

Logic and Discrete Mathematics

A Concise Introduction

WILLEM CONRADIE
VALENTIN GORANKO

WILEY

LOGIC AND DISCRETE MATHEMATICS

LOGIC AND DISCRETE MATHEMATICS

A CONCISE INTRODUCTION

Willem Conradie

*University of Johannesburg,
South Africa*

Valentin Goranko

Stockholm University, Sweden

WILEY

This edition first published 2015

© 2015 John Wiley and Sons Ltd

Registered office

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at www.wiley.com.

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. It is sold on the understanding that the publisher is not engaged in rendering professional services and neither the publisher nor the author shall be liable for damages arising herefrom. If professional advice or other expert assistance is required, the services of a competent professional should be sought

Library of Congress Cataloging-in-Publication Data

Conradie, Willem, 1978- author.

Logic and discrete mathematics : a concise introduction / Willem Conradie, Valentin Goranko.

pages cm

Includes index.

ISBN 978-1-118-75127-5 (pbk.)

1. Logic, Symbolic and mathematical--Textbooks. 2. Computer science--Mathematics--Textbooks.

I. Goranko, Valentin, author. II. Title.

QA9.C7423 2015

511.3--dc23

2014035964

A catalogue record for this book is available from the British Library.

Front cover image: Narrow walkway inside the Great Enclosure at the Great Zimbabwe ruins. Reproduced by kind permission of Kenneth Meeks.

Set in 9/11pt, PalatinoLTStd by Laserwords Private Limited, Chennai, India

Contents

List of Boxes	xiii
Preface	xvii
Acknowledgements	xxi
About the Companion Website	xxiii
1. Preliminaries	1
1.1 Sets	2
1.1.1 Exercises	7
1.2 Basics of logical connectives and expressions	9
1.2.1 Propositions, logical connectives, truth tables, tautologies	9
1.2.2 Individual variables and quantifiers	12
1.2.3 Exercises	15
1.3 Mathematical induction	17
1.3.1 Exercises	18
2. Sets, Relations, Orders	20
2.1 Set inclusions and equalities	21
2.1.1 Properties of the set theoretic operations	22
2.1.2 Exercises	26
2.2 Functions	28
2.2.1 Functions and their inverses	28
2.2.2 Composition of mappings	31
2.2.3 Exercises	33
2.3 Binary relations and operations on them	35
2.3.1 Binary relations	35
2.3.2 Matrix and graphical representations of relations on finite sets	38

2.3.3 Boolean operations on binary relations	39
2.3.4 Inverse and composition of relations	41
2.3.5 Exercises	42
2.4 Special binary relations	45
2.4.1 Properties of binary relations	45
2.4.2 Functions as relations	47
2.4.3 Reflexive, symmetric and transitive closures of a relation	47
2.4.4 Exercises	49
2.5 Equivalence relations and partitions	51
2.5.1 Equivalence relations	51
2.5.2 Quotient sets and partitions	53
2.5.3 The kernel equivalence of a mapping	56
2.5.4 Exercises	57
2.6 Ordered sets	59
2.6.1 Pre-orders and partial orders	59
2.6.2 Graphical representing posets: Hasse diagrams	61
2.6.3 Lower and upper bounds. Minimal and maximal elements	63
2.6.4 Well-ordered sets	65
2.6.5 Exercises	67
2.7 An introduction to cardinality	69
2.7.1 Equinumerosity and cardinality	69
2.7.2 Exercises	73
2.8 Isomorphisms of ordered sets. Ordinal numbers	75
2.8.1 Exercises	79
2.9 Application: relational databases	80
2.9.1 Exercises	86

3. Propositional Logic

89

3.1 Propositions, logical connectives, truth tables, tautologies	90
3.1.1 Propositions and propositional connectives. Truth tables	90
3.1.2 Some remarks on the meaning of the connectives	90
3.1.3 Propositional formulae	91
3.1.4 Construction and parsing tree of a propositional formula	92

3.1.5 Truth tables of propositional formulae	93
3.1.6 Tautologies	95
3.1.7 A better idea: search for a falsifying truth assignment	96
3.1.8 Exercises	97
3.2 Propositional logical consequence. Valid and invalid propositional inferences	101
3.2.1 Propositional logical consequence	101
3.2.2 Logically sound rules of propositional inference. Logically correct propositional arguments	104
3.2.3 Fallacies of the implication	106
3.2.4 Exercises	107
3.3 The concept and use of deductive systems	109
3.4 Semantic tableaux	113
3.4.1 Exercises	117
3.5 Logical equivalences. Negating propositional formulae	121
3.5.1 Logically equivalent propositional formulae	121
3.5.2 Some important equivalences	123
3.5.3 Exercises	124
3.6 Normal forms. Propositional resolution	126
3.6.1 Conjunctive and disjunctive normal forms of propositional formulae	126
3.6.2 Clausal form. Clausal resolution	129
3.6.3 Resolution-based derivations	130
3.6.4 Optimizing the method of resolution	131
3.6.5 Exercises	132
4. First-Order Logic	135
4.1 Basic concepts of first-order logic	136
4.1.1 First-order structures	136
4.1.2 First-order languages	138
4.1.3 Terms and formulae	139
4.1.4 The semantics of first-order logic: an informal outline	143
4.1.5 Translating first-order formulae to natural language	146
4.1.6 Exercises	147
4.2 The formal semantics of first-order logic	152
4.2.1 Interpretations	152
4.2.2 Variable assignment and term evaluation	153

4.2.3 Truth evaluation games	156
4.2.4 Exercises	159
4.3 The language of first-order logic: a deeper look	161
4.3.1 Translations from natural language into first-order languages	161
4.3.2 Restricted quantification	163
4.3.3 Free and bound variables. Scope of a quantifier	164
4.3.4 Renaming of a bound variable in a formula. Clean formulae	165
4.3.5 Substitution of a term for a variable in a formula. Capture of a variable	166
4.3.6 Exercises	167
4.4 Truth, logical validity, equivalence and consequence in first-order logic	171
4.4.1 More on truth of sentences in structures. Models and countermodels	171
4.4.2 Satisfiability and validity of first-order formulae	172
4.4.3 Logical equivalence in first-order logic	173
4.4.4 Some logical equivalences involving quantifiers	174
4.4.5 Negating first-order formulae	175
4.4.6 Logical consequence in first-order logic	176
4.4.7 Exercises	180
4.5 Semantic tableaux for first-order logic	185
4.5.1 Some derivations using first-order semantic tableau	186
4.5.2 Semantic tableaux for first-order logic with equality	189
4.5.3 Discussion on the quantifier rules and on termination of semantic tableaux	189
4.5.4 Exercises	191
4.6 Prenex and clausal normal forms	195
4.6.1 Prenex normal forms	195
4.6.2 Skolemization	197
4.6.3 Clausal forms	198
4.6.4 Exercises	199
4.7 Resolution in first-order logic	201
4.7.1 Propositional resolution rule in first-order logic	201
4.7.2 Substitutions of terms for variables revisited	201
4.7.3 Unification of terms	202
4.7.4 Resolution with unification in first-order logic	204
4.7.5 Examples of resolution-based derivations	205

4.7.6 Resolution for first-order logic with equality	207
4.7.7 Optimizations of the resolution method for first-order logic	207
4.7.8 Exercises	207
4.8 Applications of first-order logic to mathematical reasoning and proofs	211
4.8.1 Proof strategies: direct and indirect proofs	211
4.8.2 Tactics for logical reasoning	215
4.8.3 Exercises	216

5. Number Theory

219

5.1 The principle of mathematical induction revisited	220
5.1.1 Exercises	222
5.2 Divisibility	224
5.2.1 Basic properties of divisibility	224
5.2.2 Division with a remainder	224
5.2.3 Greatest common divisor	225
5.2.4 Exercises	227
5.3 Computing greatest common divisors. Least common multiples	230
5.3.1 Euclid's algorithm for computing greatest common divisors	230
5.3.2 Least common multiple	232
5.3.3 Exercises	233
5.4 Prime numbers. The fundamental theorem of arithmetic	236
5.4.1 Relatively prime numbers	236
5.4.2 Prime numbers	237
5.4.3 The fundamental theorem of arithmetic	238
5.4.4 On the distribution of prime numbers	239
5.4.5 Exercises	240
5.5 Congruence relations	243
5.5.1 Exercises	246
5.6 Equivalence classes and residue systems modulo n	248
5.6.1 Equivalence relations and partitions	248
5.6.2 Equivalence classes modulo n . Modular arithmetic	249
5.6.3 Residue systems	250

5.6.4 Multiplicative inverses in \mathbb{Z}_n	251
5.6.5 Exercises	251
5.7 Linear Diophantine equations and linear congruences	253
5.7.1 Linear Diophantine equations	253
5.7.2 Linear congruences	254
5.7.3 Exercises	256
5.8 Chinese remainder theorem	257
5.8.1 Exercises	259
5.9 Euler's function. Theorems of Euler and Fermat	261
5.9.1 Theorems of Euler and Fermat	262
5.9.2 Exercises	264
5.10 Wilson's theorem. Order of an integer	266
5.10.1 Wilson's theorem	266
5.10.2 Order of an integer	266
5.10.3 Exercises	267
5.11 Application: public key cryptography	269
5.11.1 About cryptography	269
5.11.2 The idea of public key cryptography	269
5.11.3 The method RSA	270
5.11.4 Exercises	271
6. Combinatorics	274
6.1 Two basic counting principles	275
6.1.1 Exercises	281
6.2 Combinations. The binomial theorem	284
6.2.1 Counting sheep and combinations	284
6.2.2 Some important properties	286
6.2.3 Pascal's triangle	287
6.2.4 The binomial theorem	287
6.2.5 Exercises	289
6.3 The principle of inclusion-exclusion	293
6.3.1 Exercises	296

6.4 The Pigeonhole Principle	299
6.4.3 Exercises	302
6.5 Generalized permutations, distributions and the multinomial theorem	304
6.5.1 Arranging nondistinct objects	304
6.5.2 Distributions	306
6.5.3 The multinomial theorem	308
6.5.4 Summary	310
6.5.5 Exercises	311
6.6 Selections and arrangements with repetition; distributions of identical objects	312
6.6.1 Selections with repetition	312
6.6.2 Distributions of identical objects	314
6.6.3 Arrangements with repetition	315
6.6.4 Summary	316
6.6.5 Exercises	316
6.7 Recurrence relations and their solution	318
6.7.1 Recurrence relations. Fibonacci numbers	318
6.7.2 Catalan numbers	319
6.7.3 Solving homogeneous linear recurrence relations	322
6.7.4 Exercises	327
6.8 Generating functions	329
6.8.1 Introducing generating functions	329
6.8.2 Computing coefficients of generating functions	332
6.8.3 Exercises	335
6.9 Recurrence relations and generating functions	337
6.9.1 Exercises	341
6.10 Application: classical discrete probability	343
6.10.1 Common sense probability	343
6.10.2 Sample spaces	343
6.10.3 Discrete probability	345
6.10.4 Properties of probability measures	346
6.10.5 Conditional probability and independent events	348
6.10.6 Exercises	352

7. Graph Theory**356**

7.1 Introduction to graphs and digraphs	357
7.1.1 Graphs	357
7.1.2 Digraphs	364
7.1.3 Exercises	367
7.2 Incidence and adjacency matrices	370
7.2.1 Exercises	374
7.3 Weighted graphs and path algorithms	377
7.3.1 Dijkstra's algorithm	378
7.3.2 The Floyd–Warshall algorithm	381
7.3.3 Exercises	383
7.4 Trees	385
7.4.1 Undirected trees	385
7.4.2 Computing spanning trees: Kruskal's algorithm	388
7.4.3 Rooted trees	390
7.4.4 Traversing rooted trees	392
7.4.5 Exercises	393
7.5 Eulerian graphs and Hamiltonian graphs	395
7.5.1 Eulerian graphs and digraphs	396
7.5.2 Hamiltonian graphs and digraphs	398
7.5.3 Exercises	400
7.6 Planar graphs	404
7.6.1 Exercises	408
7.7 Graph colourings	411
7.7.1 Colourings	411
7.7.2 The four- and five-colour theorems	413
7.7.3 Exercises	414

Index**419**

List of Boxes

1. Preliminaries	1
Paradoxes	19
2. Sets, Relations, Orders	20
Russell's paradox	27
Georg Cantor	35
The notation of set theory	45
Fuzzy sets	50
Zermelo–Fraenkel set theory	58
The axiom of choice	68
The continuum hypothesis	74
Hilbert's Hotel	80
Charles Sanders Peirce	88
3. Propositional Logic	89
Aristotle	100
George Boole	109
David Hilbert	112
Early proof theory	120
Augustus De Morgan	126
The Boolean satisfiability problem and NP-completeness	133

4. First-Order Logic	135
Gottlob Frege	151
Giuseppe Peano	160
Bertrand Russell	170
Alfred Tarski	184
Kurt Gödel	194
Thoralf Skolem	200
Alonzo Church	210
Alan Turing	217
5. Number Theory	219
Carl Friedrich Gauss	223
Pythagoras of Samos	229
Euclid of Alexandria	235
Primality testing	242
Richard Dedekind	247
Babylonian mathematics	252
Diophantus of Alexandria	257
Ancient Chinese mathematics	260
Pierre de Fermat	265
Ancient and Medieval Indian and Arabic mathematics	268
History of codes and ciphers	272
6. Combinatorics	274
Stirling's approximation of $n!$	283
Pascal's triangle	292
Venn diagrams	298
Lejeune Dirichlet	304
Braille	312
Blaise Pascal	317
Fibonacci	328
Combinatorial games: Nim	336

Fibonacci numbers and the golden ratio	342
Pierre-Simon de Laplace	355

7. Graph Theory **356**

Leonhard Euler	369
Ramsey theory	376
Edsger Dijkstra	385
Paul Erdős	395
William Hamilton	403
Graph theory and Sudoku	410
The four-colour problem	417

Preface

Discrete Mathematics¹ refers to a range of mathematical disciplines that study discrete structures and phenomena, unlike other classical fields of mathematics, such as analysis, geometry and topology, which study continuous structures, processes and transformations. Intuitively put, discrete structures – such as the linear order of natural numbers or the set of cities and towns in a country together with the roads between them – consists of separable, discretely arranged objects, whereas continuous structures – such as the trajectory of a moving object, the real line, the Euclidian plane or a sphere – are densely filled, with no “gaps”. The more important distinction, however, is between the types of problems studied and solved in discrete and continuous mathematics and also between the main ideas and techniques that underly and characterize these two branches of mathematics. Nevertheless, this distinction is often blurred and many ideas, methods and results from each of these branches have been fruitfully applied to the other.

The intuitive explanation above is not meant to define what should be classified as Discrete Mathematics, as every such definition would be incomplete or debatable. A more useful description would be to list what we consider to be the basic mathematical disciplines traditionally classified as Discrete Mathematics and included in most university courses on that subject: the Theory of Sets and Relations, Mathematical Logic, Number Theory, Graph Theory and Combinatorics. These are the topics covered in this book. Sometimes textbooks and courses on Discrete Mathematics also include Abstract Algebra, Classical Probability Theory, Automata Theory, etc. These topics are not included here, mainly for practical reasons.

Logic was born in the works of Aristotle as a philosophical study of reasoning some 25 centuries ago. Over the past 150 years it has gradually developed as a fundamental mathematical discipline, which nowadays has deep and mature mathematical content and also applications spreading far beyond foundational and methodological issues. While the field of Mathematical Logic is often regarded as included in the broad scope of Discrete Mathematics, in this book it is treated essentially on a par with it.

As mathematical fields of their own importance, both Logic and Discrete Mathematics are relatively young and very dynamically developing disciplines, especially since the mid 20th century, when the computer era began. Many of the most exciting current developments

¹ Not to be confused with discreetly done mathematics!

in Logic and Discrete Mathematics are motivated and inspired by applications in Computer Science, Artificial Intelligence and Bioinformatics. We have accordingly included some such selected applications in the book.

About the book

The work on this book started more than a decade ago as a loose collection of lecture notes that we wrote and used to teach courses on Logic and Discrete Mathematics, partly because we could not find available suitable textbooks that would meet our needs and requirements. Eventually we decided to write a book of our own, which would best reflect the content and features we consider most important:

1. Being logicians, we have provided a much more detailed and deeper treatment of Logic than is usual in textbooks on Discrete Mathematics. We believe that such a treatment is necessary for the proper understanding and use of Logic as a mathematical and general reasoning tool, and we consider it a distinguishing feature of the book.
2. We included only what we consider to be the core disciplines within the field of Discrete Mathematics (without claiming that ours is the only good choice) but have treated these disciplines in considerable depth for an undergraduate text. That enabled us to keep the book within reasonable size limits without compromising on the content and exposition of the topics included.
3. We have tried to keep the exposition clear and concise while still including the necessary technical detail and illustrating concepts and techniques with numerous examples.
4. We have included comprehensive sets of exercises, most of them provided with answers or solutions in an accompanying solutions manual.
5. We have also included “boxes” at the end of each section. Some contain mathematics titbits or applications of the content in the section. Others are short biographies of distinguished scientists who have made fundamental contributions to Discrete Mathematics. We hope the reader will find it inspiring to learn a little about their lives and their contributions to the fields covered in the book.

To the instructor

We have aimed this book to be suitable for a variety of courses for students in both Mathematics and Computer Science. Some parts of it are much more relevant to only one of these audiences and we have indicated them by introducing *Mathematics Track* and *Computer Science Track* markers in the text. We regard everything not explicitly on either of these tracks to be suitable for both groups.

In addition, the book can be used for designing courses on different undergraduate or lower graduate levels. Some material that could reasonably be omitted in courses at a lower undergraduate level is indicated with an *Advances Track* marker. These tracks are, of course, only suggestions, which should serve as our recommendations to the instructor.

The single stars shown in the exercises are deemed to be exercises that are more difficult, while the double stars are considered to be exercises that are challenging.

The whole book can be comfortably covered in two semester courses, or various selections can be made for a single semester course. Apart from assuming knowledge of the background material in the preliminary Chapter 1, the chapters are essentially independent and can be taught in any order. The only exception is Chapter 4 on first-order logic, which presupposes knowledge of the material on propositional logic covered in Chapter 3. Also, much of the content of Chapter 2 covers general mathematical background, useful for the rest of the book.

Acknowledgements

A number of people have contributed, one way or another, to our work on this book over the years. With the risk of inadvertently omitting some names, for which we apologize in advance, we would like to thank Gerhard Benadé, Thomas Bolander, Izak Broere, Ruaan Kellerman, Heidi Maartens, Chantel Marais and Claudette Robinson for helpful feedback, comments, technical or editorial corrections and some solutions to exercises. Besides them, thanks are due to many students at the University of Johannesburg, the University of the Witwatersrand and the Technical University of Denmark, who pointed out some errors in earlier versions of the manuscript. We thank Justin Southey for the use of the GraphScribe application authored by him, which greatly speeded up the drawing of some combinatorial graphs, and also for his suggestions on the applications of Graph Theory. Valentin is also grateful to Nina Ninova for her general support during the work on this book as well as help with collecting photos and references for many of the historical boxes. Willem would like to thank Dion Govender for his patience and support during the final months of intense work on the book. Last but not least, we wish to thank Richard Davies, Audrey Koh, Kathryn Sharples, Mark Styles and Jo Taylor from Wiley as well as Krupa Muthu from Laserwords for their kind support and technical advice during the work on this book.

About the Companion Website

This book is accompanied by a companion website:

www.wiley.com/go/conradie/logic

The website includes:

- Lecture Slides
- Quizzes

1

Preliminaries

1.1. Sets	2
1.2. Basics of logical connectives and expressions	9
1.3. Mathematical induction	17

Here we briefly review some minimal background knowledge that we will assume in the rest of the book. Besides a small amount of that nebulous quality called “mathematical maturity”, we only expect some basic concepts from set theory and mathematical induction. The reader who is familiar with these concepts can safely skip on to the next chapter.

Some notation

We denote the set of natural numbers $\{0, 1, 2, \dots\}$ by \mathbb{N} . There is some inconsistency in the mathematical literature as to whether 0 belongs to the natural numbers or not: some authors choose to include it, other do not. For our purposes it is convenient to include 0 as a natural number. Other number sets which will be of importance to us include the sets of integers \mathbb{Z} , positive integers \mathbb{Z}^+ , rational numbers \mathbb{Q} , positive rational numbers \mathbb{Q}^+ , real numbers \mathbb{R} , and positive real numbers \mathbb{R}^+ .

The product $1 \times 2 \times 3 \times \dots \times n$ of the first n positive integers turns up in many mathematical situations. It is therefore convenient to have a more compact notation for it. We accordingly define $0! = 1$ and $n! = 1 \times 2 \times 3 \times \dots \times n$, for $n \geq 1$. We read $n!$ as ‘ n factorial’. The definition of $0!$ as 1 is not supposed to carry any intuitive meaning: it is simply a useful convention.

1.1. Sets

Sets and elements. By a **set** we intuitively mean a collection of objects of any nature (numbers, people, concepts, sets themselves, etc.) that is considered as a single entity. The objects in that collection are called **elements** of the set. If an object x is an element of a set A , we denote that fact by

$$x \in A;$$

otherwise we write

$$x \notin A.$$

We also say that x is a *member* of the set A or that x belongs to A . If a set has finitely many elements (here we rely on your intuition of what *finite* is), we can describe it precisely by listing all of them, for example:

$$A = \{3, 4, 5\}.$$

We often rely on our common intuition and use ellipses, as in

$$A = \{1, 2, \dots, n\}.$$

We sometimes go further and use the same for *infinite* sets; for example, the set of natural numbers can be specified as

$$\mathbb{N} = \{0, 1, 2, 3, \dots\}.$$

Further we will discuss a more universal method of describing sets.

Equality and containment of sets. Two sets are declared **equal** if and only if they have the same elements. In other words, the sets A and B are equal, denoted as usual by $A = B$, if every element of A is an element of B and every element of B is an element of A . For example, the sets $\{a, b, c\}$ and $\{b, c, a\}$ are equal, and so are the sets $\{1, 9, 9, 7\}$, $\{1, 9, 7\}$ and $\{7, 1, 9, 1, 7, 1\}$.

A set A is a **subset** of a set B , denoted $A \subseteq B$, if every element of A is an element of B . If $A \subseteq B$, we also say that A is **included** in B , or that B **contains** A . For example, $\{3, 5\} \subseteq \{5, 4, 3\}$. Note that every set is a subset of itself.

The following facts are very useful. They are direct consequences of the definitions of equality and containment of sets.

- Two sets A and B are equal if, and only if, $A \subseteq B$ and $B \subseteq A$.
- A set A is not a subset of a set B , denoted $A \not\subseteq B$, if, and only if, there is an element of A that is not an element of B .
- A set A is not equal to a set B if A is not a subset of B or if B is not a subset of A .

A set A is a **proper subset** of a set B , denoted $A \subset B$ or $A \not\subseteq B$, if $A \subseteq B$ and $A \neq B$. In other words, A is a proper subset of B if A is a subset of B and B is *not* a subset of A , i.e. if at least one element of B is not in A . In particular, no set is a proper subset of itself. If A is not a proper subset of B , we denote that by $A \not\subset B$.

The empty set. Amongst all sets there is one that is particularly special. That is the **empty set**, i.e. the set that has no elements. By definition of equality of sets, there is only one empty

set. One might think that the empty set is a useless abstraction. On the contrary, it is a very important set, and probably the most commonly used one in mathematics (like the number 0 is the most commonly used number). That is why it has a special notation: \emptyset .

Sets and properties. Set-builder notation. We cannot always list the elements of a set, even if it is finite, so we need a more universal method for specifying sets. The commonly used method is to *describe the property that determines membership of the set*, e.g.:

“ A is the set of all objects x such that $\dots x \dots$ ”

where “ $\dots x \dots$ ” is a certain property (predicate) involving x . We use the following convenient notation, called the **set-builder notation** for the set described above:

$$A = \{x \mid \dots x \dots\}.$$

Here are some examples:

- $\{x \mid x \text{ is a negative real number}\}$ defines the set of negative real numbers;
- $\{x \mid x \text{ is a student in the MATH3029 class}\}$ defines the set of students in the MATH3029 class.
- $\{x \mid x \in \mathbb{Z} \text{ and } 3 \geq x > -2\}$ defines the set $\{-1, 0, 1, 2, 3\}$.
- $\{x \mid x = \frac{m}{n}, \text{ where } m \in \mathbb{Z}, n \in \mathbb{Z} \text{ and } n \neq 0\}$ defines the set of rational numbers.

Sometimes, we use the set-builder notation more liberally and, for instance, describe the set of rational numbers as $\left\{ \frac{m}{n} \mid m \text{ and } n \text{ are integers and } n \neq 0 \right\}$ or the set of positive real numbers as $\{x \in \mathbb{R} \mid x > 0\}$.

Operations on sets. We describe below the basic operations on sets.

Intersection. The intersection of two sets A and B is the set

$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$

consisting of all elements that are both in A and in B . If $A \cap B = \emptyset$, then A and B are called **disjoint**.

Union. The union of two sets A and B is the set

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

consisting of all elements that are in at least one of A and B .

Difference. The difference of the sets A and B is the set

$$A - B = \{x \mid x \in A \text{ and } x \notin B\}$$

consisting of all elements that are in A but not in B . An alternative notation for $A - B$ is $A \setminus B$.

For example, if $A = \{1, 2, 3, 4\}$ and $B = \{3, 4, 5, 6, 7\}$ then $A \cap B = \{3, 4\}$, $A \cup B = \{1, 2, 3, 4, 5, 6, 7\}$, $A - B = \{1, 2\}$ and $B - A = \{5, 6, 7\}$.

Universal sets and complements of sets. Often, all sets that we consider are subsets of one set, called the **domain of discourse**. We also call that set the **universe** or the **universal set**. For example, in arithmetic, the universe is usually the set of natural numbers or the set of integers, while in algebra and calculus, the universe is the set of real numbers; talking about humans, the universe is the set of all humans, etc.

Definition 1.1.1 Let a universal set \mathbf{U} be fixed and $A \subseteq \mathbf{U}$. The complement of A (with respect to \mathbf{U}) is the set

$$A' = \mathbf{U} - A.$$

The complement of a set A is sometimes also denoted by \overline{A} .

Thus, the complement of A consists of those objects from the universal set that do not belong to A . For example, if the universal set is \mathbb{R} , then the complement of the interval $(0, 2]$ is $(-\infty, 0] \cup (2, \infty)$; the complement of \mathbb{Q} is the set of irrational numbers.

Powersets. The power set of a set A is the set of all subsets of A :

$$\mathcal{P}(A) = \{X \mid X \subseteq A\}.$$

Here are some examples:

- $\mathcal{P}(\emptyset) = \{\emptyset\}$;
- $\mathcal{P}(\{a\}) = \{\emptyset, \{a\}\}$, in particular, $\mathcal{P}(\{\emptyset\}) = \{\emptyset, \{\emptyset\}\}$;
- $\mathcal{P}(\{a, b\}) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$;
- $\mathcal{P}(\{a, b, c\}) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.

Cartesian products of sets. In order to introduce the next operation we need the notion of **ordered pair**. Let a, b be any objects. Intuitively, the ordered pair of a and b , denoted (a, b) (do not confuse this with an open interval of real numbers!), is something like a set consisting of a as a *first element* (or *first component*) and b as a *second element* (or *second component*). Thus, if $a \neq b$, then the ordered pair (a, b) is *different* from the ordered pair (b, a) and each of these is different from the set $\{a, b\}$ because the elements of a set are not ordered. In particular, the ordered pair (a, a) is different from the set $\{a, a\} = \{a\}$. Here is a formal definition of an ordered pair as a set that satisfies the intuition:

Definition 1.1.2 Given the objects a and b , the **ordered pair** (a, b) is the set $\{\{a\}, \{a, b\}\}$.

Here is the fundamental property of ordered pairs:

Proposition 1.1.3 The ordered pairs (a_1, a_2) and (b_1, b_2) are equal if and only if $a_1 = b_1$ and $a_2 = b_2$.

Proof: Exercise.

Definition 1.1.4 The **Cartesian product** of the sets A and B is the set

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\},$$

consisting of all ordered pairs where the first component comes from A and the second component comes from B . In particular, we denote $A \times A$ by A^2 and call it the **Cartesian square** of A .

For example, if $A = \{a, b\}$, $B = \{1, 2, 3\}$, then

$$A \times B = \{(a, 1), (a, 2), (a, 3), (b, 1), (b, 2), (b, 3)\},$$

while

$$B \times A = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}.$$

Note that if A or B is empty, then $A \times B$ is empty too. Moreover, if A has n elements and B has m elements, then $A \times B$ has mn elements (why?). This is one of the reasons for the term “product”.

The Cartesian coordinate system in the plane is a representation of the plane as the Cartesian¹ square \mathbb{R}^2 of the real line \mathbb{R} , where we associate a unique ordered pair of real numbers (its **coordinates**) with every point in the plane.

The notion of an ordered pair can be generalized to **ordered n -tuple**, for any $n \in \mathbb{N}^+$. An n -tuple is an object of the type (a_1, a_2, \dots, a_n) where the order of the components a_1, a_2, \dots, a_n matters. We will not give a formal set theoretic definition in the style of Definition 1.2.1, but leave this as an exercise (Exercise 11).

Accordingly, the Cartesian product can be extended to n sets:

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \dots, a_n) \mid a_1 \in A_1, a_2 \in A_2, \dots, a_n \in A_n\}.$$

As before, we will use the notation A^n for $\underbrace{A \times A \times \cdots \times A}_{n \text{ times}}$.

Relations. Relations, also called **predicates**, are ubiquitous in mathematics. Relations between numbers like “being equal”, “being less than” and “being divisible by” come to mind at once. As these examples indicate, many of the relations we commonly encounter are *binary*, i.e., relations relating *two* objects at a time. It will be convenient for us to identify a binary relation with the set of all ordered pairs of elements that stand in that relation. We thus have the following definition:

Definition 1.1.5 *A binary relation on a set A is any subset of A^2 .*

For example the relation $<$ on the set \mathbb{N} of natural numbers is a binary relation, which we identify with the set

$$\{(a, b) \mid a, b \in \mathbb{N} \text{ and } a \text{ is less than } b\}.$$

The relation of “being the mother of” is a binary relation on the set of humans, which we identify with the following set of ordered pairs:

$$\{(x, y) \mid x \text{ and } y \text{ are humans and } x \text{ is the mother of } y\}.$$

Definition 1.1.6 *An n -ary relation on a set A is any subset of A^n .*

¹ The term “Cartesian” comes from the name of the French mathematician René Descartes (1596–1650), who was the first to introduce coordinate systems and to apply algebraic methods in geometry.

More generally, relations may relate objects from different sets:

Definition 1.1.7 *An **n-ary relation** between sets A_1, A_2, \dots, A_n is any subset of $A_1 \times A_2 \times \dots \times A_n$.*

If $n = 1$ we speak of a **unary relation**, when $n = 2$ of a **binary relation**, when $n = 3$ of a **ternary relation** and so on.

Unary relations correspond to sets. For instance, “being positive” is a unary relation on the set \mathbb{R} , which we identify with the set

$$\{a \mid a \in \mathbb{R} \text{ and } 0 < a\}.$$

Conversely, every set A defines the unary relation of “being a member of A ”.

The set

$$\{(f(x), a) \mid f(x) \text{ is a polynomial, } a \in \mathbb{R} \text{ and } f(a) = 0\}$$

is a binary relation between the set of all polynomials in x and the set \mathbb{R} of real numbers, relating every polynomial to its real roots. The ordered pairs $(x^2 - 4, 2)$ and $(x^2 - 4, -2)$ are in this relation. There is no pair in this relation that has $x^2 + 2$ as first component. Why?

Examples of ternary relations are the relation “ A is between B and C ” relating triples of points in the plain or the relation “ a is greater than $b + c$ ” between triples of real numbers.

Functions. Informally, a function from a set A to a set B , denoted by $f : A \rightarrow B$, can be thought a rule that assigns to every element $a \in A$ a unique element $f(a) \in B$. A is called the **domain** of f while B is the **codomain**, or the **target set** of f .

Functions need not take only one argument. The function addition on \mathbb{R} , for example, takes two arguments. In general, a function can take any finite number of arguments, but it can always be regarded as a one-argument function if we consider the domain to be the Cartesian product of the domains of the different arguments. For instance, we can think of $+$ as a one-argument function from $\mathbb{R} \times \mathbb{R}$ to \mathbb{R} . Here is the general definition.

Definition 1.1.8 *An **n-ary function** f from $A_1 \times A_2 \times \dots \times A_n$ to B , denoted*

$$f : A_1 \times A_2 \times \dots \times A_n \rightarrow B,$$

is any rule that to every ordered n -tuple $(a_1, a_2, \dots, a_n) \in A_1 \times A_2 \times \dots \times A_n$ assigns a unique value $f(a_1, a_2, \dots, a_n) \in B$.

In particular, when $A_1 = A_2 = \dots = A_n = A$ we call f an n -ary function on the set A , denoted

$$f : A^n \rightarrow A.$$

Functions are naturally associated with special relations, called their graphs. For instance, think of some familiar function $f : \mathbb{R} \rightarrow \mathbb{R}$, such as $f(x) = x^2$. The graph of this function, as drawn in the plane, consists of the sets of points (x, y) in the Cartesian plane such that $y = x^2$, i.e. the set $\{(x, x^2) \mid x \in \mathbb{R}\}$. This set completely determines the function. Here is the general definition.

Definition 1.1.9 *Given an n -ary function $f : A_1 \times \dots \times A_n \rightarrow B$ the **graph of f** is the subset of $A_1 \times A_2 \times \dots \times A_n \times B$, defined as follows:*

$$G_f = \{(a_1, a_2, \dots, a_n, f(a_1, a_2, \dots, a_n)) \mid (a_1, a_2, \dots, a_n) \in A_1 \times A_2 \times \dots \times A_n\}$$

Sometimes it is technically convenient to identify functions with their graphs. Thus, an n -ary function f can be alternatively defined as a subset of $A_1 \times A_2 \times \cdots \times A_n \times B$, such that for all $(a_1, a_2, \dots, a_n) \in A_1 \times A_2 \times \cdots \times A_n$, there is a unique $b \in B$ such that $(a_1, a_2, \dots, a_n, b) \in f$. This last condition means that if $(a_1, a_2, \dots, a_n, b) \in f$ and $(a_1, a_2, \dots, a_n, b') \in f$ then $b = b'$. Thus an n -ary function can be regarded as a special type of $(n + 1)$ -ary relation. For instance, the multiplication of two integers can be regarded as the ternary relation:

$$\{(a, b, c) \mid a, b, c \in \mathbb{Z}, \text{ and } a \times b = c\}.$$

We will generally use the standard definition of functions as rules but, whenever convenient, we may also adopt the handling functions in terms of their graphs.

1.1.1. Exercises

❶ List all elements of the following sets:

- (a) $A = \{x \in \mathbb{R} \mid x^2 - 3x = 4\}$
- (b) $B = \{y \in \mathbb{Z} \mid (y - 1)(y + 3)(2y + 3)(y + 5) = 0\}$
- (c) $C = \{x \in \mathbb{Z} \mid -3 \leq x < 3\}$
- (d) $D = \{x \in \mathbb{Z} \mid -3 \leq x < 3 \wedge x^2 - 3x = 4\}$
- (e) $E = \{x \in \mathbb{Z} \mid -3 \leq x < 3 \wedge x^2 - 3x \neq 4\}$
- (f) $F = \{x \in \mathbb{N} \mid x \text{ is an odd single-digit number}\}$

❷ Describe the following sets, using set-builder notation:

- (a) $B = \{-\sqrt{3}, \sqrt{3}\}$
- (b) $A = \{2, 4, 6, 8\}$
- (c) $A = \{2, 4, 6, 8, 10, 12, \dots\}$
- (d) $A = \{\dots, -7, -4, -1, 2, 5, 8, \dots\}$
- (e) $A = \{0, 1, 8, 27, 64, 125, \dots\}$
- (f) $A = \{-3, -2, -1, 1, 2, 3, 4, 5, 6\}$
- (g) $A = \{\dots, -6, -5, -4, -3, 5, 6, 7, 8, \dots\}$

❸ Let $A = \{a, b, c\}$. Which of the following are true?

- | | |
|-------------------------|-------------------------------|
| (a) $c \in A$ | (f) $\{a\} \in A$ |
| (b) $b \subset A$ | (g) $\{a, c\} \subset A$ |
| (c) $b \subseteq A$ | (h) $\{a, c\} \subseteq A$ |
| (d) $\{a\} \subset A$ | (i) $\{a, b, c\} \subset A$ |
| (e) $\{a\} \subseteq A$ | (j) $\{a, b, c\} \subseteq A$ |

④ Let $A = \{1, 2\}$, $B = \{1, \{2\}, A\}$. Which of the following are true?

- | | | |
|-----------------------|-------------------------|--------------------------------|
| (a) $1 \in A$ | (f) $\{2\} \subset B$ | (k) $\{A\} \in B$ |
| (b) $2 \in B$ | (g) $\{2\} \subset A$ | (l) $\{\{2\}\} \subseteq B$ |
| (c) $\{2\} \subset B$ | (h) $\{A\} \subset A$ | (m) $\{1, 2\} \in B$ |
| (d) $A \subset B$ | (i) $\{A\} \subseteq B$ | |
| (e) $2 \subset B$ | (j) $A \in B$ | (n) $\{\{1, 2\}\} \subseteq B$ |

⑤ Let $U = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be the universal set, $A = \{1, 2, 4, 7\}$ and $B = \{0, 2, 5, 6, 7, 9\}$.

Determine:

- | | | |
|----------------|-------------------|------------------------------|
| (a) $A \cap B$ | (e) $A - (A - B)$ | (i) $A \cap B'$ |
| (b) $A \cup B$ | (f) $B - (A - B)$ | (j) $(A \cup B)'$ |
| (c) $A - B$ | (g) B' | (k) $(A \cup B)' - (B - A)'$ |
| (d) $B - A$ | (h) $(A \cap B)'$ | |

⑥ Let $A = \{0\}$, $B = \{0, A\}$ and $C = \{0, \{0\}, \{A\}\}$. Determine:

- | | | |
|----------------|----------------------------------|----------------------|
| (a) $A \cap B$ | (d) $A \cap (B \cup C)$ | (g) $(C - B) - A$ |
| (b) $A \cup B$ | (e) $(B \cup C) - A$ | (h) $A \cap (C - A)$ |
| (c) $B - A$ | (f) $(A \cap B) \cup (A \cap C)$ | |

⑦ If $A = \{1, 2, 3, 4\}$ and $B = \{0, 1\}$, list the elements of $A \times B$ and of $B \times A$.

⑧ If $C = \{a, b\}$, $D = \{\alpha, \beta, \gamma, \delta, \theta\}$ and $E = \{a, c, d\}$, how many elements does each of the following sets have?

- | | |
|---------------------------|--------------------------------------|
| (a) $C \times D \times E$ | (e) $(C \cup D) \times (C \times E)$ |
| (b) $C \times (D \cap E)$ | |
| (c) $D \times (C \cap E)$ | (f) $(C \times E) \cap (E \times C)$ |
| (d) $D \cap (C \times E)$ | |

⑨ Which of the following relations are graphs of functions?

- | |
|--|
| (a) $\{(a, b) \mid a, b \in \mathbb{R} \text{ and } b = a^2\}$ |
| (b) $\{(a, b) \mid a, b \in \mathbb{R} \text{ and } a = b^2\}$ |
| (c) $\{(a, b, c) \mid a, b, c \in \mathbb{Z} \text{ and } c \text{ divides } a \times b \text{ without remainder}\}$ |
| (d) $\{(x, y) \mid x \text{ and } y \text{ are humans and } y \text{ is a parent of } x\}$ |
| (e) $\{(x, y) \mid x \text{ and } y \text{ are humans and } y \text{ is the mother of } x\}$ |

⑩ Prove Proposition 1.2.1: two ordered pairs (a_1, a_2) and (b_1, b_2) (seen as special sets as in Definition 1.2.1) are equal if and only if $a_1 = b_1$ and $a_2 = b_2$.

⑪ Generalize Definition 1.2.1 to a formal set-theoretic definition of an ordered n -tuple.

1.2. Basics of logical connectives and expressions

Here we only provide a compendium of basic logical symbols and notation, commonly used in mathematics. We will not discuss here the underlying concepts of logical languages, semantics and deduction; these will be presented in a detailed and systematic way in the chapters on logic, Chapters 3 and 4.

1.2.1. Propositions, logical connectives, truth tables, tautologies

Propositions

The basic concept of propositional logic is the **proposition**. A proposition is a sentence that can be assigned a **truth value**: true or false.

Some simple examples are:

- The Sun is hot.
- The Earth is made of cheese.
- 2 plus 2 equals 22.
- The 999-th decimal digit of the number π is 9.

Here are some sentences that are not propositions (why?):

- Are you bored?
- Please, don't go away!
- She loves him.
- x is an integer.
- This sentence is false.

Propositional logical connectives

The propositions above are very simple. They have no logical structure, so we call them **primitive** propositions. From primitive propositions one can form **compound** ones by using **logical connectives**. The most commonly used connectives are:

- **not**, called **negation**, denoted by \neg ;
- **and**, called **conjunction**, denoted by \wedge (or sometimes, by $\&$);
- **or**, called **disjunction**, denoted by \vee ;
- **if...then...**, called **implication**, or **conditional**, denoted by \rightarrow ;
- **... if and only if ...**, called **biconditional**, denoted by \leftrightarrow .

Remark 1 It is usually not grammatically correct to read compound propositions by simply inserting the names of the logical connectives in between the primitive components. A typical problem arises with the negation: one does not say “Not the earth is square”. A uniform way to get round that difficulty and to negate a proposition P is to say “It is not the case that P ”.

Thus, given the propositions

“Two plus two equals five” and “The Sun is hot”

we can form the propositions

- “It is **not** the case that two plus two equals five.”
- “Two plus two equals five **and** the Sun is hot.”
- “Two plus two equals five **or** the Sun is hot.”
- “If two plus two equals five **then** the Sun is hot.”
- “Two plus two equals five **if and only if** the Sun is hot.”

Example 1.2.1

Here is a more involved example. Suppose Mary is a particular person, about whom it is known whether she is clever, lazy, and whether she likes logic. Then, we can consider the following sentences as propositions:

“Mary is clever”, “Mary is lazy” and “Mary likes logic.”

From these, we can compose a proposition (smoothed out a bit), such as

“Mary is not clever or if she likes logic then she is clever and not lazy.”

Truth tables

How about the truth value of a compound proposition? It can be *calculated* from the truth values of the components (in much the same way as we can calculate the value of the algebraic expression $A \times (B - C) + B/A$ as soon as we know the values of A, B, C) by following the rules of “propositional arithmetic”:

- The proposition $\neg A$ is true if and only if the proposition A is false.
- The proposition $A \wedge B$ is true if and only if both A and B are true.
- The proposition $A \vee B$ is true if and only if either of A or B (possibly both) is true.
- The proposition $A \rightarrow B$ is true if and only if A is false or B is true, i.e. if the truth of A implies the truth of B .
- The proposition $A \leftrightarrow B$ is true if and only if A and B have the same truth values.

We can systematize these rules in kinds of “multiplication tables”. For that purpose, and to make it easier for symbolic (i.e. mathematical) manipulations, we introduce a special notation for the two truth values by denoting the value `true` by `T` and the value `false` by `F`. Another common notation, particularly used in computer science, is to denote `true` by `1` and `false` by `0`.

Now the rules of the “propositional arithmetic” can be tabulated by means of the following *truth tables* (p and q below stand for arbitrary propositions):

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
p	$\neg p$				
T	F	T	T	T	T
F	T	F	T	F	F
		F	T	T	F
		F	F	T	T

Computing truth values of propositions

One can see from the truth tables that *the truth value of a compound proposition does not depend on the meaning of the component propositions, but only on their truth values*. Therefore, to check the truth of such a proposition we merely need to replace all component propositions by their respective truth values and then ‘calculate’ the truth of the whole proposition, using the truth tables of the logical connectives. Thus,

- “It is not the case that two plus two equals five” is true;
- “Two plus two equals five and the Sun is hot” is false;
- “Two plus two equals five or the Sun is hot” is true;
- “If two plus two equals five, then the Sun is hot” is true (even though it does not make much sense).

Returning to Example 1.2.1, suppose we know that

“Mary is clever” is true,
 “Mary is lazy” is false,
 “Mary likes logic” is true.

Then the truth value of the compound proposition

“Mary is not clever or if she likes logic, then she is clever and not lazy”

can be determined just as easily. However, in order to do so, we first have to analyze the *syntactic structure* of the proposition, i.e. to determine how it has been composed or in what order the logical connectives occurring therein have been applied. With algebraic expressions such as $A \times (B - C) + B/A$ that analysis is a little easier thanks to the use of parentheses and the established priority order amongst the arithmetic operations. Let us also make use of parentheses and rewrite the sentence in the way we (presumably) all understand it:

“(Mary is not clever) or (if (she likes logic), then ((she is clever) and (not lazy))).”

The structure of the sentence should be clear now. We can, however, go one step further and make it look exactly like an algebraic expression by using letters to denote the occurring

Table 1.1 Logical connectives paired with set theoretic operations

Logical symbol	Set-theoretic symbol
\wedge	\cap
\vee	\cup
\neg	$'$

primitive propositions. Let us, for instance, denote

- “Mary is clever” by A ,
- “Mary is lazy” by B ,
- “Mary likes logic” by C .

Now our compound proposition can be neatly rewritten as

$$(\neg A) \vee (C \rightarrow (A \wedge \neg B)).$$

The last step is to calculate the truth value. Here we make use of our observation above and simply replace the primitive propositions A, B and C by their truth values and do the formal calculation following the truth tables step by step:

$$(\neg T) \vee (T \rightarrow (T \wedge \neg F)) = F \vee (T \rightarrow (T \wedge T)) = F \vee (T \rightarrow T) = F \vee T = T.$$

A note on the relation between logic and set theory

The astute reader would have noticed a close connection between basic, “propositional” logical connectives and their properties, studied further in Chapter 3, on the one hand, and the basic set theoretic operations and their properties, on the other. To explore this connection fully would lead us far beyond the scope of this book, but a few comments are nevertheless in order. Firstly, we can compile something like a “translation manual” of the symbols, as depicted in Table 1.1.

Using this table, one can translate any proposition into a set-theoretic expression (by regarding the primitive propositions as names of sets) and conversely. For example, the set theoretic translation of the propositional $A \vee \neg A$ would be $A \cup A'$, with A regarded as the name of a set. How are these related? Intuitively, two propositions have the same logical meaning precisely when their “set-theoretic translations” represent equal sets. We will give a more precise answer in Chapter 3.

1.2.2. Individual variables and quantifiers

Using only propositional logical connectives we cannot express very simple mathematical statements such as

- “ $x + 2$ is greater than 5”,
- “There exists y such that $y^2 = 2$ ”,

"For every real number x , if x is greater than 0, then there exists a real number y such that y is less than 0 and y^2 equals x ", etc.

Indeed, an expression such as " $x + 2$ is greater than 5" is *not* a proposition, for it can be true or false depending on the choice of x . Neither is "There exists y such that $y^2 = 2$ " a proposition until the range of possible values of y is specified: if y is an integer, then the statement is false, but if y can be any real number, then it is true. As for the third sentence above, it *is* a proposition, but its truth heavily depends on its internal logical structure and mathematical meaning of all phrases involved, and those are not tractable on a propositional level. All these sentences are from the realm of **first-order logic**, often just called **classical logic**, which we study in Chapter 4. Here we only introduce the basic symbols of first-order logic as they are widely used in the mathematical literature, and will also be used in Section 2.4, before we study logic itself.

Individual variables

Often in mathematics we deal with unknown or unspecified objects (individuals) from the domain of discourse: integers, real numbers, geometric points, elements of abstract sets, etc. In order to be able to reason and make statements about such objects, we use **individual variables** to denote them. For instance, talking about numbers, we use phrases like "Take a positive integer n ", "For every real number x greater than $\sqrt{2}$...", etc. In natural language instead of variables we usually use pronouns or other syntactic constructions but that can often lead to awkwardness or even ambiguity (e.g. "If a man owes money to another man then that man hates the other man"). Therefore, the use of variables in mathematics is indispensable. We also use variables as *placeholders* for the arguments of the various predicates and functions we deal with. Usually, individual variables will be denoted by x, y, z, \dots , possibly with indices.

Quantifiers

In both mathematical discourse and in everyday speech, we often use special phrases used to quantify over objects of our discourse, such as (note the use of individual variables here):

- "(for) every (objects) x ",
- "there is (an object) x such that",
- "for most (objects) x ",
- "there are at least 5 (objects) x such that",
- "there are more (objects) x than y such that", etc.

Such phrases are called **quantifiers**. The first two of those listed above are particularly important and most commonly used in mathematics, and many others can be expressed by means of them, so they are given special names and notation:

- The quantifier "for every" is called **universal quantifier**, denoted by \forall .
- The quantifier "there exists" is called **existential quantifier**, denoted by \exists .

Besides the phrases above, the universal quantifier is usually represented by “*all*”, “*for all*” and “*every*” while the existential quantifier can appear as “*there is*”, “*some*” and “*for some*”, particularly, in a nonmathematical discourse.

Mathematical terms and formulae

Typically, we build mathematical expressions by using individual variables and constants (names) for specific individuals (e.g. the numbers $0, -5, \sqrt{2}, \pi$) and functions and relations that we apply to these. We also use auxiliary symbols, such as parentheses, and **logical symbols**, including the propositional connectives introduced earlier, as well as equality $=$ and quantifiers. Using all these components and following certain syntactic rules, we can compose formal expressions that allow us to symbolically represent mathematical statements, to reason about them and to prove them in precise, well-structured and logically correct ways, which we will present and discuss in detail in Chapter 4.

There are two main types of expressions used in mathematics:

1. **Terms**, used to denote objects, which we build from names and variables for such objects, possibly by applying functions to them. Some examples of mathematical terms are: $1, x, (x + 1), (x + 1)(x - 2), \sqrt{(x + 1)(y - z)}, \sin(x + \pi/2)$, etc.
2. **Formulae**, used to make statements about objects, by applying relations between terms, and then composing more complex formulae from simpler ones by applying logical symbols. Here are some examples of formulae used to make statements about numbers:
 - $x > 0$, saying that (the value of) x is greater than 0. This could be true or false, depending on the actual value of x .
 - $x \times y = y \times x$, saying that multiplication is commutative. This is true for all real values of x and y but not necessarily true if x and y denote, say, matrices.
 - $x < y \rightarrow \neg y < x$, saying that if (the value of) x is less than (the value of) y then it is not the case that y is less than x . Again, this is true for all numerical values of x and y , because of the standard meaning of $<$ and $>$.
 - $\exists x(x < 0)$, saying that there is a (value of) x that is less than 0. This is true if x is an integer or real, but false if x represents a natural number.
 - $\forall x \forall x(x < y \rightarrow \neg y < x)$, saying that for all (values of) x and y , if x is less than y then it is not the case that y is less than x . As noted above, this is true if x and y represent natural numbers, because of the meaning of $<$ and $>$ on real numbers.
 - $\forall x(x > 0 \rightarrow \exists y(y > 0 \wedge y < x))$, saying that for every positive (value of) x there is a lesser, still positive (value of) y . This is true if x and y represent rational or real numbers, but false if they represent integers, when x is taken as 1.

At this stage it is sufficient to understand the meaning of such mathematical expressions and statements informally and to be able to use logical symbols to make such simple statements. In Chapter 4 these skills will be lifted to a higher level and be made much more precise.

1.2.3. Exercises

❶ Which of the following are propositions?

- (a) $2^3 + 3^2 = 17$
- (b) $2^3 + 3^2 = 71$
- (c) $2^3 + 3^2 = x$
- (d) Will you marry me?
- (e) John married on January 1st, 1999.
- (f) Mary is not happy.
- (g) I told her about John.
- (h) If John loves Lisa then Mary is not happy.

