *words* will also be good predictors, letters, or letter-bigrams are far more robust: we are likely to encounter a new document without any of the words we observed in the training set, while a document without any of the distinctive letter-bigrams is significantly less likely.

letter-bigram ab:

$$x_{ab} = \frac{\#_{ab}}{|D|}, \tag{2.3}$$

where $\#_{ab}$ is the number of times the bigram ab appears in the document, and $|D|$ is the total number of bigrams in the document (the document's length).

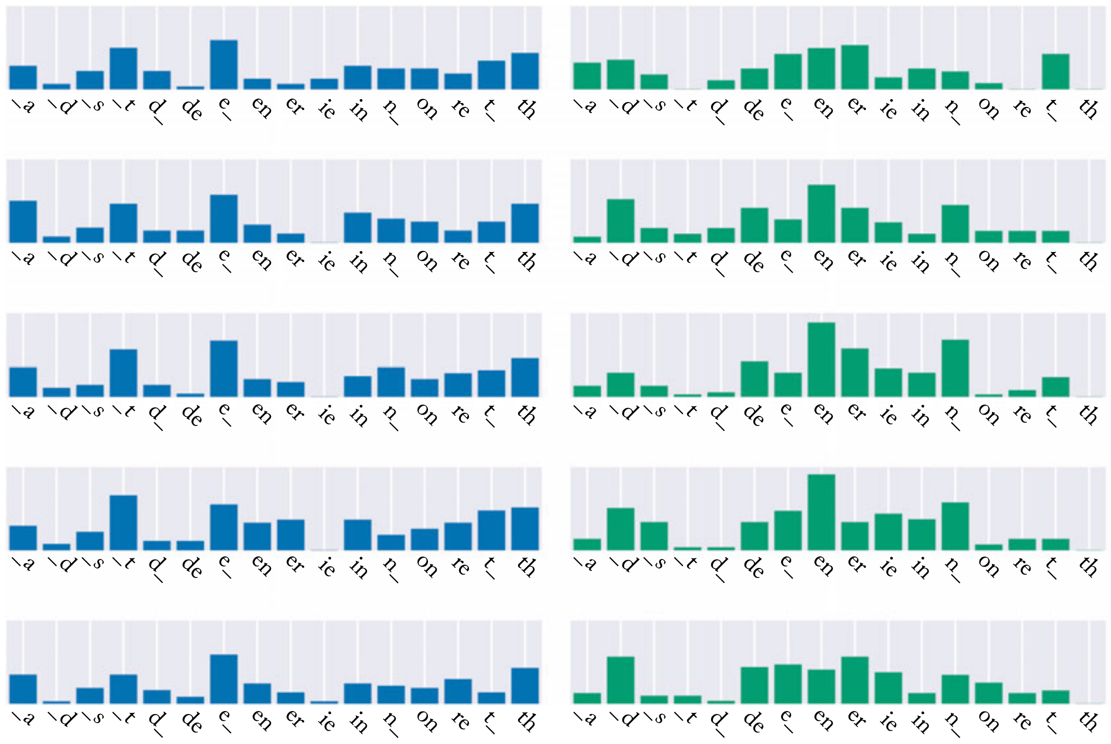
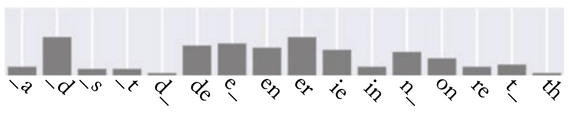


**Figure 2.2:** Character-bigram histograms for documents in English (left, blue) and German (right, green). Underscores denote spaces.

Figure 2.2 shows such bigram histograms for several German and English texts. For readability, we only show the top frequent character-bigrams and not the entire feature vectors. On the left, we see the bigrams of the English texts, and on the right of the German ones. There are clear patterns in the data, and, given a new item, such as:



you could probably tell that it is more similar to the German group than to the English one. Note, however, that you couldn't use a single definite rule such as "if it has th its English" or "if it has ie its German": while German texts have considerably less th than English, the th may and does occur in German texts, and similarly the ie combination does occur in English. The decision requires weighting different factors relative to each other. Let's formalize the problem in a machine-learning setup.