❷ If A and B are true propositions and C and D are false ones, determine the truth values of the following compound propositions:

- (a) $A \wedge (\neg B \vee C)$
- (b) $\neg(\neg A \vee D) \rightarrow \neg B$
- (c) $\neg(\neg D \wedge (B \rightarrow \neg A))$
- (d) $(C \rightarrow A) \rightarrow D$
- (e) $C \rightarrow (A \rightarrow D)$
- (f) $\neg(C \rightarrow A) \vee (C \rightarrow D)$
- (g) $(A \rightarrow \neg A) \vee (C \leftrightarrow \neg B)$

❸ Write each of the following composite propositions in a symbolic form by identifying its atomic propositions and logical structure. Then determine its truth value.

- (a) The Sun is hot and if the Earth is larger than Jupiter then there is life on Jupiter.
- (b) If the Sun rotates around the Earth or the Earth rotates around the Moon then the Sun rotates around the Moon.
- (c) The Moon does not rotate around the Earth if the Sun does not rotates around the Earth and the Earth does not rotate around the Moon.
- (d) The Earth rotate around itself only if the Sun rotates around the Earth or the Moon does not rotate around the Earth.
- (e) The Earth rotates around itself if and only if the Sun does not rotate around itself or the Moon does not rotate around itself.

❹ Determine the truth value of the proposition A in each of the following cases:

- (a) B and $B \rightarrow A$ are true.
- (b) $A \rightarrow B$ is true and B is false.

- (c) $\neg B$ and $A \vee B$ are true.
- (d) Each of $B \rightarrow \neg A$, $\neg B \rightarrow \neg C$ and C is true.
- (e) Each of $\neg C \wedge B$, $C \rightarrow (A \vee B)$ and $\neg(A \vee C) \rightarrow C$ is true.
 (Hint: consider cases for the truth of B and C .)
- ⑤ Translate the following formulae into English and determine which of them are true propositions in the set of natural numbers \mathcal{N} and which are true propositions in the set of real numbers \mathcal{R} .
- (a) $\neg \forall x(x \neq 0)$
- (b) $\forall x(x^3 \geq x)$
- (c) $\forall x(x = x^2 \rightarrow x > 0)$
- (d) $\exists x(x = x^2 \wedge x < 0)$
- (e) $\forall x(x > 0 \rightarrow x^2 > x)$
- (f) $\forall x(x = 0 \vee \neg x + x = x)$
- (g) $\forall x \forall y(x > y \vee y > x)$
- (h) $\forall x \exists y(x > y^2)$
- (i) $\forall x \exists y(x > y^2 \vee y > 0)$
- (j) $\forall x(x \geq 0 \rightarrow \exists y(y > 0 \wedge x = y^2))$
- (k) $\forall x \exists y(x > y \rightarrow x > y^2)$
- (l) $\forall x \exists y(\neg x = y \rightarrow x > y^2)$
- (m) $\exists x \forall y(x > y)$
- (n) $\exists x \forall y(x + y = x)$
- (o) $\exists x \forall y(x + y = y)$
- (p) $\exists x \forall y(x > y \vee \neg x > y)$
- (q) $\exists x \forall y(x > y \vee \neg x > y)$
- (r) $\exists x \forall y(y > x \rightarrow y^2 > x)$
- (s) $\exists x \forall y(x > y \rightarrow x > y^2)$
- (t) $\exists x \exists y(xy = x + y)$
- (u) $\forall x \forall y(x > y \rightarrow \exists z(x > z \wedge z > y))$

1.3. Mathematical induction

Mathematical induction is a method for proving statements about all natural numbers. Recall that we assume the set of natural numbers to be $\mathbb{N} = \{0, 1, 2, \dots\}$. Here is the most common version of the Principle of Mathematical Induction (PMI):

Theorem 1.3.1 (Principle of Mathematical Induction) *Suppose that some property P of natural numbers holds for 0 and whenever P holds for some natural number k , then it also holds for $k + 1$. Then P holds for every natural number.*

We write $P(n)$ to say that the property P holds for the number n . Thus, using the PMI to prove a statement of the type “ $P(n)$ holds for every natural number n ” requires proving the following two steps:

1. *Base step:* $P(0)$ holds.
2. *Induction step:* for any $k \in \mathbb{N}$, if $P(k)$ holds, then $P(k + 1)$ also holds.

We illustrate the use of the PMI with the following example.

Example 1.3.2

Prove that for every natural number n ,

$$0 + 1 + \dots + n = \frac{n(n + 1)}{2}. \quad (1.1)$$

Proof: Let us denote the statement in Equation (1.1) by $P(n)$. First, we have to verify the Base step, i.e. to show that $P(0)$ is true. Here:

$$0 = \frac{0(0 + 1)}{2}.$$

Then, to verify the Induction step, suppose that $P(k)$ is true for some $k \geq 0$. This assumption is called the *Inductive Hypothesis (IH)*. We now have to show that $P(k + 1)$ is also true. Here we go:

$$\begin{aligned} 0 + 1 + \dots + k + (k + 1) &= (0 + 1 + \dots + k) + (k + 1) \\ &= \frac{k(k + 1)}{2} + (k + 1) \\ &= \frac{k(k + 1) + 2(k + 1)}{2} \\ &= \frac{(k + 1)(k + 1 + 1)}{2}, \end{aligned}$$

where we use the Inductive Hypothesis to justify the second equality.

Thus, by the Principle of Mathematical Induction, we conclude that $P(n)$ holds for every natural number n .

Sometimes, we only need to prove that a property holds for all natural numbers greater than some $m \geq 0$. Then the PMI works just as well, by starting with a Base step $P(m)$ and applying the Induction step only to natural numbers $k \geq m$.

Example 1.3.3

Prove that $n^2 \leq 2^n$ for every natural number $n \geq 4$.

Proof: Let $P(n)$ be the statement $n^2 \leq 2^n$.

Base step. Prove that $P(4)$ is true:

$$4^2 = 16 \leq 16 = 2^4$$

Induction step. Assume that $P(k)$ is true for some $k \geq 4$ (the Inductive Hypothesis (IH)), i.e. $k^2 \leq 2^k$.

Now we have to prove that $(k+1)^2 \leq 2^{(k+1)}$. We begin with:

$$(k+1)^2 = k^2 + 2k + 1 = k \left(k + 2 + \frac{1}{k} \right)$$

Our goal is to rewrite this equation so that we can use the IH. Note that $1/k < 1$ since $k \geq 4$. Therefore

$$k \left(k + 2 + \frac{1}{k} \right) \leq k(k+2+1) = k(k+3) \leq k(k+k) = 2k^2$$

We can now use the IH to get

$$2k^2 \leq 2 \cdot 2^k = 2^{(k+1)}$$

Therefore we have that $(k+1)^2 \leq 2^{(k+1)}$, as required.

By the PMI, we conclude that $n^2 \leq 2^n$ for all natural numbers $n \geq 4$.

Later, in the beginning of the chapter on number theory (Chapter 5), we will extend and generalize the Principle of Mathematical Induction.

1.3.1. Exercises

- ① Prove, by using the PMI, that $n^2 + n$ is even for all natural numbers n .
- ② Prove that $2^n < n!$ for all natural numbers $n \geq 4$.
- ③ Show that for every natural number $n \geq 1$, the power set of the set $\{1, 2, 3, \dots, n\}$ has 2^n elements.

Paradoxes

A *paradox* is a statement or an argument that is an apparent contradiction, either with well-known and accepted truths or simply with itself. Unlike a *fallacy*, a paradox is not due to an incorrect reasoning, but it could be based on a play of words, or on using self-referential statements, or on a very subtle ambiguity in the assumptions or concepts involved. Some set-theoretic paradoxes, however, such as *Russell's paradox* (see the box at the end of Section 2.1 in Chapter 2), lead to deeper foundational issues of mathematics.

Probably the best-known logical **Paradox of the Liar** is the person saying “I am lying now”, i.e. “This sentence is false”. If the sentence is true, then it truly claims that it is false and, if it is false, then it falsely states so; hence it must be true.

There are many variations of this paradox, one of them being **Jourdain's card paradox**. This is a card, on both sides of which there is a written statement. The statement on one side of the card says “The sentence on the other side of this card is *true*” and the one on the other side says “The sentence on the other side of this card is *false*”. Think of each of these as to whether it is true or false.

We will see more on logical fallacies and set theory paradoxes further. Here we offer an assortment of some simple mathematical fallacies and genuine paradoxes.

The heap paradox. How many grains of wheat, put together, form a heap? A hundred, a million ... ? Well, neither! Indeed, one grain is certainly not a heap and, if k grains do not form a heap, adding just one more will not make a heap either. Therefore, by induction, for every $n \in \mathbb{N}$, a collection of n grains is *not a heap*. What is wrong with this argument?

All horses have the same colour. Here is another proof by induction, of the claim that *all horses have the same colour*. Indeed, there are only finitely many horses, so it suffices to prove that for every $n \in \mathbb{N}$, in any set of n horses they all have the same colour. When $n = 1$ this is certainly true. Now, suppose this claim is true for n and let us take any set of $n + 1$ horses, say H_1, H_2, \dots, H_{n+1} . Then, by the inductive hypothesis, the first n of them, H_1, H_2, \dots, H_n , have the same colour, in particular H_1, H_2 have the same colour and the last n of them, H_2, \dots, H_{n+1} , have the same colour. Therefore, they all have the same colour. Do you agree? Why?

All natural numbers are interesting. Surely, there are many interesting natural numbers, for instance 0 is the first natural number, 1 is interesting for being the first positive natural number, 2 is the first prime number, etc. However, are there any natural numbers that are *not* interesting? Suppose there are. Then, there is a least number amongst them, say k . However, then k is the *least non interesting natural number*, which is certainly quite an interesting property, isn't it? Therefore, k is in fact interesting, which is a contradiction. Therefore, all natural numbers are interesting!

Berry paradox. Every natural number can be defined in English with sufficiently many words. However, if we bound the number of words that can be used, then only finitely many natural numbers can be defined, so there will be ones that cannot be defined with that many words. Therefore, there must be a *smallest* so undefinable natural number. Now, consider the following sentence: “The least natural number that is not definable in English with less than twenty words.” There is a uniquely determined natural number that satisfies this description, so it is a *definition* in English, correct? Well, count how many words it uses.

2

Sets, Relations, Orders

2.1.	Set inclusions and equalities	21
2.2.	Functions	28
2.3.	Binary relations and operations on them	35
2.4.	Special binary relations	45
2.5.	Equivalence relations and partitions	51
2.6.	Ordered sets	59
2.7.	An introduction to cardinality	69
2.8.	Isomorphisms of ordered sets. Ordinal numbers	75
2.9.	Application: relational databases	80

Set theory is one of the most fundamental parts of modern mathematics. Every mathematical object you are likely to encounter is built (or can be built) from sets using set theoretic operations. As we saw in Chapter 1, even the functions and relations themselves are sets, namely subsets of appropriate Cartesian powers of the domain, satisfying certain conditions. In fact, all mathematical objects that appear in this book – even natural numbers – can be based on sets. For instance, graphs studied in Chapter 7 are just sets together with certain binary relations.

The systematic theory of sets was initiated by the German mathematician Georg Cantor (see the box at the end of Section 2.2) in the second half of the 19th century. Today set theory is a fully developed, fundamental and very important branch of mathematics. Because sets are such basic mathematical objects, no precise and formal explicit mathematical definition of a set can be given, but only an intuitive idea (which everybody has to a certain extent). Of course we could try to define a set as a collection of objects, but then we would have to define what we mean by a collection, and that definition would again involve terms that need to

be defined, leading to an infinite regress. It is important to realize that in the foundations of mathematics we cannot formally define *every* term that we use without falling into circularity or an infinitely regressing chain of definitions.

2.1. Set inclusions and equalities

In Chapter 1 we gave a quick review of the basic notions relating to sets, including membership, containment and equality of sets, the empty set, set-builder notation, as well as the basic operations on sets including intersection, union, difference, complement, and products. We advise the reader who needs to refresh his or her knowledge of these concepts to turn back to Chapter 1.

The emphasis in this chapter, besides learning the basics of the theory of sets and relations, will be on writing correct and mathematically rigorous proofs – a very important and useful skill to master, for which we will develop more systematic methods in the chapters on logic, Chapters 3 and 4. Many of the properties of sets proved here might seem quite obvious, yet writing precise proofs for them might still be challenging.

Let us first take a closer look at the basic operations on sets and the fundamental identities that hold of these. The key fact that we will use to prove identities between sets is the following:

Proposition 2.1.1 *For all sets A and B , $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.*

Proof: The proof is a nice and simple exercise on precise mathematical reasoning, so we will do it in full detail.

We have to prove an equivalence: “... if and only if ...”. We do that by proving the implications in both directions.

1. To prove the implication “If $A = B$, then $A \subseteq B$ and $B \subseteq A$ ” we assume the premise “ $A = B$ ” and try to reach the conclusion “ $A \subseteq B$ and $B \subseteq A$ ”. That conclusion is a conjunction, so we have to prove both “ $A \subseteq B$ ” and “ $B \subseteq A$ ”. To prove that $A \subseteq B$, by definition we must show that every element of A is an element of B . Therefore, let x be any (arbitrary) element of A . Then, because $A = B$, we also have $x \in B$. Since $x \in A$ was arbitrary, we conclude that $A \subseteq B$.

Likewise, to show that $B \subseteq A$, let x be any element of B . Because $A = B$, it follows that $x \in A$. Therefore, since $x \in B$ was arbitrary, we conclude that $B \subseteq A$.

2. To prove “If $A \subseteq B$ and $B \subseteq A$, then $A = B$ ”, again we assume the premise “ $A \subseteq B$ and $B \subseteq A$ ” and try to deduce “ $A = B$ ”. By the definition of equality of sets, we have to show that A and B have exactly the same elements. Let x be an arbitrary element. If $x \in A$, then since $A \subseteq B$, we have $x \in B$. Similarly, if $x \in B$, then since $B \subseteq A$, we have $x \in A$. Thus we have shown that, for any x , it holds that $x \in A$ iff $x \in B$, i.e. that A and B have the same elements and thus $A = B$.

As a corollary, we note that there is only one empty set.¹

Another often used fact is that the empty set is a subset of every set.

¹ A more orderly minded reader may object here that the empty sent of oranges just cannot be the same as the empty set of ideas, or the empty set of sets. However, they are, essentially, by definition.

Proposition 2.1.2 *For every set A , it holds that $\emptyset \subseteq A$.*

Proof: By definition of inclusion we need to show that every element of \emptyset is an element of A . However, because \emptyset has no elements, there is nothing to do, so we are done.²

2.1.1. Properties of the set theoretic operations

Each of the following propositions lists a number of identities relating to a particular set theoretic operation. We only prove some of these identities and leave the proofs of the rest as exercises for the reader.

Proposition 2.1.3 (Properties of the intersection) *For any sets A, B, C and X it holds that:*

1. $A \cap \emptyset = \emptyset$
2. $A \cap A = A$
3. $A \cap B = B \cap A$
4. $A \cap (B \cap C) = (A \cap B) \cap C$
5. $A \subseteq B$ if and only if $A \cap B = A$
6. $A \cap B \subseteq A$ and $A \cap B \subseteq B$
7. If $X \subseteq A$ and $X \subseteq B$, then $X \subseteq A \cap B$

Proof: We will prove properties 1 and 7.

1. To prove $A \cap \emptyset = \emptyset$ we proceed by contradiction: suppose $A \cap \emptyset$ is nonempty and take any element of x of $A \cap \emptyset$. Then, in particular, $x \in \emptyset$, which is impossible. Thus, $A \cap \emptyset$ is the empty set.
7. Suppose $X \subseteq A$ and $X \subseteq B$; we need to show that $X \subseteq A \cap B$. To that end, suppose that $x \in X$. Since $X \subseteq A$ we have $x \in A$, and since $X \subseteq B$ we also have $x \in B$. Thus, by the definition of intersection, $x \in A \cap B$.

Proposition 2.1.4 (Properties of the union) *For any sets A, B, C and X it holds that:*

1. $A \cup \emptyset = A$
2. $A \cup A = A$
3. $A \cup B = B \cup A$
4. $A \cup (B \cup C) = (A \cup B) \cup C$
5. If $A \subseteq B$ if and only if $A \cup B = B$
6. $A \subseteq A \cup B$ and $B \subseteq A \cup B$
7. If $A \subseteq X$ and $B \subseteq X$, then $A \cup B \subseteq X$

Proof: We will prove property 1. To prove that $A \cup \emptyset \subseteq A$, suppose that $x \in A \cup \emptyset$. Hence $x \in A$ or $x \in \emptyset$. However, since it cannot be the case that $x \in \emptyset$, it must be the case that $x \in A$.

² Another way of stating this is that, because \emptyset has no elements, it holds *vacuously* that every element of \emptyset is an element of A .

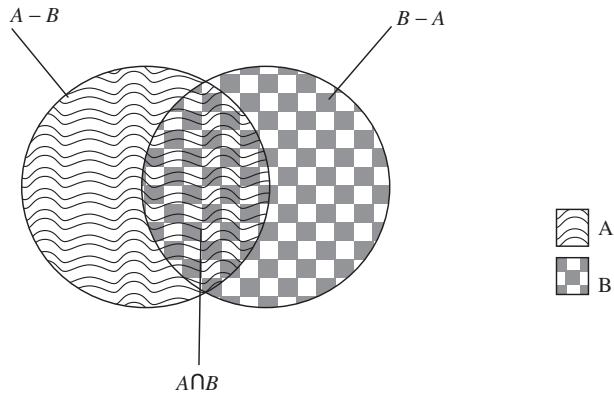


Figure 2.1: Properties of set theoretic difference

For the inclusion $A \subseteq A \cup \emptyset$, suppose that $x \in A$. Then $x \in A$ or $x \in \emptyset$, so $x \in A \cup \emptyset$.

The fourth clauses of Propositions 2.1.3 and 2.1.4 above tell us that intersection and union of sets are **associative** operations. Intuitively this means that no matter in which order we take the union (respectively, intersection) of three or more sets, the result will be the same, so we can write simply $A \cap B \cap C$ for $A \cap (B \cap C)$ or $(A \cap B) \cap C$, and also $A \cup B \cup C$ for $A \cup (B \cup C)$ or $(A \cup B) \cup C$. This also suggests the following generalizations of the intersection and union operations. Suppose that \mathcal{X} is a set of sets: we can think of $\mathcal{X} \subseteq \mathcal{P}(U)$ for some universal set U . Then we write $\bigcap \mathcal{X}$ for the intersection of *all* members of \mathcal{X} , i.e. $x \in \bigcap \mathcal{X}$ iff $x \in X$ for every set $X \in \mathcal{X}$. Similarly, the union $\bigcup \mathcal{X}$ consists of all those elements belonging to at least one $X \in \mathcal{X}$.

Example 2.1.5

Consider the set $\mathcal{X} = \{[a, \infty) \mid [a, \infty) \subseteq \mathbb{R} \text{ and } a < 0\}$ consisting of all intervals $[a, \infty)$ of the real line where a is negative. Then we claim $\bigcap \mathcal{X}$ is equal to the set $[0, \infty)$ of all non-negative real numbers. Indeed, $[0, \infty) \subseteq [a, \infty)$ for every negative real number a , so we have $[0, \infty) \subseteq \bigcap \mathcal{X}$ (by an extension of Proposition 2.1.3(7)). Conversely, if $x \notin [0, \infty)$, then $x < 0$. Then $x < x/2 < 0$, so $x \notin [x/2, \infty)$. Now since $[x/2, \infty) \in \mathcal{X}$, we conclude that $x \notin \bigcap \mathcal{X}$.

Similarly, it is easy to see that $\bigcup \mathcal{X} = \mathbb{R}$.

The difference of two sets is visualized by means of the Venn diagram in Figure 2.1 and the following proposition lists some properties of the set theoretic difference.

Proposition 2.1.6 (Properties of set theoretic difference) *For any sets A, B and C :*

- | | |
|--|--|
| 1. $A - A = \emptyset$
2. $\emptyset - A = \emptyset$
3. $A - \emptyset = A$ | 4. $A - B \subseteq A$
5. $(A - B) \cap B = \emptyset$
6. $A - B = \emptyset$ if and only if $A \subseteq B$ |
|--|--|

- | | |
|---------------------------------------|---|
| 7. $(A - B) \cap (B - A) = \emptyset$ | 10. $A - (B \cap C) = (A - B) \cup (A - C)$ |
| 8. $(A - B) \cup B = A \cup B$ | 11. $A - (B \cup C) = (A - B) \cap (A - C)$ |
| 9. $A - (A \cap B) = A - B$ | 12. $A \cup B = (A - B) \cup (B - A) \cup (A \cap B)$ |

Proof: We prove properties 1, 5 and 11.

1. Suppose that $x \in A - A$. That means $x \in A$ and $x \notin A$, which is impossible. There can therefore be no element $x \in A - A$, i.e. $A - A = \emptyset$.
5. Suppose $x \in (A - B) \cap B$. This means that $x \in (A - B)$ and $x \in B$. However, $x \in (A - B)$ means that $x \in A$ and $x \notin B$, so it cannot be the case that $x \in B$, which is a contradiction. We conclude that $(A - B) \cap B$ has no elements, i.e. $(A - B) \cap B = \emptyset$.
11. We will prove both inclusions. To prove that $A - (B \cup C) \subseteq (A - B) \cap (A - C)$, suppose that $x \in A - (B \cup C)$. Then $x \in A$ and $x \notin (B \cup C)$, i.e. $x \in A$ but $x \notin B$ and $x \notin C$. Then $x \in A - B$ and $x \in A - C$, i.e. $x \in (A - B) \cap (A - C)$.
For the inclusion $(A - B) \cap (A - C) \subseteq A - (B \cup C)$, suppose that $x \in (A - B) \cap (A - C)$. This means that $x \in (A - B)$ and $x \in (A - C)$, i.e. $x \in A$ but $x \notin B$ and $x \notin C$. Since $x \notin B$ and $x \notin C$ we have $x \notin (B \cup C)$. Combining these with $x \in A$, we have $x \in A - (B \cup C)$.

Proposition 2.1.7 (Interaction between union and intersection) *For all sets A, B and C the following hold:*

1. $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ (*distributivity of \cap over \cup*)
2. $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ (*distributivity of \cup over \cap*)
3. $A \cap (A \cup B) = A$ (*absorption*)
4. $A \cup (A \cap B) = A$ (*absorption*)

Proof: Properties 1 and 2 are illustrated in Figure 2.2 by means of Venn diagrams. We will prove only property 1 and leave the proofs of the others as exercises. The proof goes through a chain of “if and only ifs”:

$$\begin{aligned}
 &x \in A \cap (B \cup C) \\
 \text{iff } &x \in A \text{ and } x \in (B \cup C) \\
 \text{iff } &x \in A, \text{ and } x \in B \text{ or } x \in C \\
 \text{iff } &x \in A \text{ and } x \in B, \text{ or } x \in A \text{ and } x \in C \\
 \text{iff } &x \in A \cap B \text{ or } x \in A \cap C \\
 \text{iff } &x \in (A \cap B) \cup (A \cap C)
 \end{aligned}$$

Next we will consider the properties of powersets. Figure 2.3 shows a three-element set and its powerset.

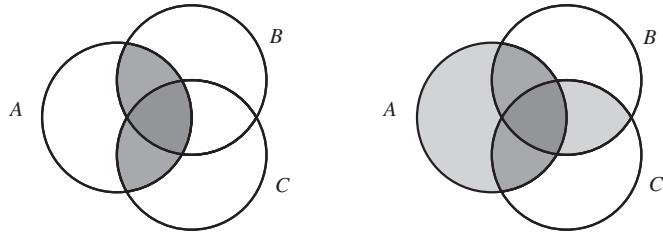


Figure 2.2: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ and $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

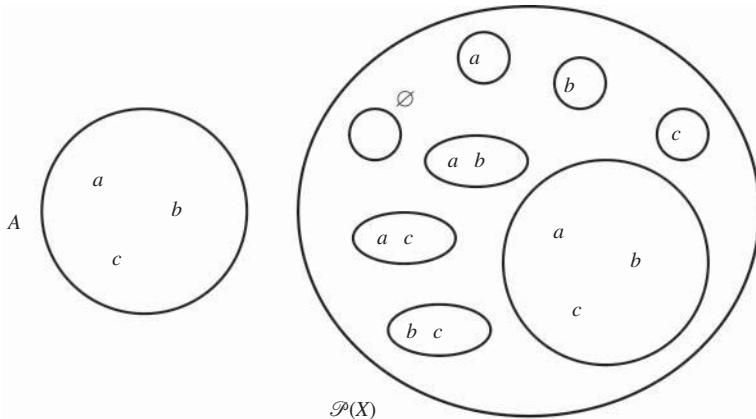


Figure 2.3: A three-element set and its powerset

Proposition 2.1.8 (Properties of the powerset) For any sets A , B and X :

1. $A \in \mathcal{P}(B)$ iff $A \subseteq B$.
2. $\emptyset \in \mathcal{P}(A)$.
3. $A \in \mathcal{P}(A)$.
4. If $A \subseteq B$ then $\mathcal{P}(A) \subseteq \mathcal{P}(B)$.
5. If $A \in \mathcal{P}(X)$ then $X - A \in \mathcal{P}(X)$.
6. If $A \in \mathcal{P}(X)$ and $B \in \mathcal{P}(X)$ then $A \cap B$, $A \cup B$ and $A - B$ are in $\mathcal{P}(X)$.
7. If X has n elements then $\mathcal{P}(X)$ has 2^n elements.

Proof: Properties 1, 2 and 3 are direct consequences of the definition of the powerset (see Section 1.1 in Chapter 1). We leave the proofs of items 4, 5 and 6 as exercises and only prove 7. In order to construct a subset A of X , for each element of X we must decide whether it will be in A or not, and these choices completely determine A . For each element of X there are therefore two choices, “in” or “out”. Thus the number of ways to form the subset A is $\underbrace{2 \times 2 \times \cdots \times 2}_{n \text{ times}} = 2^n$.

(Counting techniques like this are extensively studied in Chapter 6; see especially Section 6.1.)

Proposition 2.1.8 implies that if U is a universal set, then \cup , \cap and $'$ are operations on $\mathcal{P}(U)$. (Recall that we write A' for $U - A$.) Thus, we obtain an algebraic structure $\langle \mathcal{P}(U); \cup, \cap, ', \emptyset, U \rangle$, called the **powerset Boolean algebra of U** .

Proposition 2.1.9 (Properties of the Cartesian product) *For any sets A, B and C :*

1. $A \times (B \cup C) = (A \times B) \cup (A \times C)$, $(B \cup C) \times A = (B \times A) \cup (C \times A)$
2. $A \times (B \cap C) = (A \times B) \cap (A \times C)$, $(B \cap C) \times A = (B \times A) \cap (C \times A)$
3. $A \times (B - C) = (A \times B) - (A \times C)$

Proof: We prove the identity 1. For the inclusion $A \times (B \cup C) \subseteq (A \times B) \cup (A \times C)$, suppose that $x \in A \times (B \cup C)$. Then $x = (y, z)$ for some $y \in A$ and $z \in (B \cup C)$. Since $z \in (B \cup C)$, we have $z \in B$ or $z \in C$. Therefore $(y, z) \in A \times B$ or $(y, z) \in A \times C$, and hence $x = (y, z) \in (A \times B) \cup (A \times C)$.

For the inclusion $(A \times B) \cup (A \times C) \subseteq A \times (B \cup C)$ suppose that $x = (y, z) \in (A \times B) \cup (A \times C)$. Then $(y, z) \in (A \times B)$ or $(y, z) \in (A \times C)$. Thus $y \in A$, while $z \in B$ or $z \in C$. Thus $y \in A$, while $z \in B \cup C$, and hence $x = (y, z) \in A \times (B \cup C)$.

2.1.2. Exercises

❶ Prove that for all sets A, B, C and X it holds that:

- | | |
|--|---|
| (a) $A \cap A = A$
(b) $A \cap B = B \cap A$
(c) $A \cap (B \cap C) = (A \cap B) \cap C$
(d) $A \cap B \subseteq A$ and $A \cap B \subseteq B$
(e) $A \subseteq B$ if and only if $A \cap B = A$
(f) $A \cup A = A$ | (g) $A \cup B = B \cup A$
(h) $A \cup (B \cup C) = (A \cup B) \cup C$
(i) $A \subseteq A \cup B$ and $B \subseteq A \cup B$
(j) If $A \subseteq B$ if and only if $A \cup B = B$
(k) If $A \subseteq X$ and $B \subseteq X$, then $A \cup B \subseteq X$ |
|--|---|

❷ Prove that for all sets A, B and C it holds that:

- (a) $\emptyset - A = \emptyset$
- (b) $A - \emptyset = A$
- (c) $A - B \subseteq A$
- (d) $A - B = \emptyset$ if and only if $A \subseteq B$
- (e) $(A - B) \cap (B - A) = \emptyset$
- (f) $(A - B) \cup B = A \cup B$
- (g) $A - (A \cap B) = A - B$
- (h) $A - (B \cap C) = (A - B) \cup (A - C)$
- (i) $A \cup B = (A - B) \cup (B - A) \cup (A \cap B)$
- (j) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
- (k) $A \cap (A \cup B) = A$
- (l) $A \cup (A \cap B) = A$

③ Prove that for all sets A, B and X it holds that:

- (a) If $A \subseteq B$ then $\mathcal{P}(A) \subseteq \mathcal{P}(B)$.
- (b) If $A \in \mathcal{P}(X)$ and $B \in \mathcal{P}(X)$ then $A \cap B \in \mathcal{P}(X)$, $A \cup B \in \mathcal{P}(X)$, $A - B \in \mathcal{P}(X)$, $X - A \in \mathcal{P}(X)$.

④ Prove that for all sets A, B , and C it holds that:

- (a) $(B \cup C) \times A = (B \times A) \cup (C \times A)$
- (b) $A \times (B \cap C) = (A \times B) \cap (A \times C)$
- (c) $(B \cap C) \times A = (B \times A) \cap (C \times A)$
- (d) $A \times (B - C) = (A \times B) - (A \times C)$.

Russell's paradox

Elements of a set can be sets themselves, and their elements can be sets, etc. In the early times of set theory, at the end of the 19th century, there were no restrictions as to what conceivable collections could be regarded as sets. That, however, soon led to constructions of sets, the existence of which brought contradictions. A famous example is the set defined by the prominent British philosopher Bertrand Russell (see the box at the end of Section 4.3 in Chapter 4) in the beginning of the 20th century. Russell considered the set \mathcal{R} , defined as follows:

\mathcal{R} is the set consisting of exactly those sets which are not elements of themselves.

Even though it might look strange, this set seems legitimate until one goes through the following argument.

Since \mathcal{R} is a set, it either belongs to itself or not.

If it belongs to itself, then, by definition, it is not one of its elements, i.e. it *does not* belong to itself.

But, if it does not belong to itself, then, again by definition, it *must* belong to itself.

Thus, \mathcal{R} belongs to itself if and only if it does not belong to itself.

This contradiction is known as *Russell's paradox*. It arises from the very assumption that the set \mathcal{R} exists or, rather, that the conceivable collection \mathcal{R} can be regarded as a set. Russell's and similar paradoxes discovered around the beginning of the 20th century dealt a serious blow to the then young and naive theory of sets and caused a crisis in the very foundations of mathematics and logic. That crisis brought about the development of new philosophies of mathematics and initiated new approaches to its foundations, the most popular of which became the *axiomatic method*, promoted by the famous German mathematician David Hilbert (1862–1943) and his followers. In particular, an *axiomatic set theory* (see the box at the end of Section 2.5) was systematically developed during the 20th century, the purpose of which was, among others, to avoid paradoxes by strictly prescribing all permitted constructions of sets.

2.2. Functions

In Chapter 1 we mentioned that functions can be seen equivalently, either as mappings that assign values to elements of a set or as special types of relations. In this section it will be convenient for us to take the first view. We will therefore use the terms function and mapping interchangeably.

2.2.1. Functions and their inverses

Definition 2.2.1 A *function* (or *mapping*, or *map*) from a set A to a set B , denoted by

$$f : A \rightarrow B,$$

assigns to each element $a \in A$ a unique element $f(a) \in B$. The element $f(a)$ is called the *value of a under f* , or also the *image of a under f* . If $f(a) = b$ then a is called a *pre-image of b under f* .

The set A is called the *domain of f* , denoted $\text{dom}(f)$, and B is called the *co-domain*, or the *target set*, of f , denoted $\text{cod}(f)$.

The notion of image can be generalized from elements to subsets of the domain as follows. For any subset $X \subseteq A$ of the domain of f , the *image of X under f* is the set

$$f[X] = \{f(a) \mid a \in X\}.$$

The *image of the domain A under f* , i.e. the set of all values of f ,

$$f[A] = \{f(a) \mid a \in A\},$$

is called the *range*, or *image*, of f , also denoted by $\text{rng}(f)$.

Example 2.2.2

Figure 2.4 depicts a mapping $f : A \rightarrow B$ where $A = \{a, b, c, d\}$, $B = \{2, 3, 4, 5, 6\}$ and $f(a) = 2$, $f(b) = 2$, $f(c) = 5$, $f(d) = 4$ and $f(e) = 6$. In this example $\text{dom}(f) = A$, $\text{cod}(f) = B$ and $\text{rng}(f) = \{2, 4, 5, 6\}$.

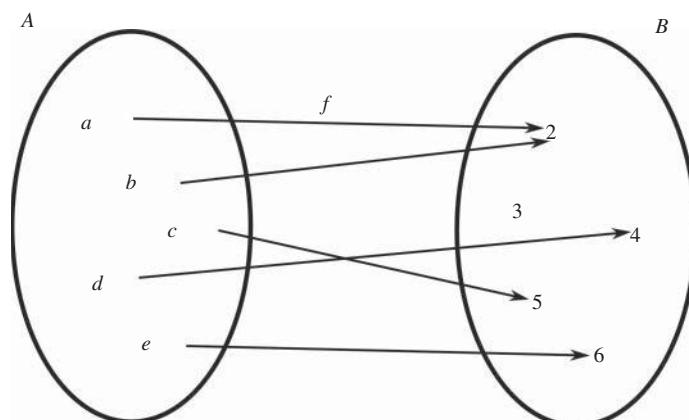


Figure 2.4: A mapping $f : A \rightarrow B$ (see Example 2.2.2)

Example 2.2.3

Let $\text{sq} : \mathbb{R} \rightarrow \mathbb{R}$ be the function that takes a real number and squares it. We use the notation

$$\text{sq} : x \mapsto x^2$$

to indicate that this is how sq , seen as a rule, works. Then $\text{dom}(\text{sq}) = \mathbb{R}$, $\text{cod}(\text{sq}) = \mathbb{R}$ and $\text{rng}(\text{sq}) = \{y \in \mathbb{R} \mid 0 \leq y\}$.

The addition function acts on pairs of real numbers, so we can see it as a function $\text{add} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ given by the rule $\text{add} : (x_1, x_2) \mapsto x_1 + x_2$. We have $\text{dom}(\text{add}) = \mathbb{R} \times \mathbb{R}$ and $\text{cod}(\text{add}) = \text{rng}(\text{add}) = \mathbb{R}$.

In the example above we could have chosen the non-negative reals as the co-domain of the squaring function, which could have made it a function $\text{sq}' : \mathbb{R} \rightarrow \mathbb{R}_0^+$, where $\mathbb{R}_0^+ = \{x \in \mathbb{R} \mid 0 \leq x\}$. However, we usually identify (unless otherwise specified) two functions f and f' that have the same domain A , and for which $f(a) = f'(a)$ for every $a \in A$, even though their co-domains may be different.

Definition 2.2.4 (Equality of functions) *Two functions f and g are equal if*

1. $\text{dom}(f) = \text{dom}(g)$,
2. $\text{cod}(f) = \text{cod}(g)$ and
3. $f(a) = g(a)$ for every $a \in \text{dom}(f)$.

Example 2.2.5

Let $f : \mathbb{N} \rightarrow \mathbb{Z}$ and $g : \mathbb{N} \rightarrow \mathbb{Z}$ be functions such that

$$f : x \mapsto x^2 - 4$$

and

$$g : x \mapsto (x - 2)(x + 2).$$

Now we have $\text{dom}(f) = \mathbb{N} = \text{dom}(g)$ and $\text{cod}(f) = \mathbb{Z} = \text{cod}(g)$. Also, for every $a \in \mathbb{N}$ we have $f(a) = a^2 - 4 = (a - 2)(a + 2) = g(a)$. Therefore, the conditions of Definition 2.2.4 are satisfied, and we conclude that $f = g$.

As discussed in Chapter 1 we can also regard a function as a set of pairs, i.e. a special type of relation. The following definition makes this more precise:

Definition 2.2.6 (Graph of a function) *The graph of a function $f : A \rightarrow B$ is the set of pairs*

$$\{(a, f(a)) \mid a \in A\}.$$

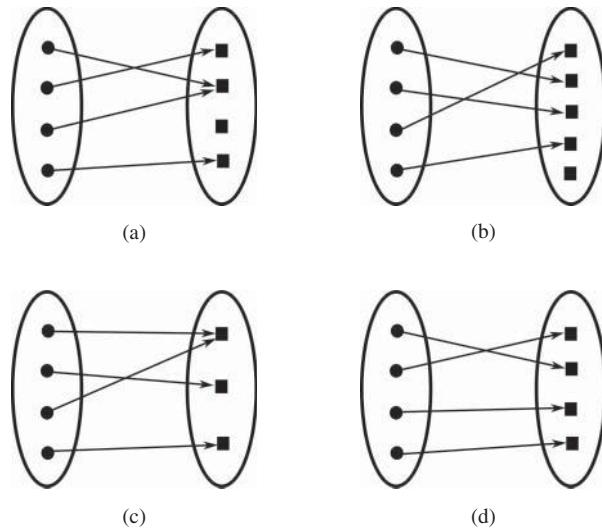


Figure 2.5: Four functions that are: (a) neither injective nor surjective; (b) injective but not surjective; (c) surjective but not injective; (d) bijective.

Definition 2.2.7 Let $X \subseteq A$ and $f : A \rightarrow B$ be a mapping. The **restriction** of f to X is the mapping $f|_X : X \rightarrow B$ defined by

$$f|_X(a) = f(a) \text{ for each } a \in X.$$

If g is a restriction of f , then f is called an **extension** of g .

Definition 2.2.8 A mapping $f : A \rightarrow B$ is (see Figure 2.5):

- **injective, or into**, if for every $a_1, a_2 \in A$,

$$f(a_1) = f(a_2) \text{ implies } a_1 = a_2;$$

- **surjective, or onto**, if

$$\text{rng}(f) = B.$$

- **bijective, or one-to-one**, if it is both injective and surjective.

Example 2.2.9

Neither the squaring function sq nor the addition function add from Example 2.2.3 is injective. For example, $\text{sq}(2) = 4 = \text{sq}(-2)$ and $\text{add}(3, 4) = 3 + 4 = 7 = 2 + 5 = \text{add}(2, 5)$. However, the restriction $\text{sq}|_{\mathbb{R}_0^+}$ of the squaring function to the non-negative real numbers is injective. The addition function add is surjective. The cubing function $\text{cb} : \mathbb{R} \rightarrow \mathbb{R}$ given by $\text{cb} : x \mapsto x^3$ is bijective.

Definition 2.2.10 Let $f : A \rightarrow B$ be injective. The *inverse* (mapping) off is the mapping

$$f^{-1} : \text{rng}(f) \rightarrow A,$$

defined by

$$f^{-1}(f(a)) = a \text{ for every } a \in A.$$

Proposition 2.2.11 For every bijective mapping f :

1. f^{-1} is a bijection.
2. $(f^{-1})^{-1} = f$.

Proof: We will prove property 1 and leave property 2 as an exercise. Since f is surjective, we have $\text{rng}(f) = B = \text{dom}(f^{-1})$. To prove that f^{-1} is injective, suppose that $b_1, b_2 \in B$ and that $f^{-1}(b_1) = f^{-1}(b_2)$. Since $\text{rng}(f) = B$, there are $a_1, a_2 \in A$ such that $f(a_1) = b_1$ and $f(a_2) = b_2$. Thus $a_1 = f^{-1}(f(a_1)) = f^{-1}(b_1) = f^{-1}(b_2) = f^{-1}(f(a_2)) = a_2$. Therefore, since we now have $a_1 = a_2$, the injectivity of f allows us to conclude that $f(a_1) = f(a_2)$, and thus that $b_1 = b_2$, as desired.

To show that f^{-1} is surjective, suppose that $a \in A$. We must show that there is a $b \in B$ such that $f^{-1}(b) = a$. Take $f(a) \in B$: we then have $f^{-1}(f(a)) = a$.

Hereafter we restrict the notion of inverse mappings to bijections only.

Example 2.2.12

The squaring function and addition functions, sq and add , from Example 2.2.3 are not injective, so neither has an inverse. If we were to restrict the domain and range of sq to the non-negative real numbers to obtain a function $\text{sq}'' : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ such that $\text{sq}'' : x \mapsto x^2$, this function would be bijective, and its inverse would be given by $(\text{sq}'')^{-1}x : \mapsto \sqrt{x}$.

2.2.2. Composition of mappings

Definition 2.2.13 If $f : A \rightarrow B$ and $g : B \rightarrow C$ then the *composition* off and g is the mapping

$$gf : A \rightarrow C$$

defined by

$$gf(a) = g(f(a)) \text{ for each } a \in A.$$

Sometimes we also denote the composition gf by fog . The reason for swapping the positions of f and g in the alternative notation will become clear later.

Whenever we refer to a composition gf of two mappings f and g , we will assume that $\text{rng}(f) \subseteq \text{dom}(g)$.

Example 2.2.14

$\text{rng}(\text{add}) = \mathbb{R} \subseteq \text{dom}(\text{sq}) = \mathbb{R}$, so we can form the composition sq add , which would give $\text{sq add}(x, y) = \text{sq}(\text{add}(x, y)) = (x + y)^2$. Notice that $\text{rng}(\text{sq}) \not\subseteq \text{dom}(\text{add}) = \mathbb{R} \times \mathbb{R}$ so the composition add sq cannot be formed.

Proposition 2.2.15 Let $f : A \rightarrow B, g : B \rightarrow C$ be any mappings (see Figure 2.6). Then:

1. If f and g are injective then gf is also injective.
2. If f and g are surjective then gf is also surjective.
3. In particular, the composition of bijections is a bijection.
4. If gf is injective then f is injective.
5. If gf is surjective then g is surjective.

Proof:

1. Suppose f and g are injective and that $a_1, a_2 \in A$ such that $gf(a_1) = gf(a_2)$. We must show that $a_1 = a_2$. Thus $g(f(a_1)) = g(f(a_2))$. However, since g is injective this means that $f(a_1) = f(a_2)$ which in turn means that $a_1 = a_2$ since f is injective.
2. Exercise.
3. Follows immediately from 1 and 2.
4. Exercise.
5. Suppose that gf is surjective and that $c \in C$. We need to find a $b \in B$ such that $gf(b) = c$. By the surjectivity of gf there is an $a \in A$ such that $gf(a) = c$, i.e. $g(f(a)) = c$. Then, $f(a) \in B$ is the desired element of B .

Proposition 2.2.16 Composition of mappings is associative, i.e. $f(gh) = (fg)h$, whenever either of these is defined.

Proof: Exercise.

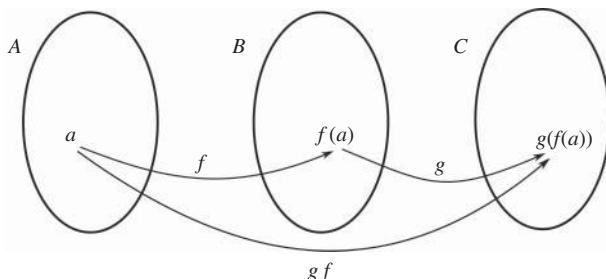


Figure 2.6: The composition of two mappings $f : A \rightarrow B$ and $g : B \rightarrow C$

Proposition 2.2.17 If $f : A \rightarrow B$, $g : B \rightarrow C$ are bijective mappings then gf has an inverse, and particularly

$$(gf)^{-1} = f^{-1}g^{-1}.$$

Proof: By Proposition 2.2.15(3) gf is bijective and therefore we know that it has an inverse $(gf)^{-1}$. Clearly $\text{dom}(f^{-1}g^{-1}) = C = \text{dom}((gf)^{-1})$ and $\text{rng}(f^{-1}g^{-1}) = A = \text{rng}((gf)^{-1})$. We need to show that $f^{-1}g^{-1}(c) = (gf)^{-1}(c)$ for all $c \in C$. To this end let $c \in C$ arbitrarily. Then, because gf is surjective, there is an $a \in A$ such that $gf(a) = c$. By definition of inverses $(gf)^{-1}(c) = (gf)^{-1}(gf(a)) = a$. Now $f^{-1}g^{-1}(c) = f^{-1}g^{-1}(gf(a)) = f^{-1}g^{-1}(g(f(a))) = f^{-1}(g^{-1}(g(f(a))))$. By the associativity of function composition (Proposition 2.2.16) the last expression is equal to $f^{-1}((g^{-1}g)(f(a))) = f^{-1}(f(a)) = a$.

Proposition 2.2.18 A mapping f is injective iff it has the following property: for every two mappings g_1 and g_2 with $\text{dom}(g_1) = \text{dom}(g_2)$ and $\text{cod}(g_1) = \text{dom}(f) = \text{cod}(g_2)$, the following left cancellation property holds:

$$\text{if } fg_1 = fg_2 \text{ then } g_1 = g_2.$$

Proof: For the sake of definiteness, suppose that $f : A \rightarrow B$. To prove the left-to-right implication suppose that f is injective and take any mappings $g_1 : C \rightarrow A$ and $g_2 : C \rightarrow A$. Further suppose that $g_1f = g_2f$. We must show that $g_1 = g_2$. Therefore, let $c \in C$ arbitrarily. Then $fg_1(c) = fg_2(c)$, i.e. $f(g_1(c)) = f(g_2(c))$. Since f is injective, it follows that $g_1(c) = g_2(c)$.

To prove the converse we will proceed by contraposition, so suppose that f is not injective. Then there are $a_1, a_2 \in A$ such that $a_1 \neq a_2$ but $f(a_1) = f(a_2)$. We need to show that the left cancellation rule does not hold. For this it is sufficient to find two mappings g_1 and g_2 such that $fg_1 = fg_2$ but $g_1 \neq g_2$. Let $g_1 : A \rightarrow A$ and $g_2 : A \rightarrow A$ be maps such that g_1 is the identity map on A , i.e. $g_1(a) = a$ for all $a \in A$ and $g_2(a) = a$ for all $a \in A - \{a_1, a_2\}$ but $g(a_1) = a_2$ and $g(a_2) = a_1$. Then it is easy to check that $fg_1 = fg_2$, but $g_1 \neq g_2$.

Proposition 2.2.19 A mapping f is surjective iff it has the following property: for every two mappings g_1 and g_2 with $\text{dom}(g_1) = \text{dom}(g_2) = \text{cod}(f)$, the following right cancellation property holds:

$$\text{if } g_1f = g_2f \text{ then } g_1 = g_2.$$

Proof: Exercise.

2.2.3. Exercises

- ① Let $f : \mathbb{Z} \rightarrow \mathbb{R}$, $g : \mathbb{R} \rightarrow \mathbb{R}$ and $h : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ be functions such that $f : x \mapsto x/3$, $g : x \mapsto x^3$ and $h : (x, y) \mapsto x \times y$.
 - (a) Describe gf . Specifically, determine $\text{dom}(gf)$ and $\text{rng}(gf)$.
 - (b) Describe gh . Specifically, determine $\text{dom}(gh)$ and $\text{rng}(gh)$.
 - (c) Is $gf = fg$?

- (d) Can one form the composition hf ?
- (e) Can one form the composition fg ?
- ② Find examples of well-known functions $f_1 : \mathbb{R} \rightarrow \mathbb{R}$, $f_2 : \mathbb{R} \rightarrow \mathbb{R}$, $f_3 : \mathbb{R} \rightarrow \mathbb{R}$ and $f_4 : \mathbb{R} \rightarrow \mathbb{R}$ such that
- f_1 is neither surjective nor injective.
 - f_2 is surjective but not injective.
 - f_3 is injective but not surjective.
 - f_4 is bijective.
- ③ Give an example of two functions $f : A \rightarrow B$ and $g : B \rightarrow C$ such that gf is injective but g is not injective.
- ④ Give an example of two functions $f : A \rightarrow B$ and $g : B \rightarrow C$ such that gf is surjective but f is not surjective.
- ⑤ Let $f : \mathbb{R} \rightarrow \mathbb{R}$, $g : \mathbb{R}^+ \rightarrow \mathbb{R}$ and $h : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ be functions such that $f : x \mapsto x + 2$, $g : x \mapsto \sqrt{x}$ and $h : (x, y) \mapsto x/y$.
- What is $\text{cod}(h)$?
 - Determine $\text{rng}(f)$, $\text{rng}(g)$ and $\text{rng}(h)$.
 - Determine which of f , g and h are injective.
 - Determine which of f , g and h are surjective.
 - Determine which of f , g and h are bijective.
- ⑥ A fixed point of a function $f : A \rightarrow A$ is an element $a \in A$ such that $f(a) = a$.
- Which are the fixed points of the squaring function sq ?
 - Give an example of a function $f : \mathbb{Z} \rightarrow \mathbb{Z}$ that has no fixed points.
 - Let $\text{abs} : \mathbb{R} \rightarrow \mathbb{R}$ be the function given by $\text{abs} : x \mapsto |x|$. How many fixed points does abs have?
- ⑦* Prove that for every bijective mapping f , it holds that $(f^{-1})^{-1} = f$.
- ⑧* Let $f : A \rightarrow B$, $g : B \rightarrow C$ be any mappings. Show that
- If f and g are surjective then gf is also surjective.
 - If gf is injective then f is injective.
- ⑨* Let $h : A \rightarrow B$, $g : B \rightarrow C$ and $f : C \rightarrow D$ be any mappings. Show that the $f(gh) = (fg)h$, i.e. show that the composition of mappings is associative.
- ⑩** Prove that a mapping f is surjective iff it has the following property: for every two mappings g_1 and g_2 with $\text{dom}(g_1) = \text{dom}(g_2) = \text{cod}(f)$, the following *right cancellation property* holds: if $g_1f = g_2f$ then $g_1 = g_2$.



Georg Cantor (1845–1918)

Georg Cantor is the founding father of set theory or, more specifically, the theory of infinite sets. The notion of infinity is a tricky topic and very easily leads to paradoxes like those of Zeno of Elea (c. 490–c. 430 BC). Because of these dangers, mathematicians up to the time of Cantor had preferred to work with “potential” infinities (e.g., for every natural number there is a greater one) rather than actual “completed” infinities (e.g. the set of all natural numbers). In 1831 Gauss said: “I protest against the use of on infinite magnitude as something completed, which is never permissible in mathematics. Infinity is merely a way of speaking, the true meaning being a limit [· · ·].” (Of course, here Gauss was probably thinking not of an infinite set, but rather of the use of infinity as a value in a calculation. Still, the quotation captures the general suspicion in which infinity was viewed.)

Cantor was the first to embrace actual infinite sets, perform operations on them and compare them: something we do as a matter of routine in mathematics today.

Cantor was born in St Petersburg in Russia in 1845, the son of a prosperous merchant father and an artistic mother. His father wanted him to become an engineer, but after it became clear that he was temperamentally unsuited for this profession, he was allowed to study mathematics, first at the University of Zurich and then the University of Berlin. In Berlin he was taught by some of the leading mathematicians of the age, including Kummer, Weierstrass and Kronecker. Cantor’s first paper on infinite sets was published shortly before his thirtieth birthday. The mathematical establishment of the day did not welcome his innovations: many mathematicians were openly hostile towards it, especially Kronecker. However, as time went by, the immense value of his work was increasingly recognized, so much so that in the early 1890s he was elected president of the German Mathematical Society. He taught at the University of Halle in Germany from 1869 until his retirement in 1913.

2.3. *Binary relations and operations on them*

We already introduced the abstract notion of a relation in Chapter 1. Binary (or two-place) relations are the most common relations in the mathematical practice, so in this section we will be exploring them in some more detail.

2.3.1. *Binary relations*

Definition 2.3.1 Given sets A and B , a **binary relation between A and B** is any subset of $A \times B$. In particular, a **binary relation on a set A** is any subset of $A^2 = A \times A$.

Let $R \subseteq A \times B$ be a binary relation between sets A and B . If $(a, b) \in R$, we will also write aRb and say that a is R -related to b .

Definition 2.3.2 Given sets A and B and a binary relation $R \subseteq A \times B$:

1. The **domain of R** is the set

$$\text{dom}(R) = \{a \in A \mid \exists b \in B(aRb)\}.$$

2. The **range of R** is the set

$$\text{rng}(R) = \{b \in B \mid \exists a \in A(aRb)\}.$$

More generally, given subsets $X \subseteq A$ and $Y \subseteq B$, we define:

3. The **direct image of X under R** is the set

$$R[X] = \{b \in B \mid \exists x \in X(xRb)\}.$$

4. The **inverse image of Y under R** is the set

$$R^{-1}[Y] = \{a \in A \mid \exists y \in Y(aRy)\}.$$

Notice that we have $\text{dom}(R) = R^{-1}[B]$ and $\text{rng}(R) = R[A]$.

Example 2.3.3

On the set of \mathbb{Z} integers the “less than” relation $<$ is the set $\{(x, y) \mid x, y \in \mathbb{Z} \text{ and } x < y\}$. Its domain and range are both equal to \mathbb{Z} . The “less than or equal to” relation \leq is the set $\{(x, y) \mid x, y \in \mathbb{Z} \text{ and } x \leq y\}$ and also has both domain and range equal to \mathbb{Z} .

The relation “properly divides” on the positive natural numbers \mathbb{N}^+ , which we will denote by div , is given by $\{(x, y) \mid x, y \in \mathbb{N}^+ \text{ and } (\exists z \in \mathbb{N})(1 < z < y \wedge x \times z = y)\}$. The domain of this relation is $\{2, 3, 4, 5, \dots\}$ while its range is the set of all composite numbers. The direct image $\text{div}[\{2\}]$ of $\{2\}$ under div is the set $\{4, 6, 8, 10, \dots\}$ consisting of multiples of 2 greater than or equal to 4, i.e. the set of positive natural numbers that can be properly divided by 2. Also $\text{div}^{-1}[\{385\}] = \{5, 7, 11, 35, 55, 77\}$, i.e. the set of proper factors of 385.

The (usual, nonstrict) divisibility relation on the non-zero integers $\mathbb{Z} - \{0\}$, denoted by $|$, is given by $\{(x, y) \mid x, y \in \mathbb{Z} - \{0\} \text{ and } \exists z \in \mathbb{Z}(x \times z = y)\}$ (cf. Definition 5.2.1).

For a non-numerical example, consider the containment relation on the powerset of a set U : $\{(X, Y) \mid X, Y \subseteq U \text{ and } X \subseteq Y\}$. Here the domain and range are both equal to $\mathcal{P}(U)$.

Example 2.3.4

Here are some binary relations on the set of humans:

- $\text{ChildOf} = \{(a, b) \mid a \text{ and } b \text{ are humans and } b \text{ is a child of } a\}$.
- $\text{ParentOf} = \{(a, b) \mid a \text{ and } b \text{ are humans and } a \text{ is a parent of } b\}$.
- $\text{IsMarriedTo} = \{(a, b) \mid a \text{ and } b \text{ are humans and } a \text{ is married to } b\}$.

Here are some other relations between humans and other things:

- OwnsDog = $\{(a, b) \mid a \text{ is a human and } b \text{ is a dog and } a \text{ owns } b\}$.
- IsCitizenOf = $\{(a, b) \mid a \text{ is a human and } b \text{ is a country and } a \text{ is a citizen of } b\}$.
- IsMemberOf = $\{(a, b) \mid a \text{ is a human and } b \text{ is a club and } a \text{ is a member of } b\}$.

Since every human has (biological) parents, the domain of the relation ChildOf is the set of all humans, while the range is the set of all humans who have children. The identification of the domains and ranges of the other relations is left as an exercise.

Some special relations on a set A :

- The **empty relation** is just the empty set \emptyset and relates no element of A .
- The **equality (identity, diagonal)** relation

$$E_A = \{(a, a) \mid a \in A\}$$

relates every element of A only to itself.

- The **universal relation** A^2 relates every element of A to every element of A .

We can also restrict the domain of a relation, just like that of a function:

Definition 2.3.5 If R is a binary relation on a set A and $B \subseteq A$, then the **restriction of R to B** , denoted $R|_B$, is the relation

$$R|_B = R \cap (B \times B) = \{(x, y) \mid (x, y) \in R \text{ and } x, y \in B\}.$$

In other words, the restriction of a relation R on a set A to a subset B of A is the relation we get by throwing out of R those ordered pairs that have one or more coordinates that do not belong to B . Equivalently, it is the relation obtained from R by keeping exactly those ordered pairs in R that have both their coordinates in B .

Example 2.3.6

One could restrict the relation \leq on \mathbb{Z} (Example 2.3.3) to any subset of \mathbb{Z} , for example to the set of odd integers or to the members of the interval from -3 to 121 .

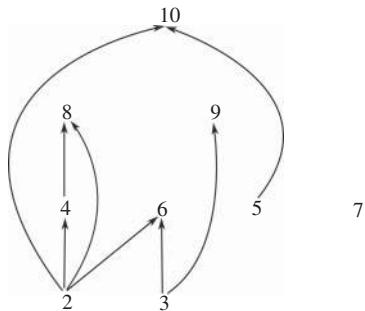
If we were to restrict the proper division relation div (Example 2.3.3) to the set of all prime numbers in \mathbb{N}^+ we would obtain the empty relation \emptyset , for no prime number properly divides any prime number.

On the other hand, if we were to restrict the division relation $|$ (Example 2.3.3) to the set P of all prime numbers in \mathbb{N}^+ we would obtain the relation

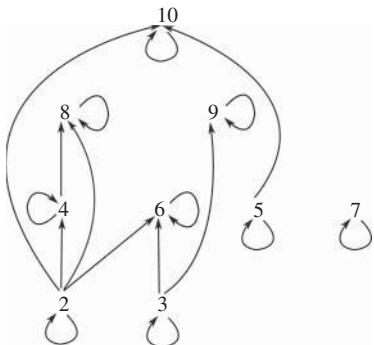
$$\{(x, y) \mid x | y \text{ and } x, y \in P\} = \{(x, x) \mid x \in P\}.$$

2.3.2. Matrix and graphical representations of relations on finite sets

Binary relations on finite sets can be conveniently represented either graphically or by means of matrices. A common way to represent a binary relation R on a finite set A graphically is to draw nodes representing the elements of A on the page in some convenient configuration, and then, for every two elements $a, b \in A$, draw an arrow from the node representing a to the node representing b if and only if aRb . For example, the proper divisibility relation div on the set $\{2, 3, \dots, 10\}$ can be represented as:



The nonstrict divisibility relation $|$ on the set $\{2, 3, \dots, 10\}$ is the same as the strict one, except that each number also divides itself. This is reflected in the graphical representation by the fact that each number now also has an arrow linking it to itself:



Another way of representing a binary relation on a finite set is by means of a matrix. If the set has n elements this will be an $n \times n$ matrix with rows and columns corresponding to the elements of the set. We then place a 1 in row i column j if iRj and a 0 otherwise. Thus, the matrix representation of the proper divisibility relation div on the set $\{2, 3, \dots, 10\}$, graphically

represented above, will be the following:

$$\begin{array}{cccccccccc} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \begin{matrix} 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{matrix} & \left(\begin{array}{cccccccccc} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right) \end{array}$$

The matrix representation of the relation \leq on the set $\{2, 3, \dots, 10\}$ is

$$\begin{array}{cccccccccc} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \begin{matrix} 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{matrix} & \left(\begin{array}{cccccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right) \end{array}$$

2.3.3. Boolean operations on binary relations

Since binary relations are sets themselves, the set operations \cup , \cap , $-$ also apply to them. Besides, the **complementation** of a relation $R \subseteq A \times B$ is defined as $R' = (A \times B) - R$.

Note that we can always assume that two relations are defined between the same sets because if $R_1 \subseteq A_1 \times B_1$ and $R_2 \subseteq A_2 \times B_2$, then both $R_1 \subseteq (A_1 \cup A_2) \times (B_1 \cup B_2)$ and $R_2 \subseteq (A_1 \cup A_2) \times (B_1 \cup B_2)$.

Example 2.3.7

Consider again the relations $<$, \leq and div from Example 2.3.3. Then $< \cup \leq = \leq$, $< \cap \leq = <$, $<' = \mathbb{Z}^2 - < \geq$ and $\leq' = \mathbb{Z}^2 - \leq = >$. Also, $\text{div} \cup < = <$ and $\text{div} \cap < = \text{div}$.

Example 2.3.8

Consider again the relation from Example 2.3.4. Unless there are people who are married to their parents, it should hold that $\text{ChildOf} \cap \text{IsMarriedTo} = \emptyset$. Also, unless some dogs are clubs, $\text{OwnsDog} \cap \text{IsMemberOf} = \emptyset$. The relation $(\text{IsCitizenOf})'$ relates every person to all those countries of which (s)he is not a citizen.

Example 2.3.9

Let F denote the sets of all female humans, D be the set of all Dachshunds, and let c denote the Rotary Club. Then $\text{ParentOf}^{-1}[F]$ is the set of all people with a female child, $\text{ParentOf}[F]$ is the set of all humans (since everyone has a mother) and $\text{IsMemberOf}^{-1}[\{c\}] \cap \text{OwnsDog}^{-1}[D]$ is the set of all members of the Rotary Club who own a Dachshund.

Proposition 2.3.10 *For any relations $R, S \subseteq A \times B$ the following hold:*

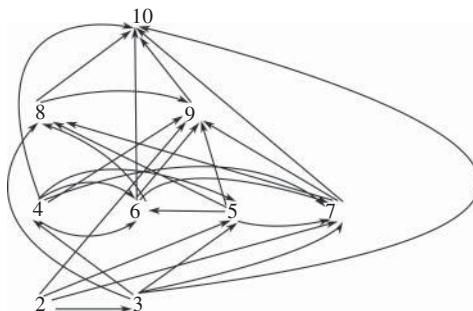
1. $\text{dom}(R \cup S) = \text{dom}(R) \cup \text{dom}(S)$
2. $\text{rng}(R \cup S) = \text{rng}(R) \cup \text{rng}(S)$
3. $\text{dom}(R \cap S) \subseteq \text{dom}(R) \cap \text{dom}(S)$
4. $\text{rng}(R \cap S) \subseteq \text{rng}(R) \cap \text{rng}(S)$

Proof: We will prove property 1. For the sake of the inclusion $\text{dom}(R \cup S) \subseteq \text{dom}(R) \cup \text{dom}(S)$, suppose that $a \in \text{dom}(R \cup S)$, i.e. there exists $b \in B$ such that $(a, b) \in R \cup S$. However, then $(a, b) \in R$ or $(a, b) \in S$, i.e. $a \in \text{dom}(R)$ or $a \in \text{dom}(S)$, so $a \in \text{dom}(R) \cup \text{dom}(S)$.

For the converse inclusion, suppose that $a \in \text{dom}(R) \cup \text{dom}(S)$, i.e. $a \in \text{dom}(R)$ or $a \in \text{dom}(S)$, so there is a $b \in B$ such that $(a, b) \in R$ or $(a, b) \in S$. However, then $(a, b) \in R \cup S$ and hence $a \in \text{dom}(R \cup S)$.

Example 2.3.11

The graphical representation of the relation $|$ on the set $\{2, 3, \dots, 10\}$ was given above. Now consider the relation $|' \cap \leq$ on the set $\{2, 3, \dots, 10\}$. Here is the graphical representation of this relation:



In effect, we have exactly those arrows linking smaller to larger numbers and that were not in the graphical representation of $|$.

2.3.4. Inverse and composition of relations

The inverse of a binary relation R is the relation obtained by swapping the components in each ordered pair in R . Formally:

Definition 2.3.12 Let $R \subseteq A \times B$. The **inverse of R** is the relation $R^{-1} \subseteq B \times A$, defined as

$$R^{-1} = \{(b, a) \mid (a, b) \in R\}.$$

Thus, we have aRb iff $bR^{-1}a$.

Example 2.3.13

Let us consider the inverses of the relations from Example 2.3.3. Here we have $<^{-1}=>$, $\leq^{-1}=\geq$, and the inverse of the “strictly divides” relation is the “proper multiple” relation. The inverse of the subset relation \subseteq on $\mathcal{P}(U)$ is the superset relation \supseteq .

Next, let us look at the inverse of some of the relations from Example 2.3.4. Here we have $\text{ChildOf}^{-1} = \text{ParentOf}$ and $\text{ParentOf}^{-1} = \text{ChildOf}$. Notice that $\text{IsMarriedTo}^{-1} = \text{IsMarriedTo}$. Such relations are called “symmetric”.

Proposition 2.3.14 For any relations $R, S \subseteq A \times B$ the following hold:

1. $\text{dom}(R^{-1}) = \text{rng}(R)$
2. $\text{rng}(R^{-1}) = \text{dom}(R)$
3. $(R^{-1})^{-1} = R$
4. $(R \cup S)^{-1} = R^{-1} \cup S^{-1}$
5. $(R \cap S)^{-1} = R^{-1} \cap S^{-1}$
6. $(R')^{-1} = (R^{-1})'$, where R' is the complement of R

Proof: We will prove properties 1 and 4. In both cases we need to prove the equality of two sets, which we will do through a chain of “iff’s”.

1. $b \in \text{dom}(R^{-1})$ iff $\exists a \in A$ such that $(b, a) \in R^{-1}$ iff $\exists a \in A$ such that $(a, b) \in R$ iff $b \in \text{rng}(R)$
2. $(b, a) \in (R \cup S)^{-1}$ iff $(a, b) \in (R \cup S)$ iff $(a, b) \in R$ or $(a, b) \in S$ iff $(b, a) \in R^{-1}$ or $(b, a) \in S^{-1}$ iff $(b, a) \in R^{-1} \cup S^{-1}$

Definition 2.3.15 Let $R \subseteq A \times B$ and $S \subseteq B \times C$. The **composition of R and S** is the binary relation $R \circ S \subseteq A \times C$ defined by

$$R \circ S = \{(a, c) \mid \exists b \in B (aRb \wedge bSc)\}.$$

In particular, when $R \subseteq A^2$, then $R \circ R$ is defined and denoted by R^2 .

Note that we can always form the composition of two relations, even if their domains and ranges do not match, by applying the same trick we did for the Boolean combinations of relations, above. Specifically, if $R \subseteq A \times B_1$ and $S \subseteq B_2 \times C$, then $R \subseteq A \times (B_1 \cup B_2)$ and $S \subseteq (B_1 \cup B_2) \times C$.

Example 2.3.16

Let us look at the compositions of some of the relations from Examples 2.3.3 and 2.3.4.

- $(a, b) \in (< \circ <) \text{ iff } a \text{ is less than an integer that is less than } b; \text{ i.e. } (< \circ <) \text{ consists of all ordered pairs of integers } (a, b) \text{ such that } a \text{ is at least 2 less than } b.$
- $< \circ \leq = <.$ Indeed, $(a, b) \in (< \circ \leq) \text{ iff there exists an integer } c \in \mathbb{Z} \text{ such that } a < c \text{ and } c \leq b \text{, iff } (a, b) \in <.$
- $\text{divo } < \text{ consists of exactly those ordered pairs } (a, b) \text{ of integers where } a \text{ properly divides a number strictly less than } b.$ For example, $(11, 23) \in (\text{divo } <) \text{ since } 11 \text{ div } 22 \text{ and } 22 < 23.$
- $< \text{ odiv}$ consists of exactly those ordered pairs (a, b) of integers where a is strictly less than a number that properly divides $b.$ For example, $(7, 16) \in (< \text{ odiv})$ since $7 < 8$ and $8 \text{ div } 16.$ Note that $< \text{ odiv}$ is not equal to $\text{divo } <$ since $(11, 23) \notin (< \text{ odiv}).$
- The relation ChildOfoOwnsDog consists of exactly those pairs (a, b) such that a is a human, b is a dog and b is owned by a parent of $a.$
- The relation $\text{IsMarriedTo} \circ \text{IsCitizenOf}$ relates humans a to countries b , where a is married to a citizen of $b.$

Proposition 2.3.17 *The composition of binary relations is associative, i.e. $(R \circ S) \circ T = R \circ (S \circ T).$*

Proof: To be definite, let $R \subseteq A \times B$, $S \subseteq B \times C$ and $T \subseteq C \times D.$ Then $(a, d) \in (R \circ S) \circ T$ iff there exists a $c \in C$ such that $(a, c) \in (R \circ S)$ and $(c, d) \in T,$ iff there exists $c \in C$ and $b \in B$ such that $(a, b) \in R$, $(b, c) \in S$ and $(c, d) \in T,$ iff there exists $b \in B$ such that $(a, b) \in R$ and $(b, d) \in S \circ T$ and iff $(a, d) \in R \circ (S \circ T).$

Note, however, that the composition of binary relations is *not commutative*. Indeed, as we have already seen in Example 2.3.16, $< \text{ odiv}$ is not equal to $\text{divo } <.$

Proposition 2.3.18 *For any binary relations R and S , it holds that $(R \circ S)^{-1} = S^{-1} \circ R^{-1}.$*

Proof: Exercise.

2.3.5. Exercises

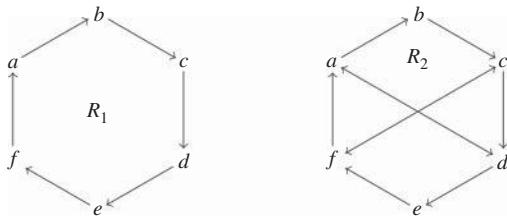
- ① Identify the domains and ranges of the relations ParentOf , IsMarriedTo , OwnsDog , IsCitizenOf and IsMemberOf from Example 2.3.4.
- ② Determine:
 - $\text{div}[\{2, 5\}]$
 - $\text{div}^{-1}[\{3, 10, 12\}]$
 - $< [\{12, 303\}]$
 - $<^{-1}[\{12, 303\}]$

- ③ Let F and M denote the sets of all female and male humans, respectively. Let E and A denote the sets of all European and all African countries, respectively, and let D be the set of all Dachshunds, S the set of all Scottish Terriers and let c denote the Rotary Club.

Then, for example, $F \cap (\text{OwnsDog}^{-1}[S] \cup \text{IsMarriedTo}^{-1}[\text{IsCitizenOf}^{-1}[A]])$ is the set of all women who own Scottish Terriers or who are married to citizens of African countries.

Determine the content of the following sets:

- (a) $D \cap \text{OwnsDog}[\text{IsCitizenOf}^{-1}[E]]$
 - (b) $S \cap \text{OwnsDog}[\text{ChildOf}^{-1}[\text{IsMemberOf}^{-1}[\{c\}]]]$
 - (c) $S \cap \text{OwnsDog}[\text{ChildOf}[\text{IsMemberOf}^{-1}[\{c\}]]]$
 - (d) $\text{ChildOf}^{-1}[\text{CitizenOf}^{-1}[A]] \cap \text{ChildOf}^{-1}[\text{CitizenOf}^{-1}[E]]$
 - (e) $\text{ChildOf}[\text{CitizenOf}^{-1}[A]] \cap \text{ChildOf}[\text{CitizenOf}^{-1}[E]]$
 - (f) $M \cap \text{ChildOf}^{-1}[\text{IsMemberOf}^{-1}[\{c\}]]$
 - (g) $F \cap \text{ChildOf}[\text{IsMemberOf}^{-1}[\{c\}]]$
- ④ The graphical representation of binary relations R_1 and R_2 on the set $\{a, b, c, d, e, f\}$ are given.



Give the graphical representations of the following relations:

- (a) $R_1 \cap R_2$
 - (b) $R_1 \cup R_2$
 - (c) $R_1 - R_2$
 - (d) R'_1
 - (e) R_1^{-1}
 - (f) $R_1 \circ R_2$
- ⑤ If R_1 and R_2 are binary relations on a set A , describe how the graphical representations of the following relations relate to the graphical representations of R_1 and R_2 :
- (a) $R_1 \cap R_2$
 - (b) $R_1 \cup R_2$
 - (c) $R_1 - R_2$

- (d) R'_1
 (e) R_1^{-1}
 (f) $R_1 \circ R_2$

- ⑥ The matrix representations of two relations S_1 and S_2 , respectively, on the set $\{\alpha, \beta, \gamma, \delta, \epsilon\}$ are given as

$$\begin{array}{c} \alpha \quad \beta \quad \gamma \quad \delta \quad \epsilon \\ \alpha \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array} \quad \begin{array}{c} \alpha \quad \beta \quad \gamma \quad \delta \quad \epsilon \\ \beta \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Determine the matrix representations of the following relations:

- (a) $S_1 \cap S_2$
 (b) $S_1 \cup S_2$
 (c) $S_1 - S_2$
 (d) S'_1
 (e) S_1^{-1}
 (f) $S_1 \circ S_2$

- ⑦ If S_1 and S_2 are binary relations on a set A , describe how you would compute the matrix representations of the following relations from the matrix representations of S_1 and S_2 :

- (a) $S_1 \cap S_2$
 (b) $S_1 \cup S_2$
 (c) $S_1 - S_2$
 (d) S'_1
 (e) S_1^{-1}
 (f) $S_1 \circ S_2$

- ⑧* Prove that for any relations $R, S \subseteq A \times B$ the following hold:

- (a) $\text{rng}(R \cup S) = \text{rng}(R) \cup \text{rng}(S)$
 (b) $\text{dom}(R \cap S) \subseteq \text{dom}(R) \cap \text{dom}(S)$
 (c) $\text{rng}(R \cap S) \subseteq \text{rng}(R) \cap \text{rng}(S)$

- ⑨ In each case, give an example of a set X and binary relations R and S on X such that

- (a) $\text{dom}(R) \cap \text{dom}(S) \not\subseteq \text{dom}(R \cap S)$
 (b) $\text{rng}(R) \cap \text{rng}(S) \not\subseteq \text{rng}(R \cap S)$

- (c) $\text{dom}(R \cap S) = \text{dom}(R) \cap \text{dom}(S)$
- (d) $\text{rng}(R \cap S) = \text{rng}(R) \cap \text{rng}(S)$
- ⑩* Prove that for any relations $R, S \subseteq A \times B$ the following hold:
- (a) $\text{rng}(R^{-1}) = \text{dom}(R)$
- (b) $(R^{-1})^{-1} = R$
- (c) $(R \cap S)^{-1} = R^{-1} \cap S^{-1}$
- (d) $(R')^{-1} = (R^{-1})'$, where R' is the complement of R
- ⑪* Prove that for any binary relations R, S it holds that $(R \circ S)^{-1} = S^{-1} \circ R^{-1}$.

The notation of set theory

Much of the notation that we use today for the set theoretic operations and concepts was first introduced by the Italian mathematician Giuseppe Peano (1858–1932), who made important contributions to set theory and logic (see the box at the end of Section 4.2 in Chapter 4). The membership symbol \in is a stylised version of the Greek letter ϵ , which is the first letter of the word $\epsilon\sigma\tauι$ meaning “is”. Peano also introduced the notations \cap and \cup for the intersection and the union of sets.

The symbols \notin for nonmembership and \emptyset for the empty set were introduced by Nicolas Bourbaki – the collective pseudonym of a group of French mathematicians who wrote highly influential and very rigorous expositions of core mathematical topics, starting in 1935. Bourbaki also introduced the terminology *injection*, *surjection* and *bijection*.

2.4. Special binary relations

In this section will be taking a closer look at some of the properties of well-known binary relations. We will also study functions seen as special types of relations.

2.4.1. Properties of binary relations

The commonly encountered binary relations usually have certain special properties. The following definition lists some of the most important properties of binary relations, expressed by formulae of first-order logic (see Section 1.2 if you are not sure about the meaning of these).

Definition 2.4.1 A binary relation $R \subseteq X^2$ is called:

- *reflexive*, if it satisfies $\forall x(xRx)$;
- *irreflexive*, if it satisfies $\forall x\neg(xRx)$, i.e. if $X^2 \setminus R$ is reflexive;
- *serial*, if it satisfies $\forall x\exists y(xRy)$.
- *functional*, if it satisfies $\forall x\exists !y(xRy)$, where $\exists !y$ means “there exists a unique y ”;
- *symmetric*, if it satisfies $\forall x\forall y(xRy \rightarrow yRx)$;
- *asymmetric*, if it satisfies $\forall x\forall y(xRy \rightarrow \neg yRx)$;
- *antisymmetric*, if it satisfies $\forall x\forall y(xRy \wedge yRx \rightarrow x = y)$;

- **connected**, if it satisfies $\forall x \forall y (xRy \vee yRx \vee x = y)$;
- **transitive**, if it satisfies $\forall x \forall y \forall z ((xRy \wedge yRz) \rightarrow xRz)$;
- **an equivalence relation**, if it is reflexive, symmetric, and transitive;
- **euclidean**, if it satisfies $\forall x \forall y \forall z ((xRy \wedge xRz) \rightarrow yRz)$.
- **a pre-order (or quasi-order)**, if it is reflexive; and transitive.
- **a partial order**, if it is reflexive, transitive and antisymmetric, i.e. an antisymmetric pre-order;
- **a strict partial order**, if it is irreflexive and transitive;
- **a linear order (or total order)**, if it is a connected partial order;
- **a strict linear order (or total order)**, if it is a connected strict partial order.

Example 2.4.2

Let us consider which of these properties hold for each of the relations $<$, \leq , div , $|$, ChildOf and IsMarriedTo from Example 2.3.3. This is summarized in the table below. We leave as an exercise to the reader to fill in the blanks.

	$<$	\leq	div	$ $	ChildOf	IsMarriedTo
Reflexive	✗	✓	✗	✓	✗	✗
Irreflexive	✓	✗	✓	✗	✓	✓
Serial	✓	✓	✗	✓	✓	✗
Functional	✗	✗	✗	✗	✗	✗
Symmetric	✗	✗			✗	✓
Asymmetric	✓	✗				
Antisymmetric	✓	✓				
Connected	✓	✓				
Transitive	✓	✓				
Equivalence relation	✗	✗				
Euclidean	✗	✗				
Pre-order	✗	✓				
Partial order	✗	✓				
Strict partial order	✓	✗				
Linear order	✗	✓				
Strict linear order	✓	✗				

Looking at the table one might be tempted to draw the conclusion that a relation is not reflexive if and only if it is irreflexive. This is not true, for consider the relation $\text{Shaves} = \{(x, y) \mid x \text{ and } y \text{ are men and } x \text{ shaves } y\}$ on the set of all men.

Proposition 2.4.3 *For any binary relation $R \subseteq X \times X$ the following hold:*

1. R is reflexive iff $E_X \subseteq R$.
2. R is symmetric iff $R^{-1} \subseteq R$ iff $R^{-1} = R$.

3. R is asymmetric iff $R^{-1} \cap R = \emptyset$.
4. R is antisymmetric iff $R^{-1} \cap R \subseteq E_X$.
5. R is connected iff $R \cup R^{-1} \cup E_X = X^2$.
6. R is transitive iff $R^2 \subseteq R$.

Proof: We will prove properties 3 and 5.

3. Suppose that R is asymmetric, i.e. $\forall x \forall y (xRy \rightarrow \neg yRx)$. Now suppose that $(x, y) \in R^{-1} \cap R$, i.e. $(x, y) \in R^{-1}$ and $(x, y) \in R$, or $(y, x) \in R$ and $(x, y) \in R$, which is impossible by the asymmetry of R . Hence, $R^{-1} \cap R = \emptyset$.
For the converse we argue contrapositively. Suppose that R is not asymmetric, i.e. $\neg \forall x \forall y (xRy \rightarrow \neg yRx)$, i.e. there exists $x, y \in X$ such that $(x, y) \in R$ and $(y, x) \in R$, i.e. $(x, y) \in R$ and $(x, y) \in R^{-1}$, i.e. $(x, y) \in R \cap R^{-1}$. We therefore conclude that $R^{-1} \cap R \neq \emptyset$.
5. Suppose R is connected. To show that $R \cup R^{-1} \cup E_X = X^2$ we only need to show that $X^2 \subseteq R \cup R^{-1} \cup E_X$ since the converse containment holds for any relation. To that end, take an arbitrary $(x, y) \in X^2$. If $x = y$ then $(x, y) = (x, x) \in E_X$ and we are done. If $x \neq y$, then since R is connected, we know that xRy or yRx , i.e. xRy or $xR^{-1}y$, or $(x, y) \in R \cup R^{-1}$. Conversely, suppose that $X^2 = R \cup R^{-1} \cup E_X$. Thus, for all $x, y \in X$, if $x \neq y$, we have $(x, y) \in R$ or $(x, y) \in R^{-1}$, i.e. for all $x, y \in X$, we have $(x, y) \in R$ or $(y, x) \in R$, which precisely means that R is connected.

2.4.2. Functions as relations

The idea that a function can be regarded as a special type of relation was already discussed in Chapter 1.

Definition 2.4.4 The *graph* of a function (mapping) $f : A \rightarrow B$ is the binary relation $G_f \subseteq A \times B$ defined by

$$G_f = \{(a, f(a)) \mid a \in A\}.$$

Proposition 2.4.5 A relation $R \subseteq A \times B$ is functional iff it is the graph of a function from A to B .

Proof: Exercise.

2.4.3. Reflexive, symmetric and transitive closures of a relation

Reflexivity, symmetry and transitivity are properties of relations that are often of particular interest, as we will see, for example, when we study equivalence relations in Section 2.5. Notice that each of these conditions requires the existence of certain ordered pairs in the relation. For example, if R is a relation of X , then reflexivity requires that $(x, x) \in R$ for each $x \in X$, while transitivity requires that $(x, z) \in R$ whenever $(x, y) \in R$ and $(y, z) \in R$. Thus if we start with a relation lacking one of these properties, it seems reasonable that we can extend it to a relation that has the property, by adding suitable pairs. We make the following definition:

Definition 2.4.6 Let $R \subseteq X \times X$ be a binary relation on a set X . Then:

- the **reflexive closure** of R , denoted R^{ref} , is the smallest reflexive binary relation on X such that $R \subseteq R^{\text{ref}}$,
- the **symmetric closure** of R , denoted R^{sym} , is the smallest symmetric binary relation on X such that $R \subseteq R^{\text{sym}}$ and
- the **transitive closure** of R , denoted R^{tran} , is the smallest transitive binary relation on X such that $R \subseteq R^{\text{tran}}$.

Remark 2 A few remarks on Definition 2.4.6 are in order. Firstly, the word “smallest” is used in the sense of set inclusion. For example, item 1 says that R^{ref} is a reflexive relation on X such that $R \subseteq R^{\text{ref}}$, and for any other reflexive relation $R' \subseteq X^2$, if $R \subseteq R'$, then $R^{\text{ref}} \subseteq R'$.

Secondly, note that in Definition 2.4.6, when speaking of the reflexive closure of R as the smallest reflexive binary relation on X such that $R \subseteq R^{\text{ref}}$, we make the implicit assumption that such a relation exists and is unique. Likewise, for the clauses for the symmetric and transitive closures. Proposition 2.4.8 below justifies these assumptions.

Example 2.4.7

Some examples:

- The transitive closure $\text{ChildOf}^{\text{tran}}$ of the ChildOf relation on the set of humans is the DescendentOf relation, which relates every human to all his/her ancestors. To see this, we reason informally as follows: ChildOf relates me to my mother and father, who in turn are related by ChildOf to their parents (my grandparents). Consequently, since $\text{ChildOf}^{\text{tran}}$ must be transitive, it will have to relate me to all my grandparents. Now my grandparents are related to their parents (my great-grandparents) by ChildOf (and hence also by $\text{ChildOf}^{\text{tran}}$), so by transitivity $\text{ChildOf}^{\text{tran}}$ must relate me to my great-grandparents, etc. Eventually, we see that $\text{ChildOf}^{\text{tran}}$ relates me to all my ancestors.
- Similarly, $\text{ParentOf}^{\text{tran}}$ relates every human to all his/her predecessors.
- Since the divisibility relation $|$ is transitive, it is itself the smallest transitive relation containing $|$, and hence $|^{\text{tran}} = |$.
- The symmetric closure $|^{\text{sym}}$ of $|$ relates every member of $\mathbb{Z} - \{0\}$ to all its multiples and all its divisors in $\mathbb{Z} - \{0\}$.

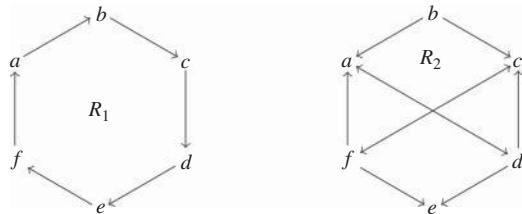
Proposition 2.4.8 Let $R \subseteq X \times X$. Then:

1. $R^{\text{ref}} = \bigcap \{S \mid R \subseteq S \subseteq X^2 \text{ and } S \text{ is reflexive}\},$
2. $R^{\text{sym}} = \bigcap \{S \mid R \subseteq S \subseteq X^2 \text{ and } S \text{ is symmetric}\},$
3. $R^{\text{tran}} = \bigcap \{S \mid R \subseteq S \subseteq X^2 \text{ and } S \text{ is transitive}\}.$

Proof: We will only prove clause 3, the proofs of 1 and 2 being very similar (see Exercise 9). Let $\mathcal{X} = \{S \mid R \subseteq S \subseteq X^2 \text{ and } S \text{ is transitive}\}$, i.e. \mathcal{X} is the set consisting of all transitive binary relations on X that contain R . We must show that $R^{\text{tran}} = \bigcap \mathcal{X}$. We begin by noting that $\mathcal{X} \neq \emptyset$, since X^2 is transitive and $R \subseteq X^2$. By definition of \mathcal{X} we have that $R \subseteq S$ for all $S \in \mathcal{X}$, so $R \subseteq \bigcap \mathcal{X}$ by Proposition 2.1.3(7). Also, $\bigcap \mathcal{X}$ is transitive: suppose that $(x, y), (y, z) \in \bigcap \mathcal{X}$. Then $(x, y), (y, z) \in S$ for all $S \in \mathcal{X}$ and, since each $S \in \mathcal{X}$ is transitive, $(x, z) \in S$ for all $S \in \mathcal{X}$. Hence $(x, z) \in \bigcap \mathcal{X}$. Thus we have established that $\bigcap \mathcal{X}$ is a transitive binary relation containing R . All that remains is to show that $\bigcap \mathcal{X}$ is the *smallest* such relation. To that end, suppose that $R' \subseteq X^2$ is transitive and $R \subseteq R'$. Then, by definition of \mathcal{X} we have $R' \in \mathcal{X}$, and hence $\bigcap \mathcal{X} \subseteq R'$.

2.4.4. Exercises

- ① Complete the table in Example 2.4.2.
- ② Give examples of binary relations, other than those mentioned in Example 2.4.2, that have the following properties:
 - (a) a relation that is transitive and reflexive but not antisymmetric;
 - (b) a partial order;
 - (c) serial but not functional;
 - (d) an equivalence relation.
- ③ The graphical representations of two binary relations, R_1 and R_2 , on the set $\{a, b, c, d, e, f\}$ are given.



Draw the graphical representations of the following relations:

- (a) R_1^{ref}
- (b) R_2^{sym}
- (c) R_1^{tran}
- (d) R_2^{tran}
- ④ Draw graphical representations of relations with the following properties:
 - (a) a relation that is serial but not functional;
 - (b) an irreflexive, functional and symmetric relation;

- (c) an irreflexive, serial and asymmetric relation;
 - (d) an equivalence relation that is not connected;
 - (e) an Euclidean relation that is serial but not functional.
- ⑤* Show that for any binary relations $R, S, T \subseteq X^2$ the following hold:
- (a) If R and S are transitive and $RoS = SoR = T$, then T is also transitive.
 - (b) If one of R and S is reflexive and $RoS = E_X$, then $R = S = E_X$.
- ⑥ Describe the following relations in the style of Example 2.4.7.
- (a) div^{ref}
 - (b) div^{sym}
 - (c) div^{tran}
 - (d) $<^{\text{ref}}$
 - (e) \leq^{sym}
 - (f) \leq^{tran}
- ⑦* Prove that for any binary relation $R \subseteq X \times X$ the following hold:
- (a) R is reflexive iff $E_X \subseteq R$.
 - (b) R is symmetric iff $R^{-1} \subseteq R$ iff $R^{-1} = R$.
 - (c) R is antisymmetric iff $R^{-1} \cap R \subseteq E_X$.
 - (d) R is transitive iff $R^2 \subseteq R$. (Recall that $R^2 = RoR$.)
- ⑧* Prove that a relation $R \subseteq A \times B$ is functional iff it is the graph of a function from $\text{dom}A$ to B .
- ⑨* Prove that for any binary relation $R \subseteq X \times X$,
- (a) $R^{\text{sym}} = \bigcap\{S \mid R \subseteq S \subseteq X^2 \text{ and } S \text{ is symmetric}\}$.
 - (b) $R^{\text{ref}} = \bigcap\{S \mid R \subseteq S \subseteq X^2 \text{ and } S \text{ is reflexive}\}$.

Fuzzy sets

Many concepts we use in everyday life are vague. For example, consider “the tall members of the American Mathematical Society”, “the democratic countries of the world”, or “the hot days in Sydney during January 2014”. If we were asked to determine the members of these sets, some cases would be clear-cut: for example, a day that reached a high of 34°C would definitely be a member of the set of hot days, while one on which the maximum temperature was 15°C would not. However, what about a day that reached 25°C ? Even membership in the set of living organisms is vague at its edges. Elephants are clearly living creatures while granite boulders are not, but things become a bit fuzzy when it comes to viruses, which scientists currently regard as falling in a sort of grey area. (Viruses cannot reproduce on their own but need to hijack the replication machinery of the cells of the living organisms that they infect; nor can they perform their own metabolic functions, all of which makes them look not alive. Yet, viruses evolve and spread their genes like living organisms.)

The notion of a *fuzzy set* can be used to reason about such vague concepts. A fuzzy set is a pair (U, μ) where U is a (universal) set and $\mu : U \rightarrow [0, 1]$ is a function assigning each element of U a real number between 0 and 1, which is its grade of membership. An element x is not included in (U, μ) if $\mu(x) = 0$, it is fully included if $\mu(x) = 1$ and it is a fuzzy member if $0 < \mu(x) < 1$. Ordinary sets can be seen as the special case of fuzzy sets where μ only takes on values 0 and 1.

(U, μ_A) is a fuzzy subset of (U, μ_B) if $\mu_A(x) \leq \mu_B(x)$ for all $x \in U$, i.e. if every element is a member if (U, μ_B) at least to the extent that it is a member of (U, μ_A) . The fuzzy intersection of (U, μ_A) and (U, μ_B) is the fuzzy set $(U, \mu_{A \cap B})$ where $\mu_{A \cap B}(x) = \min\{\mu_A(x), \mu_B(x)\}$, while their union is the fuzzy set $(U, \mu_{A \cup B})$ where $\mu_{A \cup B}(x) = \max\{\mu_A(x), \mu_B(x)\}$.

Fuzzy set theory and the fuzzy logic and mathematics built upon it has found many applications, for example in Artificial Intelligence, where it can be used to reason about real world situations where information is often incomplete or vague. Fuzzy sets were first introduced in 1965 by Lotfi A. Zadeh, a mathematician, electrical engineer, computer scientist and artificial intelligence researcher.

2.5. Equivalence relations and partitions

In many settings, mathematical and otherwise, we often want to regard two objects as “the same” or “interchangeable” if they agree with regards to a certain property. In that case we say that the objects are “equivalent”. For example, working with distances, I might want to regard two numbers as equivalent if they have the same absolute value; we can say that two composite propositions are logically equivalent if they always have the same truth value, regardless of the truth values of their primitive components; we could declare two algorithms equivalent if they always give the same output when given the same input; we could regard two insurance policies as equivalent if they give the same cover for the same premiums and under the same conditions; etc.

In this section we will take a closer look at relations of equivalence and give a precise mathematical definition of the notion of equivalence relation; we will see how an equivalence relation divides (or partitions) the set on which it is defined into equivalence classes and how, consequently, equivalence relations and partitions are two sides of the same coin.

2.5.1. Equivalence relations

Having discussed the informal notion of equivalence, let us consider how one would capture this as a formal mathematical concept. Firstly, equivalence is a relationship between two objects, i.e. a binary relation. Now it is also clear that not every binary relation can be regarded as an equivalence; for example, the binary relation $<$ on \mathbb{Z} is not such a relation. Therefore we would have to impose constraints on the relations if it is to capture the notion of equivalence: every object should be equivalent to itself; if object a is equivalent to object b , then object b must, of course, also be equivalent to object a ; if object a is equivalent to object b and object b is in turn equivalent to an object c , it seems reasonable to require that object a must also be equivalent to object c . Putting all this together yields the following definition.

Definition 2.5.1 Let X be a set and $R \subseteq X \times X$ a binary relation on X . We call R an *equivalence relation* on X if it is reflexive, symmetric and transitive, i.e. if for all $x, y, z \in X$ it holds that

1. xRx ,
2. if xRy then yRx and
3. if xRy and yRz then xRz .

Example 2.5.2

The following are equivalence relations on the set \mathbb{Z} :

1. The equality relation $=$. For every $x \in \mathbb{Z}$ we have $x = x$. If $x, y \in \mathbb{Z}$ and $x = y$ then $y = x$. If $x, y, z \in \mathbb{Z}$ and $x = y$ and $y = z$, then $x = z$.
2. The relation of having the same absolute value. Let us write $x \equiv_{\text{abs}} y$ if $|x| = |y|$ and check that this relation \equiv_{abs} satisfies the requirements of Definition 2.5.1. For any number $x \in \mathbb{Z}$ we have $|x| = |x|$, i.e. $x \equiv_{\text{abs}} x$, so \equiv_{abs} is a reflexive relation. Next, to prove that \equiv_{abs} is symmetric suppose that $x, y \in \mathbb{Z}$ and that $x \equiv_{\text{abs}} y$. This means that $|x| = |y|$. However, then $|y| = |x|$ and so $y \equiv_{\text{abs}} x$. Lastly, to prove that \equiv_{abs} is transitive, suppose that $x, y, z \in \mathbb{Z}$ and that $x \equiv_{\text{abs}} y$ and $y \equiv_{\text{abs}} z$. This means that $|x| = |y|$ and $|y| = |z|$. However, then $|x| = |z|$, so $x \equiv_{\text{abs}} z$, as desired.
3. The relation of having the same remainder when divided by 3, denoted \equiv_3 . There are of course exactly three possible remainders after division by 3, namely 0 (in the case of multiples of 3), 1 and 2. For example, $3 \equiv_3 0 \equiv_3 -3 \equiv_3 6, -2 \equiv_3 1 \equiv_3 4 \equiv_3 7$ and $-1 \equiv_3 2 \equiv_3 5 \equiv_3 8$. It is easy to prove that \equiv_3 is an equivalence relation (see Exercise 1). This type of equivalence relation will be studied in much greater detail in Section 5.5.1.
4. The relation sim of similarity on the set of all triangles in the plane.

Definition 2.5.3 Let $R \subseteq X \times X$ be an equivalence relation on X and $x \in X$. The subset

$$[x]_R = \{y \in X \mid xRy\}$$

is called the *equivalence class* (or the *cluster*) of x generated under R .

Proposition 2.5.4 For every equivalence relation R on a set X and $x, y \in X$ the following hold:

1. $x \in [x]_R$,
2. $x \in [y]_R$ implies $[x]_R = [y]_R$,
3. $x \notin [y]_R$ implies $[x]_R \cap [y]_R = \emptyset$,
4. if $[x]_R \neq [y]_R$ then $[x]_R \cap [y]_R = \emptyset$.

Proof: The proofs of (1), (2) and (4) are left as Exercise 8. We prove (3), by proving the contrapositive. Suppose that $[x]_R \cap [y]_R \neq \emptyset$. Thus, there is at least one element $z \in [x]_R \cap [y]_R$, and so xRz and yRz . By symmetry we also have zRx , and then by transitivity from yRz and zRx we get yRx , i.e. $x \in [y]_R$.

Items (1) and (2) mean that it does not matter which member of an equivalence class we choose to name or represent it: if $x, z \in [y]_R$ then $[x]_R = [y]_R = [z]_R$. Together items (1), (2) and

(3) imply (4), which, taken together, imply that any two equivalence classes are either the same or disjoint.

Example 2.5.5

Let us see what the equivalence classes of the equivalence relations of Example 2.5.2 look like.

1. The equivalence classes under $=$ are singletons, e.g. $[3]_{} = \{3\}$, $[0]_{} = \{0\}$ and $[-99]_{} = \{-99\}$. In general $[x]_{} = \{x\}$ for all $x \in \mathbb{Z}$. Since each equivalence class has only one member there is only one way to represent each class.
2. $[2]_{\equiv_{\text{abs}}} = \{-2, 2\}$, $[-50]_{\equiv_{\text{abs}}} = \{-50, 50\}$ and $[0]_{\equiv_{\text{abs}}} = \{0\}$. It is clear that the equivalence class under \equiv_{abs} of every integer x other than 0 will consist of exactly two elements, namely x and $-x$. Moreover, $[x]_{\equiv_{\text{abs}}} = [-x]_{\equiv_{\text{abs}}}$, for example $[2]_{\equiv_{\text{abs}}} = [-2]_{\equiv_{\text{abs}}}$. Notice that there are infinitely many different equivalence classes under \equiv_{abs} .
3. The equivalence class of 4 under the relation \equiv_3 consists of all those integers that have the same remainder as 4 upon division by 3, namely 1, i.e. $[4]_{\equiv_3} = \{\dots, -5, -2, 1, 4, 7, 10, \dots\}$. We also have $[0]_{\equiv_3} = \{\dots, -6, -3, 0, 3, 6, 9, \dots\}$ and $[2]_{\equiv_3} = \{\dots, -7, -4, -1, 2, 5, 8, \dots\}$. As you would expect, these are the only three equivalence classes under \equiv_3 , since there are exactly three different remainders after division by 3. Notice that we can represent each of these equivalence classes in infinitely many different ways, for example $[4]_{\equiv_3} = [-5]_{\equiv_3} = [7]_{\equiv_3} = \dots$, etc.
4. If ABC is a triangle of which all side lengths are 1 unit, then $[ABC]_{\text{sim}}$ is the set of all equilateral triangles.

2.5.2. Quotient sets and partitions

An equivalence relation divides up the set on which it is defined into equivalence classes. The resulting set of equivalence classes is an important mathematical object, known as the quotient set.

Definition 2.5.6 Let $R \subseteq X \times X$ be an equivalence relation. The set

$$X/R = \{[x]_R \mid x \in X\},$$

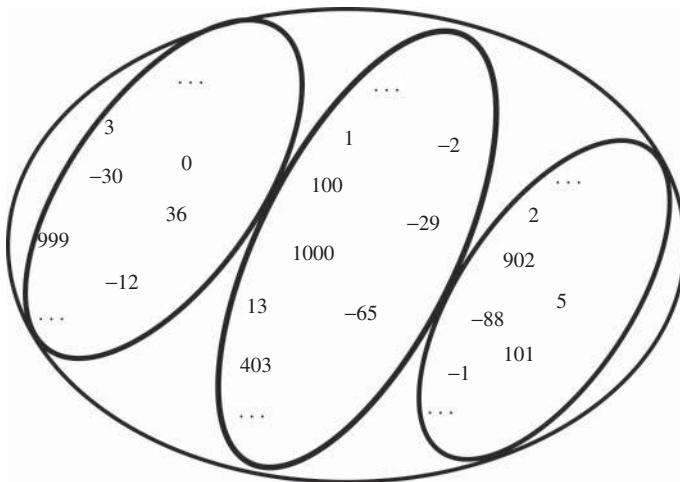
consisting of all equivalence classes of elements of X under R , is called the **quotient set of X by R** .

The equivalence class $[x]_R$, also denoted by x/R , is also called the **quotient element of x by R** .

The mapping $\eta_R : X \rightarrow X_R$ defined by $\eta_R(x) = [x]_R$ is called the **canonical mapping of X onto X/R** .

The quotient set \mathbb{Z}/\equiv_3 is graphically represented in Figure 2.7. Notice, as we already hinted above, that the different equivalence classes (in this case $[0]_{\equiv_3}$, $[1]_{\equiv_3}$ and $[2]_{\equiv_3}$) are disjoint and that $[0]_{\equiv_3} \cup [1]_{\equiv_3} \cup [2]_{\equiv_3} = \mathbb{Z}$. That is to say, the sets $[0]_{\equiv_3}$, $[1]_{\equiv_3}$ and $[2]_{\equiv_3}$ form a partition of \mathbb{Z} . Here is the formal definition of partition.

Definition 2.5.7 A **partition** of a set X is any family \mathcal{P} of nonempty and pairwise disjoint subsets of X , the union of which is X .

**Figure 2.7:** The quotient set \mathbb{Z}/\equiv_3 **Example 2.5.8**

The family $\{\{d, b, c\}, \{a\}, \{g, f\}, \{e, h\}\}$ is a partition of the set $\{a, b, c, d, e, f, g, h\}$. Indeed,

$$\{d, b, c\} \cup \{a\} \cup \{g, f\} \cup \{e, h\} = \{a, b, c, d, e, f, g, h\}$$

and

$$\begin{array}{ll} \{d, b, c\} \cap \{a\} = \emptyset & \{d, b, c\} \cap \{g, f\} = \emptyset \\ \{d, b, c\} \cap \{e, h\} = \emptyset & \{a\} \cap \{g, f\} = \emptyset \\ \{a\} \cap \{e, h\} = \emptyset & \{g, f\} \cap \{e, h\} = \emptyset. \end{array}$$

As we mentioned before, the quotient set $\{[0]_{\equiv_3}, [1]_{\equiv_3}, [2]_{\equiv_3}\}$ of \mathbb{Z} by \equiv_3 is a partition of \mathbb{Z} . This is no coincidence, as every quotient set by an equivalence relation is a partition of the set on which the relation is defined, a fact we now enshrine and prove as a theorem.

Proposition 2.5.9 *If R is an equivalence relation on a set X , then the quotient set*

$$\{[x]_R \mid x \in X\}$$

is a partition of X .

Proof: In order to show that $\{[x]_R \mid x \in X\}$ satisfies the conditions of Definition 2.5.7, we need to show that $\bigcup\{[x]_R \mid x \in X\} = X$ and that, for any two different equivalence classes $[x]_R$ and $[y]_R$, it holds that $[x]_R \cap [y]_R = \emptyset$.

Since $[x]_R \subseteq X$ for each $x \in X$ we immediately have that $\bigcup\{[x]_R \mid x \in X\} \subseteq X$. For the sake of the inclusion in the other direction, let $y \in X$ arbitrarily. By Proposition 2.5.4(1), $y \in [y]_R$ and therefore $y \in \bigcup\{[x]_R \mid x \in X\}$. We have therefore established that $\bigcup\{[x]_R \mid x \in X\} = X$.

Next suppose that $[x]_R \neq [y]_R$. By Proposition 2.5.4(4) this implies that $[x]_R \cap [y]_R = \emptyset$. This completes the proof.

Example 2.5.10

The quotient set $\mathbb{Z}/\equiv_{\text{abs}}$ of \mathbb{Z} by \equiv_{abs} is a partition of \mathbb{Z} into infinitely many sets, namely $\{0\}, \{-1, 1\}, \{-2, 2\}, \{-3, 3\}, \dots$. The canonical mapping $\eta_{\equiv_{\text{abs}}}$ sends each integer n to the set consisting of n and its additive inverse $-n$, i.e. $\eta_{\equiv_{\text{abs}}}(n) = \{-n, n\}$. This means that $\eta_{\equiv_{\text{abs}}}(n)$ is a two-element set for each nonzero integer n , but that $\eta_{\equiv_{\text{abs}}}(0) = \{-0, 0\} = \{0\}$.

Proposition 2.5.9 tells us that each equivalence relation corresponds to a partition. The converse of this is also true, as the next theorem shows.