We can again use a linear model:

$$\begin{aligned} \hat{y} &= \text{sign}(f(\boldsymbol{x})) = \text{sign}(\boldsymbol{x} \cdot \boldsymbol{w} + b) \\ &= \text{sign}(x_{aa} \times w_{aa} + x_{ab} \times w_{ab} + x_{ac} \times w_{ac} \ldots + b). \end{aligned} \tag{2.4}$$

A document will be considered English if $f(\boldsymbol{x}) \geq 0$ and as German otherwise. Intuitively, learning should assign large positive values to $\boldsymbol{w}$ entries associated with letter pairs that are much more common in English than in German (i.e., th) negative values to letter pairs that are much more common in German than in English (ie, en), and values around zero to letter pairs that are either common or rare in both languages.

Note that unlike the 2D case of the housing data (price vs. size), here we cannot easily visualize the points and the decision boundary, and the geometric intuition is likely much less clear. In general, it is difficult for most humans to think of the geometries of spaces with more than three dimensions, and it is advisable to think of linear models in terms of assigning weights to features, which is easier to imagine and reason about.

### 2.3.2 LOG-LINEAR BINARY CLASSIFICATION

The output $f(\boldsymbol{x})$ is in the range $[-\infty, \infty]$, and we map it to one of two classes $\{-1, +1\}$ using the $sign$ function. This is a good fit if all we care about is the assigned class. However, we may be interested also in the confidence of the decision, or the probability that the classifier assigns to the class. An alternative that facilitates this is to map instead to the range $[0, 1]$, by pushing the output through a squashing function such as the sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$, resulting in:

$$\hat{y} = \sigma(f(\boldsymbol{x})) = \frac{1}{1 + e^{-(\boldsymbol{x} \cdot \boldsymbol{w} + b)}}. \tag{2.5}$$

Figure 2.3 shows a plot of the sigmoid function. It is monotonically increasing, and maps values to the range $[0, 1]$, with 0 being mapped to $\frac{1}{2}$. When used with a suitable *loss function* (discussed in Section 2.7.1) the binary predictions made through the log-linear model can be interpreted as class membership probability estimates $\sigma(f(\boldsymbol{x})) = P(\hat{y} = 1 \mid \boldsymbol{x})$ of $\boldsymbol{x}$ belonging to the positive class. We also get $P(\hat{y} = 0 \mid \boldsymbol{x}) = 1 - P(\hat{y} = 1 \mid \boldsymbol{x}) = 1 - \sigma(f(\boldsymbol{x}))$. The closer the value is to 0 or 1 the more certain the model is in its class membership prediction, with the value of 0.5 indicating model uncertainty.

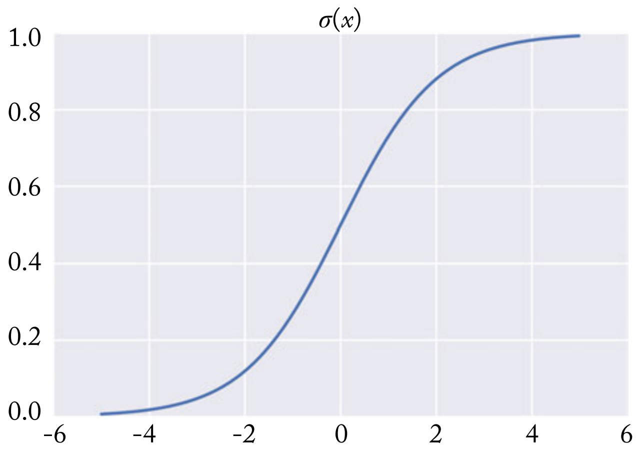


**Figure 2.3:** The sigmoid function $\sigma(x)$.

### 2.3.3 MULTI-CLASS CLASSIFICATION

The previous examples were of *binary classification*, where we had two possible classes. Binary-classification cases exist, but most classification problems are of a *multi-class* nature, in which we should assign an example to one of $k$ different classes. For example, we are given a document and asked to classify it into one of six possible languages: *English, French, German, Italian, Spanish, Other*. A possible solution is to consider six weight vectors $\boldsymbol{w}^{\text{En}}, \boldsymbol{w}^{\text{Fr}}, \ldots$ and biases, one for each language, and predict the language resulting in the highest score:[8]

$$\hat{y} = f(\boldsymbol{x}) = \underset{L \in \{\text{En},\text{Fr},\text{Gr},\text{It},\text{Sp},\text{O}\}}{\operatorname{argmax}} \quad \boldsymbol{x} \cdot \boldsymbol{w}^L + b^L. \tag{2.6}$$