Proposition 2.5.11 *If \mathcal{P} is a partition of a set X then the relation $\sim_{\mathcal{P}} \subseteq X \times X$ defined by*

$$x \sim_{\mathcal{P}} y \text{ iff } x \text{ and } y \text{ belong to the same member of } \mathcal{P}$$

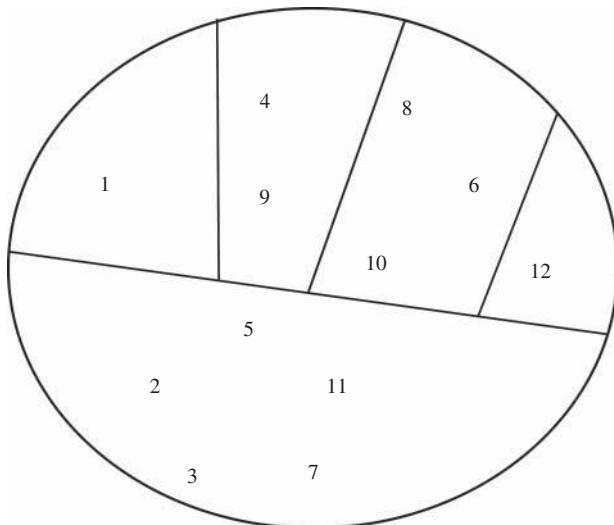
is an equivalence relation on X .

Proof: See Exercise 11.

Together Propositions 2.5.9 and 2.5.11 tell us that, for any set X , equivalence relations on X and partitions of X are the two faces of the same coin.

Example 2.5.12

Consider the partition \mathcal{P} of the set $\{1, 2, 3, \dots, 12\}$ pictured below.



Formally, $\mathcal{P} = \{\{1\}, \{2, 3, 5, 7, 11\}, \{4, 9\}, \{8, 6, 10\}, \{12\}\}$. In this case the equivalence relation $\sim_{\mathcal{P}}$ relates every number in $\{1, 2, 3, \dots, 12\}$ to those that have the same number of divisors as it. For example, each of the prime numbers 2, 3, 5, 7 and 11 has exactly two divisors, namely 1 and the number itself. So $\sim_{\mathcal{P}}$ relates all these numbers to themselves and to each other. The number 12 is the only number in the set having exactly 6 divisors (namely, 1, 2, 3, 4, 6 and 12) and so $\sim_{\mathcal{P}}$ relates 12 only to itself.

2.5.3. The kernel equivalence of a mapping

With every mapping we can associate an equivalence relation on the domain. The idea is that two elements x_1 and x_2 in the domain of a mapping f should be related if and only if $f(x_1) = f(x_2)$. Here is the formal definition.

Definition 2.5.13 Let $f : X \rightarrow Y$. Then the kernel equivalence of f is the relation $\tilde{f} \subseteq X \times X$ defined such that

$$x\tilde{f}y \text{ iff } f(x) = f(y).$$

The following theorem shows that we are justified in calling the kernel equivalence an equivalence.

Theorem 2.5.14 Let $f : X \rightarrow Y$. Then the kernel equivalence of f is an equivalence relation on X .

Proof: Exercise.

Example 2.5.15

Let $f : \mathbb{Z} \rightarrow \mathbb{Z}$ be defined by $f(x) \mapsto |x|$. We therefore have that $x\tilde{f}y$ if and only if $|x| = |y|$. In other words, \tilde{f} is exactly the same relation as \equiv_{abs} defined in Example 2.5.2.

Theorem 2.5.16 Let $f : X \rightarrow Y$, and let $\mu : X/\tilde{f} \rightarrow Y$ be the mapping defined as

$$\mu([x]_{\tilde{f}}) = f(x).$$

Then

1. μ is well defined,
2. μ is an injection and
3. $f = \mu\eta_{\tilde{f}}$.

Before we prove Theorem 2.5.16, let us discuss what it means for μ to be “well defined” and why it is important to know this about μ . Notice that the definition of $\mu([x]_{\tilde{f}})$ as $f(x)$ makes use of the particular element x used to represent the equivalence class $[x]_{\tilde{f}}$. What if $[x]_{\tilde{f}}$ had been written as $[y]_{\tilde{f}}$, where y is another element of $[x]_{\tilde{f}}$? By Proposition 2.5.4 this would mean that $[x]_{\tilde{f}} = [y]_{\tilde{f}}$. Certainly, then, we would want $\mu([x]_{\tilde{f}}) = \mu([y]_{\tilde{f}})$, i.e. we would not want the value of μ to depend on the particular choice of representative of the equivalence class, but simply on the equivalence class itself. This is what it means for μ to be *well defined*. Therefore, to prove that μ is well defined, it would be enough to show that if $[x]_{\tilde{f}} = [y]_{\tilde{f}}$, then $\mu([x]_{\tilde{f}}) = \mu([y]_{\tilde{f}})$. Now we are ready for the proof.

Proof:

1. As explained above, to prove that μ is well defined, we must show that $\mu([x]_{\tilde{f}}) = \mu([y]_{\tilde{f}})$ whenever $[x]_{\tilde{f}} = [y]_{\tilde{f}}$. Accordingly, suppose that $[x]_{\tilde{f}} = [y]_{\tilde{f}}$. Then $x\tilde{f}y$ and hence $f(x) = f(y)$, i.e. $\mu([x]_{\tilde{f}}) = \mu([y]_{\tilde{f}})$, so μ is well defined.

2. Suppose $\mu([x]_{\tilde{f}}) = \mu([y]_{\tilde{f}})$. Then $f(x) = f(y)$, i.e. $x \tilde{f} y$. Then it follows by Proposition 2.5.4 that $[x]_{\tilde{f}} = [y]_{\tilde{f}}$ and so μ is injective.
3. Finally, $(\mu \eta_{\tilde{f}})(x) = \mu(\eta_{\tilde{f}}(x)) = \mu([x]_{\tilde{f}}) = f(x)$.

The above theorem shows, in particular, that every mapping can be represented as a composition of a surjective mapping followed by an injective mapping.

Example 2.5.17

Consider again the mapping $f : \mathbb{Z} \rightarrow \mathbb{Z}$ defined by $f(x) \mapsto |x|$. Suppose we were to define a map $\gamma : \mathbb{Z}/_{\tilde{f}} \rightarrow \mathbb{Z}$ such that $\gamma : [x]_{\tilde{f}} \mapsto x - 1$. This γ would be an example of a map that is *not well defined* on $\mathbb{Z}/_{\tilde{f}}$. For example, $[-2]_{\tilde{f}} = [2]_{\tilde{f}}$, but $\gamma([-2]_{\tilde{f}}) = -2 - 1 = -3$ while $\gamma([2]_{\tilde{f}}) = 2 - 1 = 1$.

2.5.4. Exercises

- ① Prove that the relation \equiv_3 defined in Example 2.5.2 is an equivalence relation on \mathbb{Z} .
- ② Let $X = \{a, b, c, d, e, f, g, h\}$. For each of the following families of subsets of X , determine whether or not it is a partition of X . If it is a partition of X draw the graphical representation of the corresponding equivalence relation.
 - (a) $\{\{a, c, e, g\}, \{b, d\}, \{h, f, c\}\}$
 - (b) $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}\}$
 - (c) $\{\{a, c, e, g\}, \{d, f, h\}\}$
 - (d) $\{\{a, c, e, g\}, \{b, h\}, \{d, f\}\}$
- ③ (a) Show that the relation of being parallel (denoted \parallel) is an equivalence relation on the set L of all lines in the Cartesian plane.
 (b) How many elements are there in L/\parallel ?
 (c) Give an example of a binary relation on L that is not an equivalence relation.
- ④ Let 2^* be the set of all strings of 0s and 1s of finite length. We can think of 2^* as the set of all binary representations of positive integers. Let n be a positive integer and let \sim_n be the binary relation on 2^* that relates two elements if either (i) they are both shorter than n positions or (ii) they are both at least n positions long and they agree in their first n positions. For example, we will have $010 \sim_4 11$ (because both are shorter than 4 positions) and $101111001010 \sim_4 101100$, but $101 \not\sim_4 1011$ and $11011 \not\sim_4 11111$.
 - (a) Prove that \sim_n is an equivalence relation on 2^* .
 - (b) Describe the equivalence class $[1]_{\sim_1}$.
 - (c) How many elements does $[1]_{\sim_3}$ have?
 - (d) How many elements does $[1]_{\sim_n}$ have?

- ⑤ Draw the graphical representation of the relation $\sim \mathcal{P}$ in Example 2.5.12.
- ⑥* Prove that if $R \subseteq \mathbb{Z} \times \mathbb{Z}$ is an equivalence relation on \mathbb{Z} , then $= \subseteq R$, i.e. prove that the equality relation is contained in every equivalence relation on \mathbb{Z} . Conclude that $=$ is the smallest equivalence relation on \mathbb{Z} .
- ⑦* Let R_1 and R_2 be equivalence relations on a set X .
- Is it always the case that $R_1 \cap R_2$ is an equivalence relation on X ? Justify your answer.
 - Is it always the case that $R_1 \cup R_2$ is an equivalence relation on X ? Justify your answer.
 - Is it always the case that R_1^c is an equivalence relation on X ? Justify your answer.
- ⑧* Let R be an equivalence relation on a set X . Prove that for all $x, y \in X$ the following hold (see Proposition 2.5.4):
- $x \in [x]_R$.
 - $x \in [y]_R$ implies $[x]_R = [y]_R$.
 - If $[x]_R \neq [y]_R$ then $[x]_R \cap [y]_R = \emptyset$.
- ⑨* Prove that a relation $R \subseteq X^2$ is an equivalence relation on X if and only if it is reflexive and euclidean.
- ⑩** Let E_1 and E_2 be equivalence relations in a set X . Prove that the composition $E_1 \circ E_2$ is an equivalence relation if and only if the two relations commute, i.e. if and only if

$$E_1 \circ E_2 = E_2 \circ E_1.$$

- ⑪** Prove Proposition 2.5.11.

Zermelo–Fraenkel set theory

Russell's paradox (see the box in Section 2.1) arises from the assumption that any collection of objects that can be described forms a set. This assumption is made in what is known as *naive set theory*. To avoid such paradoxes we need to be more explicit about what we allow to be a set. This is done by adopting an *Axiomatic Set Theory*, which states explicitly which sets we assume to exist and in which ways new sets can be constructed from old. The best-known example of such a system is *Zermelo–Fraenkel set theory*, which is due to the German mathematician Ernst Zermelo (1871–1953) and the German-born Israeli mathematician Abraham Fraenkel (1891–1965). It can be argued that its eight axioms, possibly with the addition of the Axiom of Choice (see the box in Section 2.6), are sufficient to build all mathematical objects, i.e. that a mathematical object exists if and only if it can be shown to exist on the basis of these axioms. As such, Zermelo–Fraenkel set theory provides a foundation for mathematics.

Some of the axioms are a bit involved, so we will not list them all here, but will only provide a few examples to give an idea of the spirit of axiomatic set theory.

Axiom of Pairing. For any sets a and b there exists a set $\{a, b\}$ that contains exactly a and b as elements. Expressed in first-order logic this becomes

$$\forall a \forall b \exists c \forall x [x \in c \leftrightarrow x = a \vee x = b].$$

Axiom of Union. For every set x there exists a set $y = \bigcup x$, i.e. a set containing exactly the elements of the elements of x . Expressed in first-order logic this becomes

$$\forall x \exists y \forall u (u \in y \leftrightarrow \exists z (z \in x \wedge u \in z)).$$

Powerset Axiom. The powerset of any set exists. In first-order logic:

$$\forall x \exists y \forall u (u \in y \leftrightarrow \forall v (v \in u \rightarrow v \in x)).$$

Zermelo–Fraenkel set theory is often abbreviated to ZF, while ZFC denotes a system with the addition of the Axiom of Choice.

2.6. Ordered sets

Many well-known binary relations impose some type of order on the sets on which they are defined. For example, the relation \leq orders the integers \mathbb{Z} from smaller to larger; the relation \subseteq orders the members of the powerset $\mathcal{P}(X)$ from “smaller to larger” according to containment; the Ancestor relation orders humans into generation according to descent. Such ordering relations are extremely important in the mathematical sciences, and are the topic of the current section.

2.6.1. Pre-orders and partial orders

Partial orders and the more general pre-orders were already mentioned in Section 2.4. Partial orders are the most fundamental of the ordering relations that we want to study, and we therefore recall their definition.

Definition 2.6.1 Let X be a set and R a binary relation on X . The relation R is called a **pre-order** if it is reflexive and transitive. If, moreover, R is also antisymmetric, then it is called a **partial order**.

If R is a partial order on X , then the pair (X, R) is known as a **partially ordered set** or as a **poset**. If the ordering relation R is clear from the context we will often simply refer to X on its own as a poset.

Example 2.6.2

The pairs (\mathbb{N}, \leq) , (\mathbb{Z}, \leq) , (\mathbb{Q}, \leq) and (\mathbb{R}, \leq) are all posets. The pair $(\{1, 2, 3, \dots, 10\}, |)$, where $|$ is the relation of divisibility, is a poset. For any set X , the pair $(\mathcal{P}(X), \subseteq)$ is a poset.

When we want to speak about an arbitrary poset, we will often denote it as (X, \leq) , using the symbol “ \leq ” to denote the partial order relation, even though the elements of X need not be numbers of any sort. Given a poset (X, \leq) the following notations are standardly used:

$$\begin{array}{ll} x \geq y & \text{means} \\ x < y & \text{means} \\ x > y & \text{means} \end{array} \quad \begin{array}{l} y \leq x \\ x \leq y \text{ and } x \neq y \\ y < x \end{array}$$

Let (X, \leq) be a poset. For elements $x, y \in X$ we say that x is **covered by** y if

1. $x < y$ and
2. for every $z \in X$, if $x \leq z < y$ then $x = z$.

We use the notation $x < y$ to indicate that x is covered by y . Notice that x being covered by y means that $x < y$ and that there is no element of X strictly in between x and y .

Example 2.6.3

In the poset (\mathbb{Z}, \leq) every integer n is covered by exactly one integer, namely $n + 1$. In symbols, $n < n + 1$.

In the poset (\mathbb{Q}, \leq) no element is covered by another element. The same is true of the poset (\mathbb{R}, \leq) .

Let $Y = \{a, b, c, d, e, f\}$ and consider the poset $(\mathcal{P}(Y), \subseteq)$. The element $\{a, b\}$ is covered by exactly those subsets of Y that extend it with one element of Y , i.e. the subsets $\{a, b, c\}$, $\{a, b, d\}$, $\{a, b, e\}$ and $\{a, b, f\}$. The empty set \emptyset is covered by exactly the singleton subsets of Y , while Y is not covered by anything.

The following proposition shows that by taking a suitable quotient we can transform any pre-order into a partial order. It is a typical application of a quotient construction.

Proposition 2.6.4 *Let R be a pre-order on X . Then:*

1. *The relation \sim on X , defined by*

$$x \sim y \text{ iff } xRy \text{ and } yRx,$$

is an equivalence relation on X .

2. *The relation \tilde{R} on X/\sim , defined by*

$$[x]_\sim \tilde{R} [y]_\sim \text{ iff } xRy,$$

*is a well-defined partial order on X/\sim , called **the partial order induced by R** .*

Proof: We leave the proof of part (1) as Exercise 11 and prove only part (2). To show that \tilde{R} is well defined means, as usual, to show that the definition of \tilde{R} does not depend on the particular choice of representatives of equivalence classes. In particular, we need to show that if $[x]_\sim = [x']_\sim$ and $[y]_\sim = [y']_\sim$, then $[x]_\sim \tilde{R} [y]_\sim$ iff $[x']_\sim \tilde{R} [y']_\sim$. To that purpose, suppose that $[x]_\sim = [x']_\sim$ and $[y]_\sim = [y']_\sim$ and that $[x]_\sim \tilde{R} [y]_\sim$. We need to show that $[x']_\sim \tilde{R} [y']_\sim$. Since $[x]_\sim \tilde{R} [y]_\sim$ we have xRy . From the facts $[x]_\sim = [x']_\sim$ and $[y]_\sim = [y']_\sim$ it follows by Proposition 2.5.4 that $x \sim x'$ and $y \sim y'$. Now applying the definition of \sim to these last two facts we get xRx' , $x'Rx$, yRy' and $y'Ry$. Next applying the fact that R is transitive to the facts $x'Ry$ and $y'Ry'$ yields $x'Ry'$. Then, by the definition of \tilde{R} we have $[x']_\sim \tilde{R} [y']_\sim$, as we wanted. A symmetric argument proves that $[x]_\sim \tilde{R} [y]_\sim$ if $[x']_\sim \tilde{R} [y']_\sim$.

Now that we have established that \tilde{R} is well defined, we proceed to prove that it is a partial order X/\sim .

Reflexivity. Since R is reflexive we have xRx for all $x \in X$ and hence that $[x]_\sim \tilde{R} [x]_\sim$ for all $[x]_\sim \in X/\sim$.

Transitivity. Suppose $[x]_\sim \tilde{R} [y]_\sim$ and $[y]_\sim \tilde{R} [z]_\sim$. By the definition of \tilde{R} we have xRy and yRz and hence by the transitivity of R that xRz . Again applying the definition of \tilde{R} we get $[x]_\sim \tilde{R} [z]_\sim$.

Antisymmetry. Suppose that $[x]_\sim \tilde{R} [y]_\sim$ and $[y]_\sim \tilde{R} [x]_\sim$. By the definition of \tilde{R} we have xRy and yRx and, consequently, by the definition of \sim that $x \sim y$. Then, by Proposition 2.5.4, $[x]_\sim = [y]_\sim$.

Every subset of a poset is itself a poset with respect to the restriction of the partial order, as the following proposition shows. We leave the proof as Exercise 12.

Proposition 2.6.5 *If (X, \leq) is a poset and $Y \subseteq X$, then $(Y, \leq|_Y)$ is also a poset.*

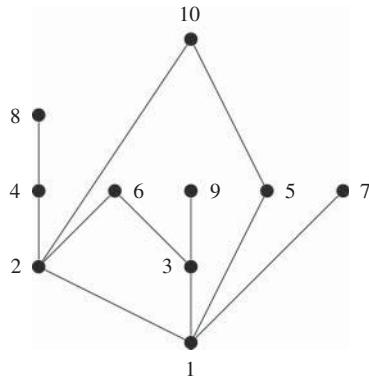
2.6.2. Graphical representing posets: Hasse diagrams

We have already seen how useful graphical representations can be for visualizing and understanding relations (Section 2.3.2). In particular, they can be applied to partial orders. The properties of partial orders allow for a simplified graphical representation. Informally speaking, we will omit all arrows linking nodes to themselves, since reflexivity implies that these are always present; we will also omit arrows, the existence of which can be inferred from the transitivity of the relation; lastly, we will link nodes with line segments instead of arrows but place them in such a way that the “direction” of the relational links is always upwards on the page. Such a representation of a poset is known as its **Hasse diagram**. More formally, the Hasse diagramme of a finite poset (X, \leq) is constructed as follows:

1. Place a node in the plane. Call it $p(x)$, representing each $x \in X$.
2. Draw a line segment linking $p(x)$ and $p(y)$ iff $x < y$.
3. Do this in such a way that
 - (a) if $x < y$ then $p(x)$ is lower than $p(y)$ in the plane and
 - (b) line segments never cross in a node, but may cross elsewhere.

Example 2.6.6

Consider the poset consisting of the integers 1 to 10 together with the divisibility relation $|$ restricted to this set. Here is its Hasse diagram:

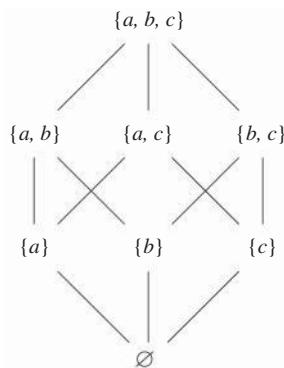


Notice that, even though there is no line segment from 2 to 8, the diagram does imply that $2 \mid 8$, since the line segments from 2 to 4 and from 4 to 8 mean that $2 \mid 4$ and $4 \mid 8$ and hence $2 \mid 8$ by the transitivity of \mid . On the other hand, the diagram does not imply that 6 divides 10, even though 10 is placed above 6, because they are not connected by a line segment.

If you try to draw the graphical representation (as in Section 2.3.2) of this relation you will appreciate how much simpler and less cluttered the Hasse diagrams are.

Example 2.6.7

Here is the Hasse diagram of the poset consisting of the power set of the three element set $\{a, b, c\}$ together with the set inclusion relation \subseteq :



2.6.3. Lower and upper bounds. Minimal and maximal elements

There are special elements in a partially ordered set: for example, there may be a smallest or largest element, or – more generally – maximal or minimal elements, in the whole set or in a given subset. Here we will define and discuss these and related notions.

Definition 2.6.8 Let (X, \leq) be a poset and $Y \subseteq X$.

- An element $x \in X$ is a **lower bound for Y in X** if $x \leq y$ for every $y \in Y$.
- An element $x \in X$ is the **greatest lower bound**, also called **infimum**, of Y in X if
 - x is a lower bound for Y in X and
 - $x' \leq x$ for every lower bound x' of Y in X .
- An element $x \in X$ is an **upper bound for Y in X** if $x \geq y$ for every $y \in Y$.
- An element $x \in X$ is the **least upper bound**, also called **supremum**, of Y in X if
 - x is an upper bound for Y in X , and
 - $x \leq x'$ for every upper bound x' of Y in X .

When the poset X is clear from the context we will simply speak of a lower bound of Y , omitting “in X ”, and similarly for upper bounds, infima and suprema.

When speaking of the supremum or infimum in the definition above we are making the implicit assumption that such an element is unique, if it exists. The next proposition shows that we are justified in this.

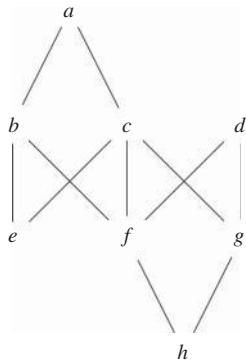
Proposition 2.6.9 Let (X, \leq) be a poset and $Y \subseteq X$.

1. If Y has a infimum in X then it is unique.
2. If Y has a supremum in X then it is unique.

Proof: We prove (1). Suppose that x and x' are infima of Y . Then, since x is a lower bound of Y we must have $x \leq x'$. Similarly, since x' is a lower bound of Y we must have $x' \leq x$. Then $x = x'$, by the antisymmetry of \leq .

Example 2.6.10

Consider the poset depicted below:



- The upper bounds for $\{e, b, f\}$ are a and b . The least of these upper bounds is b , so $\{e, b, f\}$ has a supremum, namely b , which, in this case, is itself a member of the set $\{e, b, f\}$. This is always the case for a set that has a greatest element, in this case b .
- The upper bounds for $\{e, g\}$ are a and c , and the least upper bound is c . This shows that the supremum of a set need not itself be a member of the set.
- The upper bounds for $\{e, f\}$ are a , b and c . In this case the least upper bound does not exist: it cannot be a , since $b \leq a$, it cannot be b , since $b \not\leq c$ and it cannot be c , since $c \not\leq b$.

The possibilities for infima are similar (see Exercise 3).

Even if a subset of a poset does not have a least element, it may have minimal elements, i.e. ones that are not greater than any element in the subset. For example, in the poset introduced in Example 2.6.10 the subset $\{b, c, e, f\}$ does not have a least element. However, no element in it is less than e or less than f . Likewise, maximal elements can be defined. This is made precise in the following definition.

Definition 2.6.11 Let (X, \leq) be a poset and $Y \subseteq X$.

1. An element $x \in Y$ is called **minimal** in Y if there is no element of Y strictly less than x , i.e. for every $y \in Y$, if $y \leq x$ then $x = y$.
2. An element $x \in Y$ is called **maximal** in Y if there is no element of Y strictly greater than x , i.e. for every $y \in Y$, if $y \geq x$ then $x = y$.

Note that the least (respectively, greatest) element of Y , if it exists, is the only minimal (respectively, maximal) element of Y .

Example 2.6.12

Consider the poset $(\mathbb{Z}^+, |)$ of positive integers with the divisibility relation $|$. The whole set \mathbb{Z}^+ has a least element, namely 1, but no maximal (hence, no greatest) elements.

The subset $\{2, 3, 4, \dots\}$ has no least element while it has infinitely many minimal elements, namely all primes.

2.6.4. Well-ordered sets

Well-ordered sets are a special type of poset in which it is impossible to “go downwards” forever, i.e. one in which whenever following the relation $>$ one will always come to a stop after a finite number of steps. Linear orders with this property are of particular interest, as it is possible to formulate an induction principle for them that generalizes the principle of mathematical induction. Let us make these intuitions a bit more precise:

Definition 2.6.13 *An ascending (respectively, strictly ascending) chain in a poset (X, \leq) is any finite or infinite sequence x_1, x_2, x_3, \dots of elements of X such that $x_1 \leq x_2 \leq x_3 \leq \dots$ (respectively, $x_1 < x_2 < x_3 < \dots$).*

A descending (respectively, strictly descending) chain in a poset (X, \leq) is any finite or infinite sequence x_1, x_2, x_3, \dots of elements of X such that $x_1 \geq x_2 \geq x_3 \geq \dots$ (respectively, $x_1 > x_2 > x_3 > \dots$).

Example 2.6.14

$1, 2, 2, 5, 6, 7, 7, 7, 101$ is an ascending chain in the poset (\mathbb{N}, \leq) . It is not strictly ascending. The chain $1, 1, 1, \dots$ is both ascending and descending in the poset (\mathbb{N}, \leq) . $2, 4, 8, 16, 32, \dots$ is an infinite strictly ascending chain in the poset $(\mathbb{N}, |)$. The chain $0, -2, -4, -6, -8, -10, \dots$ is an infinite strictly descending one in the poset (\mathbb{Z}, \leq) .

Definition 2.6.15 *A poset is called well-founded if it contains no infinite strictly descending chains.*

For instance, every finite poset is well founded.

Proposition 2.6.16 *The poset $(\mathcal{P}(X), \subseteq)$, where X is any infinite set, is not well founded.*

Proof: See Exercise 9.

Definition 2.6.17 *A well-founded linear order is called a well order.*

Example 2.6.18

(\mathbb{N}, \leq) is a well order, while (\mathbb{Z}, \leq) , (\mathbb{Q}, \leq) and (\mathbb{R}, \leq) are not. In particular, $0, -1, -2, -3, \dots$ is an infinite strictly descending chain in each of the posets (\mathbb{Z}, \leq) , (\mathbb{Q}, \leq) and (\mathbb{R}, \leq) .

Proposition 2.6.19 *The lexicographic order in \mathbb{N}^2 defined by*

$$\langle x_1, y_1 \rangle \leq \langle x_2, y_2 \rangle \text{ iff } x_1 < x_2 \text{ or } (x_1 = x_2 \text{ and } y_1 \leq y_2)$$

is a well order on \mathbb{N}^2 .

Proof: Exercise.

Proposition 2.6.20 *A poset (X, \leq) is well founded if and only if every nonempty subset of X has a minimal element.*

In particular, a linear order (X, \leq) is a well order if and only if every nonempty subset of X has a least element.

Proof: We prove the first claim and leave the second one as Exercise 13.

- (\Rightarrow) Suppose that (X, \leq) is well founded but, for the sake of contradiction, that there is a subset $\emptyset \neq Y \subseteq X$ such that Y has no minimal element. Take an arbitrary element of Y and call it y_1 . Since y_1 cannot be a minimal element of Y there must be another element, y_2 say, such that $y_2 \in Y$ and $y_1 > y_2$. Again, since y_2 cannot be a minimal element of Y there must be another element, y_3 say, such that $y_3 \in Y$ and $y_1 > y_2 > y_3$. Continuing in this way we can construct an infinite descending chain, contradicting our assumption that (X, \leq) is well founded. (This step of the proof requires a special axiom of set theory known as the **Axiom of Choice**; see the box at the end of this section.)
- (\Leftarrow) We prove the contrapositive. Suppose that (X, \leq) is not well founded. If (X, \leq) is not well founded it must contain an infinite descending chain, say $x_1 > x_2 > x_3 > \dots$. Now let $Y = \{x_1, x_2, x_3, \dots\}$. Clearly, Y is a nonempty subset of X that does not have a least element.

As promised, we now generalize the principle of mathematical induction to induction on arbitrary well-founded sets.

Theorem 2.6.21 (Induction principle for well-founded sets) *Let (X, \leq) be a well-founded set and $P \subseteq X$ be such that for every $x \in X$, if all elements of X less than x belong to P , then x itself belongs to P . Then $P = X$.*

Proof: Assume the contrary, i.e. $X \setminus P \neq \emptyset$. Then $X \setminus P$ has a minimal element x . Then all elements of X less than x belong to P ; hence x must belong to P – a contradiction.

Example 2.6.22

The set of formulae of propositional logic (see Definition 3.1.3) together with the subformula relation forms a well-founded set, which is not linear. Many properties of propositional formulae are proved by induction, using Theorem 2.6.21. For example, it can be proven in this way that every propositional formula is logically equivalent to one involving only the connectives \wedge and \neg .

2.6.5. Exercises

① Draw the Hasse diagrams of the following posets:

- (a) The set of integers $\{-2, 0, \dots, 8, 10\}$ with the \leq relation restricted to it.
- (b) The set $\mathcal{P}(\{0, 1, 2\})$ together with the \subseteq relation.
- (c) The set $X = \{a, b, c, d, e\}$ together with the relation

$$R = (X \times X) \cup \{(a, b), (a, c), (a, d), (a, e), (b, c), (e, d), (e, c), (a, c)\}.$$

② A binary string is a finite sequence of 0s and 1s, e.g. 0, 1001 and 11010101100. The empty string is the string of length 0 and is denoted by ϵ . We say that a string s is an initial segment of a string t if t can be obtained from s by adding zero or more 0s and/or 1s to the end of s . For example, 10 is an initial segment of each of the strings 101, 10, 1011 and 10101.

- (a) Show that the set of binary strings together with the initial segment relation forms a poset.
- (b) Show that the set of binary strings of length at most 4 together with the initial segment relation restricted to this set forms a poset.
- (c) Draw the Hasse diagram of the poset in question 2(b).
- ③ Give an example of a poset (X, \leq) and subsets $Y_1, Y_2, Y_3 \subseteq X$ such that
 - (a) Y_1 does not have an infimum in X ,
 - (b) Y_2 has an infimum in X that is also a member of Y_2 ,
 - (c) Y_3 has an infimum in X but that is not a member of Y_3 .
- ④ Consider the poset (\mathbb{Z}, \leq) where \leq is the usual ordering on the integers.
 - (a) Give an example of a proper subset of \mathbb{Z} that has no upper bounds.
 - (b) Give an example of a proper subset of \mathbb{Z} that has no lower bounds but does have an upper bound.
 - (c) Are there subsets of \mathbb{Z} with infinitely many upper bounds? Motivate.
 - (d) Are there subsets of \mathbb{Z} with finitely many upper bounds? Motivate.
 - (e) Is there a subset of \mathbb{Z} that has lower bounds but no infimum? Motivate.
- ⑤ Repeat Exercise 4, but this time for the poset (\mathbb{Q}, \leq) . What changes are there if the exercise is repeated for the poset (\mathbb{R}, \leq) ?
- ⑥ Consider the poset $(\mathcal{P}(X), \subseteq)$, where X is a nonempty set. Show that every subset $\mathcal{X} \subseteq \mathcal{P}(X)$ has both a supremum and an infimum in $(\mathcal{P}(X), \subseteq)$.
- ⑦ Give an example of a poset that has a unique minimal element, which is not a least one.
- ⑧ Give an example of a poset that has infinitely many maximal elements.
- ⑨* Let X be an infinite set. Show that the poset $(\mathcal{P}(X), \subseteq)$ is not well founded. (Hint: in order to do this, it is sufficient to exhibit an infinite strictly descending chain of elements of $\mathcal{P}(X)$. The key idea is that what remains after removing a single element from an infinite set is again an infinite set.)

⑩* Let (X, \leq) be a poset and $Y \subseteq X$. Show that if Y has a supremum in X , then it is unique.

⑪* Let R be a pre-order in X . Prove that the relation \sim on X defined by

$$x \sim y \text{ iff } xRy \text{ and } yRx$$

is an equivalence relation on X (see Proposition 2.6.4).

⑫* Show that if (X, \leq) is a poset and $Y \subseteq X$, then $(Y, \leq|_Y)$ is also a poset.

⑬* Complete the proof of Proposition 2.6.20, i.e. show that a linear order (X, \leq) is a well order if and only if every nonempty subset of X has a least element.

The axiom of choice

The axiom of choice states that, given any set $\{A_i \mid i \in I\}$ of non-empty, pairwise disjoint sets A_i , there exists a set X containing exactly one element from each A_i . Various different, but equivalent, forms of the axiom of choice are known and used, including the often-encountered principle known as *Zorn's Lemma*.

The axiom is not needed in case there are only finitely many A_i s or if the set X can be explicitly defined in some way. For example, suppose that each set A_i is a subset of the natural numbers and that the A_i s are pairwise disjoint. Then, by the well-ordering of the natural numbers (see Section 2.6.4), each A_i has a least element. We could then define X to consist of the least element from each of the sets A_i . So defined, X would contain exactly one element from each A_i , but its existence would not depend on the axiom of choice.

Another example is due to Bertrand Russell (see the box at the end of Section 4.3 in Chapter 4): suppose I have denumerably many pairs of shoes $\{P_i \mid i \in \mathbb{N}\}$. I could construct a set containing one shoe from each pair by, for example, choosing the left shoe from each pair. In contrast, if I had denumerable many pairs of socks $\{S_i \mid i \in \mathbb{N}\}$, because the socks in a pair are identical, I would not be able to specify such a procedure for picking exactly one from each pair. I would need to appeal to the axiom of choice to justify the existence of a set containing exactly one sock from each pair.

The axiom of choice cannot be derived from the other axioms of Zermelo–Fraenkel (see Section 2.5) set theory and must consequently be added as a separate axiom if we want to use it. There is some controversy attached to the axiom of choice as some mathematicians have objected to it on philosophical grounds. Also, using this axiom one can prove the existence of nonmeasurable sets of real numbers, which can be used to prove some rather unbelievable claims, such as the *Banach–Tarski paradox*, stating intuitively that a ball of radius 1 can be partitioned into finitely many subsets from which a ball of radius 2 can be composed. Therefore, even though the vast majority of mathematicians accept the axiom of choice as a valid principle, it is still customary to explicitly note its use in a proof.



2.7. An introduction to cardinality

In this section we look into two somewhat more advanced topics. First, we investigate what it means for two sets to have the same “size” – something that is intuitively clear in the case of finite sets, but rather trickier when it comes to infinite sets. Secondly, we explore the different shapes that well orders can come in, which leads us to the notion of ordinal.

2.7.1. Equinumerosity and cardinality

When do two sets have the same size? Intuitively, “size” means the “number of elements”. For finite sets the answer is easy: simply count the elements in the two sets and if you came up with the same number in both cases they have the same size, otherwise not. When we try to extend this to infinite sets, for example when comparing the sizes of the sets \mathbb{N} and \mathbb{Z} , we run into a problem: we cannot count the number of natural numbers or the number of integers because these sets are both infinite. Still, it seems that there are more – perhaps, twice as many – integers than there are natural numbers: one positive and one negative integer with the same absolute value per positive natural number. However, is that true? As we will see below, this is, in fact, not true: actually, the sets \mathbb{N} and \mathbb{Z} have the same size! Perhaps even more surprising is the fact that \mathbb{N} and \mathbb{Q} also have the same size. Are we therefore saying that all infinite sets have the same size? No! We will show that \mathbb{R} is strictly larger in size than \mathbb{N} , and in fact there are infinitely many possible different “infinite sizes” of a set.

The standard way used to compare the sizes of sets takes us back to the very origins of counting. Scholars believe that counting originated as a means of keeping track of possessions, like livestock. In a typical scenario (see Figure 2.8), a herder would make a mark on a post or put a pebble in the bag for each of his sheep as it left the sheep pen in the morning. In the evening these marks would be crossed off, or the pebbles taken out of the bag, one by one, as the sheep returned to the pen. If all marks were crossed off or all pebbles taken out, he knew that all his sheep were safely in the pen, but if a mark or a pebble remained, that meant that a sheep was missing. The idea being used here is that of a one-to-one correspondence (or a bijection) between the set of sheep and the set of marks or pebbles.

Definition 2.7.1 Two sets X and Y are said to be *equinumerous*, or *equipotent*, if there exists a bijection between them. We use the notation $X \approx Y$ to indicate that X and Y are equinumerous.

Example 2.7.2

The sets \mathbb{N} and \mathbb{Z} are equinumerous. To show this we need to find a bijective mapping from \mathbb{N} on to \mathbb{Z} . We let $f : \mathbb{N} \rightarrow \mathbb{Z}$ be the mapping defined by

$$f(n) = \begin{cases} \frac{n}{2}, & n \text{ is even} \\ -\frac{n+1}{2}, & n \text{ is odd} \end{cases}.$$

It is easy to show that f is a bijection; the details are left as Exercise 3.

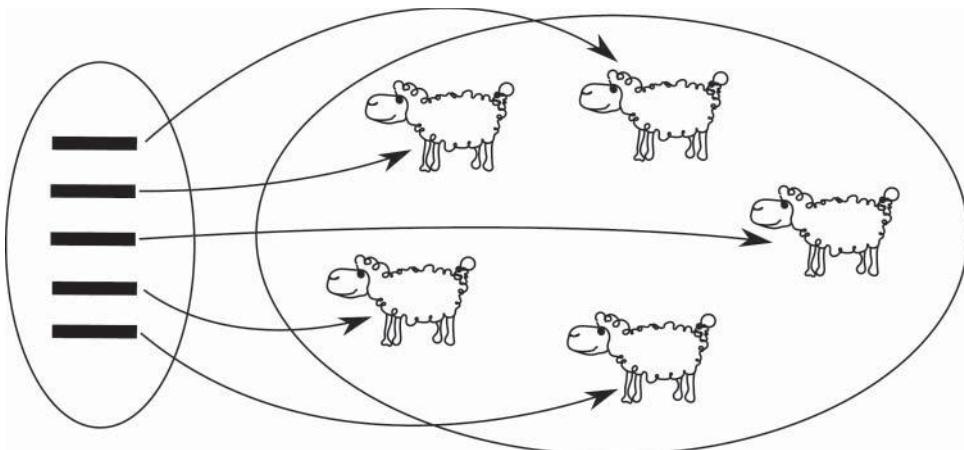


Figure 2.8: A one-to-one correspondence between marks on a post and a flock of sheep

Example 2.7.3

Now, as promised, we show that \mathbb{N} and \mathbb{Q} are equinumerous, by constructing a function from \mathbb{N} on to \mathbb{Q} . It is not so easy to specify this function with an explicit formula as we did in Example 2.7.2. Instead we describe how the desired function works. We begin by arranging all fractions a/b , with $a \in \mathbb{Z}$ and $b \in \mathbb{N} - \{0\}$, in a grid as follows:

...	$\frac{-3}{1} *$	$\frac{-2}{1} *$	\leftarrow	$\frac{-1}{1} *$	\leftarrow	$\frac{0}{1} *$	\rightarrow	$\frac{1}{1} *$	\rightarrow	$\frac{2}{1} *$	\rightarrow	$\frac{3}{1} *$...	
	↑	↓		↑		↓		↑		↑		↓		
...	$\frac{-3}{2} *$	$\frac{-2}{2} *$		$\frac{-1}{2} *$	\leftarrow	$\frac{0}{2} *$	\leftarrow	$\frac{1}{2} *$	\leftarrow	$\frac{2}{2} *$	\leftarrow	$\frac{3}{2} *$...	
	↑	↓		↑		↓		↑		↑		↓		
...	$\frac{-3}{3} *$	$\frac{-2}{3} *$	\rightarrow	$\frac{-1}{3} *$	\rightarrow	$\frac{0}{3} *$	\rightarrow	$\frac{1}{3} *$	\rightarrow	$\frac{2}{3} *$	\rightarrow	$\frac{3}{3} *$...	
	↑			↑				↑				↓		
...	$\frac{-3}{4} *$	\leftarrow	$\frac{-2}{4} *$	\leftarrow	$\frac{-1}{4} *$	\leftarrow	$\frac{0}{4} *$	\leftarrow	$\frac{1}{4} *$	\leftarrow	$\frac{2}{4} *$	\leftarrow	$\frac{3}{4} *$...
	⋮	⋮		⋮		⋮		⋮		⋮		⋮		

Since every rational can be represented in this way, every rational number occurs in the grid. In fact every rational occurs infinitely often, e.g. $1/2$ occurs as $1/2, 2/4, 3/6, 4/8, \dots$. Next we trace a path through the grid as indicated by the arrows, starting at $0/1$. We put a star on each rational number when it occurs on the path for the first time, e.g. we star $1/1$ but not $2/2$. Now define $g : \mathbb{N} \rightarrow \mathbb{Q}$ by specifying that g maps n to the $(n+1)$ st starred rational number. Thus $g(0) = 0/1, g(1) = 1/1, g(2) = 1/2, g(3) = -1/2, g(4) = -1/1$, etc. Since the range of g consists of exactly the starred fractions in the grid, it follows that g is injective. Also, since every rational occurs in the grid and is visited by the path, it will be the image of some natural number under g , i.e. g is surjective.

Until now we have been relying on an intuitive understanding of what it means for a set to be finite or infinite. It is time we made these notions a little more precise. We use the notation \mathbb{N}_n , where $n \in \mathbb{N}$, for the set $\{0, 1, \dots, n - 1\}$. Note that \mathbb{N}_n has n elements and, in particular, that $\mathbb{N}_0 = \emptyset$.

Definition 2.7.4 A set X is *finite* if it is equinumerous to \mathbb{N}_n for some $n \in \mathbb{N}$. If X is not finite, then it is *infinite*.

The finite sets together with those equinumerous to \mathbb{N} are called countable:

Definition 2.7.5 A set is called *denumerable* (or *countably infinite*) if it is equinumerous to \mathbb{N} . A set is *countable* if it is either finite or denumerable. A set that is not countable is *uncountable*.³

We have seen that the natural numbers, the integers and the rationals are all countable sets. What about the set of real numbers? We will show that this is an uncountable set, but first we will need a preliminary lemma. Since subsets are “smaller” than their supersets it seems plausible that subsets of countable sets should be countable. This is indeed the case, as stated by the next lemma. The proof is left as an exercise (see Exercise 8).

Lemma 2.7.6 If a set X is countable and $Y \subseteq X$, then Y is also countable.

Theorem 2.7.7 The set \mathbb{R} of real numbers is not countable.

Proof: We will show that the subset $I = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$ is uncountable. The theorem then follows by Lemma 2.7.6. Suppose, for the sake of contradiction, that there is a surjective function $f : \mathbb{N} \rightarrow I$. We make use of the fact that every number in I has a decimal representation of the form $0.a_1a_2a_3\dots$, where each a_i is a digit between 0 and 9. (It can be shown that if a real number has two decimal representations, one representation always terminates in repeating 0s and the other in repeating 9s. Specifically, $1 = 0.\dot{9}$.) We can enumerate the numbers in the range of f as follows:

$$\begin{aligned} f(0) &= 0 \cdot a_{01}a_{02}a_{03}\dots a_{0n}\dots \\ f(1) &= 0 \cdot a_{11}a_{12}a_{13}\dots a_{1n}\dots \\ f(2) &= 0 \cdot a_{21}a_{22}a_{23}\dots a_{2n}\dots \\ &\vdots \\ f(n) &= 0 \cdot a_{n1}a_{n2}a_{n3}\dots a_{nn}\dots \\ &\vdots \end{aligned}$$

We will now construct a number in I that is not a value of f . Let $y = 0 \cdot y_1y_2y_3\dots y_n\dots$, where

$$y_n = \begin{cases} 2 & \text{if } a_{nn} \geq 5 \\ 7 & \text{if } a_{nn} \leq 4 \end{cases}.$$

³In some texts the word *countable* is used more narrowly to refer to countably infinite sets. It is good to be aware of these differing usages in order to avoid the confusion that this can cause.

Note that y cannot be equal to $f(n)$ for any $n \in \mathbb{N}$. Indeed, by construction, $y_n \neq a_{nn}$, because y and a_{nn} differ in the n th decimal place. (Note also that y is not a real number with two decimal representations as it does not contain 0 or 9 as a digit.) Thus, $y \notin \text{rng}(f)$, contradicting the assumption that f is on to I .

We began this subsection by noting that we could compare the sizes of finite sets by counting their elements, i.e. we can associate a number with each set (its size) and compare the numbers. We would like to be able to associate in this way some abstract “numbers” also with infinite sets. This should be done in such a way that equinumerous sets are of the same associated “number”. The abstract number associated with a set in this way is called the **cardinal number**, or the **cardinality**, of the set.

Definition 2.7.8 *With any set X we associate a cardinal number, denoted $|X|$, in such a way that $|X| = |Y|$ iff $X \approx Y$. Specifically:*

1. *For any finite set S the cardinality $|S|$ of S is its number of elements, i.e. the natural number n for which $S \approx \mathbb{N}_n$.*
2. $|\mathbb{N}| = \aleph_0$. Hence, the cardinality of every denumerably infinite set is \aleph_0 .
3. $|\mathbb{R}| = c$, where c is known as the **cardinality of the continuum**.

The natural numbers are the **finite cardinal numbers**; all other cardinals numbers are **infinite cardinal numbers**.

Therefore, for example, we have $\{a, b, c\} = 3$, $|\mathbb{N}_n| = n$ and $|\{0, 2, 4, 6, 8, 10, \dots\}| = \aleph_0$.

To be able to compare different cardinal numbers we impose the following ordering:

Definition 2.7.9 *Let λ and κ be cardinal numbers and K and L sets such that $|K| = \kappa$ and $|L| = \lambda$. Then we define*

$$\kappa \leq \lambda \text{ if and only if there exists an injection } f : K \rightarrow L.$$

We write $\kappa < \lambda$ if $\kappa \leq \lambda$ and $\kappa \not\approx \lambda$.

Proposition 2.7.10 *The relation \leq defined on cardinal numbers in Definition 2.7.9 is a well-defined linear order. Moreover, if $X \subseteq Y$, then $|X| \leq |Y|$.*

Proof: We only sketch an outline of the proof. The details are left as Exercise 12.

To prove that \leq is well defined on cardinal numbers, we need to show that it does not depend on particular choices of sets of the given cardinality. That is, we need to show that if $|K_1| = \kappa = |K_2|$ and $|L_1| = \lambda = |L_2|$, then there exists an injection $f : K_1 \rightarrow L_1$ if and only if there exists an injection $f : K_2 \rightarrow L_2$. This is an easy exercise.

Then, to show that \leq is of linear order we need to show the following:

- \leq is reflexive. For that it is enough to show that, for every set X , there exists an injection $f : X \rightarrow X$.
- \leq is transitive. For that it is enough to show that if there exist injections $f : X \rightarrow Y$ and $g : Y \rightarrow Z$, then there exists an injection $h : X \rightarrow Z$.

- \leq is antisymmetric. For that it is enough to show that if there exists injections $f : X \rightarrow Y$ and $g : Y \rightarrow X$, then $X \approx Y$. This is the claim of a well-known theorem in Set Theory, known as the *Cantor–Schröder–Bernstein Theorem*, the proof of which can be found in most advanced texts on Set Theory and will not be given here.
- Lastly, \leq is total, i.e. that for any sets X and Y either $|X| \leq |Y|$ or $|Y| \leq |X|$. In other words, we need to prove that either there exists an injection $f : X \rightarrow Y$ or an injection $g : Y \rightarrow X$. The proof of this requires the use of the Axiom of Choice. In fact, it can be shown that this assertion is equivalent to the Axiom of Choice. Again, a proof of this claim can be found in most advanced textbooks on Set Theory.

Showing that $|X| \leq |Y|$ when $X \subseteq Y$ is easy – see Exercise 12.

We are now in a position to compare the cardinal numbers we have encountered so far.

Theorem 2.7.11 (Comparing cardinal numbers)

1. $n \leq \aleph_0$ for every natural number $n \in \mathbb{N}$.
2. $\aleph_0 < c$.
3. \aleph_0 is the least infinite cardinal, i.e. $\aleph_0 \leq \kappa$ for all infinite cardinals κ .

Proof: We leave item (1) as Exercise 11. We omit the proof of item (3), which requires the use of the Axiom of Choice, and only prove (2). Since $\mathbb{N} \subseteq \mathbb{R}$ we know by Proposition 2.7.10 that $|\mathbb{N}| \leq |\mathbb{R}|$, i.e. $\aleph_0 \leq c$. To obtain the strict inequality it is therefore enough to know that $\aleph_0 \not\approx c$, which is exactly what Theorem 2.7.7 says.

2.7.2. Exercises

- ① Show that \mathbb{Z} is equinumerous to the set E of even integers.
- ② Determine the cardinalities of the following sets:
 - (a) $\{\alpha, \beta, \gamma, \delta, \epsilon\}$
 - (b) $\{x \in \mathbb{Z} \mid x^2 = 4\}$
 - (c) $\{x \in \mathbb{Q} \mid 3 \leq x \leq 4\}$
 - (d) $\{(x, y) \in \mathbb{R} \mid x^2 + y^2 = 3\}$
- ③ Show that the function f defined as

$$f(n) = \begin{cases} \frac{n}{2}, & n \text{ is even} \\ -\frac{n+1}{2}, & n \text{ is odd} \end{cases}$$

in Example 2.7.2 is a bijection between \mathbb{N} and \mathbb{Z} .

- ④ For any $n \in \mathbb{N}$, let S_n be the set of binary sequences of length n . Show that $\mathcal{P}(\mathbb{N}_n) \approx S_n$.
- ⑤ Show that if A and B are finite sets then $A \times B$ is finite.
- ⑥ Let \mathcal{X} be a set of sets. Show that the equinumerosity relation restricted to \mathcal{X} is an equivalence relation.

- ⑦ Show that if S is a countable set and $a \notin S$, then the set $S \cup \{a\}$ is also countable.
- ⑧ Show that if a set X is countable and $Y \subseteq X$, then Y is also countable. (Hint: it is sufficient to show that every subset of \mathbb{N} is countable.)
- ⑨ Let A and B be countable sets. Show that the following sets are countable:
- $A \cap B$
 - $A \cap B$
 - $A \times B$
- ⑩ For any $n \in \mathbb{N}$, let F_n be the set $\{f \mid f : \mathbb{N}_n \rightarrow \{0, 1\}\}$ of all functions from \mathbb{N}_n into the two-element set $\{0, 1\}$. Show that $\mathcal{P}(\mathbb{N}_n) \approx F_n$.
- ⑪** Prove that $\aleph_0 < \aleph_0$ for every natural number $n \in \mathbb{N}$.
- ⑫** With this exercise we fill in the details left out in the proof of Proposition 2.7.10.
- Show that if $|K_1| = \kappa = |K_2|$ and $|L_1| = \lambda = |L_2|$, then an injection $f : K_1 \rightarrow L_1$ exists if and only if an injection $f : K_2 \rightarrow L_2$ exists.
 - Show that for every set X there exists an injection $f : X \rightarrow X$. Now conclude that \leq is transitive on cardinal numbers.
 - Show that, for all sets X , Y and Z , if there exist injections $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ then there exists an injection $h : X \rightarrow Z$. Now conclude that \leq is transitive on cardinal numbers.
 - Show that if $X \subseteq Y$ then there exists an injection $f : X \rightarrow Y$. Now conclude that if $X \subseteq Y$ then $|X| \leq |Y|$.

The continuum hypothesis

We have seen that the cardinality of the set of natural numbers is strictly less than the cardinality of the set of real numbers, i.e.

$$|\mathbb{N}| = \aleph_0 < \mathfrak{c} = |\mathbb{R}|.$$

However, are there sets that have cardinality strictly in between \aleph_0 and \mathfrak{c} ? The statement that there are no such sets is known as the *Continuum Hypothesis* (CH). If \aleph_1 denotes the least cardinal number greater than \aleph_0 , then the CH asserts that $\aleph_0 = \mathfrak{c}$, i.e. that the first cardinal greater than \aleph_0 is equal to the cardinality of the continuum of real numbers, whence the name “continuum hypothesis”.

The problem to determine whether the CH is true or false was the first one on Hilbert’s (see the box at the end of Section 3.3 in Chapter 3) famous list of 23 open problems, published in 1900. *Hilbert’s Problems* exerted a very significant influence on the mathematics of the 20th century, and problem 1 was no exception. In 1940 Kurt Gödel (see the box at the end of Section 4.4 in Chapter 4) showed that the CH cannot be disproved within ZFC (see the box in Section 2.5). In 1963 Paul Cohen proved that the CH cannot be proven within ZFC either. For this work he was awarded the Fields Medal, which is the highest award in mathematics. The combined consequence of these two

results is that the CH is *independent* of ZFC, which means that our fundamental intuitions about sets, as captured by the axioms of ZCF, do not imply anything about the (non-)existence of sets with cardinality in between \aleph_0 and \mathfrak{c} .

How does one interpret this situation? Those who feel that mathematics is a human creation might say that this means that it is up to mathematicians to decide whether they want the CH to be true or false. In other words, we can decide whether we want sets with cardinalities strictly in between that of the naturals and that of the reals to exist or not. Depending on what we decide, we can simply add an axiom to our set theory that would make it so. On the other hand, those who feel that mathematical objects have an existence independent of human thought, often referred to as Platonists, might claim that the CH is either true or false, but we just do not know yet. Accordingly, they would claim that ZFC is simply too weak to capture all there is to know about the independently existing realm of sets.



2.8. Isomorphisms of ordered sets. Ordinal numbers

In the previous subsection we were comparing the sizes of sets and the associated numbers were cardinal numbers. Here we will be looking specifically at well-ordered sets and compare the “lengths” of their orderings. The first tool we will need for this endeavour is the notion of an order preserving map.

Definition 2.8.1 Let (X, \leq_X) and (Y, \leq_Y) be posets. An **order preserving map** between X and Y is a map $f : X \rightarrow Y$ that respects the order, i.e. for all $x_1, x_2 \in X$

$$x_1 \leq_X x_2 \text{ if and only if } f(x_1) \leq_Y f(x_2).$$

If, in addition, f is a bijection, it is called an **order isomorphism** between X and Y . If there is an order isomorphism between (X, \leq_X) and (Y, \leq_Y) we say that they are **order isomorphic** and write $(X, \leq_X) \cong (Y, \leq_Y)$, or simply $X \cong Y$ if the ordering relations are clear from the context.

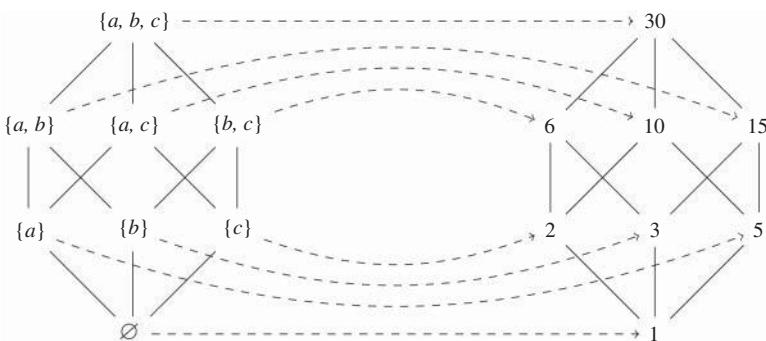
This definition has the following immediate consequence.

Proposition 2.8.2 Every order preserving map is an injection.

Proof: Let $f : X \rightarrow Y$ be an order preserving map between posets (X, \leq_X) and (Y, \leq_Y) . Suppose $f(x_1) = f(x_2)$. Therefore, $f(x_1) \leq_Y f(x_2)$. Since f is order-preserving, it follows that $x_1 \leq_X x_2$. Similarly, $f(x_2) \leq_Y f(x_1)$ and hence $x_2 \leq_X x_1$. We conclude that $x_1 = x_2$.

Example 2.8.3

1. The map $f : \mathbb{N} \rightarrow \mathbb{Z}$ which sends every natural number to itself in an order preserving map. It is not an order isomorphism, since it is not surjective.
2. The map $g : \mathbb{N} \rightarrow \mathbb{N}$ defined by $f : n \mapsto 2n$ is an order preserving map between \mathbb{N} and itself. Again, it is not an order isomorphism as it is not surjective.
3. Below are Hasse diagrams of two-order isomorphic posets. The action of the isomorphism is indicated by the dashed arrows. On the left is the poset consisting of the powerset of the three-element set $\{a, b, c\}$ with the inclusion relation \subseteq . The poset on the right consists of the set $\{1, 2, 3, 5, 6, 10, 15, 30\}$ with the divisibility relation.



Saying that two posets are isomorphic means that they are essentially the same poset: even though the elements might have different names, there are “equal numbers” of them in both posets and, as far as the ordering relation is concerned, they are arranged in exactly the same way. The word “isomorphic” means “to have the same shape”, and most kinds of mathematical structures have an associated notion of isomorphism: we talk of isomorphic first-order structures, isomorphic graphs, isomorphic groups and many more. If two mathematical structures are isomorphic, a mathematician would regard them as the same structure “up to isomorphism”.

The following proposition says that, up to isomorphism, there is only one well-ordered set of any given finite cardinality. For the proof, see Exercise 7.

Proposition 2.8.4 *If (X, \leq_X) and (Y, \leq_Y) are finite well-ordered sets such that $|X| = |Y|$, then $(X, \leq_X) \cong (Y, \leq_Y)$.*

To every well order X we want to assign a number $\text{ord}(X)$, called the ordinality of X , in such a way that $\text{ord}(X) = \text{ord}(Y)$ if and only if $X \cong Y$. Together with the ordinals we want to define a well-ordering relation in such a way that, for every set of ordinals, there is a least ordinal strictly greater than every ordinal in the set.

The least ordinal is the natural number 0. The next ordinal is the natural number 1, etc. In fact, all natural numbers are ordinals, and they are ordered by the usual ordering of the natural numbers. These are the **finite ordinals**. The least ordinal greater than all finite ordinals is called ω . It corresponds to the ordering of all natural numbers \mathbb{N} . The next ordinal is $\omega + 1$, then $\omega + 2$, $\omega + 3$ and so on. The least ordinal greater than all members of $\{\omega + n \mid n \in \mathbb{N}\}$ is $\omega + \omega$, written $\omega \cdot 2$. The next ordinal is $\omega \cdot 2 + 1$, then $\omega \cdot 2 + 2$ and so on. The least ordinal greater than all members of $\{\omega \cdot 2 + n \mid n \in \mathbb{N}\}$ is $\omega \cdot 3$. Now the least ordinal greater than all members of $\{\omega \cdot m + n \mid n, m \in \mathbb{N}\}$ is $\omega \cdot \omega = \omega^2$. The least ordinal greater than all ordinals in $\{\omega^2 \cdot n \mid n \in \mathbb{N}\}$ is ω^3 , and the least ordinal greater than all ordinals ω^n , $n \in \mathbb{N}$, is ω^ω . Summarizing:

$$\begin{aligned} 0 < 1 < 2 < \dots < \omega < \omega + 1 < \omega + 2 < \dots < \omega \cdot 2 < \omega \cdot 2 + 1 < \dots < \omega \cdot n < \omega \cdot n + 1 < \dots \\ < \omega^2 < \omega^2 + 1 < \dots < \omega^2 \cdot 2 < \omega^2 \cdot 2 + 1 < \dots < \omega^2 \cdot n < \omega^2 \cdot n + 1 < \dots < \omega^3 < \omega^3 + 1 < \dots \\ < \omega^n < \omega^n + 1 < \dots < \omega^\omega < \dots . \end{aligned}$$

Every ordinal α has a direct successor, namely $\alpha + 1$, which is the least ordinal strictly greater than it. Not every ordinal has a direct predecessor, however: the direct predecessor of 3 is 2 and the direct predecessor of $\omega + 1$ is ω , but ω has no direct predecessor. Ordinals with direct predecessors are called **successor ordinals** and those with predecessors but with no direct ones are **limit ordinals**. For example, all positive natural numbers are successor ordinals, while ordinals like ω , $\omega \cdot 2$, ω^5 and ω^ω are limit ordinals.

In fact, we can define all ordinals as special sets, so that every ordinal α is the well-ordered set $\{\beta \mid \beta < \alpha\}$ of all ordinals strictly less than α , as follows. Since there are no ordinals strictly less than 0, we declare the ordinal $\bar{0}$ to be identical with the empty set \emptyset . The only ordinal strictly less than 1 is 0, so the ordinal $\bar{1}$ is the set $\{0\} = \{\emptyset\}$. Similarly, the ordinal $\bar{2}$ is $\{0, 1\} = \{\emptyset, \{\emptyset\}\}$, $\bar{3}$ is $\{0, 1, 2\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$, and so forth. In general, for any natural number n , the set $\mathbb{N}_n = \{0, \dots, n - 1\}$ together with the usual ordering on natural numbers is the well-ordered set of ordinals less than n , i.e. it is the ordinal \bar{n} . Hereafter, we will identify every finite ordinal \bar{n} with the number n . Further, the ordinal ω is the same as the well-ordered set \mathbb{N} of all natural numbers; then $\omega + 1 = \{0, 1, 2, \dots, \omega\}$, etc.

Eventually, it can be shown that every well-ordered set is order isomorphic to a unique ordinal. We can therefore define the “ordinality” of a well-ordered set unambiguously.

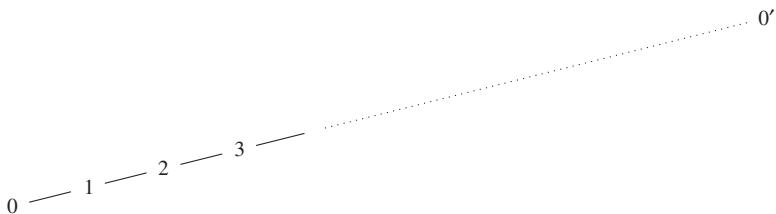
Definition 2.8.5 *The ordinality of a well ordered set X , denoted $\text{ord}(X)$, is the unique ordinal number α such that $\alpha \cong X$.*

In particular, $\text{ord}(\mathbb{N}_n) = n$ for every natural number n and $\text{ord}(\mathbb{N}) = \omega$.

Here are two more examples of ordinalities of well-ordered sets.

Example 2.8.6

Consider the well-ordered set $\mathbb{N}^* = \mathbb{N} \cup \{0'\}$ consisting of the natural numbers with their usual ordering together with the new element $0'$, which we declare to be greater than every natural number.



Let $f : \mathbb{N}^* \rightarrow \omega + 1$ be the map defined by

$$f(x) = \begin{cases} x & \text{if } x \in \mathbb{N} \\ \omega & \text{if } x = 0' \end{cases}.$$

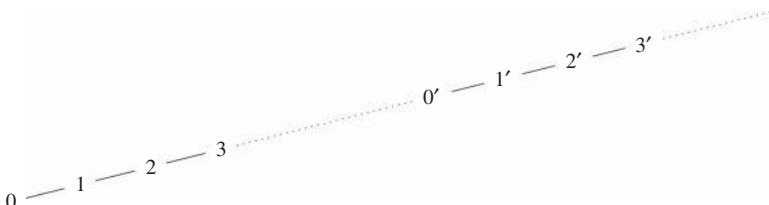
This is an order isomorphism between \mathbb{N}^* and $\omega + 1$. Thus $\text{ord}(\mathbb{N}^*) = \omega + 1$.

Example 2.8.7

Consider the well ordered set $A = \mathbb{N} \cup \mathbb{N}'$ where $\mathbb{N}' = \{n' \mid n \in \mathbb{N}\}$ together with the well-ordering relation \leq_A defined as $\leq_A =$:

$$\begin{aligned} &\{(n, m) \in \mathbb{N} \times \mathbb{N} \mid n \leq m\} \cup \\ &\{(n', m') \in \mathbb{N}' \times \mathbb{N}' \mid n \leq m\} \cup \\ &\{(n, m') \mid n \in \mathbb{N} \text{ and } m' \in \mathbb{N}'\}. \end{aligned}$$

In other words, A consists of two copies of the natural numbers, each with the usual ordering, but the second copy placed “after” the first such that each of its members is greater than each member of the first copy.



Let $g : A \rightarrow \omega \cdot 2$ be the map defined by

$$g(x) = \begin{cases} x & \text{if } x \in \mathbb{N} \\ \omega + n & \text{if } x = n' \end{cases}.$$

This is an order isomorphism between A and $\omega \cdot 2$ (see Exercise 4), so $\text{ord}(A) = \omega \cdot 2$.

2.8.1. Exercises

- ① Let $X = \{\emptyset, \{1\}, \{2\}, \{1, 3\}, \{1, 4\}, \{2, 5\}, \{2, 6\}\}$. Show that (X, \subseteq) is order isomorphic to the set of binary strings of length at most 4 ordered by the initial segment relation (see Exercise 2 from Section 2.6.5).
- ② Show that if two posets are order isomorphic then they are equinumerous.
- ③ Show that the map defined in Example 2.8.6 is an order isomorphism.
- ④ Show that the map defined in Example 2.8.7 is an order isomorphism.
- ⑤ Show that $\omega \cdot n$ is a countable set, for every $n \in \mathbb{N}$.
- ⑥ Let X be a nonempty set and let $2^X = \{f \mid f : X \rightarrow \{0, 1\}\}$ be the set of all maps from X into the set $\{0, 1\}$. Define a relation \leqq on 2^X such that $f_1 \leqq f_2$ iff $f_1(x) \leq f_2(x)$ for all $x \in X$.
 - (a) Show that $(2^X, \leqq)$ is a poset.
 - (b) Show that $(2^X, \leqq)$ is order-isomorphic to $(\mathcal{P}(X), \leq)$.
- ⑦** The aim of this exercise is to prove Proposition 2.8.4.
 - (a) Let (Z, \leq_Z) be a well order and z_0 the least element of Z . Prove that $(Z - \{z_0\}, \leq'_Z)$ is a well order, where \leq'_Z is the restriction of \leq_Z to $Z - \{z_0\}$.
 - (b) Let (Z, \leq_Z) be a well order and a an element not occurring in Z . Prove that $(Z \cup \{a\}, \leq_Z^+)$ is a well order, where $\leq_Z^+ = \leq_Z \cup \{(a, a)\} \cup \{(a, z) \mid z \in Z\}$. That is to say, $(Z \cup \{a\}, \leq_Z^+)$ is obtained from (Z, \leq_Z) by adding the new element a and declaring it to be the new least element.
 - (c) Let (X, \leq_X) and (Y, \leq_Y) be well orders, $a, b \notin X \cup Y$, and $(X \cup \{a\}, \leq_X^+)$ and $(Y \cup \{b\}, \leq_Y^+)$ obtained by adding a and b as respective new least elements of X and Y . Then prove that if $(X, \leq_X) \cong (Y, \leq_Y)$ then $(X \cup \{a\}, \leq_X^+) \cong (Y \cup \{b\}, \leq_Y^+)$.
 - (d) Now prove that if (X, \leq_X) and (Y, \leq_Y) are finite well-ordered sets such that $|X| = |Y|$, then $(X, \leq_X) \cong (Y, \leq_Y)$. Do this by induction in $n = |X|$ using the three claims above for the induction step.

Hilbert's Hotel

Here is a story that originally appeared in the 1947 book *One, Two, Three, ... Infinity* by the physicist George Gamow (1904–1968). In a certain city there is a hotel, called *Hilbert's Hotel*, with denumerably many rooms, numbered 0, 1, 2, 3, 4, One evening a weary traveler arrives at the front desk and asks for a room, only to be informed that the hotel is completely full. However, after a moment's thought the clerk says: "Wait a minute, Sir. I think we can still accommodate you, but it will take a bit of rearranging. I will ask every guest to move to the room with number one greater than the room he or she is currently occupying. That way room 0 will become vacant and we can put you in there."

So even though this hotel may be completely booked out, it can always accommodate one more guest, and then one more, and then another, and so on. Of course, this is possible because we can always define an injection from an infinite set to a proper subset of itself. In the example above the injection is $f : \mathbb{N} \rightarrow \mathbb{N}$, given by $f(n) = n + 1$, which increases the room number of each current guest by one and thus making room for the new guest in room 0, where no current guest is sent. In fact it is not essential that the new guest be placed in room 0: we could vacate room n for him by simply moving every one in a room with number n or higher up one room. Moreover, even if countably many new guests arrive at the same time after Hilbert's Hotel is fully occupied, the receptionist can still accommodate them, by asking the guest in room n to move to room $2n$ for every $n \in \mathbb{N}$, thus freeing all odd-numbered rooms.

This is an example of an apparently paradoxical situation, which arises when we try to think of infinite versions of finite physical objects, like hotels. Since all sets in the physical world are finite, our intuitions are often baffled by the infinite.



2.9. Application: relational databases

A relational database consists of a number of tables filled with data items. Here are three tables from the database of a veterinary clinic.

The rows in a table form n -tuples – 3-tuples, 5-tuples and 3-tuples, respectively, in the case of Tables 2.1, 2.2 and 2.3 – with the coordinates of the tuples coming from particular sets – in Table 2.1 from the sets of character strings, real numbers and the set {Y,N}, respectively. Looking at a table in this way suggests that it can be seen as an n -ary relation between sets. This is precisely what the **relational database model** does and what enables it to bring the mathematical theory of relations to bear on the problems of data manipulation and retrieval. As our tables show, apart from the n -tuples forming the rows of the table (and the elements in the relation) we also need names for the columns, which are referred to as the **attributes** of the table.

Table 2.1 Clients

Client Name	Account balance	Payments-up-to-date
N Bates	-45.93	N
L du Plessis	+0.12	Y
V Tshabalala	-76.01	Y
A Winterbottom	-235.54	N

Table 2.2 Pets

Pet Name	Type	Breed	Owner	Vet Name
Fluffy	Dog	Maltese	L du Plessis	Dr Dolittle
Nefertiti	Cat	Sphinx	A Winterbottom	Dr Dolittle
Mopsy	Dog	Cross breed	L du Plessis	Dr Yang
Tiger	Cat	Domestic shorthair	A Winterbottom	Dr Dolittle
Butch	Dog	Bulldog	V Tshabalala	Dr Yang
Montgomery	Cat	Domestic longhair	N Bates	Dr Khumba

Table 2.3 Vets

Vet Name	Staff ID	Day off
Dr Dolittle	A64B8	Sat
Dr Yang	A57C6	Mon
Dr Khumba	A55B9	Sun

Definition 2.9.1 An *n*-ary **relation schema** $\mathbf{R}(A_1, A_2, \dots, A_n)$ is an *n*-tuple (A_1, A_2, \dots, A_n) of attributes each with an associated **data type**, denoted $\text{DT}(A_i)$. Here attributes are simply names and the data type $\text{DT}(A_i)$ is a set.

An instance of an *n*-ary relation schema $\mathbf{R}(A_1, A_2, \dots, A_n)$ is an *n*-ary relation

$$R \subseteq \text{DT}(A_1) \times \text{DT}(A_2) \times \dots \times \text{DT}(A_n).$$

In other words, an instance of $R(A_1, A_2, \dots, A_n)$ is a relation with matching arity (*n*-ary) and where each coordinate from each *n*-tuple in the relation is of the right data type.

A **relational database schema** \mathcal{D} is a set of relation schemas and an instance of \mathcal{D} is a set consisting of an instance of each of the relation schemas in \mathcal{D} .

Consider the relation schemas:

- CLIENT(Client Name, Account balance, Payments-up-to-date) with
 - $\text{DT}(\text{Client Name}) = \text{STRING}$;
 - $\text{DT}(\text{Account balance}) = \mathbb{R}$;

- $\text{DT}(\text{Payments-up-to-date}) = \{Y, N\}$.
- $\text{PET}(\text{Pet Name}, \text{Type}, \text{Breed}, \text{Owner}, \text{Vet Name})$ with
 - $\text{DT}(\text{Pet Name}) = \text{STRING}$;
 - $\text{DT}(\text{Type}) = \{\text{Cat}, \text{Dog}, \text{Bird}, \text{Reptile}, \text{Rodent}\}$;
 - $\text{DT}(\text{Breed}) = \text{STRING}$;
 - $\text{DT}(\text{Owner}) = \text{STRING}$;
 - $\text{DT}(\text{Vet Name}) = \text{STRING}$.
- $\text{VET}(\text{Vet Name}, \text{Staff ID}, \text{Day off})$ with
 - $\text{DT}(\text{Vet Name}) = \text{STRING}$;
 - $\text{DT}(\text{Staff ID}) = \text{STRING}$;
 - $\text{DT}(\text{Name}) = \text{DAY} = \{\text{Mon}, \text{Tue}, \text{Wen}, \text{Thu}, \text{Fri}, \text{Sat}, \text{Sun}\}$.

The relations given by Tables 2.1, 2.2 and 2.3 are instances of the relation schemas CLIENT, PET and VET, respectively. These three tables, taken together, are an instance of the relational database schema {CLIENT, PET, VET}.

Relational databases are not simply stores of information: their true usefulness lies in the queries that can be made on them. For example, we might want to retrieve the account balances of all clients who have pets treated by Dr Yang, or retrieve a list of all vets who treat cats and who have their day off on Wednesday. These queries may be answered by performing appropriate set theoretic operations on the relations (=tables). The study of operations on relations dates back to the work of mathematicians like Augustus De Morgan (1806–1871), Charles Peirce (1839–1914) and Ernst Schröder (1841–1902). There is a standard set of such operations used for database queries, known as the relational calculus, which was introduced by Edgar F. Codd in the 1970s.

Intersection of relations

Let R and S be n -ary relations that are instances of the same relation schema. Then we can form their set-theoretic intersection, namely

$$R \cap S = \{(a_1, \dots, a_n) \mid (a_1, \dots, a_n) \in R \text{ and } (a_1, \dots, a_n) \in S\}.$$

Difference of relations

Let R and S be n -ary relations that are instances of the same relation schema. Then we can form their set-theoretic difference, namely

$$R - S = \{(a_1, \dots, a_n) \mid (a_1, \dots, a_n) \in R \text{ and } (a_1, \dots, a_n) \notin S\}.$$

Flattened Cartesian product

Let R be an n -ary and S an m -ary relation. Then we can form their flattened Cartesian product, namely the $(n + m)$ -ary relation

$$R \times S = \{(a_1, \dots, a_n, b_1, \dots, b_m) \mid (a_1, \dots, a_n) \in R \text{ and } (b_1, \dots, b_m) \in S\}.$$

Note that this is different from the ordinary Cartesian product, which would yield a set of ordered pairs

$$\{((a_1, \dots, a_n), (b_1, \dots, b_m)) \mid (a_1, \dots, a_n) \in R \text{ and } (b_1, \dots, b_m) \in S\}$$

instead of a set of $(n + m)$ -tuples.

Projection

Let R be an n -ary relation that is an instance of a relation schema $\mathbf{R}(A_1, A_2, \dots, A_n)$. If $\{A_{i_1}, \dots, A_{i_m}\} \subseteq \{A_1, A_2, \dots, A_n\}$ we can form the **projection** of R on to coordinates $(A_{i_1}, \dots, A_{i_m})$, namely

$$\begin{aligned} \pi_{A_{i_1}, \dots, A_{i_m}}(R) \\ = \{(a_1, \dots, a_m) \mid \text{there exists } (b_1, \dots, b_n) \in R \text{ such that } a_1 = b_{i_1}, \dots, a_m = b_{i_m}\}. \end{aligned}$$

In other words, the projection function forms a new relation by keeping only the indicated coordinates from the tuples in the original relation, rearranged in the order indicated. For example, if $Pets$ is the relation defined in Table 2.2, then the projection $\pi_{\text{Owner}, \text{Pet Name}}(Pets)$ is the relation given in Table 2.4.

Selection

The selection operation allows one to select the elements of a relation (= rows in a table) that satisfy a certain condition. The conditions are expressions built in the following way:

Atomic conditions (1). If A_1 and A_2 are attributes of the same data type and c is a constant of this data type, then $A_1 = A_2$, $A_1 \neq A_2$, $A_1 = c$ and $A_1 \neq c$ are atomic conditions.

Table 2.4 $\pi_{\text{Owner}, \text{Pet Name}}(Pet)$

Owner	Pet Name
L du Plessis	Fluffy
A Winterbottom	Nefertiti
L du Plessis	Mopsy
A Winterbottom	Tiger
V Tshabalala	Butch
N Bates	Montgomery

Table 2.5 $\Theta_{\text{Vet}=\text{DrDolittle} \wedge \text{Type}=\text{Cat}}(\text{Pets})$

Pet Name	Type	Breed	Owner	Vet
Nefertiti	Cat	Sphinx	A Winterbottom	Dr Dolittle
Tiger	Cat	Domestic shorthair	A Winterbottom	Dr Dolittle

Table 2.6 $\Theta_{\text{Accountbalance} < -100 \wedge \text{Payments-up-to-date} = N}(\text{Clients})$

Client Name	Account balance	Payments-up-to-date
A Winterbottom	-235.54	N

Atomic conditions (2). If A_1 and A_2 are attributes of the same data type, c is a constant of the data type and \leq is a linear order on the elements of this data type, then $A_1 \leq A_2$, $A_1 < A_2$, $A_1 \not\leq A_2$, $A_1 \leq c$, $A_1 < c$, $A_1 \not\leq c$, $c \leq A_2$, $c < A_2$, $c \not\leq A_2$ are atomic conditions.

Compound conditions. If ϕ_1 and ϕ_2 are conditions, then $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$ are conditions.

Here are some conditions using attributes from our running veterinary example: $\text{Vet} = \text{DrDolittle} \vee \text{Vet} = \text{DrYang}$, $\text{Vet} = \text{DrDolittle} \wedge \text{Type} = \text{Cat}$, $\text{Day off} \neq \text{Saturday}$ and $\text{Accountbalance} < -100 \wedge \text{Payments-up-to-date} = N$.

The selection operation applies conditions to relations (= tables) and returns the elements (= rows) that satisfy the condition. To be precise, if ϕ is a condition and R a relation, then

$$\Theta_\phi(R) = \{(a_1, \dots, a_n) \in R \mid (a_1, \dots, a_n) \text{ satisfies } \phi\}.$$

For example, the results of selection $\Theta_{\text{Vet}=\text{DrDolittle} \wedge \text{Type}=\text{Cat}}(\text{Pets})$ and $\Theta_{\text{Accountbalance} < -100 \wedge \text{Payments-up-to-date} = N}(\text{Clients})$ are given in Tables 2.5 and 2.6, respectively.

Definition 2.9.2 Relational algebra consists of the following five operations on relations: union, difference, flattened Cartesian product, projection and selection.

As we have seen, the operations of relational algebra can be used to answer queries on databases. However, the real power of these operations lies in their composition. Suppose we want to retrieve a list of all pets with the days of the week on which their vets are off, needed for scheduling appointments. Here is what we do: since this query asks for information that is spread over two tables, namely Pets and Vets, we first need to bring this information into one table. The only operation of relational algebra that can do this for us is the flattened Cartesian product. Accordingly we form

$$\text{Pets} \times \text{Vets}.$$

This table will contain rows in which the values of the attribute Vet Name coming from table Pets does not match with the value of the attribute Vet Name coming from table Vets. To eliminate such rows we apply the selection operation as follows:

$$\Theta_{\text{Vets.Vet Name} = \text{Pets.Vet Name}}(\text{Pets} \times \text{Vets}),$$

Table 2.7 $\Theta_{Vets.Vet\ Name=Pets.Vet\ Name}(Pets \times Vets)$

Pet Name	Type	Breed	Owner	Pets.Vet Name	Vets.Vet Name	Staff ID	Day off
Fluffy	Dog	Maltese	L du Plessis	Dr Dolittle	Dr Dolittle	A64B8	Sat
Nefertiti	Cat	Sphinx	A Winterbottom	Dr Dolittle	Dr Dolittle	A64B8	Sat
Mopsy	Dog	Cross breed	L du Plessis	Dr Yang	Dr Yang	A57C6	Mon
Tiger	Cat	Domestic shorthair	A Winterbottom	Dr Dolittle	Dr Dolittle	A64B8	Sat
Butch	Dog	Bulldog	V Tshabalala	Dr Yang	Dr Yang	A57C6	Mon
Montgomery	Cat	Domestic longhair	N Bates		Dr Khumba	A55B9	Sun

Table 2.8 $Pets \bowtie Vets$

Pet Name	Type	Breed	Owner	Vet Name	Staff ID	Day off
Fluffy	Dog	Maltese	L du Plessis	Dr Dolittle	A64B8	Sat
Nefertiti	Cat	Sphinx	A Winterbottom	Dr Dolittle	A64B8	Sat
Mopsy	Dog	Cross breed	L du Plessis	Dr Yang	A57C6	Mon
Tiger	Cat	Domestic shorthair	A Winterbottom	Dr Dolittle	A64B8	Sat
Butch	Dog	Bulldog	V Tshabalala	Dr Yang	A57C6	Mon
Montgomery	Cat	Domestic longhair	N Bates	Dr Khumba	A55B9	Sun

where $Vets.Vet\ Name$ and $Pets.Vet\ Name$ is the Vet Name attributes coming from the $Vets$ and $Pets$ tables, respectively. The resulting table (Table 2.7) will still contain some redundancy, since the name of the vet will be listed twice in each row.

To eliminate redundancy, we can apply the following instance of the projection function:

$$\pi_{Pet\ Name, Pet\ Type, Pet\ Breed, Pet\ Owner, Pet\ .Vet\ Name, Vets.\ Staff\ ID, Vets.\ Day\ off}$$

which keeps every column in $\Theta_{Vets.Vet\ Name=Pets.Vet\ Name}(Pets \times Vets)$ except the copy of Vet Name, which comes from the $Vets$ table. The result is given in Table 2.8. The operation we just performed on $Pets$ and $Vets$ is a very commonly used one and is known as the natural join. Here is the formal definition:

Natural join

Let R and S be two relations which share the attributes A_1, \dots, A_k . Then the **natural join** of R and S is given by

$$R \bowtie S = \pi_{\langle \text{list} \rangle}(\Theta_{R.A_1=S.A_1, \dots, R.A_k=S.A_k}(R \times S))$$

where $\langle \text{list} \rangle$ is the list of all attributes of R or S , other than $S.A_1, \dots, S.A_k$. The result of applying the natural join to $Pets$ and $Vets$ is given in table 2.8.

2.9.1. Exercises

❶ Consider Tables 2.9–2.12 from the database of a university. Compute the following tables:

- (a) $\text{Enrollment2014} \cap \text{Enrollment2013}$
- (b) $\text{Enrollment2014} - \text{Enrollment2013}$
- (c) $\text{Student} \times \text{Lecturer}$
- (d) $\pi_{\text{Name}, \text{StudyYear}}(\text{Student})$
- (e) $\Theta_{1 < \text{StudyYear}}(\text{Student})$
- (f) $\text{Student} \bowtie \text{Enrollment2014}$
- (g) $\pi_{\text{Name}, \text{Course}}(\text{Student} \bowtie \text{Enrollment2014})$

❷ Write down relational algebra queries on Tables 2.9–2.12 that would produce:

- (a) The student ID of every student that was enrolled for Psychology both in 2013 and 2014.
- (b) The student IDs of all students enrolled for Algebra in 2013.
- (c) All the lecturers of courses for which a second year student was enrolled for in 2014.
- (d) The names of all lecturers of S. Nkosi in 2014.

❸ Write down relational algebra queries on Tables 2.1–2.3 that would produce:

- (a) The names of all dogs treated by Dr Yang.
- (b) The name and owner's name of every dog in the database.
- (c) The names of all dogs who are treated by a vet who is off on Sundays.
- (d) The names of all clients who have pets treated by a vet who is off on Sundays.

Table 2.9 *Student*

Name	Student ID	Study Year
W Adams	186753	2
S Nkosi	268356	1
T Ling	647537	3
U de Klerk	723934	1

Table 2.10 *Enrollment2013*

Student ID	Course
647537	Psychology
647537	AI
268356	French
186753	Algebra
723934	Algebra

Table 2.11 Enrollment2014

Student ID	Course
268356	Algebra
647537	Psychology
723934	Algebra
268356	Chemistry
186753	Psychology
647537	French

Table 2.12 Lecturer

Lecturer Name	Course
W Morton	Algebra
F Jung	Psychology
D Shkatov	AI
B Mamba	Chemistry
J Southey	French
Y Jacobs	Calculus

Table 2.13 Lecturer \bowtie Enrollment2014

Lecturer Name	Course
W Morton	Algebra
F Jung	Psychology
B Mamba	Chemistry
J Southey	French

- ④ An often used operation on relations is the semi-join, denoted $R \bowtie S$. The semi-join is similar to the natural join. It reruns those tuples in R for which there is a tuple in S such that the shared attributes match. The result of applying the semi-join to *Lecturer* and *Enrollment2014* is given in Table 2.13.

Like the natural join, the semi-join can be expressed in terms of the operations of relational algebra.

- (a*) Express the semi-join operation in terms of the operations of relational algebra.
- (b) Calculate *Lecturer* \bowtie *Enrollment2013*.
- (c) Show that $R \bowtie S = (R \bowtie S) \bowtie S$.

- ⑤ Another operation on relations useful for answering queries is the **division operation**. Let R and S be relations such that the attributes of S form a strict subset of the attributes of R . Then $R \div S$ consists of the restriction of tuples in R to attributes only occurring in R and such that their combination with every tuple in S yields a tuple in R . To be specific, suppose that

R is an n -ary relation that is an instance of the relation schema $\mathbf{R}(A_1, \dots, A_{n-m}, B_1, \dots, B_m)$ and that S is an m -ary relation that is an instance of the relation schema $\mathbf{S}(B_1, \dots, B_m)$. Then

$$R \div S = \{(a_1, \dots, a_{n-m}) \mid (a_1, \dots, a_{n-1}, b_1, \dots, b_m) \in R \text{ for all } (b_1, \dots, b_m) \in S\}.$$

- (a) Calculate:

$$\text{Student} \bowtie \text{Enrollment2014} \div \Theta_{\text{Course}=\text{Algebra} \vee \text{Course}=\text{Chemistry}} \text{Enrollment2014}.$$

- (b) What query is answered by the calculation in 5(a)?
- (c) Write a query in relation algebra, also using \div and any other operations we have introduced so far, which would return the details of all clients who have all their pets treated by Dr Dolittle.
- (d*) Express the division operation in terms of the operations of relational algebra.



Charles Sanders Peirce (1839–1914)

Charles Peirce was an American universal scientist, philosopher, logician and mathematician. He made pioneering contributions to early mathematical logic and the algebra of relations, philosophy, scientific methodology and many other fields. He is regarded as a founding father of *pragmatism*, a philosophical theory and method of scientific research, and of *semiotics*, the study of the meaning of symbols.

Peirce made many important scientific contributions in an extremely wide range of disciplines – too many to mention here. While he was trained and worked as a chemist and geodesist, he regarded himself mainly as a logician, philosopher and mathematician. His pioneering work on the algebraic and logical study of relations, starting with his 1870 paper *Logic of Relatives*, laid the foundations for the relational model of databases developed by Edgar F. Codd, a scientific descendant of Peirce. He further developed Boole's work on algebraic logic, where he showed that a single binary logical operator (logical NOR, now also called Peirce's arrow) suffices to express all other Boolean connectives. He made contributions to cardinal arithmetic for infinite numbers, years before Cantor's first attempts, to the theory of infinitesimals and to the mathematics of continua. He introduced first-order and second-order quantification, independent from Frege and Peano. He also did important work in many other branches of pure and applied mathematics, and was one of the founders of modern statistics. Notably, already in 1886 – more than 50 years before the first digital computers were constructed – he showed how Boolean logical operations could be carried out by electrical switching circuits.

While Russell and Whitehead did not yet know of his work when they published *Principia Mathematica* in 1910 and did not mention him there, in 1959 Russell wrote: "Beyond doubt [...] he was one of the most original minds of the later nineteenth century, and certainly the greatest American thinker ever." In 1934, the philosopher Paul Weiss called Peirce "the most original and versatile of American philosophers and America's greatest logician" and Karl Popper viewed him as "one of the greatest philosophers of all time".

3

Propositional Logic

3.1. Propositions, logical connectives, truth tables, tautologies	90
3.2. Propositional logical consequence. Valid and invalid propositional inferences . .	101
3.3. The concept and use of deductive systems	109
3.4. Semantic tableaux	113
3.5. Logical equivalences. Negating propositional formulae	121
3.6. Normal forms. Propositional resolution	126

Logic studies and guides our reasoning – more precisely, *logically correct reasoning*. It does so by providing formal and unambiguous *logical languages*, where all statements are expressed by means of *logical formulae*. Formulae have precise *syntax* and grammar and their meaning is given by a precise *semantics*, based on a formally defined notion of *truth*. The fundamental notions in logic that capture the notion of logically correct reasoning are the notions of *logical consequence* and, in particular, *logical validity*. These can be “computed” by purely mechanical procedures called *deductive systems*.

There are two levels of logical expressions and reasoning in classical logic. The lower level is *propositional logic*, which we will discuss in this chapter, and the higher level is *first-order logic*, also known as *predicate logic*, which is the topic of the next chapter.

Propositional logic is about reasoning with *propositions*. These are sentences built, using *propositional logical connectives*, from primitive statements called *atomic propositions*, which can be assigned a *truth value*: *true* or *false*. These truth values propagate over all propositions through *truth tables* for the propositional connectives.

In the chapter we explain how to compute the truth value of propositions and how to reason correctly using propositions, by introducing the notion of *propositional logical consequence*. We then present several *deductive systems* used for deriving logical consequences in

propositional logic and show how they can be used for checking the logical correctness of propositional reasoning.

3.1. Propositions, logical connectives, truth tables, tautologies

3.1.1. Propositions and propositional connectives. Truth tables

Recall from Chapter 1 that a proposition is a sentence that can be assigned a **truth value**: true denoted by `T` or false denoted by `F`. Propositions can be either primitive (atomic) or built from primitive propositions by using propositional logical connectives, the most common of which are:

- **not**, called **negation**, denoted by \neg ;
- **and**, called **conjunction**, denoted by \wedge (or, sometimes, by $\&$);
- **or**, called **disjunction**, denoted by \vee ;
- **if ... then ...**, called **implication**, or **conditional**, denoted by \rightarrow ;
- **... if and only if ...**, called **biconditional**, denoted by \leftrightarrow .

These logical connectives are also called **Boolean connectives**, after the famous mathematician and logician, George Boole.

The propositional connectives act as functions on the truth values, determined by their truth tables:

p	$\neg p$	p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
T	F	T	T	T	T	T	T
F	T	F	F	F	T	F	F

The truth value of a compound proposition is then computed by replacing the atomic propositions occurring in it by their truth values and then doing the formal calculations following these truth tables.

3.1.2. Some remarks on the meaning of the connectives

The use and meaning of the logical connectives in natural language does not always match their formal logical meaning. For instance, quite often the conjunction is loaded with a causal relationship that makes the common sense meanings of the sentences “He threw the stone and the window broke” and “The window broke and he threw the stone” quite different, while they have the same truth value, by the truth table of the conjunction. Also, the disjunction is often used in an *exclusive* sense, e.g. in “I shall win or I shall die”, while in formal logic we use it by convention in an *inclusive* sense, so “You will win or I will win” will be true if we both win.

Amongst all logical connectives, however, the implication seems to be the most questionable one. For instance, it is not so easy to accept that a proposition such as “If $2+2=5$, then the Moon is made of cheese”, if it makes any sense at all, should be true. The leading motivation

in determining the truth behaviour of the implication is, of course, the logical meaning we assign to it. The proposition $A \rightarrow B$ means:

If A is true, then B must be true.

Note that if A is not true, then the implication $A \rightarrow B$ requires *nothing* regarding the truth of B . Therefore, there is *only one case where that proposition should be regarded as false*, namely when A is true, and yet B is not true. In all other cases we have no reason to consider it false. Hence, for it to be a proposition, it must be regarded as true. This argument justifies the truth table of the implication. It is very important to understand this particular truth table, because the implication is the logical connective that is most closely related to the concepts of logical reasoning and deduction.

Remark 2 One can also think of an implication as a promise, e.g. Johnny's father tells him: "If you pass your logic exam, then I'll buy you a motorbike." Now consider the four possible situations (Johnny passes/fails his exam and his father buys/does not buy him a motorbike) and see in which of them the promise is kept (the implication is true) and in which it is broken (the implication is false).

Some terminology: the proposition A in the implication $A \rightarrow B$ is called the **antecedent** and the proposition B is the **consequent** of the implication.

The implication $A \rightarrow B$ can be expressed in many different but logically equivalent ways, which one should be able to recognize:

- A implies B
- If A (then) B .
- A only if B .
- B if A .
- B whenever A .
- A is sufficient for B . (Meaning: the truth of A is sufficient for the truth of B .)
- B is necessary for A . (Meaning: the truth of B is necessary for A to be true.)

3.1.3. Propositional formulae

If we only discuss particular propositions, our study of logic would be no more useful than a study of algebra based on particular equalities such as $2 + 3 = 5$ or $12345679 \times 9 = 111111111$. Instead, we should look at *schemes of propositions* and their properties, like we study algebraic formulae and equations and their properties. To begin with, we use **propositional constants**: T , which always takes a truth value `true`, and \perp , which always takes a value `false`. We also use **propositional variables** (also called **Boolean variables**) p, q, r, \dots , possibly indexed, to denote unspecified propositions, just in the same way as we use algebraic variables to denote unspecified or unknown numbers. Using propositional constants and variables, the logical connectives and parentheses whenever necessary, we can form **propositional formulae** in the way in which we are used to forming algebraic expressions from variables and arithmetic

operations. Here are a few examples of propositional formulae:

$$\top, p, \neg\perp, \neg\neg p, p \vee \neg q, p_1 \wedge \neg(p_2 \rightarrow (\neg p_1 \wedge \perp)).$$

Note that there are infinitely many possible propositional formulae, so we cannot list them all here. However, there is a simple and elegant way to give a precise definition of propositional formulae, viz. the so-called **inductive definition** (or **recursive definition**). It consists of the following clauses (**formation rules**):

1. Every propositional constant or variable is a propositional formula.
2. If A is a propositional formula then $\neg A$ is a propositional formula.
3. If A, B are propositional formulae then $(A \vee B)$, $(A \wedge B)$, $(A \rightarrow B)$ and $(A \leftrightarrow B)$ are propositional formulae.

Hereafter we will be omitting the “unnecessary” pairs of parentheses, whenever that would not lead to syntactic ambiguity.

Propositional formulae are also called **Boolean formulae**. Note that the notion of propositional formula is used in its own definition; this is the idea of *recursion*. The definition works as follows: the first rule above gives us some initial stock of propositional formulae; as we keep applying the other rules, we construct more and more formulae and use them further in the definition. Eventually, every propositional formula can be obtained in several (finitely many!) steps of applying these rules. Thus, one can think of the definition above as a “construction manual” prescribing how new objects (here, propositional formulae) can be built from already constructed ones.

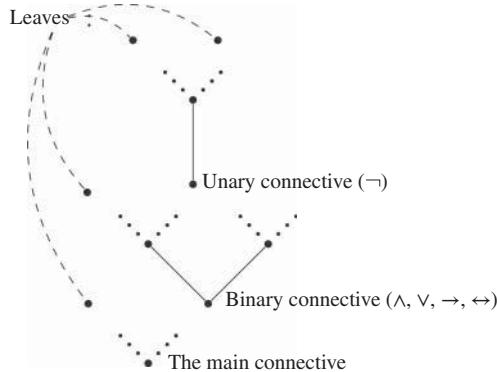
The formulae that occur in the process of the construction of a formula A are called **subformulae** of A . The last propositional connective introduced in the construction of A is called the **main connective of A** and the formula(e) to which it is applied is/are **the main subformula(e) of A** . We will make all these more precise in what follows.

3.1.4. Construction and parsing tree of a propositional formula

The sequence of formulae constructed in the process of applying the definition and ending with A is called a **construction sequence of the formula A** . A formula has many construction sequences and a construction sequence may contain many redundant formulae. A better notion capturing the construction of a formula is the **construction tree** of that formula. A construction tree is a tree-like directed graph with nodes labelled with propositional constants, variables and propositional connectives, such that:

1. Every leaf is labelled by a propositional constant or variable.
2. Propositional constants and variables label only leaves.
3. Every node labelled with \neg has exactly one successor node.
4. Every node labelled with any of \wedge , \vee , \rightarrow or \leftrightarrow has exactly two successor nodes – a *left* successor and a *right* successor.

Thus, a construction tree looks like this:



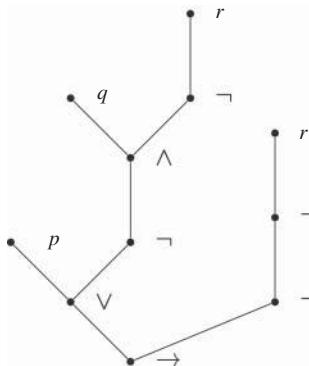
Every construction tree defines a formula C , built starting from the leaves and going towards the root, by applying at every node the formula construction rule corresponding to the label at that node. The formulae constructed in the process are precisely the subformulae of C and the propositional connective labelling the root of the construction tree of a formula C is the main connective of C .

Example 3.1.1 (Construction tree)

Formula:

$$(p \vee \neg(q \wedge \neg r)) \rightarrow \neg \neg r.$$

Construction tree:



The **parsing tree** of a formula looks the same as its construction tree, but is produced in inverse order, starting from the main connective (if any), drawing edges to the main components and then recursively producing the parsing trees for each of them.

3.1.5. Truth tables of propositional formulae

One can think of a propositional formula as a scheme that becomes a proposition whenever we substitute propositions for all occurring propositional variables. Of course, we mean *uniform*

substitutions; that is, the same variables are replaced by the same propositions throughout the formula. We cannot assign a truth value to a propositional formula before we assign concrete propositions, or at least concrete *truth values*, to all occurring propositional variables, for the same reason that we cannot evaluate $x(y + z)$ before we know the values of x, y, z . Remember, in order to evaluate the truth value of a compound proposition, we only need to know the truth values of the occurring atomic propositions, and not of the propositions themselves.

For instance, if we substitute “0.5 is an integer” for p , “2 is less than 3” for q and “The Moon is larger than the Sun” for r in the propositional formula

$$(p \vee \neg(q \wedge \neg r)) \rightarrow \neg\neg r$$

we find that the resulting proposition is true:

$$\begin{aligned} (F \vee \neg(T \wedge \neg F)) \rightarrow \neg\neg F &= (F \vee \neg(T \wedge T)) \rightarrow \neg T = (F \vee \neg T) \rightarrow F \\ &= (F \vee F) \rightarrow F = F \rightarrow F = T. \end{aligned}$$

If, however, we substitute any true propositions for p and q and a false proposition for r , then the resulting proposition will be false:

$$\begin{aligned} (T \vee \neg(T \wedge \neg F)) \rightarrow \neg\neg F &= (T \vee \neg(T \wedge T)) \rightarrow \neg T = (T \vee \neg T) \rightarrow F \\ &= (T \vee F) \rightarrow F = T \rightarrow F = F. \end{aligned}$$

We can tabulate the “truth behaviour” of any propositional formula in a truth table where we list all possible combinations of truth values of the occurring variables, and for each combination we calculate the corresponding truth value of the formula. We can do that by successively calculating the truth values of all occurring subformulae, as we did just now. For example, the truth table for the formula above is compiled as follows:

p	q	r	$\neg r$	$\neg\neg r$	$q \wedge \neg r$	$\neg(q \wedge \neg r)$	$p \vee \neg(q \wedge \neg r)$	$(p \vee \neg(q \wedge \neg r)) \rightarrow \neg\neg r$
T	T	T	F	T	F	T	T	T
T	T	F	T	F	T	F	T	F
T	F	T	F	T	F	T	T	T
T	F	F	T	F	F	T	T	F
F	T	T	F	T	F	T	T	T
F	T	F	T	F	T	F	F	T
F	F	T						
F	F	F						

The last two rows are left to the reader as exercises.

Truth tables can be somewhat simplified if we notice that every occurrence of a logical connective in a propositional formula determines a *unique subformula* where that occurrence is the main connective, i.e. the connective applied last in the construction of the subformula. For instance, the conjunction occurring in the formula $(p \vee \neg(q \wedge \neg r)) \rightarrow \neg\neg r$ is the main connective of the subformula $q \wedge \neg r$, the first negation is the main connective of $\neg(q \wedge \neg r)$ and the implication is the main connective of the whole formula. Now we can simplify the truth table of the formula by listing only the occurring variables and the whole formula and by calculating the

truth table of every subformula in the column below the corresponding main connective:

p	q	r	$(p \vee \neg q) \wedge (\neg r)$	\rightarrow	$\neg p$	$\neg q$	$\neg r$
T	T	T	T T T T F F T	T	T F T		
T	T	F	T T F T T T F	F	F T F		
T	F	T	T T T F F F T	T	T F T		
T	F	F	T T T F F T F	F	F T F		
F	T	T	F T T F F T T	T	T F T		
F	T	F	F F F T T T F	T	F T F		
F	F	T		...			
F	F	F		...			

Thus, we see that a propositional formula can represent a true or a false proposition, depending of the choice of the occurring atomic propositions, or rather on their truth values.

3.1.6. Tautologies

Definition 3.1.2 A **tautology** is a propositional formula that obtains a truth value `true` for any assignment of truth values to the occurring variables. Tautologies are also called **logically valid formulae**.

Thus, every tautology always renders a true proposition. So, it represents a **logical law** in the same way that the identity $x + y = y + x$ represents a law of arithmetic and, therefore, holds no matter what values we assign to x and y .

Here are a few simple tautologies that represent some important laws of propositional logic:

- $(p \vee \neg p)$: the **law of excluded middle** – every proposition p is either true or false (and in the latter case $\neg p$ must be true);
- $\neg(p \wedge \neg p)$: the **law of non-contradiction** – it cannot be the case that both p and $\neg p$ are true;
- $((p \wedge (p \rightarrow q)) \rightarrow q)$: if p is true and it is true that p implies q , then q is true.

How can one determine if a propositional formula A is a tautology? Quite easily: complete the truth table of A and check if it always takes a truth value `true`. For example, let us check the formulae mentioned above:

p	$\neg p$	$(p \vee \neg p)$	$(p \wedge \neg p)$	$\neg(p \wedge \neg p)$
T	F	T	F	T
F	T	T	F	T

p	q	$(p \rightarrow q)$	$(p \wedge (p \rightarrow q))$	$((p \wedge (p \rightarrow q)) \rightarrow q)$
T	T	T	T	T
T	F	F	F	T
F	T	F	F	T
F	F	T	F	T

The opposite concept of a tautology is a **contradictory formula** – a formula that always takes a truth value `false`. For instance, $(p \wedge \neg p)$ is a contradictory formula.

Question 1 How are the concepts of tautology and contradictory formula related?

Definition 3.1.3 A propositional formula that obtains a truth value `true` for some assignment of truth values to the variables is called **satisfiable**.

Example 3.1.4

The formula $p \wedge \neg q \wedge r$ is satisfied by the assignment $p : T$, $q : F$, $r : T$. The formula $p \wedge \neg p$ is not satisfiable.

Question 2 How are the concepts of tautology and satisfiable formula related?

3.1.7. A better idea: search for a falsifying truth assignment

Checking tautologies with truth tables is straightforward, but laborious and – let's admit – rather boring. There is a somewhat streamlined method, whereby we attempt to show that the formula is *not* a tautology by finding an appropriate **falsifying assignment**, i.e. a combination of truth values of the propositional variables which renders it false. If we succeed in finding such an assignment, then the formula is indeed not a tautology; if, however, we end up with a *contradiction* (e.g. some variable must be both true and false) in *every* possible case during our exploration, then we have *proved that the formula cannot be falsified*, and hence it is a tautology.¹ The search for a falsifying assignment is based on a step-by-step decomposition of the formula using the truth tables of the propositional connectives occurring in it. Let us see how this works on some examples.

Example 3.1.5

To save on some writing, we will use **signed formulae** $A : T$, meaning '*A must be true*', and $A : F$, meaning '*A must be false*'.

1. Consider $\neg(p \rightarrow \neg q) \rightarrow (p \vee \neg r)$.

To falsify it, it must be the case that $\neg(p \rightarrow \neg q) : T$ and $p \vee \neg r : F$.

For the former, $p \rightarrow \neg q : F$, and hence $p : T$ and $\neg q : F$.

For the latter, $p : F$ and $\neg r : F$.

Therefore, p must be both true and false, which is impossible. Thus, our attempt to falsify the formula has failed, and hence it is a tautology.

2. Consider now $\neg p \rightarrow \neg(p \vee \neg q)$.

To falsify it, it must be the case that $\neg p : T$ while $\neg(p \vee \neg q) : F$.

Therefore $p : F$ and $p \vee \neg q : T$.

For the latter, $p : T$ or $\neg q : T$.