The six sets of parameters $\boldsymbol{w}^L \in \mathbb{R}^{784}$, $b^L$ can be arranged as a matrix $\boldsymbol{W} \in \mathbb{R}^{784 \times 6}$ and vector $\boldsymbol{b} \in \mathbb{R}^6$, and the equation re-written as:

$$\begin{aligned} \hat{\boldsymbol{y}} &= f(\boldsymbol{x}) = \boldsymbol{x} \cdot \boldsymbol{W} + \boldsymbol{b} \\ \text{prediction} = \hat{y} &= \underset{i}{\operatorname{argmax}}\, \hat{\boldsymbol{y}}_{[i]}. \end{aligned} \tag{2.7}$$

Here $\hat{\boldsymbol{y}} \in \mathbb{R}^6$ is a vector of the scores assigned by the model to each language, and we again determine the predicted language by taking the argmax over the entries of $\hat{\boldsymbol{y}}$.

[8]There are many ways to model multi-class classification, including binary-to-multi-class reductions. These are beyond the scope of this book, but a good overview can be found in Allwein et al. [2000].

## 2.4 REPRESENTATIONS

Consider the vector $\hat{\boldsymbol{y}}$ resulting from applying Equation 2.7 of a trained model to a document. The vector can be considered as a *representation* of the document, capturing the properties of the document that are important to us, namely the scores of the different languages. The representation $\hat{\boldsymbol{y}}$ contains strictly more information than the prediction $\hat{y} = \text{argmax}_i\ \hat{\boldsymbol{y}}_{[i]}$: for example, $\hat{\boldsymbol{y}}$ can be used to distinguish documents in which the main language in German, but which also contain a sizeable amount of French words. By clustering documents based on their vector representations as assigned by the model, we could perhaps discover documents written in regional dialects, or by multilingual authors.

The vectors $\boldsymbol{x}$ containing the normalized letter-bigram counts for the documents are also representations of the documents, arguably containing a similar kind of information to the vectors $\hat{\boldsymbol{y}}$. However, the representations in $\hat{\boldsymbol{y}}$ is more compact (6 entries instead of 784) and more specialized for the language prediction objective (clustering by the vectors $\boldsymbol{x}$ would likely reveal document similarities that are not due to a particular mix of languages, but perhaps due to the document's topic or writing styles).

The trained matrix $\boldsymbol{W} \in \mathbb{R}^{784\times 6}$ can also be considered as containing learned representations. As demonstrated in Figure 2.4, we can consider two views of $\boldsymbol{W}$, as rows or as columns. Each of the 6 columns of $\boldsymbol{W}$ correspond to a particular language, and can be taken to be a 784-dimensional vector representation of this language in terms of its characteristic letter-bigram patterns. We can then cluster the 6 language vectors according to their similarity. Similarly, each of the 784 rows of $\boldsymbol{W}$ correspond to a particular letter-bigram, and provide a 6-dimensional vector representation of that bigram in terms of the languages it prompts.

Representations are central to deep learning. In fact, one could argue that the main power of deep-learning is the ability to learn good representations. In the linear case, the representations are interpretable, in the sense that we can assign a meaningful interpretation to each dimension in the representation vector (e.g., each dimension corresponds to a particular language or letter-bigram). This is in general not the case—deep learning models often learn a cascade of representations of the input that build on top of each other, in order to best model the problem at hand, and these representations are often not interpretable—we do not know which properties of the input they capture. However, they are still very useful for making predictions. Moreover, at the boundaries of the model, i.e., at the input and the output, we get representations that correspond to particular aspects of the input (i.e., a vector representation for each letter-bigram) or the output (i.e., a vector representation of each of the output classes). We will get back to this in Section 8.3 after discussing neural networks and encoding categorical features as dense vectors. It is recommended that you return to this discussion once more after reading that section.