¹ This method of proof is called *indirect proof* or *proof by contradiction* and will be discussed in more detail later.

Let us consider both cases:

Case 1 $p : T$. This contradicts $p : F$.

Case 2 $\neg q : T$. Then $q : F$. In this case we have not reached any contradiction and there is nothing more we can do in order to obtain one. Indeed, we can check that $p : F$ and $q : F$ renders the formula false.

3. A contradiction in the truth values can be reached on any *subformula*, not necessarily a variable. For instance, take $(p \vee \neg q) \rightarrow (\neg p \rightarrow (p \vee \neg q))$. For it to be false, $(p \vee \neg q) : T$ and $(\neg p \rightarrow (p \vee \neg q)) : F$ must be the case. From the latter, $\neg p : T$ and $(p \vee \neg q) : F$, which contradicts the former.

4. Finally, take $((p \wedge \neg q) \rightarrow \neg r) \leftrightarrow ((p \wedge r) \rightarrow q)$. For it to be false, there are two possible cases:

Case 1 $((p \wedge \neg q) \rightarrow \neg r) : T$ and $((p \wedge r) \rightarrow q) : F$. The latter implies $(p \wedge r) : T$ and $q : F$, hence $p : T$, $q : F$, $r : T$. For the former, two subcases are possible:

Case 1.1 $\neg r : T$. Then $r : F$, which is a contradiction with $r : T$.

Case 1.2 $(p \wedge \neg q) : F$. Then:

Case 1.2.1 $p : F$ – a contradiction with $p : T$.

Case 1.2.1 $\neg q : F$. Then $q : T$ – again a contradiction, now with $q : F$.

Note that the consideration of these subcases could have been avoided if we had noticed that the assignment $p : T$, $q : F$, $r : T$ renders $((p \wedge \neg q) \rightarrow \neg r) : F$.

Case 2 $((p \wedge r) \rightarrow q) : T$ and $(p \wedge \neg q) \rightarrow \neg r) : F$. The former implies $(p \wedge \neg q) : T$ and $\neg r : F$, i.e., $p : T$, $q : F$, $r : T$. Again, we can either notice that this assignment renders $((p \wedge r) \rightarrow q) : F$ or consider the cases for $((p \wedge r) \rightarrow q) : T$ and see that all lead to a contradiction.

Thus, in neither case can the formula be falsified, so it is a tautology.

This method can be mechanized completely into a formal procedure, called **semantic tableaux**, which will be presented in Section 3.4.

3.1.8. Exercises

- ① Determine the antecedent and the consequent in each of the following implications:
 - (a) If John talks everyone else listens.
 - (b) Everyone else listens if John talks.
 - (c) John talks only if everyone else listens.
 - (d) An integer is positive if its cube is positive.

(Hint: to make it easier, introduce a name for the object in question, e.g. “An integer n is positive if the cube of n is positive.”)

- (e) An integer is positive only if its cube is positive.
 - (f) A function is continuous whenever it is differentiable.
 - (g) The continuity of a function is necessary for it to be differentiable.
 - (h) The continuity of a function is sufficient for it to be differentiable.
- ② Which of the following conditions are sufficient and which are necessary for the truth of “The integer n is composite”?
- (a) “ n is divisible by 3.”
 - (b) “ n is divisible by 6.”
 - (c) “ n is even.”
 - (d) “ n has at least two different factors.”
 - (e) “ n has more than two different factors.”
 - (f) “ $n = 15$.”
 - (g) “ n has a factor different from n .”
 - (h) “ n has a prime factor.”
 - (i) “ n has a prime factor different from n .”
- ③ Let P, Q, R and S be propositions. Using only the truth tables of the propositional connectives and informal reasoning, show that:
- (a) if the propositions P and $P \rightarrow Q$ are true, then Q is true;
 - (b) if $P \rightarrow Q$, $Q \rightarrow R$ and $P \wedge S$ are true, then $R \wedge S$ is true;
 - (c) if $P \rightarrow Q$, $R \vee (S \wedge \neg Q)$, $\neg R$ are true, then P is false;
 - (d) if $\neg R$, $\neg(P \wedge \neg R)$, $\neg P \rightarrow \neg Q$ are all true, then Q is false;
 - (e) if the propositions $(P \vee Q) \rightarrow R$ and $P \vee R$ are true, then R is true.
(Hint: suppose the contrary.)
- ④ Let P, Q, R and S be propositions. Using truth tables, check whether:
- (a) if $Q \rightarrow (R \wedge S)$ is true and $Q \wedge S$ is false, then $R \wedge Q$ is false;
 - (b) if $P \rightarrow Q$ and $Q \rightarrow (R \vee S)$ are true and $P \rightarrow R$ is false, then $\neg R \rightarrow S$ is true;
 - (c) if $\neg P \rightarrow (\neg Q \vee \neg R)$, $Q \wedge (P \vee R)$ are true, then P is true;
 - (d) if $P \rightarrow Q$ and $Q \rightarrow (R \vee S)$ are true and $P \rightarrow R$ is false, then S is true;
 - (e) if $Q \rightarrow (R \wedge S)$ is true and $Q \wedge S$ is false, then Q is false.
- (Hint: regard P, Q, R and S as propositional variables and inspect the truth tables of the respective formulae.)

⑤ Construct the truth tables of the following propositional formulae and determine which of them (if any) are tautologies, which are contradictory formulae and which are satisfiable formulae.

- (a) $\neg(p \rightarrow \neg p)$
- (b) $p \vee (p \rightarrow \neg p)$
- (c) $p \wedge (q \vee \neg q)$
- (d) $\neg((p \wedge \neg p) \rightarrow q)$
- (e) $\neg p \wedge \neg(p \rightarrow q)$
- (f) $((p \rightarrow q) \rightarrow p) \rightarrow p$
- (g) $(p \vee \neg q) \rightarrow \neg(q \wedge \neg p)$
- (h) $(p \rightarrow q) \wedge (q \rightarrow r) \wedge \neg(\neg p \vee r)$
- (i) $\neg(\neg p \leftrightarrow q) \rightarrow (r \vee \neg q)$
- (j) $\neg((p \wedge \neg q) \rightarrow r) \leftrightarrow (\neg(q \vee r) \rightarrow \neg p)$

⑥ Lastly, some logical puzzles.² On the remote planet Nologic there are two types of intelligent creatures:

- *truth tellers*, who always tell the truth, and
- *liars*, who – you guessed it – always lie.

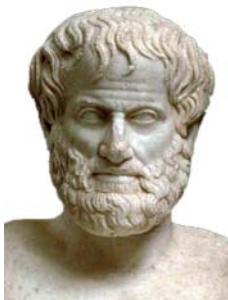
It is not possible to distinguish them by appearance, but only by the truth or falsity of the statements they make.

- (a) Once a space stranger visited Nologic and met there two inhabitants, P and Q. He asked them: "Is any of you a liar?" "At least one of us is a liar", replied P. Can you find out what P and Q are?
- (b) Next, the stranger met two other inhabitants and I asked one of them again "Is any of you a truth teller?" He got a 'yes' or 'no' answer and from that answer he was able to determine for each of them whether he is a liar or not. What was the answer and what was the stranger's conclusion?
- (c) Walking about Nologic, the stranger met two other inhabitants, A and B, and asked A, "Is any of you a liar?" "If B is a liar, then I am a liar, too", replied A. What are A and B?
- (d*) The stranger went on and met three locals, X, Y and Z and asked X: "Are you a liar?" X answered something that the stranger did not hear, so he asked Y: "What did X say?" "He said that he is a liar", replied Y. Then Z added "Don't believe Y, he is a liar." Who are liars?
- (e*) The stranger went on. In the evening, he began to look for a shelter for the night, but was very cautious because he knew that some of the inhabitants were man-eaters and

² For many more of these, we warmly recommend to the reader the marvellous books by Raymond Smullyan, including *What is the name of this book?* and *Lady or the Tiger? And Other Logic Puzzles*.

it was not possible to recognize them by appearance. Then he met three inhabitants, C, D and E. He asked C "How many of you are truth-tellers?" "Flam flim", answered C in her language. "What did she say?", asked the stranger D. "Just one", replied D. "Do not trust D, he is a liar. Come with me, I'm not a man-eater", said E. "No, come with me, I'm not a man-eater", countered D. What should the stranger do?

- (f*) The stranger decided to go back to the port where his spaceship was, but he got lost. After a long walk he got to a fork in the road. He knew that one of the two roads would take him to the spaceship port, but did not know which one. Luckily, he saw two of the inhabitants whom he had met before, so he knew that one of them was a liar and the other a truth-teller, but could not remember who was who.
- Can the stranger ask just one question to any of these inhabitants in order to find out for sure which is the correct road to the spaceship port? If so, what question should he ask?



Aristotle (384–322 BC)

Aristotle was one of the greatest ancient Greek philosophers and, according to *Encyclopaedia Britannica*, "the first genuine scientist in history". He joined Plato's Academy in Athens at the age of 18 and did scholarly work there for about 20 years. After the death of Plato, he left Athens and, at the request of King Philip of Macedonia, became the private teacher of his son, Alexander the Great, during the years 356–323 BC.

It is believed that only about a third of Aristotle's original works have survived. In them he made influential contributions to just about every field of science and arts existing in his time: philosophy, poetry, theatre, music, rhetoric, politics, government and ethics, physics, geology, biology, etc. Aristotle's legacy, not only in philosophy, continues to be the object of active academic study today. For instance, his ideas of systematic classification of plants and animals are still in use in our times.

Aristotle is regarded—amongst his many other seminal contributions – as the founding father of Logic. He was the first to propose the use of formal logical languages in the study of reasoning and to undertake and accomplish a systematic study of the principles of correct reasoning, unsurpassed during the next more than 2000 years. His treatises were collected by his followers under the name *Organon* (from the Greek word meaning "instrument", "tool"), consisting of six books: *Categories*, *On Interpretation*, *Prior Analytics*, *Posterior Analytics*, *Topics* and *On Sophistical Refutations*. In the *Organon* Aristotle developed a theory of deduction for a special kind of logical arguments called *syllogisms*, built on several forms of expressions, like "*All As are (not) Bs*", "*Some As are (not) Bs*", "*No As are Bs*", etc. A typical example of a syllogism is "*All men are mortal. Socrates is a man. Therefore, Socrates is mortal*". In his logical system, now called the *Syllogistic*, there are 256 different types of syllogisms, classified according to their logical form. Aristotle further classified these types as logically valid and invalid, and gave compelling counter-examples to the invalid ones. He also proved many "meta-properties" of correct logical reasoning, thus anticipating the rigorous, mathematical study of modern logic.

3.2. Propositional logical consequence. Valid and invalid propositional inferences

Logic is not concerned with what is true and what is false, but rather with the *correctness of argumentation and reasoning*. Here, we discuss what it means for an argument to be logically correct or not, by formalizing the concept of (*propositional*) *logical consequence*.

First are a couple of examples, providing some food for thought.

Syllogisms, first studied by Aristotle, are specific patterns of arguments. A typical example of a syllogism (not due to Aristotle) is:

All logicians are clever.	
All clever people are rich.	
All logicians are rich.	

The way we actually read this is:

If all logicians are clever and all clever people are rich, then all logicians are rich.

This sounds like a correct piece of reasoning, and it is one, *but* – and this is very important to understand – *it does not mean that the conclusion is necessarily true*. (In fact, unfortunately, it is not.) What then makes it correct?

Here is another one:

All natural numbers are integers.	
Some integers are odd numbers.	
Some natural numbers are odd numbers.	

Note that all of the statements above are true. Therefore, is this a correct argument? If you think so, then how about this: take the same argument and replace the words “natural numbers” by “mice”, “integers” by “animals” and “odd numbers” by “elephants”. This will not change the logical form of the argument and, therefore, should not change the correctness. The result speaks for itself:

All mice are animals.	
Some animals are elephants.	
Some mice are elephants.	

So, what makes an argument logically correct? We will answer this question in full later in the book. Now we will provide and discuss the answer for a simple type of logical argument that only involves propositional reasoning, called **propositional arguments**.

3.2.1. Propositional logical consequence

The intuition behind logically correct reasoning is simple: starting from true premises it should always lead to true conclusions. Let us first make this intuition precise.

Definition 3.2.1 A propositional formula B is a **logical consequence** of the propositional formulae A_1, \dots, A_n , denoted $A_1, \dots, A_n \models B$, if B is true whenever all A_1, \dots, A_n are true, that is:

Every truth assignment to the variables occurring in A_1, \dots, A_n and B that renders the formulae A_1, \dots, A_n true also renders the formula B true.

When $A_1, \dots, A_n \models B$, we also say that B follows logically from A_1, \dots, A_n or that A_1, \dots, A_n logically imply B .

In the context of $A_1, \dots, A_n \models B$, the formulae A_1, \dots, A_n are called **assumptions** while the formula B is called the **conclusion**. When $A_1, \dots, A_n \models B$ is not the case, we write $A_1, \dots, A_n \not\models B$. Thus, if $A_1, \dots, A_n \models B$ then every substitution of propositions for the variables occurring in A_1, \dots, A_n, B that turns the formulae A_1, \dots, A_n into true propositions also turns the formula B into a true proposition.

In order to check if $A_1, \dots, A_n \models B$ we can simply complete the truth tables of A_1, \dots, A_n, B and check row by row if the following holds: whenever all formulae A_1, \dots, A_n have a truth value T in that row, B must have a truth value T as well.³ If that holds in every row in the table, then B does follow logically from A_1, \dots, A_n ; if that fails in at least one row, then B does not follow logically from A_1, \dots, A_n .

Thus, B does not follow logically from A_1, \dots, A_n if and only if there is a truth value assignment that renders all formulae A_1, \dots, A_n true and B false.

Example 3.2.2

Let us start with a few simple cases:

1. Any formula B follows logically from any set of formulae that contains B . (Why?)
2. Q follows logically from P and $P \rightarrow Q$, for any formulae P and Q .

First note that, whatever the formulae P and Q are, any truth assignment eventually renders each of them true or false, and all combinations of these truth values are possible. Therefore, we can just as well treat P and Q as propositional variables and consider the truth tables for P, Q and $P \rightarrow Q$:

P	Q	\parallel	P	$ $	$P \rightarrow Q$	\parallel	Q
T	T		T		T		T
T	F		T		F		F
F	T		F		T		T
F	F		F		T		F

Indeed, in every row where the third and fourth entries are T, the fifth entry is also T.

3. Is it true that $p \vee q, q \rightarrow p \models p \wedge q$? Let us check the definition:

p	q	\parallel	$p \vee q$	$ $	$q \rightarrow p$	\parallel	$p \wedge q$
T	T		T		T		T
T	F		T		T		F
F	T	
F	F	

We see that in the second row both assumptions are true while the conclusion is false, and this suffices to conclude that $p \vee q, q \rightarrow p \not\models p \wedge q$, so there is no need to complete the truth table any further.

³ Of course, it is possible for B to be true without all, or any, of A_1, \dots, A_n being true.

A logical consequence can be expressed quite simply in terms of tautologies:

The formula B follows logically from A_1, \dots, A_n if and only if the formula $(A_1 \wedge \dots \wedge A_n) \rightarrow B$ is a tautology.

In particular, a formula A is a tautology if and only if it follows logically from the empty set of formulae. Indeed, if $\models A$ then, trivially, $\emptyset \models A$. Now, to see that $\emptyset \models A$ implies $\models A$, suppose $\emptyset \models A$ and take *any* truth assignment. Note that it satisfies *every formula* from \emptyset . Why? Well, there can be *no* formula in \emptyset that is not satisfied, because there are no formulae in \emptyset at all! Then, since $\emptyset \models A$, that truth assignment must also satisfy A . Thus, A is satisfied by every truth assignment.⁴

Checking logical consequences can be streamlined, just like checking tautologies, by searching for a falsifying assignment; i.e. in order to check if B follows logically from A_1, \dots, A_n we look for an assignment of truth values to the variables occurring in A_1, \dots, A_n, B that renders all A_1, \dots, A_n true and B false. If we succeed, we have shown that B does not follow logically from A_1, \dots, A_n ; otherwise we try to prove that no such assignment is possible by showing that the assumption that it exists leads to a contradiction.

Example 3.2.3

Show that $p \rightarrow r$ and $q \rightarrow r$ logically imply $(p \vee q) \rightarrow r$.

- Method 1 (truth tables)

p	q	r	$p \rightarrow r$	$q \rightarrow r$	$p \vee q$	$(p \vee q) \rightarrow r$
T	T	T	T	T	T	T
T	T	F	F	F	T	F
T	F	T	T	T	T	T
T	F	F	F	T	T	F
F	T	T	T	T	T	T
F	T	F	T	F	T	F
F	F	T	T	T	F	T
F	F	F	T	T	F	T

In every row where the truth values of $p \rightarrow r$ and $q \rightarrow r$ are T, the truth value of $(p \vee q) \rightarrow r$ is also T.

- Method 2 (search for falsifying assignment). Recall that $A : T$ (respectively $A : F$) means that A has, or is required to have, a truth value T (respectively F). Now, suppose that for some assignment $(p \rightarrow r) : T$, $(q \rightarrow r) : T$ and $((p \vee q) \rightarrow r) : F$. Then $(p \vee q) : T$ and $r : F$, and hence $p : T$ or $q : T$.

Case 1: $p : T$. Then $(p \rightarrow r) : F$ – a contradiction.

Case 2: $q : T$. Then $(q \rightarrow r) : F$ – again, a contradiction.

Thus, there is no assignment that falsifies the logical consequence.

⁴ Note that, what we did here was a piece of logical reasoning based on the very same concepts of truth and logical consequence that we are discussing. When reasoning about logic, this kind of bootstrapping argument is inevitable!

In the next section we are going to develop this method systematically.

3.2.2. Logically sound rules of propositional inference. Logically correct propositional arguments

Definition 3.2.4 A rule of propositional inference (inference rule, for short) is a scheme:

$$\frac{P_1, \dots, P_n}{C}$$

where P_1, \dots, P_n, C are propositional formulae. The formulae P_1, \dots, P_n are called premises of the inference rule and C is its conclusion.

An instance of an inference rule, obtained by uniform substitution of propositions for the variables occurring in all formulae of the rule, is called a propositional inference, or a propositional argument based on that rule.

Definition 3.2.5 An inference rule is (logically) sound if its conclusion follows logically from the premises. A propositional inference is (logically) correct if it is an instance of a logically sound inference rule.

Example 3.2.6

Here are some simple but important examples:

1. The inference rule (known as the Detachment rule, or Modus Ponens)

$$\frac{p, p \rightarrow q}{q}$$

is sound (as we have already seen).

Therefore, the inference

$$\begin{array}{c} \text{Chantel is singing.} \\ \text{If Chantel is singing, then Chantel is happy.} \\ \hline \text{Chantel is happy} \end{array}$$

is logically correct, being an instance of that inference rule.

2. The inference rule

$$\frac{p, q \rightarrow p}{q}$$

is not sound: if p is true and q is false, then both premises are true, while the conclusion is false.

Therefore the inference

$$\begin{array}{c} \text{2 plus 2 equals 4.} \\ \text{If 5 is greater than 3, then 2 plus 2 equals 4.} \\ \hline \text{5 is greater than 3.} \end{array}$$

is not logically correct (despite the truth of the conclusion), being an instance of a logically unsound rule.

Some remarks are in order here.

- It is important to realize that the logical correctness of an inference *does not always guarantee the truth of the conclusion, but only when all premises are true*. In other words, if at least one premise of a logically correct inference is false, then the conclusion may also be false. For instance, the inference

$$\begin{array}{c} 5 \text{ divides } 6. \\ \text{If } 5 \text{ divides } 6, \text{ then } 5 \text{ divides } 11. \\ \hline 5 \text{ divides } 11. \end{array}$$

is logically correct (being an instance of the detachment rule) in spite of the falsity of the conclusion; this does not contradict our idea of logical correctness of an inference, because the first premise is false. (What about the second premise?)

- Conversely, if the conclusion of an inference happens to be true, this does not necessarily mean that the inference is logically correct, as in Example 3.2.6(2).
- Moreover, it may happen that the truth of the premises of an inference *does imply* the truth of the conclusion, and yet the inference is not *logically* correct. For instance, the correctness of the inferences

$$\begin{array}{c} \text{Today is Monday.} \\ \hline \text{Tomorrow will be Tuesday.} \end{array}$$

or

$$\begin{array}{c} a = 2, a + b = 5 \\ \hline b = 3 \end{array}$$

is based *not* on logical consequence but, in the first case, on the commonly known fact that Tuesday always follows Monday, and in the second case on some laws of arithmetic. Indeed, the first inference is based on the rule

$$\frac{p}{q}$$

and the second one (though the statements in that rule are not real propositions) on

$$\frac{p, q}{r},$$

both of which are clearly unsound.

To summarize, the meaning and importance of logically correct inferences is that only such inferences guarantee that *if all premises are true, then the conclusions will also be true*. That is why only logically correct inferences are safe to be employed in our reasoning.

Example 3.2.7

Let us now look at a few more examples.

1. The inference rule

$$\frac{q, p \vee \neg q}{p}$$

is logically sound. You can check this in two ways: by applying the definition or by showing that the corresponding formula

$$(q \wedge (p \vee \neg q)) \rightarrow p$$

is a tautology. In particular, the inference

$$\begin{array}{c} \text{Josephine is singing or Josephine does not feel happy.} \\ \text{Josephine feels happy.} \\ \hline \text{Josephine is singing.} \end{array}$$

is valid, being an instance of the rule above.

2. Now, take the argument

If a divides b or a divides c , then a divides bc .

a divides bc .

a does not divide b .

Therefore a divides c .

where a, b, c are certain integers. This argument is an instance of the following inference rule:

$$\frac{(p \vee q) \rightarrow r, r, \neg p}{q}.$$

Let us see if we can invalidate this rule. For that we need an assignment such that $((p \vee q) \rightarrow r) : T, r : T, \neg p : T$; hence $p : F$ and $q : F$. Indeed, the assignment $p : F, q : F, r : T$ renders all premises true and the conclusion false. (Check this.)

Therefore, the rule is not logically sound and hence the argument above is not logically correct.

3.2.3. Fallacies of the implication

As an application let us analyze some very common forms of correct and incorrect reasoning related to implications. Given the implication

$$A \rightarrow B$$

we can form the so-called **derivative implications**:

- the **converse** of $A \rightarrow B$: $B \rightarrow A$;

- the **inverse** of $A \rightarrow B$: $\neg A \rightarrow \neg B$;
- the **contrapositive** of $A \rightarrow B$: $\neg B \rightarrow \neg A$.

Now, suppose we know that $A \rightarrow B$ is true. What can we say about the truth of its derivatives? To answer that question, look at the inferences

$$\frac{A \rightarrow B}{B \rightarrow A}, \quad \frac{A \rightarrow B}{\neg A \rightarrow \neg B}, \quad \frac{A \rightarrow B}{\neg B \rightarrow \neg A}$$

Exercise

Show that the first two of these inferences are incorrect, while the third one is correct.

Therefore, the truth of $A \rightarrow B$ only implies the truth of its contrapositive, but not the truth of the converse, or inverse. These are mistakes that people often make, respectively called the **fallacy of the converse implication** and the **fallacy of the inverse implication**. For example, the truth of the implication "If it has just rained, then the tennis court is wet" does not imply that either of "If the tennis court is wet, then it has just rained" and "If it has not just rained, then the tennis court is not wet" is true – for someone may have just watered the court on a clear sunny day – but it certainly implies that "If the tennis court is not wet, then it has not just rained".

In fact, one can easily check that the inference

$$\frac{\neg B \rightarrow \neg A}{A \rightarrow B}$$

is also logically correct. This is the basis of the method of **proof by contraposition**, which will be discussed later.

3.2.4. Exercises

- ❶ Show that the following is true:

The formula B follows logically from A_1, \dots, A_n if and only if the formula $(A_1 \wedge \dots \wedge A_n) \rightarrow B$ is a tautology.

- ❷ Show that the first two of the following inference rules, corresponding to the derivative implications, are logically unsound, while the third one is sound.

(a)
$$\frac{p \rightarrow q}{q \rightarrow p}$$

(b)
$$\frac{p \rightarrow q}{\neg p \rightarrow \neg q}$$

(c)
$$\frac{p \rightarrow q}{\neg q \rightarrow \neg p}$$

- ❸ Using truth tables, check if the following inference rules are valid.

(a)
$$\frac{p \rightarrow q, \neg q \vee r, \neg r}{\neg p}$$

(b)
$$\frac{p \rightarrow q, p \vee \neg r, \neg r}{\neg q \vee r}$$

$$(c) \frac{\neg p \rightarrow \neg q, q, \neg(p \wedge \neg r)}{r}$$

$$(d) \frac{((p \wedge q) \rightarrow r), \neg(p \rightarrow r)}{q \rightarrow r}$$

- ④ Write down the inference rules on which the following arguments are based and check their logical validity, using truth tables.

- (a) In the following argument, x is a certain number.

x is greater than 3.

x is greater than or equal to 3.

- (b) In the following argument, a is a certain number.

If a is greater than -1 , then a is greater than -2 .

a is not greater than -2 .

a is not greater than -1 .

- (c)

If the triangle ABC has a right angle, then it is not equilateral.
The triangle ABC does not have a right angle.

Therefore, the triangle ABC is equilateral.

- (d)

If Bill likes logic, then he is clever.

If Bill is clever, then he is rich.

Therefore, if Bill likes logic, then he is rich.

- (e) In the following argument n is a certain integer.

If n is divisible by 2 and n is divisible by 3, then n is divisible by 6.

If n is divisible by 6, then n is divisible by 2.

n is not divisible by 3.

Therefore, n is not divisible by 6.

- ⑤ For each of the following implications construct the converse, inverse and the contrapositive, phrased in the same way.

- (a) If a is greater than -1 , then a is greater than -2 .

- (b) If x is divisible by 6 then x is not prime.

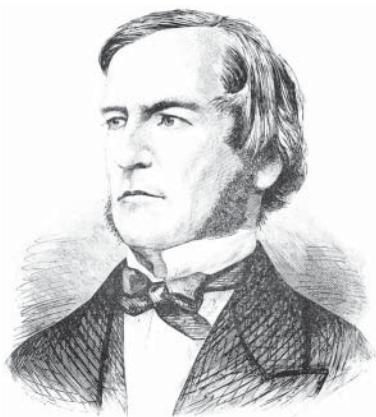
- (c) x is positive only if its square is positive.

- (d) The triangle ABC is equilateral whenever its medicentre and orthocentre coincide.

- (e) For the function f to be continuous it is sufficient that it is differentiable.

- (f) For the function f not to be differentiable it is sufficient that it is discontinuous.

- (g) For the integer n to be prime it is necessary that it is not divisible by 10.



George Boole (2.11.1815–8.12.1864)

George Boole was an English mathematician and the first to propose and develop an algebraic approach to the study of logical reasoning. Boole's first contribution to logic was a pamphlet called *Mathematical Analysis of Logic*, written in 1847. He published his main work on logic, *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities*, in 1854. In it Boole developed a general mathematical method of logical inference and thus laid the foundations of modern mathematical logic. His system proposed a formal algebraic treatment of propositions by processing only their two possible truth values: yes–no, true–false, zero–one. For instance, in Boole's system, if x stands for "white things", then $1 - x$ stands for "non-white things", and if also y stands for "sheep", then xy stands for "white sheep", etc. Then $x(1 - x)$ denotes things that are both white and non-white, which is impossible, so a proposition of the shape $x(1 - x)$ is always false, i.e. has a truth value 0. Thus an algebraic law emerges: $x(1 - x) = 0$, etc. The resulting algebraic system is known today as (the simplest) *Boolean algebra*.

Boole also argued that symbolic logic is needed in other mathematical disciplines, especially in probability theory. He wrote: "...no general method for the solution of questions in the theory of probabilities can be established which does not explicitly recognise those universal laws of thought which are the basis of all reasoning..."

Propositional logic today is somewhat different from Boole's system of logic 150 years ago, but the underlying ideas are the same. That is why propositional logic is also often called *Boolean logic* in recognition of Boole's groundbreaking contribution. Boolean logic is not only fundamental as a system of formal logical reasoning but it also provides the mathematical basis of the *logical circuits* underlying the architecture of the modern digital computers.

3.3. The concept and use of deductive systems

The most fundamental concept in Logic is logical consequence. It extends the concept of logical validity and is the basis of logically correct reasoning. While verifying logical consequence in propositional logic is conceptually simple and technically easy (though possibly computationally expensive), this is not so any more for the full *first-order logic*, which we will introduce later in Chapter 4, nor for the variety of *nonclassical logics*, which we will not discuss here. In fact, verifying logical consequence (in particular, logical validity) is usually an *infinite task*, as it generally requires checking infinitely many possible "models", rather than a finite number of simple truth assignments. Thus, a different approach for proving logical validity and consequence, not based on the semantic definition, becomes practically necessary. Such an approach is provided by the notion of the *deductive system*: a formal, mechanical – or, at least mechanizable – procedure for *derivation (inference, deduction)* of formulae, or lists ("sequents") of formulae, by applying precise *inference rules* and possibly using some formulae that are postulated as derived, called *axioms*. The underlying idea of deductive systems is to *simulate*

and substitute the semantic notion of logical consequence with the formal, mechanical notion of *deductive consequence*, which is based only on the syntactic shape of the formulae to which inference rules are applied, but not on their meaning. Thus, at least theoretically, any person with no knowledge and understanding of the meaning of logical formulae and logical consequence, or even a suitably programmed computer, should be able to do successful derivations in a given logical deductive system.

The idea of logical deductive systems goes back to antiquity, most notably to the book *Organon* of the famous ancient Greek philosopher and founding father of Formal Logic, Aristotle (384–322 BC). Perhaps the first real prototype of a deductive system can be found in the fundamental work *Elements* of the “father of Geometry”, the greatest ancient Greek mathematician Euclid (ca. 300 BC), who provided a systematic development of elementary geometry based on several simple assumptions about points and lines, called *postulates* (such as “Every two different points determine exactly one line”, “For every line there is a point not belonging to that line”, etc.). Using these postulates and some informal logical reasoning, other geometric facts are derived and thus the entire body of Euclidean geometry is eventually built. The logical concept of a deductive system gradually emerged much later, notably through the ideas of one of the most influential philosophers and mathematicians of all times, Gottfried Leibniz (1646–1716) for a *characteristica universalis* (universal language) and a *calculus ratiocinator* (calculus of reasoning). The first really formal deductive system, however, was only constructed in the late 19th century by the mathematician and philosopher Gottlob Frege (1848–1925) in his seminal book *Begriffsschrift* (1879), for his prototype of the modern first-order logic. Later, David Hilbert⁵ (1862–1943), one of the leading mathematicians of the late 19th and early 20th century, re-worked Euclid’s system of geometry developed in the *Elements* into a rigorous and mathematically precise treatment that resulted in his book *Foundations of Geometry* (1899) and the mathematician Giuseppe Peano (1858–1932) developed a formal system of arithmetic, still known as Peano’s axiomatic system, in his most important work *Formulario Mathematico*. The concept of formal deductive system was further developed by the famous philosophers Bertrand Russell (1872–1970) and Alfred North Whitehead (1861–1947) in their three-volume book *Principia Mathematica* (1910–1913), and by David Hilbert and Wilhelm Ackermann (1896–1962) in their book *Principles of Mathematical Logic* (1928), followed by the two-volume book *Grundlagen der Mathematik* of Hilbert and Paul Bernays (1888–1977), published in 1934 and 1939. These books were most influential for the development of logic and the foundations of mathematics in the first half of the 20th century. In particular, Hilbert vigorously promoted the idea of building the whole body of mathematics as a formal *axiomatic system* – a deductive system mainly based on axioms and on a very few and simple rules of inference. Thus, Hilbert was the leading proponent of the development of the *formal, axiomatic approach* in mathematics, meant to replace the semiformal notion of mathematical proof by the completely formalized notion of derivation in an axiomatic system, thus freeing mathematics from paradoxes and other possible contradictions and putting it on sound logical foundations.⁶

⁵ Read more on Hilbert in the box at the end of this section.

⁶ In the early 1930s it turned out from the groundbreaking works of the most famous logician of the 20th century Kurt Gödel (1906–1978) that Hilbert’s idea was only partly realizable, for purely logical validity and consequence in first-order logic, but it was not realizable for “mathematical” consequence, even in relatively simple mathematical systems such as arithmetic.

There are different types of **deductive systems**, but they all share several common aspects and principles:

- A deductive system works within a *formal logical language*, where derivations are performed on formal expressions, called *formulae* (or “*sequents*”, being finite sequences of formulae), which are special words—finite strings of symbols – in the formal language, having precise *syntax*.⁷
- The main component of a deductive system is the notion of *derivation (inference, deduction)* from a given set of assumptions, based on a set of precisely specified *rules of inference*. The idea, again, is that by systematically applying these rules, one can *derive (infer, deduce)* formulae from other formulae that are already derived, or *assumed* as given, and called *assumptions (premises)*. In addition, axiomatic systems also allow for an initial set of formulae, called *axioms*, to be accepted as derived without applying any rules of inference. Thus, the axioms can always be used as premises in derivations.

If a formula A can be derived from a set of assumptions Γ in a deductive system D , we denote this by

$$\Gamma \vdash_D A$$

and say that A is a **deductive consequence** from Γ in D . In particular, if $\Gamma = \emptyset$, we write

$$\vdash_D A$$

and say that A is a **theorem** of D .

- A very important aspect of deductive systems is that derivations in them are *completely mechanical procedures*, which do not require any intelligence or understanding of the meaning of the formulae or rules involved; in fact, such meaning need not be specified at all. Thus, derivations in a given deductive system can be performed by a mechanical device, such as computer, without any human intervention, as long as the axioms and rules of inference of the deductive system have been programmed into it.
- While deductive systems are not concerned with the meaning (*semantics*) of the formulae they derive, they are designed with the purpose of deriving *valid, and only valid*, logical consequences from the assumptions. A deductive system with this property is called *sound*. In particular, every theorem in a sound deductive system must be true; i.e. in the case of propositional logic it must be a tautology. Formally, a deductive system D is **sound** (or **correct**) for a given logical semantics (i.e. well-defined notions of logical validity and consequence) if D can *only* derive logically valid consequences, i.e.:

$$A_1, \dots, A_n \vdash_D C \text{ implies } A_1, \dots, A_n \models C.$$

In particular:

$$\vdash_D C \text{ implies } \models C.$$

- A deductive system D is **complete** for a given logical semantics if D can derive *every* valid logical consequence, i.e.:

$$A_1, \dots, A_n \models C \text{ implies } A_1, \dots, A_n \vdash_D C.$$

⁷ In this aspect a deductive system is much like a programming language.

In particular:

$$\models C \text{ implies } \vdash_D C,$$

i.e. a complete deductive system can derive *every* logically valid formula.

A deductive system D is **adequate** for a given semantics if it is both sound and complete for it, i.e.:

$$A_1, \dots, A_n \models C \text{ if and only if } A_1, \dots, A_n \vdash_D C.$$

In particular:

$$\models C \text{ if and only if } \vdash_D C.$$

The most popular types of deductive systems for classical logic are: *Axiomatic systems*, *Natural deduction*, *Sequent calculi*, *Semantic tableaux* and *Resolution*. In this book we only present Semantic tableaux and Resolution, as they are the easiest to use and automatize.



David Hilbert (23.01.1862–14.02.1943)

David Hilbert was one of the leading and most influential mathematicians of the late 19th and early 20th centuries. He studied at the University of Königsberg and in 1895 was appointed Professor of Mathematics at the University of Göttingen – the world leading research center for mathematics at that time – where he remained for the rest of his life. Hilbert's contributions to mathematics were encyclopaedic, ranging from his fundamental ideas in mathematical logic and the foundations of mathematics to revolutionizing the theory of invariants and the theory of algebraic number fields, introducing (what is now called) Hilbert spaces, and founding the field of functional analysis as the main instrument of modern mathematical physics. At the end of the 19th century Hilbert completely revised and modernized Euclid's seminal work *Elements* (see the box at the end of Section 5.3 in Chapter 5) in his book *Foundations of Geometry* published in 1899, with which he put the foundations of geometry in a formal axiomatic setting. With this work Hilbert also promoted the formal, axiomatic approach to mathematics and essentially launched what became known as *Hilbert's programme* for axiomatizing and formalizing the entire body of mathematics. This was the most influential development in the foundations of mathematics throughout the 20th century. By putting forward and persistently pursuing that programme for many decades, Hilbert became one of the founders of proof theory – which he called *metamathematics* – a major branch of mathematical logic.

In his historical speech *The Problems of Mathematics*, delivered at the Second International Congress of Mathematicians in Paris in 1900, Hilbert presented his famous list of 23 unsolved mathematical problems, the solutions of which would, according to him, guide and chart the development of mathematics in the 20th century. Many of Hilbert's 23 problems were solved during the 20th century and their solutions influenced much of the development of mathematics at the time. Others, such as Goldbach's conjecture and the Riemann hypothesis, are still famous unsolved problems.

Hilbert was baptized and raised in the Protestant church, but later on he became an agnostic. He argued that mathematical truth was independent of the existence of God or other a priori assumptions. In his 1930 retirement address he responded to the Latin maxim: "*Ignoramus et ignorabimus*" ("We do not know, we shall not know"), used in the 19th century in connection with the perceived limits of scientific knowledge, by stating his famous mathematician's credo: "*Wir müssen wissen. Wir werden wissen*" ("We must know. We will know"). These words became the epitaph on his tombstone in Göttingen.

3.4. Semantic tableaux

Here we will formalize the idea of testing the validity of propositional formulae and inferences by a systematic search for a falsifying truth assignment into a formal (deductive) system, called the semantic tableaux. The first versions of the semantic tableaux were designed in the 1950s by the logicians Evert Beth and Jaakko Hintikka and further developed in the 1960s and later by Raymond Smullyan, Melvin Fitting and others.

In order to prove a logical consequence $A_1, \dots, A_n \models B$, the method of semantic tableaux attempts to show that there is no truth assignment that falsifies it, which is equivalent to showing that the formulae A_1, \dots, A_n and $\neg B$ cannot be satisfied simultaneously.

The system of semantic tableaux (ST) is based on **formula-decomposition rules**, which reduce the truth or falsity of a formula to the truth or falsity of its main subformulae. These rules can be extracted from the truth tables of the propositional connectives as follows:

1. \neg :

($\neg T$) For $\neg A$ to be true, A must be false.

($\neg F$) For $\neg A$ to be false, A must be true.

2. \wedge :

($\wedge T$) For $A \wedge B$ to be true, A must be true and B must be true.

($\wedge F$) For $A \wedge B$ to be false, A must be false or B must be false.

3. \vee :

($\vee T$) For $A \vee B$ to be true, A must be true or B must be true.

($\vee F$) For $A \vee B$ to be false, A must be false and B must be false.

4. \rightarrow :

($\rightarrow T$) For $A \rightarrow B$ to be true, A must be false or B must be true.

($\rightarrow F$) For $A \rightarrow B$ to be false, A must be true and B must be false.

5. \leftrightarrow :

($\leftrightarrow T$) For $A \leftrightarrow B$ to be true, A and B must be true or A and B must be false.

($\leftrightarrow F$) For $A \leftrightarrow B$ to be false, A must be true and B must be false or A must be false and B must be true.

These rules can be formalized further as follows:

Rules for semantic tableaux in propositional logic

Non-branching rules (α -rules)		Branching rules (β -rules)	
$(\wedge T)$	$A \wedge B : T$ \downarrow $A : T, B : T$	$(\wedge F)$	$A \wedge B : F$ $\swarrow \searrow$ $A : F, B : F$
$(\vee F)$	$A \vee B : F$ \downarrow $A : F, B : F$	$(\vee T)$	$A \vee B : T$ $\swarrow \searrow$ $A : T, B : T$
$(\rightarrow F)$	$A \rightarrow B : F$ \downarrow $A : T, B : F$	$(\rightarrow T)$	$A \rightarrow B : T$ $\swarrow \searrow$ $A : F, B : T$
$(\neg T)$	$\neg A : T$ \downarrow $A : F$	$(\leftrightarrow T)$	$A \leftrightarrow B : T$ $\swarrow \searrow$ $A : T, B : T \quad A : F, B : F$
$(\neg F)$	$\neg A : F$ \downarrow $A : T$	$(\leftrightarrow F)$	$A \leftrightarrow B : F$ $\swarrow \searrow$ $A : T, B : F \quad A : F, B : T$

Using these rules we can search *systematically* for an assignment, falsifying a given formula. In the process of this systematic application of the rules we build a “search tree”, called a **tableau**.

If we are to derive the consequence $A_1, \dots, A_n \models B$ we start by placing the signed formulae

$$A_1 : T, \dots, A_n : T, B : F$$

at the root of the tree and then extend it downwards by applying repeatedly the decomposition rules to signed formulae appearing on the tree as follows:

1. A branch on the tree is selected.
2. A signed formula occurring on that branch is selected to which a decomposition rule has not been applied yet on that branch.
3. The respective decomposition rule is applied to that signed formula. As a result, one (for the α -rules) or two (for the β -rules) successor nodes are added, labelled with the signed formulae which that rule introduces.

We repeat that procedure along every branch of the tableau until:

- either a **contradictory pair of signed formulae** – i.e. a pair of the type $A : T, A : F$ – appears on the branch, in which case we declare that branch **closed** (meaning that no assignment can ever be found which satisfies all signed formulae on that branch), mark it by \times and do not extend it any further

or

- no new applications of decomposition rules are possible on that branch, i.e. every signed formula appearing on the branch has already been decomposed further on that branch. Then we say that the branch is **saturated**. If a saturated branch is not closed yet, we declare it **open** and mark it by \circlearrowleft . If the tableau construction ends with at least one open branch, the tableau is declared **open**.

Note that an open branch defines an assignment for all variables occurring on that branch as follows: a variable p is true under that assignment if $p : T$ appears on the branch and is false otherwise (in which case, $p : F$ may or may not appear on the branch). It is easy to check that such an assignment will satisfy *all* signed formulae on the branch. That can be done step by step by going up the branch. Eventually, all signed formulae at the root will be satisfied, which will falsify the logical consequence $A_1, \dots, A_n \models B$ and will thus prove it invalid.

On the other hand, if all branches of the tableau close, then the entire tableau is declared **closed**. This means that the systematic search for an assignment satisfying the input formulae listed at the root has failed. Then we say that B is **derived from** A_1, \dots, A_n in ST, which we denote by

$$A_1, \dots, A_n \vdash_{\text{ST}} B.$$

In particular, if $\emptyset \vdash_{\text{ST}} B$ then we say that B is a **theorem of ST**.

It can be proved that the semantic tableaux is *sound and complete*, i.e.:

$$A_1, \dots, A_n \vdash_{\text{ST}} B \text{ iff } A_1, \dots, A_n \models B.$$

In particular, the tableau applied to $B : F$ closes iff B is a tautology.

Example 3.4.1

Let us demonstrate the method of semantic tableaux on some of the examples we did informally in Example 3.1.5:

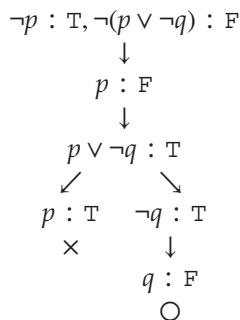
1. Show that $\neg(p \rightarrow \neg q) \rightarrow (p \vee \neg r)$ is a tautology:

$$\begin{array}{c} \neg(p \rightarrow \neg q) \rightarrow (p \vee \neg r) : F \\ \downarrow \\ \neg(p \rightarrow \neg q) : T, (p \vee \neg r) : F^1 \\ \downarrow \\ p \rightarrow \neg q : F \\ \downarrow \\ p : T, \neg q : F \\ \downarrow \\ q : T \\ \downarrow^1 \\ p : F, \neg r : F \\ \times \end{array}$$

Thus, $\vdash_{\text{ST}} \neg(p \rightarrow \neg q) \rightarrow (p \vee \neg r)$ and hence $\models \neg(p \rightarrow \neg q) \rightarrow (p \vee \neg r)$.

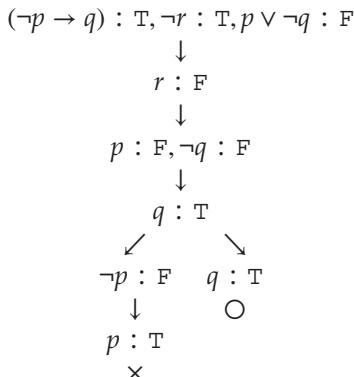
Note that no branching rules were applicable in this example, so the tableau consists of only one branch, which closes because the complementary pair $p : T, p : F$ appears on it. The superscript ¹ is used as a label indicating where the decomposition rule is applied to the formula $(p \vee \neg r) : F$. This label is not needed for the derivation; it has only been added for bookkeeping purposes and as an aid to following the construction of the tableau, and will not be systematically used further.

2. Check if $\neg p \models \neg(p \vee \neg q)$.



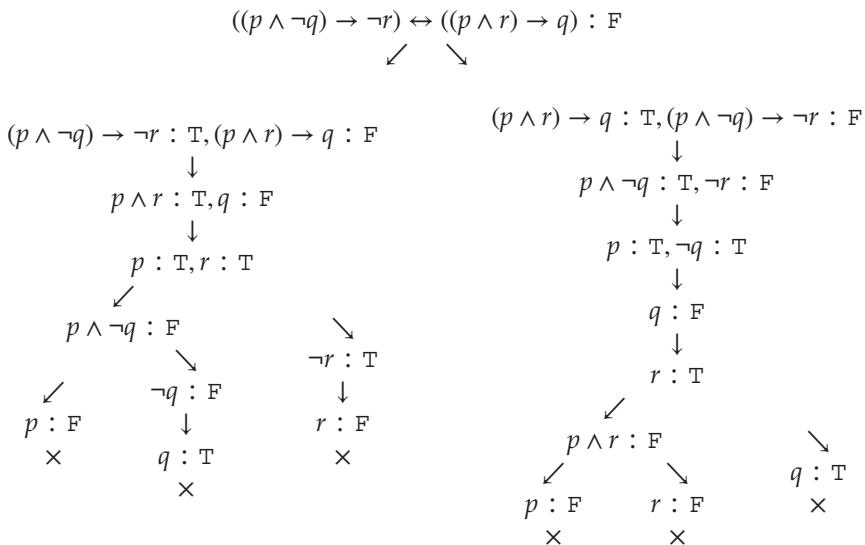
The left-hand branch closes, but the right-hand one does not close and no more decomposition rules are applicable on it. Thus it is declared open, and therefore the whole tableau is open, which means that $\neg p \not\models \neg(p \vee \neg q)$. Indeed, the right-hand branch provides a falsifying valuation for that consequence: $p : F, q : F$. (Check it!)

3. Check whether $(\neg p \rightarrow q), \neg r \models p \vee \neg q$:



Again, the tableau is open, and a falsifying valuation is provided by the open branch: $p : F, q : T, r : F$. Note that we chose to first decompose the subformulae $\neg r : T$ and $(p \vee \neg q) : F$ because they are nonbranching (α -formulae) and then $\neg p \rightarrow q : T$, which is branching (β -formula). This does not make a difference in the outcome of the tableau but saves some work, so it is a generally preferable strategy.

4. Check whether $((p \wedge \neg q) \rightarrow \neg r) \leftrightarrow ((p \wedge r) \rightarrow q)$ is a tautology.



The tableau closes and hence the formula is a tautology.

Lastly, here are a few further remarks:

- At any stage during the construction of a semantic tableau, more than one decomposition rule may be applicable. Depending of the order of their application, the tableau tree may develop differently, but the final result – whether the tableau closes or not – will always be the same. That order is therefore inessential. However, it may affect the size of the tableau tree. As a general guideline, it is better to first apply the non-branching rules before applying the branching rules (if there is a choice) as this would defer the branching and would reduce the size of the tableau.
- Note that no node in a tableau can have more than two children nodes. Thus, if a node in your tableau has three or more children then *there is something wrong!*
- Although we can use equivalences like $p \wedge (q \vee p) \equiv p$ and $\neg(p \wedge q) \equiv \neg p \wedge \neg q$ when reasoning informally, these equivalences do not form part of the rules of the deductive system of ST and should *not* be used in the construction of the tableaux tree. As we emphasized in Section 3.3, derivations within a deductive system may only use the rules of that system, in order to be completely mechanizable.

3.4.1. Exercises

For each of the following exercises use the method of semantic tableaux.

- ① Check which of the following formulae are tautologies.

- (a) $((p \rightarrow q) \rightarrow q) \rightarrow q$
- (b) $((q \rightarrow p) \rightarrow q) \rightarrow q$
- (c) $((p \rightarrow q) \wedge (p \rightarrow \neg q)) \rightarrow \neg p$
- (d) $((p \vee q) \rightarrow \neg r) \rightarrow \neg(\neg q \wedge r)$
- (e) $((p \rightarrow q) \wedge (p \rightarrow r)) \rightarrow (p \rightarrow (q \wedge r))$
- (f) $((p \rightarrow r) \wedge (q \rightarrow r)) \rightarrow ((p \vee q) \rightarrow r)$
- (g) $((p \rightarrow r) \vee (q \rightarrow r)) \rightarrow ((p \vee q) \rightarrow r)$
- (h) $((p \rightarrow r) \wedge (q \rightarrow r)) \rightarrow ((p \wedge q) \rightarrow r)$
- (i) $p \rightarrow ((q \rightarrow r) \rightarrow ((p \rightarrow q) \rightarrow r))$
- (j) $(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$

② Check which of the following logical consequences are valid.

- (a) $\neg p \rightarrow q, \neg p \rightarrow \neg q \vDash p$
- (b) $p \rightarrow (q \wedge r) \vDash (p \rightarrow q) \wedge (p \rightarrow r)$
- (c) $(p \wedge q) \rightarrow r \vDash (p \rightarrow r) \wedge (q \rightarrow r)$
- (d) $(p \wedge q) \rightarrow r \vDash (p \rightarrow r) \vee (q \rightarrow r)$
- (e) $(p \vee q) \rightarrow r \vDash (p \rightarrow r) \wedge (q \rightarrow r)$
- (f) $(\neg p \wedge q) \rightarrow \neg r, r \vDash p \vee \neg q$

③ Let P, Q, R and S be propositions. Check whether:

- (a) if $P \rightarrow Q$, $Q \rightarrow R$ and $P \wedge S$ are true, then $R \wedge S$ is true;
- (b) if R and $(P \wedge \neg R)$ are false and $\neg P \rightarrow \neg Q$ is true, then Q is false;
- (c) if $Q \rightarrow (R \wedge S)$ is true and $Q \wedge S$ is false, then $R \wedge Q$ is false;
- (d) if $P \rightarrow Q$ and $Q \rightarrow (R \vee S)$ are true and $P \rightarrow R$ is false, then S is true;
- (e) if $Q \rightarrow (R \wedge S)$ is true and $Q \wedge S$ is false, then Q is false.

④ Check the soundness of each of following inference rules.

- (a) Rule of propositional resolution:

$$\frac{p \vee r, \quad q \vee \neg r}{p \vee q}$$

- (b)

$$\frac{(p \vee q) \rightarrow r, \quad p \vee r}{r}$$

- (c)

$$\frac{p \rightarrow q, \quad \neg q \vee r, \quad \neg r}{\neg p}$$

(d)

$$\frac{p \rightarrow q, p \vee \neg r}{q \rightarrow r}$$

(e)

$$\frac{\neg p \rightarrow \neg q, \neg(p \wedge \neg r), \neg r}{\neg q}$$

(f)

$$\frac{p \rightarrow q, r \vee (s \wedge \neg q), q}{\neg p}$$

(g)

$$\frac{p \rightarrow q, q \rightarrow (r \vee s), \neg(p \rightarrow r)}{s}$$

⑤ Check the logical correctness of each of the following propositional arguments.

(a)

If the triangle ABC has a right angle, then it is not equilateral.

The triangle ABC does not have a right angle.

Therefore, the triangle ABC is equilateral.

(b) In the following argument n is a certain integer.

If n is divisible by 2 and n is divisible by 3, then n is divisible by 6.

If n is divisible by 6, then n is divisible by 2.

n is not divisible by 3.

Therefore n is not divisible by 6.

(c)

If Bill likes logic, then he is clever.

If Bill is clever, then he is rich.

Therefore, if Bill likes logic, then he is rich.

(d)

Socrates is wise or not happy.

If Socrates is not wise then Socrates is not a philosopher.

Therefore, if Socrates is a philosopher then Socrates is not happy.

(e)

If Christina is not a good student, then she is not clever or she is lazy.

If Christina is clever, then she is not lazy.

Therefore, if Christina is clever, then she is a good student.

(f)

If Victor studies hard and he is clever, then he will go to a university.

If Victor does not study hard, then he will not get a good job.

Therefore, if Victor is clever, then he will go to a university or will not get a good job.

(g)

Johnny will not learn logic if he skips lectures

Johnny will pass the exam if he learns logic and does not skip lectures.

Therefore if Johnny skips lectures or does not pass the exam then he will not learn logic.

- (h) Replace the second premise of the propositional argument in the previous question with "Johnny will pass the exam only if he learns logic and does not skip lectures" and determine whether the resulting argument is logically correct.

Early proof theory

The idea of proof as demonstrative argumentation goes back to **Aristotle**, but the idea of a formal proof as a sequence of precisely specified derivation steps in a formal system of logical and mathematical deduction was perhaps first suggested in the late 17th century by **Leibniz**, with his *calculus ratiocinator*. This idea has been systematically developed only since the late 19th century, first by **Frege** (see the box at the end of Section 4.1 in Chapter 4) and later mainly by **Hilbert** (see Section 3.3), in the context of his programme on the foundations of mathematics.

Hilbert's concept of a formal proof, or *derivation*, in an *axiomatic system* was a purely syntactic, mechanically implementable and verifiable procedure producing a sequence of formulae, derived by means of explicitly specified simple rules of inference by using axioms – formulae assumed to be given without derivation – or other already derived formulae, ending up with the desired conclusion. The main purpose was to avoid any paradoxes and contradictions in mathematics by doing all mathematical proofs within such a formal system of deduction, which could be proved to be *consistent*, i.e. free from contradictions. Such a system, based on first-order logic, was proposed by Hilbert and **Ackermann** in their book *Principles of Mathematical Logic* and Hilbert's ultimate goal was to prove that that system was both consistent and complete, i.e. capable of deriving every mathematical truth. Later, Hilbert's idea of axiomatic systems, as a paradigm of a *deductive system*, was extended by adding the possibility of making and withdrawing *assumptions* in derivations, i.e. formulae that are neither axioms nor have been derived. First versions of such systems, now called *natural deduction*, were independently developed by **Stanisław Jaśkowski** in 1929 and **Gerhard Gentzen** in 1934. Gentzen's system involved two kinds of inference rules: *introduction rules* and *elimination rules* for each logical connective, an idea that turned out to be very important in proof theory. In his thesis *Untersuchungen über das Logische Schliessen* (*Investigations into Logical Inference*), completed in 1933 under Bernays' supervision, he wrote that his aim was formal analysis of mathematical proofs as they occur in practice.

Gentzen's motivation was to provide a purely syntactic, combinatorial proof of the consistency of Peano's axiom system of arithmetic (see the box at the end of Section 4.2). In this he was partly

successful. In order to analyse derivations better and to prove a special property of derivations that could be used for the proof of consistency, Gentzen also introduced the *sequent calculus*, a deduction system based not just on formulae but on *sequents*—pairs of lists of formulae, assumptions on the left and conclusions on the right – with a set of rules for manipulating and deriving sequents.

A few years later, in 1931, Kurt Gödel (see the box at the end of Section 4.5) proved his Second Incompleteness Theorem, stating that no sufficiently expressive deductive system, such as Peano's system of arithmetic, can prove its own consistency – suitably encoded as a formula in that system – unless it is inconsistent (in which case it can derive *any formula* by using the sound logical rule *Ex Falso Quodlibet*), and therefore an absolute and finitary proof of consistency of such system is impossible. Thus, Hilbert's dream of formalizing the whole of mathematics was shattered and Gentzen's ultimate goal was proved unattainable. Still, proof theory flourished as one of the main branches of mathematical logic after Gentzen's ground-breaking work, and it has achieved a lot in analyzing the notion of derivation and in proving relative consistency results. The method of *semantic tableaux*, also known as *analytic tableaux*, was first introduced as a system of logical deduction in the mid 1950s, independently by the Dutch logician **Evert Beth** and the Finnish philosopher **Jaakko Hintikka**. It was further developed and simplified by the American logician, philosopher and magician, **Raymond Smullyan**, in the late 1960s. Since then, systems of semantic tableaux have also been developed for a great variety of *non-classical* logical systems.

3.5. Logical equivalences. Negating propositional formulae

3.5.1. Logically equivalent propositional formulae

Definition 3.5.1 Propositional formulae A and B are **logically equivalent**, which will be denoted by $A \equiv B$, if for every assignment of truth values to the variables occurring in them they obtain the same truth values.

Being a little imprecise (you will see why), we say that A and B are logically equivalent if they have the same truth tables. The following are examples of logically equivalent formulae:

- $p \vee \neg p \equiv \top$ (**excluded middle**) and $p \wedge \neg p \equiv \perp$ (**contradiction**).
- $p \equiv (\neg \neg p)$: a double negation of a proposition is equivalent to the proposition itself.
- $\neg(p \wedge q) \equiv (\neg p \vee \neg q)$ and $\neg(p \vee q) \equiv (\neg p \wedge \neg q)$. These are known as **De Morgan's laws**.

Let us check the first one, using the simplified version of truth tables:

p	q	\neg	$(p \wedge q)$	$\neg(p \wedge q)$	\neg	p	\vee	\neg	q
T	T	F	T	T	T	F	T	F	F
T	F	T	T	F	F	F	T	T	F
F	T	T	F	F	T	F	T	F	T
F	F	T	F	F	F	T	F	T	F

- $(p \wedge (p \vee q)) \equiv (p \wedge p)$. There is a small problem here: it is not quite clear what it means to say that the truth tables of these formulae are the same, for the first one contains two variables

(p and q) while the second one contains only p . We can always think that q occurs *vacuously* in the second formula and include it in its truth table:

p	q	$(p \wedge (p \vee q))$	$(p \wedge p)$
T	T	T	T
T	F	T	T
F	T	F	F
F	F	F	F

Warning: do not confuse the logical connective \leftrightarrow and logical equivalence between formulae. These are different things but there is a simple relation between them:

The propositions A and B are logically equivalent if and only if the proposition $A \leftrightarrow B$ is a tautology.

Indeed, both mean the same: A and B always take the same truth values.

Checking logical equivalence can be streamlined, just like checking validity or logical consequence, by searching systematically for a falsifying assignment: in order to check if $A \equiv B$ we try to construct an assignment of truth values to the variables occurring in A and B that renders one of them true and the other false. If such an assignment exists, the formulae are *not* logically equivalent; otherwise they are. For example, let us check the second of De Morgan's laws: $\neg(p \vee q) \equiv (\neg p \wedge \neg q)$. There are two possibilities for an assignment to falsify that equivalence:

- (i) $\neg(p \vee q) : T, (\neg p \wedge \neg q) : F$. Then $p \vee q : F$ and hence $p : F$, and $q : F$, but then $(\neg p \wedge \neg q) : T$ – a contradiction.
- (ii) $\neg(p \vee q) : F, (\neg p \wedge \neg q) : T$. Then $\neg p : T$ and $\neg q : T$ and hence $p : F$, and $q : F$, but then $\neg(p \vee q) : T$ – a contradiction again.

Thus, there is no such assignment; hence the logical equivalence holds.

The idea above can be formalized further, using the semantic tableaux, by reducing logical equivalence to a tautology:

The propositions A and B are logically equivalent if and only if $\models A \leftrightarrow B$.

Alternatively, sometimes it is better to apply the semantic tableaux by reducing logical equivalence to logical consequences:

The propositions A and B are logically equivalent if and only if $A \vDash B$ and $B \vDash A$.

This approach allows us to establish which of the two formulae implies the other, if any, in case the equivalence does not hold.

In the case of the example above, check that both tableaux, for $\neg(p \vee q) \vDash (\neg p \wedge \neg q)$ and for $(\neg p \wedge \neg q) \vDash \neg(p \vee q)$, close.

3.5.2. Some important equivalences

Let us now discuss some kinds of important and useful logical equivalences. Verifying them is an easy but useful exercise.

Equivalences expressing algebraic laws for the logical connectives

We begin with some important logical equivalences, which are used, for example for equivalent transformations of propositional formulae to so-called *conjunctive and disjunctive normal forms*, to be introduced later in Section 3.6.

- **Idempotency:** $p \wedge p \equiv p$ and $p \vee p \equiv p$.
- **Commutativity:** $p \wedge q \equiv q \wedge p$ and $p \vee q \equiv q \vee p$.
- **Associativity:** $(p \wedge (q \wedge r)) \equiv ((p \wedge q) \wedge r)$ and $(p \vee (q \vee r)) \equiv ((p \vee q) \vee r)$. This property allows us to omit the parentheses in multiple conjunctions and disjunctions.
- **Absorption:** $p \wedge (p \vee q) \equiv p$ and $p \vee (p \wedge q) \equiv p$.
- **Distributivity:** $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$ and $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$.

Equivalences mutually expressing logical connectives

The following logical equivalences can be used to define logical connectives in terms of others:

- $\neg A \equiv A \rightarrow \perp$. (We sometimes use this equivalence in colloquial expressions, such as “If this is true then I can fly”, meaning to say: “This cannot be true”.)
- $A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$. This equivalence allows us to consider the biconditional as a definable connective, which we will do often.
- $A \vee B \equiv \neg(\neg A \wedge \neg B)$.
- $A \wedge B \equiv \neg(\neg A \vee \neg B)$.
- $A \rightarrow B \equiv \neg A \vee B \equiv \neg(A \wedge \neg B)$.
- $A \vee B \equiv \neg A \rightarrow B$.
- $A \wedge B \equiv \neg(A \rightarrow \neg B)$.

Thus, we see that each of \wedge , \vee , \rightarrow can be expressed by means of any other of them, using negation.

Some simplifying equivalences

Other useful logical equivalences can be used to simplify formulae:

- $A \vee \neg A \equiv \top; A \wedge \neg A \equiv \perp$
- $A \wedge \top \equiv A; A \wedge \perp \equiv \perp$
- $A \vee \top \equiv \top; A \vee \perp \equiv A$
- $\neg A \rightarrow \neg B \equiv B \rightarrow A$ (Every implication is equivalent to its contrapositive.)

Negating propositional formulae. Negation normal form

In mathematical and other arguments we sometimes have to negate a formalized statement and then use the result in further reasoning. For that it is useful to transform, up to logical equivalence, the formula formalizing the statement into **negation normal form**, where negation may only occur in front of propositional variables. Such transformation can be done by importing all occurrences of negations inside the other logical connectives step by step, using the following equivalences, some of which we already know:

- $\neg\neg A \equiv A$
- $\neg(A \wedge B) \equiv \neg A \vee \neg B$
- $\neg(A \vee B) \equiv \neg A \wedge \neg B$
- $\neg(A \rightarrow B) \equiv A \wedge \neg B$
- $\neg(A \leftrightarrow B) \equiv (A \wedge \neg B) \vee (B \wedge \neg A)$

Example 3.5.2

Equivalent transformation to negation normal form:

$$\begin{aligned} & \neg((A \vee \neg B) \rightarrow (\neg C \wedge D)) \\ & \equiv (A \vee \neg B) \wedge \neg(\neg C \wedge D) \\ & \equiv (A \vee \neg B) \wedge (\neg\neg C \vee \neg D) \\ & \equiv (A \vee \neg B) \wedge (C \vee \neg D) \end{aligned}$$

3.5.3. Exercises

❶ Verify the following logical laws:

- Idempotency:* $p \wedge p \equiv p, p \vee p \equiv p$
- Commutativity:* $p \wedge q \equiv q \wedge p, p \vee q \equiv q \vee p$
- Associativity:* $(p \wedge (q \wedge r)) \equiv ((p \wedge q) \wedge r), (p \vee (q \vee r)) \equiv ((p \vee q) \vee r)$
- Absorption:* $p \wedge (p \vee q) \equiv p, p \vee (p \wedge q) \equiv p$
- Distributivity:* $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r), p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
- De Morgan's laws:* $\neg(p \wedge q) \equiv (\neg p \vee \neg q), \neg(p \vee q) \equiv (\neg p \wedge \neg q)$

❷ Prove the following logical equivalences:

- $(p \rightarrow q) \wedge (p \rightarrow r) \equiv p \rightarrow (q \wedge r)$
- $(p \rightarrow q) \vee (p \rightarrow r) \equiv p \rightarrow (q \vee r)$

- (c) $((p \vee q) \rightarrow r), (p \rightarrow r) \wedge (q \rightarrow r)$
- (d) $p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$
- (e) $\neg(p \leftrightarrow q) \equiv (p \wedge \neg q) \vee (q \wedge \neg p)$
- (f) $\neg(p \leftrightarrow q) \equiv \neg p \leftrightarrow q \equiv p \leftrightarrow \neg q$
- (g) $p \leftrightarrow q \equiv q \leftrightarrow p$
- (h) $p \leftrightarrow (q \leftrightarrow r) \equiv (p \leftrightarrow q) \leftrightarrow r$

③ Determine which of the following pairs of formulae are logically equivalent:

- (a) $p \rightarrow q, q \rightarrow p$
- (b) $p \rightarrow q, \neg p \rightarrow \neg q$
- (c) $p \rightarrow \neg q, q \rightarrow \neg p$
- (d) $\neg(p \rightarrow \neg q), p \wedge q$
- (e) $((p \rightarrow q) \rightarrow q) \rightarrow q, p \vee q$
- (f) $(p \rightarrow r) \wedge (q \rightarrow r), (p \wedge q) \rightarrow r$
- (g) $(p \rightarrow r) \vee (q \rightarrow r), (p \vee q) \rightarrow r$
- (h) $((p \wedge q) \rightarrow r), (p \rightarrow r) \vee (q \rightarrow r)$
- (i) $p \rightarrow (q \rightarrow r), (p \rightarrow q) \rightarrow r$
- (j) $p \leftrightarrow (q \leftrightarrow r), q \leftrightarrow (p \leftrightarrow r)$
- (k) $p \rightarrow (q \rightarrow r), (p \rightarrow q) \rightarrow (p \rightarrow r)$

④ Negate each of the following propositional formulae and transform the result to an equivalent formula in a negation normal form.

- (a) $(p \vee \neg q) \wedge \neg p$
- (b) $(p \rightarrow \neg q) \rightarrow p$
- (c) $p \rightarrow (\neg q \rightarrow p)$
- (d) $(p \leftrightarrow \neg q) \rightarrow \neg r$
- (e) $p \rightarrow (\neg q \leftrightarrow r)$



Augustus De Morgan (27.6.1806–18.3.1871)

Augustus De Morgan was a British mathematician, logician and a popularizer of mathematics. Influenced by George Boole (see Section 3.2), he pioneered the application of algebraic methods to the study of logic in the mid 19th century, thus becoming one of the founding fathers of modern mathematical logic. In particular, he was the first to formulate the logical equivalences now known as *de Morgan's laws*.

De Morgan was born in Madura, India, and became blind in one eye soon after his birth. He graduated from Trinity College, Cambridge, and in 1828 became the first Professor of Mathematics at the newly established University College in London, where he taught during most of his academic life.

He was an enthusiastic and prolific writer of over 700 popular articles in mathematics for the *Penny Cyclopaedia*, aimed at advancing broad education in mathematics in Britain. In one of them, in 1838, he formally introduced the term “mathematical induction” and made the so far informally used method of mathematical induction a precise mathematical technique. He also wrote books, such as *Trigonometry and Double Algebra* and *The Differential and Integral Calculus*. In 1847 he published his main work on mathematical logic, called *Formal Logic: The Calculus of Inference, Necessary and Probable*, which was used for a very long time and was last reprinted in 2003. De Morgan was also a passionate collector of mathematical puzzles and curiosities, paradoxes, etc., many of which he included in his book *A Budget of Paradoxes* published in 1872 (now digitalized and available on the Internet).

In 1866 De Morgan became one of the founders and the first President of the London Mathematical Society.

There is a crater on the Moon named De Morgan after him.

3.6. Normal forms. Propositional resolution

Here we will present another system for the derivation of logical consequences, which is particularly suitable for automated and computer-aided reasoning as it is based on a single rule of inference, called **(propositional) resolution**. Just like the semantic tableaux, it is sound and complete, i.e. it can derive all and only the valid logical consequences.

The Resolution Rule is very simple:

$$\frac{A \vee C, B \vee \neg C}{A \vee B},$$

where A, B, C are any propositional formulae. The formula $A \vee B$ is called a **resolvent** of $A \vee C$ and $B \vee \neg C$, and we write $A \vee B = \text{Res}(A \vee C, B \vee \neg C)$.

The Rule of Propositional Resolution is logically sound. Exercise 1.4(a) asked you to check that.

3.6.1. Conjunctive and disjunctive normal forms of propositional formulae

The Resolution Rule is particularly efficient when applied to formulae that are pre-processed into so-called *clausal form*. In order to define that we need some technical preparation.

Definition 3.6.1 First, we need some terminology:

1. A **literal** is a propositional variable or its negation.
2. An **elementary disjunction** (respectively **elementary conjunction**) is any literal or a disjunction (respectively conjunction) of two or more literals.
3. A **disjunctive normal form (DNF)** is any elementary conjunction or a disjunction of two or more elementary conjunctions.
4. A **conjunctive normal form (CNF)** is any elementary disjunction or a conjunction of two or more elementary disjunctions.

Example 3.6.2

Some examples illustrating these notions are:

- $p, \neg q, p \vee \neg q, p \vee \neg p \vee q \vee \neg r$ are elementary disjunctions;
- $p, \neg q, \neg p \wedge q, \neg p \wedge q \wedge \neg r \wedge \neg p$ are elementary conjunctions;
- $p, \neg q, p \wedge \neg q, p \vee \neg q, (p \wedge \neg p) \vee \neg q, (r \wedge q \wedge \neg p) \vee (\neg q \wedge p) \vee (\neg r \wedge p)$ are disjunctive normal forms;
- $p, \neg q, p \wedge \neg q, p \vee \neg q, p \wedge (\neg p \vee \neg q), (r \vee q \vee \neg r) \wedge \neg q \wedge (\neg p \vee r)$ are conjunctive normal forms.

The normal forms are convenient for various symbolic manipulations and logical computations. Also, they are used for construction and optimization of logical circuits in computer systems design. That is why the following fact is quite important.

Theorem 3.6.3 Every propositional formula is equivalent to a disjunctive normal form and to a conjunctive normal form.

We will give two methods for constructing such equivalent normal forms.

Method 1 This method is based on the following algorithm, which transforms any formula into a DNF, respectively CNF.

1. Eliminate all occurrences of \leftrightarrow and \rightarrow using the logical equivalences

$$A \rightarrow B \equiv \neg A \vee B,$$

$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A).$$

2. Import all negations to stand directly in front of the propositional variables, using the logical equivalences listed above.
3. For DNF: distribute all conjunctions over disjunctions using the distributive law: $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$.
4. Respectively, for CNF: distribute all disjunctions over conjunctions using the other distributive law: $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$.

Throughout this process the formulae can be simplified by using commutativity, associativity and idempotency of \vee and \wedge , as well as:

$$p \vee \neg p \equiv \top, p \wedge \neg p \equiv \perp, ; p \wedge \top \equiv p, p \wedge \perp \equiv \perp; p \vee \top \equiv \top, p \vee \perp \equiv p.$$

Example 3.6.4

$$\begin{aligned} (p \wedge \neg r) \rightarrow (p \leftrightarrow \neg q) &\equiv \neg(p \wedge \neg r) \vee ((p \rightarrow \neg q) \wedge (\neg q \rightarrow p)) \\ &\equiv (\neg p \vee \neg \neg r) \vee ((\neg p \vee \neg q) \wedge (\neg \neg q \vee p)) \\ &\equiv \neg p \vee r \vee ((\neg p \vee \neg q) \wedge (q \vee p)) \end{aligned}$$

For a DNF we further distribute \wedge over \vee and simplify:

$$\begin{aligned} &\equiv \neg p \vee r \vee (((\neg p \vee \neg q) \wedge q) \vee ((\neg p \vee \neg q) \wedge p)) \\ &\equiv \neg p \vee r \vee ((\neg p \wedge q) \vee (\neg q \wedge q)) \vee ((\neg p \wedge p) \vee (\neg q \wedge p)) \\ &\equiv \neg p \vee r \vee ((\neg p \wedge q) \vee \perp) \vee (\perp \vee (\neg q \wedge p)) \\ &\equiv \neg p \vee r \vee (\neg p \wedge q) \vee (\neg q \wedge p). \end{aligned}$$

For a CNF we distribute \vee over \wedge and simplify:

$$\begin{aligned} &\equiv (\neg p \vee r \vee \neg p \vee \neg q) \wedge (\neg p \vee r \vee q \vee p) \\ &\equiv (\neg p \vee r \vee \neg q) \wedge (\top \vee r \vee q) \\ &\equiv (\neg p \vee r \vee \neg q) \wedge \top \\ &\equiv \neg p \vee r \vee \neg q. \end{aligned}$$

As we can see, in this case the CNF turns out to be a DNF as well, even simpler than the one we obtained above. The problem of *minimization* of normal forms, which is of practical importance, will not be discussed here.

Method 2 This second method constructs the normal forms directly from the truth table of the given formula. We will outline it for DNF.

Given the truth table of the formula A we consider all rows (i.e. all assignments of truth values to the occurring variables) where the truth value of A is true. If there are no such rows, the formula is a contradiction and a DNF for it is, for example, $p \wedge \neg p$. Otherwise, with every such row we associate an elementary conjunction in which all variables assigned value T occur positively, while those assigned value F occur negated. For instance, the assignment F, T, F to the variables p, q, r is associated with the elementary conjunction $\neg p \wedge q \wedge \neg r$. Note that such an elementary conjunction is true only under the assignment with which it is associated.

Example 3.6.5

The formula $p \leftrightarrow \neg q$ has a truth table

p	q	$p \leftrightarrow \neg q$
T	T	F
T	F	T
F	T	T
F	F	F

The corresponding DNF is $(p \wedge \neg q) \vee (\neg p \wedge q)$. Check this!



3.6.2. Clausal form. Clausal resolution

Definition 3.6.6 (Clausal form) Some basic notions related to clausal forms:

1. A **clause** is essentially an elementary disjunction $l_1 \vee \dots \vee l_n$ but written as a set of literals $\{l_1, \dots, l_n\}$.
2. The **empty clause** is a clause containing no literals; a **unit clause** is a clause containing only one literal.
3. A **clausal form** is a (possibly empty) set of clauses, written as a list: $C_1 \dots C_k$. It represents the conjunction of these clauses.

Thus, every CNF can be rewritten in a clausal form, and therefore every propositional formula is equivalent to one in a clausal form.

For instance, the clausal form of the CNF formula $(p \vee \neg q \vee \neg r) \wedge \neg p \wedge (\neg q \vee r)$ is $\{p, \neg q, \neg r\} \{\neg p\} \{\neg q, r\}$.

The Resolution Rule can be rewritten for clauses as follows:

$$\frac{\{A_1, \dots, C, \dots, A_m\} \{B_1, \dots, \neg C, \dots, B_n\}}{\{A_1, \dots, A_m, B_1, \dots, B_n\}}.$$

The clause $\{A_1, \dots, A_m, B_1, \dots, B_n\}$ is a **resolvent** of the clauses $\{A_1, \dots, C, \dots, A_m\}$ and $\{B_1, \dots, \neg C, \dots, B_n\}$.

Example 3.6.7

Some examples of applications of clausal resolution:

$$\frac{\{p, q, \neg r\} \{\neg q, \neg r\}}{\{p, \neg r\}},$$

$$\frac{\{\neg p, q, \neg r\} \{r\}}{\{\neg p, q\}},$$

$$\frac{\{\neg p\} \{p\}}{\{\}}.$$

Note that two clauses can have more than one resolvents, e.g.:

$$\frac{\{p, \neg q\} \{\neg p, q\}}{\{p, \neg p\}}, \quad \frac{\{p, \neg q\} \{\neg p, q\}}{\{\neg q, q\}}.$$

However, it would be *wrong* to apply the Resolution Rule for both pairs of complementary literals concurrently and obtain

$$\frac{\{p, \neg q\} \{\neg p, q\}}{\{\}}.$$

Indeed, check that the above inference is not logically correct.

3.6.3. Resolution-based derivations

The idea behind the method of resolution is similar to that of semantic tableaux. In order to determine whether a logical consequence $A_1, \dots, A_n \models B$ holds using the method of resolution, we negate the conclusion B and transform each of the formulae $A_1, \dots, A_n, \neg B$ into clausal form. Then we test whether the resulting set of clauses is unsatisfiable, by looking for a resolution-based derivation of the empty clause from that set of clauses. Formally, this can be defined as follows.

Definition 3.6.8 *A resolution-based derivation of a formula B from a list of formulae A_1, \dots, A_n is a derivation of the empty clause $\{\}$ from the set of clauses obtained from the formulae $A_1, \dots, A_n, \neg B$, by successive applications of the Rule of Propositional Resolution.*

Example 3.6.9

Using Propositional Resolution, check the logical consequence $p \rightarrow q, q \rightarrow r \models p \rightarrow r$.

Solution

First, we transform each of $p \rightarrow q, q \rightarrow r, \neg(p \rightarrow r)$ to clausal form. The resulting set of clauses is:

$$C_1 = \{\neg p, q\}, C_2 = \{\neg q, r\}, C_3 = \{p\}, C_4 = \{\neg r\}.$$

Now we apply the Resolution Rule successively:

$$C_5 = \{q\} = \text{Res}(C_1, C_3);$$

$$C_6 = \{r\} = \text{Res}(C_2, C_5);$$

$$C_7 = \{\} = \text{Res}(C_4, C_6).$$

The derivation of the empty clause completes the proof that $p \rightarrow q, q \rightarrow r \models p \rightarrow r$.

Example 3.6.10

Check whether $(\neg p \rightarrow q), \neg r \models p \vee (\neg q \wedge \neg r)$ holds.

Solution

First transform $(\neg p \rightarrow q), \neg r, \neg(p \vee (\neg q \wedge \neg r))$ to clausal form:

$$C_1 = \{p, q\}, C_2 = \{\neg r\}, C_3 = \{\neg p\}, C_4 = \{q, r\}.$$

Now, applying resolution successively:

$$C_5 = \text{Res}(C_1, C_3) = \{q\};$$

$$C_6 = \text{Res}(C_2, C_4) = \{q\}.$$

At this stage, no new applications of the Propositional Resolution Rule are possible hence the empty clause is not derivable. Therefore, $(\neg p \rightarrow q), \neg r \not\models p \vee (\neg q \wedge \neg r)$.

Remark 3 Remember that, even though the clauses are represented as lists of literals, they are actually sets, so if a literal occurs more than once in a clause, the repeated occurrences may be removed with no harm. In fact, such repeated occurrences must be removed whenever they occur, else they may prevent the derivation of the empty clause. For instance, the set of clauses $\{p, q\}, \{\neg p, q\}, \{p, \neg q\}, \{\neg p, \neg q\}$ is not satisfiable but the empty clause cannot be derived without deletion of repeating literals in the resolvents. Check this!

3.6.4. Optimizing the method of resolution

The method of resolution is amenable to various optimizations by means of special additional rules and derivation strategies that we will not discuss here. We only mention some simple rules that remove redundant clauses:

- *Deletion of repeating literals*: we have already discussed this as a necessary processing rule. For instance, we derive

$$\frac{\{p, \neg q, \neg r\} \{q, \neg r\}}{\{p, \neg r\}} \text{ instead of } \frac{\{p, \neg q, \neg r\} \{q, \neg r\}}{\{p, \neg r, \neg r\}}.$$

- *Tautology deletion*: if a clause contains a complementary pair of literals, then it is a tautology and therefore is of no use for deriving a contradiction, i.e. the empty clause, so it can be removed.
- *Subsumption deletion*: if a clause C contains (as a set of literals) a clause C' , then it is *subsumed* by C' , and hence can be safely removed from the set of clauses, because every derivation of the empty clause using C can be reduced to a possibly shorter derivation using C' instead.
- *Removal of all clauses containing a monopolar literal*: if a clause C contains a literal l such that its complementary literal \bar{l} does not occur in any clause of the current set, then C can be removed from the set, because every derivation of the empty clause using C can be simplified to a possibly shorter derivation not using C . The reason is simple: the literal l will be inherited in

every resolvent clause produced by using C as a parent clause because the complementary literal \bar{l} can never appear in a clause produced from the current clause set.

3.6.5. Exercises

Some of the following exercises deal with Propositional Resolution. They are intended only for the Computer science track.

- ❶ Prove that the Rule of Propositional Resolution is logically sound.
- ❷ Show that the disjunction of all elementary conjunctions associated with the rows in the truth table of a formula A is logically equivalent to A .
- ❸* Outline a similar method for construction of a CNF from the truth table of a formula as in Example 3.6.5. Prove your claim.
- ❹ Construct a DNF and a CNF, equivalent to each of the following formulae, using both methods.
 - (a) $\neg(p \leftrightarrow q)$
 - (b) $((p \rightarrow q) \wedge \neg q) \rightarrow p$
 - (c) $(p \leftrightarrow \neg q) \leftrightarrow r$
 - (d) $p \rightarrow (\neg q \leftrightarrow r)$
 - (e) $(\neg p \wedge (\neg q \leftrightarrow p)) \rightarrow ((q \wedge \neg p) \vee p)$
- ❺ Using the method of resolution check which of the following formulae are tautologies.
 - (a) $((p \rightarrow q) \rightarrow q) \rightarrow q$
 - (b) $((q \rightarrow p) \rightarrow q) \rightarrow q$
 - (c) $((p \rightarrow q) \wedge (p \rightarrow \neg q)) \rightarrow \neg p$
 - (d) $((p \vee q) \rightarrow \neg r) \rightarrow \neg(\neg q \wedge r)$
 - (e) $((p \rightarrow q) \wedge (p \rightarrow r)) \rightarrow (p \rightarrow (q \wedge r))$
 - (f) $((p \rightarrow r) \wedge (q \rightarrow r)) \rightarrow ((p \wedge q) \rightarrow r)$
 - (g) $((p \rightarrow r) \vee (q \rightarrow r)) \rightarrow ((p \vee q) \rightarrow r)$
 - (h) $((p \rightarrow r) \wedge (q \rightarrow r)) \rightarrow ((p \vee q) \rightarrow r)$
 - (i) $p \rightarrow ((q \rightarrow r) \rightarrow ((p \rightarrow q) \rightarrow r))$
 - (j) $(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$
- ❻ Using the method of resolution check the following logical consequences.
 - (a) $\neg p \rightarrow q, \neg p \rightarrow \neg q \models p$
 - (b) $p \rightarrow r, q \rightarrow r \models (p \vee q) \rightarrow r$
 - (c) $(p \wedge q) \rightarrow r \models (q \rightarrow r) \vee (p \rightarrow r)$

- (d) $p \rightarrow q, p \vee \neg r, \neg r \models \neg q \vee r$
- (e) $(p \vee q) \rightarrow r \models (p \rightarrow r) \wedge (q \rightarrow r)$
- (f) $(\neg p \wedge q) \rightarrow \neg r, r \models p \vee \neg q$
- (g) $p \rightarrow q, r \vee (s \wedge \neg q), \models \neg r \rightarrow \neg p$
- (h) $p \rightarrow q, q \rightarrow (r \vee s), \neg(p \rightarrow r) \models s$

The Boolean satisfiability problem and NP-completeness

The Boolean satisfiability (SAT) is the problem of deciding whether a given propositional (Boolean) formula is satisfiable, i.e. whether there is a truth assignment to the propositional (Boolean) variables occurring in the formula that makes it true. It is probably the most popular *NP-complete decision problem*, the meaning of which we will explain here.

Firstly, NP stands for “nondeterministic polynomial” time. An algorithmic decision problem (which requires an answer Yes/No for every input) is in the *complexity class NP* if, intuitively, there is a fixed polynomial f such that for every input where the answer is Yes there is a piece of evidence (proof, or witness) that can be verified “quickly and efficiently”, which is assumed to mean *in polynomial time*, i.e. in a number of steps that is bounded by the value $f(n)$ where n is the length of the input. Equivalently, and again intuitively, a problem is in NP if there is an algorithm that can run on a hypothetical *nondeterministic* computing device (e.g. a nondeterministic Turing machine) to always solve the problem and, when the answer is Yes, to produce that answer in a polynomially bounded amount of time. Simply put, an NP problem may be computationally difficult (slow) to *find* a solution but it is easy (quick) to *check* whether a proposed, or guessed, solution is correct.

Of course, every decision problem that can be solved (for any answer) by an algorithm that works in *deterministic polynomial time* – i.e. one that runs on a usual digital computer and always takes a number of steps that is bounded by the value of a fixed polynomial in the length of the input – is in NP. Besides, there may be problems that are in NP but for which there is no algorithm that can solve them in polynomial time. The SAT problem seems to be a case in point. Indeed, it is in NP because whenever a Boolean formula is satisfiable, any given satisfying truth assignment can be verified very quickly, in a number of steps that is linear in length of the input formula. Alternatively, the SAT problem can be solved efficiently with a nondeterministic computing device that guesses a satisfying assignment, if there is one, and then verifies in linear time that it does indeed work. Equivalently, one can think of solving SAT by using a hypothetical *unboundedly parallel* computing device, simply by checking *simultaneously for all truth assignments* whether any of them renders the formula true.

On the other hand, there is *no currently known algorithm*, run on a usual digital computer, that can *always* solve the SAT problem efficiently, i.e. in polynomial time. Indeed, all methods for solving the SAT problem that are known so far – including truth tables, semantic tableau and propositional resolution, and many more modern, extremely clever and practically efficient algorithms for solving SAT – require *in the worst case* a number of steps that is exponential in the number of variables occurring in the input formula, and therefore exponential in the length of the formula. This is so because there are 2^n possible truth assignments for a set of n propositional variables.

There are many hundreds of other very important algorithmic problems that are in NP, including problems of scheduling, colouring of maps with a given number of colours, finding the prime factorization of a given integer, etc.

Some of the problems in NP, such as SAT, are also known to be *NP-complete*. An algorithmic decision problem X is NP-complete if solving any problem that is in NP can be reduced efficiently (i.e. in a number of steps bounded by some fixed polynomial in the length of the input) to solving the problem X. Therefore, any algorithm solving an NP-complete problem can be transformed into an at most “polynomially slower” (i.e. not *much* slower) algorithm solving any given problem in NP. Therefore, if an efficient (always using a number of steps polynomial in the size of the input) algorithm solving SAT is found, it can be suitably modified to solve efficiently any other problem in NP.

Can there be an efficient (always working in a polynomial number of steps) algorithm solving SAT, or any other NP-complete problem? This has been an open problem since 1971 when it was first stated by Stephen Cook in his seminal paper “*The complexity of theorem proving procedures*”. It is now literally a **million dollar question**, being one of the *7 Millennium Prize Problems* stated by the Clay Mathematics Institute in 2000, each bearing a 1 000 000 US\$ prize tag for the first published solution that is officially accepted as correct by the scientific community. The problem is known as “P=NP?” and is considered to be currently the most challenging and important unsolved problem in computer science. Most computer scientists believe that such miracle algorithm, solving NP-complete problems in polynomial time, does not exists at all.

4

First-Order Logic

4.1.	Basic concepts of first-order logic	136
4.2.	The formal semantics of first-order logic	152
4.3.	The language of first-order logic: a deeper look	161
4.4.	Truth, logical validity, equivalence and consequence in first-order logic	171
4.5.	Semantic tableaux for first-order logic	185
4.6.	Prenex and clausal normal forms	195
4.7.	Resolution in first-order logic	201
4.8.	Applications of first-order logic to mathematical reasoning and proofs	211

Propositional logic can formalize simple patterns of reasoning, but it is certainly insufficient for knowledge representation and practical reasoning. In this chapter we will extend it to the system of **first-order logic**, which is essentially sufficient to express any common or mathematical statements and therefore provides an adequate framework for logical reasoning. That is why it is usually regarded as *the language of mathematics*.

First-order logic builds on propositional logic by using *constants* and *variables* for objects of any nature (numbers, sets, points, human beings, etc.), *functions* and *predicates* over objects, as well as *quantifiers* over them.

Here we will present and discuss the basics of first-order logical languages and structures, and the fundamental notions of truth of a first-order formula in a structure, as well as validity, logical consequence and logical equivalence of first-order formulae. We will discuss the use of first-order languages for knowledge representation with emphasis on translations between natural language and first-order logic.

First-order logical consequence can be established by using deductive systems, like in propositional logic. We will extend the systems of semantic tableaux and resolution to sound and complete systems for first-order logic. Unlike the propositional case, however, *no sound and complete deductive system for first-order logic can be designed to always terminate its search for a derivation*, even when such a derivation exists. In fact, it was proved in 1936 by one of the most influential logicians of the 20th century, Alonso Church, that the problem of whether a given first-order logical consequence is valid is *not algorithmically decidable*.

At the end of the chapter we discuss and illustrate how first-order logic can be used for structuring correct mathematical reasoning.

4.1. Basic concepts of first-order logic

As we noted in Chapter 1, propositional logic cannot grasp the logical structure of sentences that involve variables, functions, predicates and quantification. All these can be captured in *first-order (classical) logic* (also known as *predicate logic*), the basic concepts of which we shall introduce and discuss briefly here. First-order logic has two major aspects:

- precise *syntax* involving a formal language, called a *first-order language*, that enables us to express statements in a uniform way, by means of *formulae*;
- formal *semantics*, specifying the meaning of all components of the language by means of their *interpretation* in a mathematical structure, and formal *truth definitions*, extending the truth tables for the propositional connectives.¹

Here we will discuss in a rather informal manner the basic components of the syntax and semantics of predicate logic and their relevance to mathematical reasoning.

4.1.1. First-order structures

First-order logic can be used to formalize statements and reasoning in just about any mathematical or nonmathematical situation, by representing them as appropriate (*first-order structures*). First we will discuss the basic components of a structure, and then will give a more formal definition.

- **Domains of discourse.** When we reason and make statements about objects, we have in mind a certain domain of discourse. In mathematics, that domain usually consists of *numbers* (integers, rationals, reals, etc.), *vectors*, *geometric figures* (points, lines, triangles, spheres, etc.), *sets*, *graphs*, etc. A domain in a nonmathematical discourse may consist of material objects, human beings, ideas or anything else. In either case, it is important that we have specified our domain of discourse, so that we know what we are talking about. Moreover, as one of the sentences mentioned above suggests, the same statement can be true or false, depending on the domain in which it is considered.

¹ Those who have some experience with computer programming languages should find the concepts of syntax and semantics familiar.

- **Predicates.** When we reason about the objects from our domain of discourse, we usually make statements about properties they have or do not have. For instance, talking about integers, we may discuss properties like being “positive”, “divisible by 3”, “not greater than 1999”, etc. Triangles can be “obtuse”, “equilateral”, etc. A human being can be “female”, “male”, “young”, etc.

The logical term for a property is *predicate*. Predicates need not only concern a single object, like all the examples mentioned above. Those are called *unary predicates*. We also deal with *binary predicates*, relating two objects, such as: “_ is less than _”, “_ is divisible by _” (for numbers) or “_ is a son of _”, “_ loves _” (for humans), etc.; *ternary predicates*, relating three objects, such as “_ is between _ and _” (for points on a line), “_ and _ are the parents of _”, (for humans), etc. In general, we can talk about *n-ary predicates*, relating *n* objects.

Note that, as long as we specify the meaning (semantics) of a predicate and the objects that it relates, that predicate becomes true or false, i.e. represents a proposition. For example, the propositions “12 is divisible by 6” and “13 is divisible by 7” are both instances of the binary predicate “_ is divisible by _”. Furthermore, we can connect predicates, by using propositional connectives, in compound propositions, just as we did earlier with propositions.

- **Functions.** Typically in mathematics (and not only there) we use *functions* to represent operations that, applied to one or several objects, determine an object. Depending on the number of arguments, we talk about *unary*, *binary*, etc., functions. Standard examples are all arithmetic operations and, more generally, all algebraic functions we have studied at school; and so are “the mother of _”, “the older grandfather of _”, etc., in the domain of humans.

Note that in our logical framework all functions will be considered *total*, i.e. defined for all possible values of the argument or tuple of arguments in the domain. This is quite often not the case, e.g. the subtraction in the domain of natural numbers, division in the domain of reals or the function “the daughter of _” in the domain of humans. This problem has an easy (albeit artificial) fix, good enough for our formal purposes: we designate an element *u* (for “*undefined*”) from the domain and make the function total by assigning *u* to be its value for all (tuples of) arguments for which it is not defined. For instance, we can extend division by 0 in the domain of reals by putting $x/0 = 17$ for any real *x*. This may seem reckless, but if proper care is taken when reasoning about division, it will not lead to confusion or contradiction. There will be more on this later.

- **Constants.**² Some objects in the domain can be distinguished in a way that would allow us to make direct references to them, by giving them *names*. Such distinguished objects are called *constants*. Examples in the domain of real numbers are 0, 1, π , *e*, etc. Note that the notion of a constant pertains to the language rather than to the domain. For instance, in common calculus we do not have a special name for the least positive solution of the equation $\cos x = x$, but if for some purposes we need to refer directly to it, then we can extend our mathematical language by giving it a name, i.e. make it a constant in our domain.

Now we are ready to define the notion of a *structure*.

Definition 4.1.1 *A (first-order) structure consists of a nonempty domain and a family of distinguished functions, predicates and constants in that domain.*

² Note that the use of the word “constant” in first-order logic is not exactly the same as in the common mathematical language.

Here are some examples of structures, which will often be used further:

- \mathcal{N} , with domain being the set of natural numbers \mathbb{N} , the unary function s (successor, i.e. $s(x) = x + 1$), the binary functions $+$ (addition) and \times (multiplication), the predicates $=$, $<$ and $>$, and the constant 0.
- \mathcal{Z} : likewise, but with the domain being the set of integers \mathbb{Z} and the additional function $-$ (subtraction).
- With the same functions and predicates we take the domain to be the set of rational numbers \mathbb{Q} or the reals \mathbb{R} . The obtained structures will be denoted by \mathcal{Q} and \mathcal{R} respectively.
- For those familiar with some abstract algebra, algebraic structures such as groups, rings, fields are all examples of first-order structures. Usually, algebraic structures only involve functions and constants, and no predicates except $=$ and possibly $<$.
- \mathcal{H} : the domain is the set of all humans, with functions m (for “the mother of”) and f (for “the father of”), unary predicates M (for “man”) and W (for “woman”), binary predicates P (for “parent of”), C (for “child of”), L (for “loves”), and constants (names), e.g. John, Mary, Adam, Eve.
- \mathcal{G} : the domain is the set of all points and lines in the plane, with unary predicates P for “point”, L for “line” and the binary predicate I for “incidence” between a point and a line.
- \mathcal{S} : the domain is the collection of all sets, with binary predicates $=$ for equality and \in for membership.³

4.1.2. First-order languages

In order to refer to objects, functions and predicates in our domain, we give them *names*. In mathematics, they are typically symbols or abbreviations, such as $5, \pi, +, \sqrt{}, \sin, =, <$; the proper names for humans, etc. Together with logical connectives, variables and some auxiliary symbols, these symbolic names determine formal logical languages called *first-order languages*. In particular, with every structure S we can associate a **first-order language** \mathcal{L}_S for that structure, containing:

1. **Functional, predicate and constant symbols**, used as names for the functions, predicates and constants we consider in the structure. All these are referred to as **nonlogical symbols**.
2. **Individual variables**. Quite often, particularly in mathematics, we deal with unknown or unspecified objects (individuals) from the domain of discourse. In order to be able to reason and make statements about such objects, we use *individual variables* to denote them. For instance, talking about numbers, we use phrases like “Take a positive integer n ”, “For every real number x greater than $\sqrt{2} \dots$ ”, etc. In natural language instead of variables we usually use pronouns or other syntactic constructions but that often leads to awkwardness or even ambiguity (e.g. “If a man owes money to another man then that man hates the other man”). Therefore the use of variables in mathematics is indispensable and it is very important to learn how to use them properly.

³ Strictly speaking, this is not a structure because the collection of all sets is not a proper set, but we can obtain a structure if we relativize it to the family of all subsets of a given set.

We also use variables as *placeholders* for the arguments of the various predicates and functions we deal with. Thus, we can talk about the (unary) functions, such as $f(x) = x + 1$ and $m(z) = \text{"the mother of } z\text{"}$, or the (binary) predicates $x < y$ and $\text{"}x \text{ loves } y\text{"}$, etc. Usually, for variables we will be using letters u, v, w, x, y, z , possibly indexed.

3. **Auxiliary symbols**, such as (), etc.
4. Last and most important, **logical symbols**, including:
 - (a) **propositional connectives**, which we already know, and
 - (b) **quantifiers**. Often we use specific phrases to *quantify* over objects of our discourse, such as:⁴

"(for) every (object) x ",
 "there is (an object) x such that",
 "for most (objects) x ",
 "there are at least 5 (objects) x such that",
 "there are more (objects) x than y such that",
 etc.

The first two of these are particularly important in mathematics and many others can be expressed by means of them, so they are given special names and notation:

The quantifier "for every" is called the **universal quantifier**, denoted by \forall .

The quantifier "there exists" is called the **existential quantifier**, denoted by \exists .

These are the only quantifiers used in first-order languages, so they are called **first-order quantifiers**.⁵

Besides the phrases above, the universal quantifier is usually represented by "all", "for all" and "every" while the existential quantifier can appear as "there is", "some" and "for some", particularly, in a nonmathematical discourse.

4.1.3. Terms and formulae

Using all these components, and following certain syntactic rules, we can compose formal expressions that allow us to represent mathematical statements symbolically, to reason about them and to prove them in a precise, well-structured and logically correct way.

There are two basic syntactic categories in a first-order language: *terms* and (*first-order*) *formulae*.

Terms

Terms are formal expressions (think of algebraic expressions) built from constant symbols and individual variables, using functional symbols. Terms are used to denote specified or unspecified *individuals*, i.e. elements of the domain.

⁴ Note the use of variables here.

⁵ The term "first-order" refers to the fact that quantification in these languages is only allowed over individuals in the universe of discourse – called "first-order objects" – but not over more complex objects such as sets, relations, functions, etc.

Here is the formal *inductive* definition of **terms of the first-order language \mathcal{L}** :

1. Every constant symbol in \mathcal{L} is a term.
2. Every individual variable in \mathcal{L} is a term.
3. If t_1, \dots, t_n are terms and f is an n -ary functional symbol in \mathcal{L} , then $f(t_1, \dots, t_n)$ is a term in \mathcal{L} .

We denote the set of terms of \mathcal{L} by $\text{TM}(\mathcal{L})$. The set of variables occurring in a term t is denoted by $\text{VAR}(t)$. Terms that do not contain variables are called **constant terms**, or **ground terms**.

Examples

1. In the language $\mathcal{L}_{\mathcal{N}}$: to start with, 0 , x , $s(0)$, $s(s(0))$, ..., are all ground terms. We will denote the term $s(\dots s(0)\dots)$, where s occurs n times, by \mathbf{n} .⁶

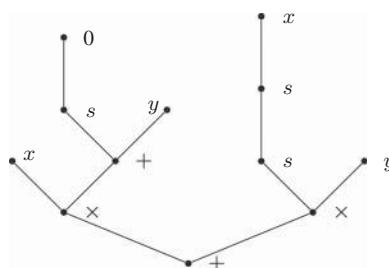
More examples of terms are:

- $+(2, 2)$, which in a more familiar infix notation written as $2 + 2$;
- $3 \times y$ (written in the usual infix notation);
- $(x^2 + x) + 5$, where x^2 is an abbreviation of $x \times x$;
- $x_1 + s((y_2 + 3) \times s(z))$, etc.

2. In the “human” language $\mathcal{L}_{\mathcal{H}}$:

- Mary;
- x ;
- $m(\text{John})$ (intended meaning: “*the mother of John*”);
- $f(m(y))$ (intended meaning: “*the father of the mother of y*”), etc.

With every term we can associate a construction tree and a parsing tree, just like with propositional formulae. For instance, here is the parsing tree of the term $t = +(\times(x, +(\mathbf{1}, y)), \times(s(s(x)), y))$ (recall that $\mathbf{1} = s(\mathbf{0})$):



Respectively, for every term t we define the set $\text{sub}(t)$ of **subterms** being all terms used in the construction of that term, i.e. all terms with construction trees rooted as subtrees at

⁶One should distinguish *the term \mathbf{n}* , which is a syntactic object, just a string of symbols, from *the number n*, which is a mathematical entity.

nodes of the construction tree of t . For instance, for the term t in the example above $\text{sub}(t) = \{t, \times(x, +(\mathbf{1}, y)), x, +(\mathbf{1}, y), \mathbf{1}, \mathbf{0}, y, \times(s(s(x)), y), s(s(x)), s(x)\}$.

Atomic formulae

Applying predicate symbols to terms we can build *atomic formulae*. These are the simplest first-order formulae. They have no internal logical structure and correspond to atomic propositions in propositional logic. Here is the formal definition:

If t_1, \dots, t_n are terms in a language \mathcal{L} and p is an n -ary predicate symbol in \mathcal{L} , then $p(t_1, \dots, t_n)$ is an **atomic formula** in \mathcal{L} .

We denote the set of atomic formulae of \mathcal{L} by $\text{AFOR}(\mathcal{L})$.

Examples

1. In $\mathcal{L}_{\mathcal{N}}$, using the traditional infix notation:

- $\mathbf{1} < \mathbf{2}$, or in formal notation: $<(\mathbf{1}, \mathbf{2})$,
- $x = \mathbf{2}$,
- $\mathbf{5} < (x + \mathbf{4})$,
- $\mathbf{2} + s(x_1) = s(s(x_2))$,
- $(x^2 + x) + \mathbf{5} = \mathbf{0}$,
- $x \times (y + z) = x \times y + x \times z$, etc.

2. In $\mathcal{L}_{\mathcal{H}}$:

- $x = m(\text{Mary})$ (intended meaning: “ x is the mother of Mary”).
- $L(f(y), y)$ (intended meaning: “the father of y loves y ”), etc.

Formulae

Finally, using atomic formulae and logical connectives we can build compound formulae, just like in propositional logic, except that now we can also use quantifiers.

The inductive definition of **formulae of a first-order language** \mathcal{L} naturally extends the definition of propositional formulae:

1. Every atomic formula in \mathcal{L} is a formula in \mathcal{L} .
2. If A is a formula in \mathcal{L} then $\neg A$ is a formula in \mathcal{L} .
3. If A, B are formulae in \mathcal{L} then $(A \vee B), (A \wedge B), (A \rightarrow B), (A \leftrightarrow B)$ are formulae in \mathcal{L} .
4. If A is a formula in \mathcal{L} and x is a variable, then $\forall x A$ and $\exists x A$ are formulae in \mathcal{L} .

We denote the set of formulae of \mathcal{L} by $\text{FOR}(\mathcal{L})$. The set of variables occurring in a formula A is denoted by $\text{VAR}(A)$. Formulae that do not contain variables are called **ground formulae**.

Examples

1. In \mathcal{L}_Z :

- $(5 < x \wedge x^2 + x - 2 = 0)$;
- $\exists x(5 < x \wedge x^2 + x - 2 = 0)$;
- $\forall x(5 < x \wedge x^2 + x - 2 = 0)$;
- $(\exists y(x = y^2) \rightarrow (\neg x < 0))$;
- $\forall x(\exists y(x = y^2) \rightarrow (\neg x < 0))$; etc.

2. In \mathcal{L}_H :

- $\text{John} = f(\text{Mary}) \rightarrow \exists x L(x, \text{Mary})$;
- $\exists x \forall z (\neg L(z, y) \rightarrow L(x, z))$;
- $\forall x (\exists y (x = m(y)) \rightarrow \exists z (L(z, x)))$.

Exercise 4.1.2 What is the intended meaning of each of these formulae?

Some syntactic conventions

As in propositional logic, we will allow ourselves to omit parentheses wherever possible without producing ambiguity. For that purpose we will, again, impose a priority order amongst the logical connectives:

- the unary connectives: negation and quantifiers have the strongest binding power, i.e. the highest priority,
- then come the conjunction and disjunction,
- then the implication and
- the biconditional has the lowest priority.