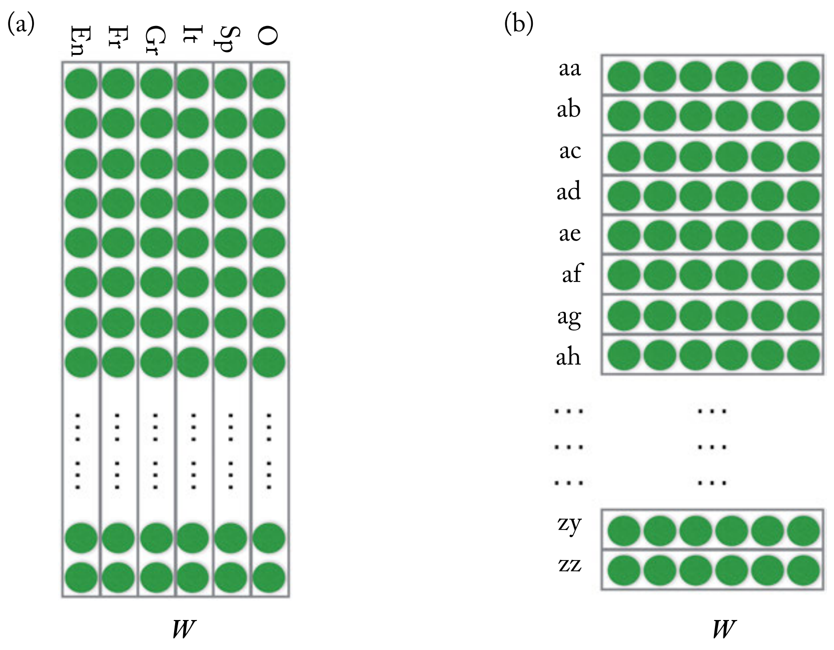


**Figure 2.4:** Two views of the $\boldsymbol{W}$ matrix. (a) Each column corresponds to a language. (b) Each row corresponds to a letter bigram.

## 2.5 ONE-HOT AND DENSE VECTOR REPRESENTATIONS

The input vector $\boldsymbol{x}$ in our language classification example contains the normalized bigram counts in the document $D$. This vector can be decomposed into an average of $|D|$ vectors, each corresponding to a particular document position $i$:

$$\boldsymbol{x} = \frac{1}{|D|}\sum_{i=1}^{|D|} \boldsymbol{x}^{D_{[i]}}; \tag{2.8}$$

here, $D_{[i]}$ is the bigram at document position $i$, and each vector $\boldsymbol{x}^{D_{[i]}} \in \mathbb{R}^{784}$ is a *one-hot* vector, in which all entries are zero except the single entry corresponding to the letter bigram $D_{[i]}$, which is 1.

The resulting vector $\boldsymbol{x}$ is commonly referred to as an *averaged bag of bigrams* (more generally *averaged bag of words*, or just *bag of words*). Bag-of-words (BOW) representations contain information about the identities of all the "words" (here, bigrams) of the document, without considering their order. A one-hot representation can be considered as a bag-of-a-single-word.

The view of the rows of the matrix $\boldsymbol{W}$ as representations of the letter bigrams suggests an alternative way of computing the document representation vector $\hat{\boldsymbol{y}}$ in Equation (2.7). Denoting by $\boldsymbol{W}^{D_{[i]}}$ the row of $W$ corresponding to the bigram $D_{[i]}$, we can take the representation $\boldsymbol{y}$ of a document $D$ to be the average of the representations of the letter-bigrams in the document:

$$\hat{\boldsymbol{y}} = \frac{1}{|D|}\sum_{i=1}^{|D|} \boldsymbol{W}^{D_{[i]}}. \tag{2.9}$$

This representation is often called a *continuous bag of words* (CBOW), as it is composed of a sum of word representations, where each "word" representation is a low-dimensional, continuous vector.

We note that Equation (2.9) and the term $\boldsymbol{x} \cdot \boldsymbol{W}$ in Equation (2.7) are equivalent. To see why, consider:

$$\begin{aligned}
\boldsymbol{y} &= \boldsymbol{x} \cdot \boldsymbol{W} \\
&= \left(\frac{1}{|D|}\sum_{i=1}^{|D|} \boldsymbol{x}^{D_{[i]}}\right) \cdot \boldsymbol{W} \\
&= \frac{1}{|D|}\sum_{i=1}^{|D|} (\boldsymbol{x}^{D_{[i]}} \cdot \boldsymbol{W}) \\
&= \frac{1}{|D|}\sum_{i=1}^{|D|} \boldsymbol{W}^{D_{[i]}}.
\end{aligned} \tag{2.10}$$

In other words, the continuous-bag-of-words (CBOW) representation can be obtained either by summing word-representation vectors or by multiplying a bag-of-words vector by a matrix in which each row corresponds to a dense word representation (such matrices are also called *embedding matrices*). We will return to this point in Chapter 8 (in particular Section 8.3) when discussing feature representations in deep learning models for text.

## 2.6 LOG-LINEAR MULTI-CLASS CLASSIFICATION

In the binary case, we transformed the linear prediction into a probability estimate by passing it through the sigmoid function, resulting in a log-linear model. The analog for the multi-class case is passing the score vector through the *softmax* function:

$$\text{softmax}(\boldsymbol{x})_{[i]} = \frac{e^{\boldsymbol{x}_{[i]}}}{\sum_j e^{\boldsymbol{x}_{[j]}}}. \tag{2.11}$$

Resulting in:

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b})$$
$$\hat{\boldsymbol{y}}_{[i]} = \frac{e^{(\boldsymbol{x}\boldsymbol{W}+\boldsymbol{b})_{[i]}}}{\sum_j e^{(\boldsymbol{x}\boldsymbol{W}+\boldsymbol{b})_{[j]}}}. \tag{2.12}$$

The *softmax* transformation forces the values in $\hat{\boldsymbol{y}}$ to be positive and sum to 1, making them interpretable as a probability distribution.

## 2.7 TRAINING AS OPTIMIZATION

Recall that the input to a supervised learning algorithm is a *training set* of $n$ training examples $x_{1:n} = x_1, x_2, \ldots, x_n$ together with corresponding labels $y_{1:n} = y_1, y_2, \ldots, y_n$. Without loss of generality, we assume that the desired inputs and outputs are vectors: $\boldsymbol{x}_{1:n}$, $\boldsymbol{y}_{1:n}$.[9]

The goal of the algorithm is to return a function $f()$ that accurately maps input examples to their desired labels, i.e., a function $f()$ such that the predictions $\hat{\boldsymbol{y}} = f(\boldsymbol{x})$ over the training set are accurate. To make this more precise, we introduce the notion of a *loss function*, quantifying the loss suffered when predicting $\hat{\boldsymbol{y}}$ while the true label is $\boldsymbol{y}$. Formally, a loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ assigns a numerical score (a scalar) to a predicted output $\hat{\boldsymbol{y}}$ given the true expected output $\boldsymbol{y}$. The loss function should be bounded from below, with the minimum attained only for cases where the prediction is correct.

The parameters of the learned function (the matrix $\boldsymbol{W}$ and the biases vector $\boldsymbol{b}$) are then set in order to minimize the loss $L$ over the training examples (usually, it is the sum of the losses over the different training examples that is being minimized).

Concretely, given a labeled training set $(\boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n})$, a per-instance loss function $L$ and a parameterized function $f(\boldsymbol{x}; \Theta)$ we define the corpus-wide loss with respect to the parameters $\Theta$ as the average loss over all training examples:

$$\mathcal{L}(\Theta) = \frac{1}{n}\sum_{i=1}^{n} L(f(\boldsymbol{x}_i; \Theta), \boldsymbol{y}_i). \tag{2.13}$$

In this view, the training examples are fixed, and the values of the parameters determine the loss. The goal of the training algorithm is then to set the values of the parameters $\Theta$ such that the value of $\mathcal{L}$ is minimized:

$$\hat{\Theta} = \operatorname*{argmin}_{\Theta} \mathcal{L}(\Theta) = \operatorname*{argmin}_{\Theta} \frac{1}{n}\sum_{i=1}^{n} L(f(\boldsymbol{x}_i; \Theta), \boldsymbol{y}_i). \tag{2.14}$$

Equation (2.14) attempts to minimize the loss at all costs, which may result in *overfitting* the training data. To counter that, we often pose soft restrictions on the form of the solution. This

[9]In many cases it is natural to think of the expected output as a scalar (class assignment) rather than a vector. In such cases, $\boldsymbol{y}$ is simply the corresponding one-hot vector, and $\text{argmax}_i\ \boldsymbol{y}_{[i]}$ is the corresponding class assignment.