Example 4.1.3

$\forall x((\exists y(x = y^2)) \rightarrow ((\neg(x < 0)) \vee (x = 0)))$ can be simplified to $\forall x(\exists y x = y^2 \rightarrow \neg x < 0 \vee x = 0)$.

On the other hand, sometimes we will be using redundant parentheses in order to improve the readability.

With every first-order formula we can associate a construction tree and a parsing tree, just like with propositional formulae. The only differences are that the leaves of the construction/parsing tree of a formula are labelled with atomic formulae rather than propositional variables and that internal nodes can also be labelled with pairs $\langle \text{quantifier}, \text{individual variable} \rangle$, such as $\forall x$ or $\exists x$, and these nodes have single successor nodes.

Respectively, for every first-order formula A we define its **main connective**, being the one labeling the root of the construction/parsing tree of the formula, and the set $\text{sub}(A)$ of

subformulae of A , being all formulae with construction trees rooted as subtrees at nodes of the construction tree of A .

First-order instances of propositional formulae

Definition 4.1.4 Given a propositional formula A , any uniform substitution of first-order formulae for the propositional variables in A produces a first-order formula, called a **first-order instance** of A .

Example: The formula $((5 < x) \wedge \neg \exists y(x = y^2)) \rightarrow (\exists y(x = y^2) \vee (5 < x))$ is a first-order instance of $(p \wedge \neg q) \rightarrow (q \vee p)$.

Unique readability of terms and formulae

A very important property of formal logical languages, which gives them a technical advantage over natural languages not only in mathematics but in every exact science, is the syntactic unambiguity that allows for making precise statements. Without going into details here, we will only state some syntactic facts about first-order terms and formulae which will guarantee that every term or a formula has the *unique readability property*, intuitively meaning that it can be “read” (parsed) in an essentially unique way and is therefore syntactically unambiguous. More precisely, every term or formula has an essentially unique construction tree (respectively parsing tree) up to the order of listing of the successor nodes.

Proposition 4.1.5 Let \mathcal{L} be any first-order language. Then:

1. Every occurrence of a functional symbol in a term t from $\text{TM}(\mathcal{L})$ is the beginning of a unique subterm of t .
2. Every occurrence of \neg , \exists , \forall , or a predicate symbol in a formula A from $\text{FOR}(\mathcal{L})$ is the beginning of a unique subformula of A .

4.1.4. The semantics of first-order logic: an informal outline

The precise meaning of logical formulae is determined by their formal *semantics*, which we are going to present here. We offer two versions of that presentation: the first one, here, is quick and informal; the second, in the next section, is more detailed and somewhat more formal and abstract, but proper and precise.

Consider an arbitrarily fixed first-order language \mathcal{L} . In order to determine the meaning of the statement expressed by a given formula of our language \mathcal{L} we first need to fix its *interpretation*. It is a first-order structure – the *structure of discourse* – that “matches” the language \mathcal{L} , in a sense that all distinguished functions, predicates and constants in the structure should have names in \mathcal{L} , i.e. respective functional, predicate and constant symbols, and, conversely, all these nonlogical symbols of the language \mathcal{L} should correspond to explicitly defined functions, predicates and constants in the structure. The semantics – in particular, the truth value – of a formula is “computed” in the given interpretation compositionally, using the values of the occurring variables and the meaning of the logical and nonlogical symbols occurring in it and following the construction tree of the formula.

When the structure of discourse and the interpretation of the nonlogical symbols occurring in the formula are fixed, we need to *assign values* to the individual variables of the language, or at least to those occurring in the formula (in fact, as we will see later, not even to all of them but only to those that are not quantified over). These values are elements of the structure of discourse. Thus, every variable refers to the element in the structure that is its value.

As a running example, consider the following formula from the language $\mathcal{L}_{\mathcal{N}}$ of arithmetic:

$$\exists x(\neg(x < y + z) \wedge (z \times x < (x \times (y + 1))) \rightarrow (x = z + 2)).$$

We consider the intended interpretation, namely the structure \mathcal{N} with the standard meanings of $<$, $+$, \times , 1 , 2 . Take an assignment of values in \mathcal{N} to the variables as follows: $x = 5$, $y = 3$, $z = 2$ (we are not interested in the values of the other variables).

Once the interpretation and the variable assignment in the structure of discourse are fixed, the value of every term can be computed step by step, following the construction tree of that term, just like we evaluate arithmetic expressions following the addition and multiplication tables. Eventually, every term of the language is evaluated into a unique element of the structure – its value.

In our example, the terms occurring in the formula are evaluated as follows:

$$y + z = 5, z \times x = 10, x \times (y + 1) = 20, z + 2 = 4.$$

Next, we are ready to “compute” the meaning of every atomic formula: all we need to do is to apply the predicate indicated in the formula to the values of the terms appearing as its arguments. That meaning is a truth value: `true` or `false`, just like in propositional logic. Once these truth values are computed, we can treat atomic formulae just like propositional variables.

In our example, the atomic subformulae of the formula acquire truth values as follows: $x < (y + z)$ becomes $5 < 5$, the truth value of which is `false`, $z \times x < x \times (y + 1)$ becomes $10 < 20$, which has the truth value `true`, and $x = (z + 2)$ becomes $5 = 4$, which has truth value `false`.

Once we have the truth values of the atomic formulae computed, we can compute the truth value of any first-order formula, following the structure of that formula, just like we computed the truth values of propositional formulae.

Let us start computing the truth values of the subformulae in our example: since $x < (y + z)$ is false, we find that $\neg(x < y + z)$ is true. Then, since $z \times x < x \times (y + 1)$ is true, we find that $(\neg(x < y + z) \wedge (z \times x < x \times (y + 1)))$ is true. Hence, since $x = z + 2$ is false, $\neg(x < y + z) \wedge (z \times x < x \times (y + 1)) \rightarrow (x = z + 2)$ is false.

Now we come across an essential new step: the quantifiers. This is where computing the meaning, i.e. truth value, of a formula becomes generally difficult. Conceptually, there is no problem, as we understand quite well what the quantifiers mean intuitively:

- For the existential quantifier: $\exists x A$ is true if *there is* a possible value of the variable x in the given structure of discourse – i.e. *there is* an element of that structure assigned as a value to x – that renders the formula A true.
- Respectively, for the universal quantifier: $\forall x A$ is true if *every* possible value of the variable x in the given structure of discourse – i.e. *every* element of that structure, assigned as a value to x – renders the formula A true.

At this point we have informally defined the notion of a *formula A being true in a structure S under a variable assignment v*, denoted

$$S, v \models A.$$

Well, that is all. Sounds simple and, in a sense, it *is* simple. The difficulty is only technical: the structure may have many – possibly, infinitely many – elements, and checking that all of them (respectively some of them) make the formula *A* true can be a long – possibly, infinitely long – and hard task. Therefore in practice one tries to *guess* a suitable value of *x* in the case of existential quantification, or respectively come up with a *uniform* argument, that works for all possible values of *x*, in the case of universal quantification.

In our example, in order to determine the truth value of the entire formula, we need to find out whether there is any natural number that, assigned as a value to *x* while *y* and *z* keep the same values as before, will render the subformula $\neg(x < y + z) \wedge (z \times x < (x \times (y + 1))) \rightarrow (x = z + 2)$ true. In this case it is not difficult to guess such a value – it suffices to find one that makes the antecedent of the implication false. For that, we can take any value that makes $\neg(x < y + z)$ false, i.e. $x < y + z$ true, for instance $x = 0$. Thus, the formula $\exists x(\neg(x < y + z) \wedge (z \times x < (x \times (y + 1))) \rightarrow (x = z + 2))$ is true in \mathcal{N} .

Now, let us change the existential quantifier into a universal one. Is the formula $\forall x(\neg(x < y + z) \wedge (z \times x < (x \times (y + 1))) \rightarrow (x = z + 2))$ still true in \mathcal{N} ? In order to determine that, in principle we have to consider all natural numbers as possible values of *x* and determine the truth value of $\neg(x < y + z) \wedge (z \times x < (x \times (y + 1))) \rightarrow (x = z + 2)$ for each of them. However, in our case this is not necessary, for we already know a value of *x* that makes that subformula false, and therefore we find that $\forall x(\neg(x < y + z) \wedge (z \times x < (x \times (y + 1))) \rightarrow (x = z + 2))$ is false in \mathcal{N} .

However what if the existentially quantified formula were false or the universally quantified formula were true? Would we have to do all the infinitely many checks then? Usually a uniform argument can be found that proves the case. For instance, one can prove that $\forall x((x + 1)^2 = x^2 + 2x + 1)$ by straightforward calculations or that $\exists x(x^2 + x + 1 \leq 0)$ is false, by using some high school algebra.

In general, however, this question is much deeper than it looks, so we will leave it open here, but will only mention that it was proved⁷ in 1973 (by the collective effort of several logicians and mathematicians, including Martin Davis, Yuri Matiyasevich, Hilary Putnam and Julia Robinson) that there is no general algorithm that can answer such questions, even when the formula is a very simple one of the type $\exists x(P(x) = 0)$, where *P(x)* is a polynomial with integer coefficients. The truth of some formulae in the language $\mathcal{L}_{\mathcal{N}}$ or simple extensions of it has been settled with a formal proof only after hundreds of years of futile attempts of many great minds. One famous example is *Fermat's Last Theorem*,⁸ which can be stated by a simple one-line formula, using exponentiation in \mathcal{N} :

$$\forall x \forall y \forall z \forall n ((x \neq 0 \wedge y \neq 0 \wedge z \neq 0 \wedge n > 2) \rightarrow x^n + y^n \neq z^n)$$

⁷ This was a negative answer to Hilbert's Tenth Problem, one of the 23 famous open challenges to the mathematics of the 20th century posed by David Hilbert at the World Congress of Mathematics in 1900.

⁸ Read more on Fermat's Last Theorem in Chapter 5 on Number Theory.

The truth of yet other such formulae is still unsettled, even after centuries of attempts, e.g. the famous *Goldbach conjecture* saying that “*Every even integer greater than 2 is the sum of two prime natural numbers*”.⁹ Formalizing this statement in \mathcal{L}_N is given later as an exercise.

This is the end of our quick and informal exposition. In Section 4.2 we provide a more detailed and formal presentation.

4.1.5. Translating first-order formulae to natural language

First-order formulae formalize statements about first-order structures and these statements can be translated back to natural language. Of course, a formula is just a string of symbols, so a formal translation could simply consist in writing all symbols in words. That, however, would be of no use for *understanding the meaning* of the formula. While evaluating the truth of the formula in a given structure *formally* does not require translating that formula to natural language and understanding its intuitive meaning, that is essential for practically carrying out the truth evaluation procedure. Indeed, the meaning of (the statement expressed by) a first-order formula in a given interpretation is closely related to its truth in that interpretation.

In fact, one can argue that understanding the logical meaning of a formula, spelled out in a natural language, and determining its truth are the two sides of the same coin.

Example 4.1.6

Let us look at some examples of translating to natural language and evaluating the truth of first-order formulae, now interpreted in the structure of real numbers \mathcal{R} . Note that a good translation is usually not word by word, but it needs some polishing and rephrasing in the target language so that it eventually sounds natural and makes good sense.

1. $\exists x(x = x^2)$
 - “There is a real number which equals its square.” (True, take $x = 0$.)
2. $\forall x(x < \mathbf{0} \rightarrow x^3 < \mathbf{0})$
 - “Every negative real number has a negative cube.” (True.)
3. $\forall x\forall y(xy > \mathbf{0} \rightarrow (x > \mathbf{0} \vee y > \mathbf{0}))$.
 - “If the product of (any) two real numbers is positive, then at least one of them is positive.” (False: take $x = y = -1$.)
4. $\forall x(x > \mathbf{0} \rightarrow \exists y(y^2 = x))$

⁹ A natural number is prime if it has no other positive integer divisors but 1 and itself (see Chapter 5).

- “Every positive real number is a square of a real number.” (True: algebraic fact.)
5. $\exists x \forall y (xy < 0 \rightarrow y = 0)$
- “There is a real number x such that, for every real number y , if xy is negative, then y is 0.” (True or false?)

Now, some examples of formulae in \mathcal{L}_H :

1. $\text{John} = f(\text{Mary}) \wedge L(\text{John}, \text{Mary})$
 - “John is the father of Mary and he loves her.”
2. $(\text{John} = f(\text{Mary})) \rightarrow \exists x L(x, \text{Mary})$
 - “If John is the father of Mary then (there is) someone (who) loves Mary.”
3. $\exists x \forall z (\neg L(z, y) \rightarrow L(x, z))$
 - “There is someone (x) who loves everyone who does not love y .”
 - (Note that y stands for an unspecified person here, so this is not a proposition.)
4. $\forall x (\exists y (x = m(y)) \rightarrow \forall z (x = m(z) \rightarrow L(x, z)))$
 - “Every mother loves all her children.”

4.1.6. Exercises

- ❶ Define suitable first-order structures and introduce appropriate first-order languages for the following types of mathematical structures: graphs, groups, rings, matrices, vector spaces.
- ❷ Define suitable first-order structures and introduce appropriate first-order languages for the following data types: Booleans, strings, lists, trees.
- ❸ Which of the following are syntactically correct terms in the language \mathcal{L}_N ? Rewrite them in the usual infix notation.
 - (a) $\times(x, y)$
 - (b) $\times(2, x, y)$
 - (c) $+xy$
 - (d) $+(5, \times(x, +(3, y)))$
 - (e) $+(\times(x, +(3, y)), \times(+(\mathbf{3}, y), x))$
 - (f) $\times(s(\times(2, x), +(3, y)))$

- (g) $s(x(x(2, x), + (3, s)))$
- (h) $s(x(x(2, x), + (3, y)))$
- (i) $x(s(x(2, x)), s(+s(3), y)))$;
- (j) $s(x(s(s(x)), s(s(s(2))))), s(2))$

- ④ Construct the parsing tree of each of the syntactically correct terms from the previous exercise and determine the set of its subterms.
- ⑤* Prove that every occurrence of a functional symbol in a term t is the beginning of a unique subterm of t .
- ⑥ Which of the following are syntactically correct formulae in the language \mathcal{L}_N ? Assume that terms are written in the usual infix notation and parentheses can be omitted whenever possible or added for better readability.
- (a) $\neg\forall x(x = \mathbf{0})$
 - (b) $\forall\neg x(x = \mathbf{0})$
 - (c) $\forall x(\neg x = \mathbf{0})$
 - (d) $\forall x(x \neg = \mathbf{0})$
 - (e) $\forall x(x = \neg\mathbf{0})$
 - (f) $\neg\forall x(\neg x = \mathbf{0})$
 - (g) $\neg\exists x(\neg x = \mathbf{0.5})$
 - (h) $\forall x(x \vee \neg x)$
 - (i) $\forall x\forall y((x + y) \rightarrow (y + x))$
 - (j) $\forall x((x < \mathbf{0}) \rightarrow \forall x(x > \mathbf{0}))$
 - (k) $\forall x((x + \mathbf{0}) \rightarrow \forall x(x > \mathbf{0}))$
 - (l) $\forall x(\forall x(x > \mathbf{0}) \rightarrow x < \mathbf{0})$
 - (m) $\forall y(x < x \rightarrow \forall x(x < x))$
 - (n) $\forall y((y < y) \neg\forall x(x < x))$
- ⑦ Construct the parsing tree of each of the syntactically correct formulae from the previous exercise and determine its main connective and the set of its subformulae.
- ⑧* Prove that every occurrence of a predicate symbol \neg , \exists or \forall in a formula A from $\text{FOR}(\mathcal{L})$ is the beginning of a unique subformula of A .
- ⑨** Prove that every occurrence of any binary connective $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ in a formula A is in a context $(B_1 \circ B_2)$ for a unique pair of subformulae B_1 and B_2 of A .
- ⑩ Evaluate the occurring terms and determine the truth of the following atomic formulae in \mathcal{R} :
- (a) $x(s(s(x)), \mathbf{2}) = x(x, s(s(\mathbf{2}))),$ where x is assigned the value 3.
 - (b) $+(\mathbf{5}, x(x, +(\mathbf{3}, y))) > s(x(x(\mathbf{3}, x), \mathbf{2})),$ where x is assigned the value $\frac{1}{2}$ and y is assigned the value -5.

(c) $(s(x(x(x(x, x), x))) < x(s(x, + (x(x, x), - (1, x))))$, for each of the following values of x : 2, $\sqrt{2}$, π .

- ⑩ Determine the first-order instances of the propositional formula

$$A = (\neg p \rightarrow (q \vee \neg(p \wedge q)))$$

where:

(a) $\neg(f(x) = y)$ is substituted for p and $\exists x(x > y)$ is substituted for q .

(b) $\neg(x = y)$ is substituted both for p and q .

- ⑪ Is the formula $(\neg(x = y) \rightarrow (\exists x(x > y) \vee \neg((y = x) \wedge \exists x(x > y))))$ a first-order instance of the propositional formula A from the previous exercise?

- ⑫ Consider the structure \mathcal{N} and a variable assignment v such that $v(x) = 1$, $v(y) = 2$, $v(z) = 3$. Determine the truth in \mathcal{N} for v of each of the following formulae by using informal semantic reasoning.

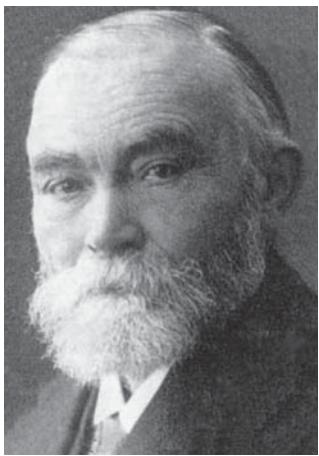
- (a) $z > y \rightarrow x > y$
- (b) $\exists x(z > y \rightarrow x > y)$
- (c) $\exists y(z > y \rightarrow x > y)$
- (d) $\exists z(z > y \rightarrow x > y)$
- (e) $\forall y(z > y \rightarrow x > y)$
- (f) $\exists x \forall y(z > y \rightarrow x > y)$
- (g) $\exists x \forall z(z > y \rightarrow x > y)$
- (h) $\exists z \forall y(z > y \rightarrow x > y)$
- (i) $\exists z \forall x(z > y \rightarrow x > y)$
- (j) $\exists y \forall x(z > y \rightarrow x > y)$
- (k) $\exists y \forall z(z > y \rightarrow x > y)$
- (l) $\forall x \exists y(z > y \rightarrow x > y)$
- (m) $\forall x \exists z(z > y \rightarrow x > y)$
- (n) $\forall y \exists x(z > y \rightarrow x > y)$
- (o) $\forall y \exists z(z > y \rightarrow x > y)$
- (p) $\forall z \exists x(z > y \rightarrow x > y)$
- (q) $\forall z \exists y(z > y \rightarrow x > y)$

- ⑬ Consider the first-order language \mathcal{L}_H for humans, with functions m for “the mother of”, f for “the father of”, the unary predicates M for “man”, W for “woman”, and the binary predicates, $C(x,y)$ meaning “ x is a child of y ”, $K(x,y)$ meaning “ x knows y ”, and $L(x,y)$ meaning “ x likes y ”. Translate each of following first-order sentences of the language \mathcal{L}_H into English and try to determine its truth in the domain of human beings.

- (a) $\forall x(\mathsf{W}(x) \rightarrow \forall z(\mathsf{C}(z, x) \rightarrow \mathsf{m}(z) = x))$
- (b) $\forall x(\exists y \mathsf{f}(y) = x \rightarrow \neg \exists z \mathsf{m}(z) = x)$
- (c) $\neg \exists x(\mathsf{M}(x) \wedge \forall z(\mathsf{C}(z, x) \rightarrow \neg \mathsf{K}(x, z)))$
- (d) $\forall x \exists y(\neg \mathsf{P}(y, x) \wedge \forall z(\mathsf{P}(x, z) \rightarrow \exists u(\mathsf{P}(u, z) \wedge \mathsf{P}(y, u))))$
- (e) $\forall x(\exists y(x = \mathsf{m}(y)) \rightarrow \forall z(\exists u \mathsf{C}(z, u) \rightarrow \mathsf{L}(x, z)))$
- (f) $\exists x(\mathsf{W}(x) \wedge \forall y(\mathsf{M}(y) \wedge \mathsf{L}(y, x) \rightarrow \neg \mathsf{L}(x, y)))$
- (g) $\neg \exists x(\exists y \mathsf{C}(x, y) \wedge \exists z(\exists u(z = \mathsf{m}(u) \wedge \neg \mathsf{L}(z, u)) \wedge \mathsf{L}(x, z)))$

15 Translate each of the following formulae from the language for reals \mathcal{L}_R extended with the unary predicate l for integers into English and determine their truth in \mathcal{R} for any given variable assignment, by using informal semantic reasoning.

- (a) $\exists x \forall y(x + y = y)$
- (b) $\exists x \forall y(x + y > y)$
- (c) $\exists x \forall y(x + y > x)$
- (d) $\forall x \exists y(x + y = x - y)$
- (e) $\forall x \exists y(x + y > y)$
- (f) $\forall x \exists y(x + y > x)$
- (g) $\exists x \forall y(xy = y)$
- (h) $\exists x \forall y(xy > y)$
- (i) $\forall x \exists y(xy > y)$
- (j) $\forall x \exists y(xy > x)$
- (k) $\forall x(\mathsf{l}(x) \rightarrow x^2 > 0)$
- (l) $\neg \forall x \exists y(\mathsf{l}(y) \wedge x > y)$
- (m) $\forall x(\mathsf{l}(x) \rightarrow \exists y(\mathsf{l}(y) \wedge x < y))$
- (n) $\exists x(\mathsf{l}(x) \wedge \neg \forall y(x < y))$
- (o) $\neg \exists x \forall y(\mathsf{l}(y) \rightarrow x > y)$
- (p) $\forall x \forall y(x < y \rightarrow \exists z(x < z \wedge z < y))$
- (q) $\forall x \forall y(x < y \rightarrow \exists z(\mathsf{l}(z) \wedge x < z \wedge z < y))$



Friedrich Ludwig Gottlob Frege (8.11.1848–26.7.1925)

Gottlob Frege was a German mathematician, logician and philosopher, who did profound and groundbreaking work on the philosophy of mathematics, logic and language. He is one of the founders of modern first-order logic, where his contribution is regarded by many as the most important development in logic since Aristotle.

Frege's first major scientific work, and also his most influential for the future development of mathematical logic, was his book *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens* (*Concept notation, a formal language modeled on that of arithmetic, for pure thought*) published in 1879. In this book he systematically developed a system of logic, with rather unusual notation but very close in spirit to the modern-day first-order logic (also known as "predicate calculus"), in which he explicitly introduced

quantifiers, quantification and logical formulae. He also formalized the notion of a "proof" in a way that is still used today. Frege had realized that natural everyday language is often imprecise and ambiguous, and therefore unreliable for performing rigorous mathematical reasoning. He therefore wanted to develop a precise, unambiguous language for expressing mathematical statements and a formal logical system for mathematical reasoning. Frege held the view, known as *logicism*, that the whole of mathematics should be reducible to logic. In the preface to the *Begriffsschrift* he stated that he wanted to prove all basic results in arithmetic "by means of pure logic". He pursued this idea further in his next major work, *The Foundations of Arithmetic*, published in 1884, where he made a lot of progress towards his goal by formalizing much of arithmetic in purely logical terms.

The reaction of Frege's contemporary mathematicians and philosophers to his work was disappointingly weak and shallow, as very few of them were able to understand his groundbreaking ideas. Particularly devastating was the criticism of Cantor (see the box at the end of Section 2.2 in Chapter 2), who held philosophical views on the foundations of mathematics very close to Frege's own. Nevertheless, Frege went on developing and elaborating his philosophy of logic and mathematics and was about to publish the 2nd volume of his *The Foundations of Arithmetic* when, in 1902, he received a letter from the young philosopher and mathematician Bertrand Russell (see Section 4.3), who pointed out a contradiction in Frege's system of axioms, revealed by Russell's paradoxical set of all sets that are not elements of themselves. This was a heavy blow to Frege, who attempted (as it turned out later, unsuccessfully) to fix the problem in an appendix to his book by amending one of his axioms, but he was so strongly shaken by that discovery that he stopped working on that project and never published the planned 3rd volume. Thus, he never succeeded to reduce the entire mathematics to logic. Much later, in the early 1920s, Frege came to the conclusion that this goal was impossible and decided, instead, that the whole of mathematics should be based on geometry, but he never managed to develop his ideas until the end of his life.

However, Frege exerted a strong influence on the views and work of Peano (see Section 4.2), Wittgenstein, Husserl, Carnap and Russell. Gradually, after his death, his revolutionary work in the philosophy of mathematics, logic and language started to receive its due recognition for the major scientific importance it has.



4.2. The formal semantics of first-order logic

We now consider an arbitrarily fixed first-order language \mathcal{L} . The formulae of \mathcal{L} are meant to express statements about structures of discourse “matching” the language \mathcal{L} . The meaning of the formulae is relative to the given structure of discourse and is “computed” compositionally, following the structure of the formula, the values of the occurring variables and the meaning of the logical and nonlogical symbols occurring in it. The precise meaning of logical formulae is determined by their formal *semantics*, which we are going to present here in a more detailed and somewhat more formal and precise manner, a method we recommend to the diligent reader. If you have less time or patience and have found the short and informal version sufficiently clear, you may skip the detailed one and proceed further with the next section. If, however, you encounter problems with the understanding and use of first-order logic further on, return to this section and go through the formal presentation of the semantics of first-order formulae.

4.2.1. Interpretations

To repeat, formulae are meant to express statements. The meaning of the statement expressed by a formula cannot be determined only from the symbols occurring in the formula and its shape, i.e. from the *syntax*. The meaning of a formula is relative to the structure of discourse and to the *interpretation*, i.e. the associated meaning of the nonlogical symbols of the formula in that structure. Formally, an **interpretation** of a first-order language \mathcal{L} involves a first-order structure \mathcal{S} and a mapping, which assigns:

- to every constant symbol c in \mathcal{L} a designated constant $c^{\mathcal{S}}$;
- to every n -ary functional symbol f in \mathcal{L} a designated n -ary function $f^{\mathcal{S}}$;
- to every n -ary predicate symbol P in \mathcal{L} a designated n -ary predicate $P^{\mathcal{S}}$.

For instance:

- The structure \mathcal{N} provides an interpretation of the language $\mathcal{L}_{\mathcal{N}}$. It is the intended, or **standard interpretation** of $\mathcal{L}_{\mathcal{N}}$.
- There are many other, natural and “unnatural” (unintended), interpretations of the language $\mathcal{L}_{\mathcal{N}}$, e.g. in the structure of reals \mathcal{R} with the standard meanings of the arithmetic operations; or still in the set of naturals, but with nonstandard meanings, e.g. $s(x)$ meaning $x + 13$, $+(x, y)$ meaning $x \times y + 1$ and $\times(x, y)$ meaning $|x - y|$; or even in a non-numerical domain with arbitrary and meaningless interpretations.
- Every graph (see Chapter 7) is an intended (standard) interpretation of the language of graphs $\mathcal{L}_{\text{Graph}}$, containing one binary relational symbol R for the edge relation.

- Likewise, every (mathematical) group is an intended (standard) interpretation of the language of groups $\mathcal{L}_{\text{Group}}$, containing one binary functional symbol \circ (the group operation), one unary functional symbol $'$ (the inverse) and one constant symbol e (the identity).
- Again, there are many more, “almost-standard” interpretations of \mathcal{L}_G (e.g. monoids with $x' = x$), as well as completely nonstandard interpretations.

Once interpreted, the meaning of a formula can be “computed”, as in the propositional case, from the interpretation of the nonlogical symbols and the meaning of the logical ones. The rules for computing this meaning determine the **semantics** of first-order logic. We will spell out these rules in some technical detail.

4.2.2. Variable assignment and term evaluation

In order to compute the truth value of a formula in a given structure S we first have to evaluate the terms occurring in it, i.e. to determine the elements of the structure denoted by these terms. For that, we must first assign values to the variables in the structure S , by means of a **variable assignment** in S , which is a mapping $v : \text{VAR} \rightarrow |S|$ from the set of variables VAR to the domain of S .

The evaluation of a term is now done just like the evaluation of an algebraic expression, which we know from primary school: starting with the values of the variables and constant symbols we systematically apply the functions in S that interpret the respective functional symbols occurring in the term; i.e. when evaluating a term $f(u_1, \dots, u_m)$, we first compute the values of the arguments u_1, \dots, u_m and then apply the interpretation of f in S to these values.

Formally, due to the unique readability of terms, every variable assignment $v : \text{VAR} \rightarrow |S|$ in a structure S can be uniquely extended to a mapping $v^S : \text{TM}(\mathcal{L}) \rightarrow |S|$, called **term evaluation**, such that, for every n -tuple of terms t_1, \dots, t_n and an n -ary functional symbol f ,

$$v^S(f(t_1, \dots, t_n)) = f^S(v^S(t_1), \dots, v^S(t_n)),$$

where f^S is the interpretation of f in S .

Once a variable assignment v in the structure S is fixed, every term t in $\text{TM}(\mathcal{L})$ is evaluated into a unique element $v^S(t)$ of S (or just $v(t)$ when S is fixed), called the **value of the term t under the variable assignment v** .

The following proposition implies that the value of any term only depends on the assignment of values to the variables occurring in that term. We leave the proof as an exercise.

Proposition 4.2.1 *For any given first-order language \mathcal{L} , term $t \in \text{TM}(\mathcal{L})$ and an \mathcal{L} -structure S , if v_1, v_2 are variable assignments in S that assign the same values to all variables in $\text{VAR}(t)$, then $v_1^S(t) = v_2^S(t)$.*

Example 4.2.2

If x is assigned a value 3 and y is assigned a value 5 in \mathcal{N} , then the term $s(s(x) \times y)$ is formally evaluated as follows:

$$\begin{aligned} v^{\mathcal{N}}(s(s(x) \times y)) \\ = s^{\mathcal{N}}(v^{\mathcal{N}}(s(x) \times y)) \\ = s^{\mathcal{N}}(v^{\mathcal{N}}(s(x)) \times^{\mathcal{N}} v^{\mathcal{N}}(y)) \\ = s^{\mathcal{N}}(s^{\mathcal{N}}(v^{\mathcal{N}}(x)) \times^{\mathcal{N}} v^{\mathcal{N}}(y)) \\ = s^{\mathcal{N}}(s^{\mathcal{N}}(3) \times^{\mathcal{N}} 5) \\ = s^{\mathcal{N}}((3 + 1) \times^{\mathcal{N}} 5) \\ = ((3 + 1) \times 5) + 1 \\ = 21. \end{aligned}$$

Likewise, $v^{\mathcal{N}}(\mathbf{1} + (x \times s(s(2)))) = 13$.

For a non-numerical term evaluation, if x is assigned the value ‘Mary’ in \mathcal{H} then $v^{\mathcal{H}}(\mathbf{f}(\mathbf{m}(x))) =$ the father of the mother of Mary.

Computing the truth values of formulae

We are now ready to start computing truth values of formulae, beginning with the logically simplest ones: atomic formulae.

Truth of atomic formulae

Eventually, we want to define the notion of a **formula A being true in a structure S under a variable assignment v** , denoted

$$S, v \models A,$$

compositionally on the structure of the formula A , beginning with the case when A is an atomic formula.

The truth value of the atomic formula $P(t_1, \dots, t_n)$ is determined by the interpretation of the predicate symbol P in S , applied to the tuple of arguments $v^S(t_1), \dots, v^S(t_n)$:

$$S, v \models P(t_1, \dots, t_n) \text{ iff } P^S \text{ holds (is true) for } v^S(t_1), \dots, v^S(t_n).$$

Otherwise, we write $S, v \not\models P(t_1, \dots, t_n)$.

The following proposition implies that the truth value of any atomic formula only depends on the assignment of values to the variables occurring in that formula. It follows quite easily from Proposition 4.2.1 and we leave the proof as an exercise.

Proposition 4.2.3 *For any given first-order language \mathcal{L} , atomic formula $A \in \text{AFOR}(\mathcal{L})$ and an \mathcal{L} -structure S , if v_1, v_2 are variable assignments in S that assign the same values to all variables in $\text{VAR}(A)$, then $S, v_1 \models A$ iff $S, v_2 \models A$.*

Example 4.2.4

If L is a binary predicate symbol interpreted in \mathcal{N} as “less than” and the variables x and y are assigned values as above, then:

- $\mathcal{N}, v \models L(\mathbf{1} + (x \times s(s(2))), s(s(x) \times y))$ iff $L^{\mathcal{N}}((\mathbf{1} + (x \times s(s(2))))^{\mathcal{N}}, (s(s(x) \times y))^{\mathcal{N}})$ iff $13 < 21$, which has a truth value `true`, while the atomic formula $\mathbf{1} + (x \times s(s(2))) = s(s(x) \times y)$ has a truth value `false`.
- Likewise, $\mathcal{N}, v \models 8 \times (x + s(s(y))) = (s(x) + y) \times (x + s(y))$ iff $(8 \times (x + s(s(y))))^{\mathcal{N}} = ((s(x) + y) \times (x + s(y)))^{\mathcal{N}}$ iff $80 = 81$, which is false.

Semantics of the propositional connectives

The truth values propagate over the propositional connectives according to their truth tables, as in propositional logic:

- $S, v \models \neg A$ iff $S, v \not\models A$;
- $S, v \models (A \wedge B)$ iff $S, v \models A$ and $S, v \models B$;
- $S, v \models (A \vee B)$ iff $S, v \models A$ or $S, v \models B$;
- $S, v \models (A \rightarrow B)$ iff $S, v \not\models A$ or $S, v \models B$;
- and likewise for $(A \leftrightarrow B)$.

Semantics of the quantifiers

Finally, the truth of formulae $\forall x A(x)$ and $\exists x A(x)$ is computed according to the meaning of the quantifiers and the truth values of A . First we need to define a technical notion. Given a structure S and two variable assignments v and v' in S , we say that v' is an x -variant of v if v' coincides with v on every variable except possibly x . Equivalently, v' is an x -variant of v if there exists an element $a \in S$ such that $v' = v[x := a]$, where $v[x := a]$ is obtained from v by redefining $v(x)$ to be a .

- $\forall x A(x)$ is true if *every object* a from the domain of S , assigned as a value of x , satisfies (i.e., renders true) the formula A . Formally, $S, v \models \forall x A(x)$ if $S, v' \models A(x)$ for every variable assignment v' that is an x -variant of v .
- $\exists x A(x)$ is true if *there is an object* a from the domain of S that, assigned as a value of x , satisfies the formula A . Formally, $S, v \models \exists x A(x)$ if $S, v' \models A(x)$ for some variable assignment v' that is an x -variant of v .

The following theorem implies that the truth value of any formula only depends on the assignment of values to the variables occurring in that formula. (We will see later that not all of these variables matter.) We leave the proof as an exercise.

Theorem 4.2.5 *For any given first-order language \mathcal{L} , formula $A \in \text{FOR}(\mathcal{L})$ and an \mathcal{L} -structure S , if v_1, v_2 are variable assignments in S that assign the same values to all variables in $\text{VAR}(A)$, then $S, v_1 \models A$ iff $S, v_2 \models A$.*

Some examples

Example 4.2.6

Consider the structure \mathcal{N} and a variable assignment v such that $v(x) = 0, v(y) = 1, v(z) = 2$. Then:

- $\mathcal{N}, v \models \neg(x > y)$.
- $\mathcal{N}, v \models \exists x(x > y)$. Indeed, $\mathcal{N}, v[x := 2] \models x > y$.
- In fact, the above holds for any value assignment of y and therefore $\mathcal{N}, v \models \forall y \exists x(x > y)$.
- On the other hand, $\mathcal{N}, v \models \exists x(x < y)$, but $\mathcal{N}, v \not\models \forall y \exists x(x < y)$. Why?
- What about $\mathcal{N}, v \models \exists x(x > y \wedge z > x)$?
- For the same variable assignment in the structure of rationals, $\mathcal{Q}, v \models \exists x(x > y \wedge z > x)$. Does this hold for every variable assignment in \mathcal{Q} ?

4.2.3. Truth evaluation games

There is an equivalent but somewhat more intuitive – and even possibly more entertaining – way to evaluate the truth of first-order formulae in given structures. It is done by playing a special kind of a two-player game, called the **formula evaluation game**.¹⁰ The two players are called **Verifier** and **Falsifier**.¹¹

The game is played on a given first-order structure S , variable assignment v in it and a formula A , the truth of which is to be evaluated in the structure S for the assignment v . As suggested by the names of the players, the objective of Verifier in the game is to defend and verify the claim that $S, v \models A$, while the objective of Falsifier is to attack and refute that claim.

The game goes in rounds and, in each round, exactly one of the players – depending on the current “game configuration” – has to make a move according to rules that we will specify below, until the game ends. The current game configuration (S, w, C) consists of the structure S , an assignment w in S and a formula C (the truth of which is to be evaluated in S for the assignment w). The **initial configuration** is (S, v, A) . We will identify every such game with its initial configuration.

At every round, the player to make a move, as well as the possible moves, are determined by the main connective of the formula in the current configuration (S, w, C) by rules that closely resemble the truth definitions for the logical connectives. Here are the rules:

- If the formula C is atomic, the game ends.
If $S, w \models C$ then Verifier wins; otherwise Falsifier wins.
- If $C = \neg B$ then Verifier and Falsifier *swap their roles* and the game continues with the configuration (S, w, B) . Swapping the roles means that Verifier wins the game $(S, w, \neg B)$ iff Falsifier

¹⁰ Also known as the model checking game.

¹¹ Also known by various other names, e.g. Proponent and Opponent, Eloise and Abelard, Eve and Adam. For the sake of convenience, and without prejudice, here we will assume that both players are male.

wins the game (S, w, B) and Falsifier wins the game $(S, w, \neg B)$ iff Verifier wins the game (S, w, B) .

Intuition: verifying $\neg B$ is equivalent to falsifying B and vice versa.

- If $C = C_1 \wedge C_2$ then Falsifier chooses $i \in \{1, 2\}$ and the game continues with the configuration (S, w, C_i) .

Intuition: for Verifier to defend the truth of $C_1 \wedge C_2$ he should be able to defend the truth of *any* of the two conjuncts, so it is up to Falsifier to question the truth of either of them.

- If $C = C_1 \vee C_2$ then Verifier chooses $i \in \{1, 2\}$ and the game continues with the configuration (S, w, C_i) .

Intuition: for Verifier to defend the truth of $C_1 \vee C_2$ it suffices to be able to defend the truth of at least one of the two disjuncts, so he can choose which one.

- If $C = C_1 \rightarrow C_2$ then Verifier chooses $i \in \{1, 2\}$ and, depending on that choice, the game continues respectively with the configuration $(S, w, \neg C_1)$ or (S, w, C_2) .

Intuition: $C_1 \rightarrow C_2 \equiv \neg C_1 \vee C_2$.

- If $C = \exists x B$ then Verifier chooses an element $a \in S$ and the game continues with the configuration $(S, w[x := a], B)$.

Intuition: by the truth definition of $\exists x B$, verifying that $S, w \models \exists x B$ amounts to verifying that $S, w[x := a] \models B$ for some suitable element $a \in S$.

- If $C = \forall x B$ then Falsifier chooses an element $a \in S$ and the game continues with the configuration $(S, w[x := a], B)$.

Intuition: by the truth definition of $\forall x B$, falsifying $S, w \models \forall x B$ amounts to falsifying $S, w[x := a] \models B$ for some suitable element $a \in S$.

It is easy to see that any formula evaluation game always ends in a finite number of steps, because the number of logical connectives in the formula in the current configuration strictly decreases after every move until an atomic formula is reached. It is also obvious that the game always ends with one of the players winning. Now clearly the winner of such a game depends not only on the truth or falsity of the claim $S, v \models A$ but also on how well the players play the game, so we suppose they always play a best possible move. Thus, the game will be won by the player who *has a winning strategy* for that game, i.e. a rule that, for every possible configuration from which that player is to move assigns such a move, he is *guaranteed to eventually win the game, no matter how the other player plays*. It is not quite obvious that one of the players is sure to have a winning strategy in every such game, but it follows from a more general result in game theory. This also follows from the following claim – which we take here for granted – relating the existence of a winning strategy to the truth of the formula in the initial configuration.

Theorem 4.2.7 *For every configuration (S, v, A) :*

1. $S, v \models A$ if and only if Verifier has a winning strategy for the evaluation game (S, v, A) .
2. $S, v \not\models A$ if and only if Falsifier has a winning strategy for the evaluation game (S, v, A) .

How does one find a winning strategy for one of the players in a truth evaluation game? We see from the theorem above that this problem is as difficult as determining the truth of a first-order formula in a given structure and variable assignment, so there is no general magic

recipe. Still, a sensible translation of the formula to natural language that helps to understand its meaning would also help the player who has a winning strategy in the evaluation game to come up with the right strategy.

Example 4.2.8

Consider the structure \mathcal{N} and the variable assignment v such that $v(x) = 0, v(y) = 1, v(z) = 2$.

1. Verifier has a winning strategy for the game $(\mathcal{N}, v, \forall y \exists x (x > y + z))$.

Indeed, the first move of the game is by Falsifier, who has to choose an integer n and the game continues from configuration $(\mathcal{N}, v[y := n], \exists x (x > y + z))$.

Now, Verifier has to choose an integer m . For every given $n \in \mathcal{N}$ Verifier can choose, for example, $m = n + 3$. Then he wins the game $(\mathcal{N}, v[y := n][x := m], (x > y + z))$ because $n + 3 > n + 2$.

Thus, he has a winning strategy for the game $(\mathcal{N}, v[y := n], \exists x (x > y + z))$, for any $n \in \mathcal{N}$. Hence, he has a winning strategy for the game $(\mathcal{N}, v, \forall y \exists x (x > y + z))$.

Therefore, $\mathcal{N}, v \models \forall y \exists x (x > y + z)$.

In fact, the above winning strategy for Verifier is easy to generalize for *any* assignment of value to z , thus showing that $\mathcal{N}, v \models \forall z \forall y \exists x (x > y + z)$.

2. Verifier has a winning strategy for the game $(\mathcal{N}, v, \forall x (y < x \vee x < z))$.

Indeed, the first move in the game is by Falsifier, who has to choose an integer n and the game continues from configuration $(\mathcal{N}, v[x := n], (y < x \vee x < z))$ in which Verifier must choose one of the disjuncts $y < x$ and $x < z$.

The strategy for Verifier goes as follows: if Falsifier has chosen $n > 1$ then Verifier chooses the disjunct $y < x$ and wins the game $(\mathcal{N}, v[x := n], y < x)$; if Falsifier has chosen $n \leq 1$ then Verifier chooses the disjunct $x < z$ and wins the game $(\mathcal{N}, v[x := n], x < z)$.

Therefore, $\mathcal{N}, v \models \forall x (y < x \vee x < z)$.

3. Falsifier has a winning strategy for the game $(\mathcal{N}, v, \forall x (x < z \rightarrow \exists y (y < x)))$.

Indeed, let Falsifier choose 0 in the first move. Then the game continues from configuration $(\mathcal{N}, v[x := 0], (x < z \rightarrow \exists y (y < x)))$ and now Verifier is to choose the antecedent or the consequent of the implication.

If Verifier chooses the antecedent, then the game continues from configuration $(\mathcal{N}, v[x := 0], \neg(x < z))$, which is won by Falsifier because the game $(\mathcal{N}, v[x := 0], x < z)$ is won by Verifier, since $0 < 2$.

If Verifier chooses the consequent of the implication, then the game continues from configuration $(\mathcal{N}, v[x := 0], \exists y (y < x))$ and Verifier is to choose a suitable value for y . However, whatever $n \in \mathcal{N}$ Verifier chooses, he loses the game $(\mathcal{N}, v[x := 0][y := n], y < x)$ because $n < 0$ is false for every $n \in \mathcal{N}$.

Thus, the Verifier has no winning move after the first move of Falsifier choosing 0 as a value for x . Therefore, $\mathcal{N}, v \not\models \forall x (x < z \rightarrow \exists y (y < x))$.

Furthermore, the winning strategy for Falsifier in the game $(\mathcal{N}, v, \forall x (x < z \rightarrow \exists y (y < x)))$ is a winning strategy for Verifier in the game $(\mathcal{N}, v, \neg \forall x (x < z \rightarrow \exists y (y < x)))$; hence $\mathcal{N}, v \models \neg \forall x (x < z \rightarrow \exists y (y < x))$.

4.2.4. Exercises

- ① Consider the structure \mathcal{N} and a variable assignment v such that $v(x) = 1, v(y) = 2, v(z) = 3$. Determine the truth in \mathcal{N} for v of each of the following formulae by using the formal truth definition. You may wish to use formula evaluation games as well.
- $\exists x \forall y \forall z (z > y \rightarrow x > y)$
 - $\exists y \forall x \forall z (z > y \rightarrow x > y)$
 - $\exists z \forall x \forall y (z > y \rightarrow x > y)$
 - $\forall y \exists x \forall z (z > y \rightarrow x > y)$
 - $\forall x \exists y \forall z (z > y \rightarrow x > y)$
 - $\forall z \exists x \forall y (z > y \rightarrow x > y)$
 - $\forall x \forall y \exists z (z > y \rightarrow x > y)$
 - $\forall y \forall z \exists x (z > y \rightarrow x > y)$
 - $\forall z \forall x \exists y (z > y \rightarrow x > y)$
- ② Translate the following first-order formulae into English and determine which of them represent true propositions when interpreted in \mathcal{R} , by using the formal truth definition. You may wish to use formula evaluation games as well.
- $\neg \forall x (x \neq 0)$
 - $\forall x (x^3 \geq x)$
 - $\forall x (x = x^2 \rightarrow x > 0)$
 - $\exists x (x = x^2 \wedge x < 0)$
 - $\exists x (x = x^2 \rightarrow x < 0)$
 - $\forall x (x > 0 \rightarrow x^2 > x)$
 - $\forall x (x = 0 \vee \neg x + x = x)$
 - $\forall x ((x = x^2 \wedge x > 1) \rightarrow x^2 < 1)$
 - $\forall x \forall y (x > y \vee y > x)$
 - $\forall x \exists y (x > y^2)$
 - $\forall x \exists y (x > y^2 \vee y > 0)$
 - $\forall x (x \geq 0 \rightarrow \exists y (y > 0 \wedge x = y^2))$
 - $\forall x \exists y (x > y \rightarrow x > y^2)$
 - $\forall x \exists y (\neg x = y \rightarrow x > y^2)$
 - $\exists x \forall y (x > y)$

- (p) $\exists x \forall y (x + y = x)$
- (q) $\exists x \forall y (x + y = y)$
- (r) $\exists x \forall y (x > y \vee \neg x > y)$
- (s) $\exists x \forall y (x > y \vee \neg x > y)$
- (t) $\exists x \forall y (y > x \rightarrow y^2 > x)$
- (u) $\exists x \forall y (x > y \rightarrow x > y^2)$
- (v) $\exists x \exists y (xy = x + y)$
- (w) $\forall x \exists y \forall z (xy = yz)$
- (x) $\exists x \forall y \exists z ((x + y)z = \mathbf{1})$
- (y) $\forall x \forall y (x > y \rightarrow \exists z (x > z \wedge z > y))$
- (z) $\forall x \exists z \forall y (x > z \rightarrow z > y)$

③* Prove Proposition 4.2.1, by induction on the length of terms.

④ Prove Proposition 4.2.3, by using Proposition 4.2.1.

⑤* Prove Theorem 4.2.5, by induction on the length of formulae.



Giuseppe Peano (27.8.1858–20.4.1932)

Giuseppe Peano was an Italian mathematician and one of the founders of mathematical logic and set theory. He played a key role in the axiomatization of the arithmetic that is now known as the *Peano system of axioms*.

Peano was born and grew up on a farm in Northern Italy. He graduated from the University of Turin in 1880 where he subsequently took up a position and worked for most of his career as Professor of Mathematics.

Peano was strongly influenced by the logical system developed by Frege and, along with Russell, did a lot to popularize and further develop these ideas. In 1889 Peano published his famous axioms, now called *Peano axioms*, which defined the natural numbers in terms of sets. Here are Peano's main axioms:

1. Zero is a number.
2. The successor of any number is another number.
3. There are no two numbers with the same successor.
4. Zero is not the successor of a number.
5. (Axiom of induction) If a set S of numbers contains zero and also the successor of every number in S, then every number is in S.

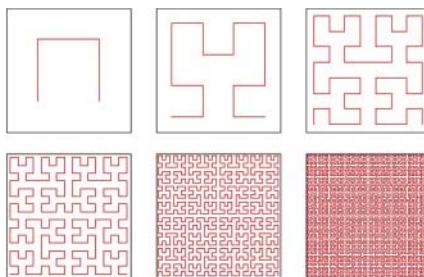
In 1890 Peano constructed a “space-filling curve” – a continuous curve filling the entire unit square, an early example of a fractal – as a counterexample to the claim that a continuous curve can be enclosed in an arbitrarily small region.

In 1892 Peano embarked on the extremely ambitious project, called *Formulario Mathematico*, intended as an encyclopedia of all mathematical formulae and theorems expressed in a symbolic language. Many of the symbols that Peano used in the *Formulario* are still in use today. Thus monumental work, eventually completed in 1908, had a strong influence in terms of its rigorous exposition and modern style, but Peano’s use of it for teaching was liked by neither his students nor his colleagues.

In 1903 Peano started developing an international auxiliary language called *Latino sine flexione*, later also called *Interlingua*, using Latin vocabulary but with very simplified grammar to make it easier to learn. He published the final edition of *Formulario Mathematico* in *Latino sine flexione*, which was an added reason for its lack of popularity.

Eventually, Peano received wide recognition and many awards for his crucial contributions to modernizing mathematics and making the mathematical notation and reasoning more rigorous, based on mathematical logic.

Below is an illustration of the first of several steps of an iterative construction of Peano’s curve, eventually defined as the limit of the construction, filling up the whole square.



4.3. The language of first-order logic: a deeper look

4.3.1. Translations from natural language into first-order languages

When learning a new language, it is usually more difficult to translate into that language than to translate from it into our own. This is likewise the case for first-order languages. In both cases the best learning strategy is practicing.

1. Let us start with translating

“There is an integer greater than 2 and less than 3.”

Surely, this is false, but now we are not concerned with its truth, only with its translation to \mathcal{L}_z . It is immediate:

$$\exists x(x > 2 \wedge x < 3).$$

Note that **2** and **3** here are *terms*, not the numbers 2 and 3.

2. Now, does the same translation work for \mathcal{L}_Q ? No, because the quantification is over *all elements of the domain of discourse*, so the same formula would say in \mathcal{L}_Q "There is a rational number greater than 2 and less than 3", which is true, but not what we wanted to say. So, what now? If we want to refer to *integers* while the domain of discourse is a larger set, viz. the rational numbers, we need to extend the language with an additional predicate I which says that the element of the domain *is an integer*. Now the translation is easy:

$$\exists x(I(x) \wedge x > 2 \wedge x < 3).$$

We will come back to this trick later again.

3. Let us now translate

"There is no real number, the square of which equals -1 ."

to \mathcal{L}_R . How about

$$\exists x(\neg x^2 = -1)?$$

Although the sentence begins with "There is ... ", do not be confused: this is *not an existential statement, but a negation of one*. It actually says

"It is not true that there is a real number the square of which equals -1 ."

so the correct translation is

$$\neg \exists x(x^2 = -1).$$

Here are some nonmathematical examples.

1. Take the sentence

"Every man loves a woman."

It looks deceptively simple; how about just $\forall x \exists y L(x, y)$? Wrong! This only means "Everybody loves somebody".

This sentence presents a problem we have already seen before: the quantifiers in our formal language range over the *whole* domain of discourse, in our case over all human beings. So whenever we write $\forall x$ in this context it will mean "every human being", not "every man"; likewise, using $\exists x$ we say "there is a human being", but we cannot say "there is a woman".

Thus, in order to formalize our sentence in the language of first-order logic, we need to resolve this problem. For that, we use again the little trick with the predicates, called **restricted quantification** (see more on that further). After thinking a little on the sentence, we realize that it can be rephrased in the following way which is more convenient to formalize in first-order logic, even though it sounds a bit awkward:

"For every human, if he is a man then there is a human who is a woman and the man loves that woman."

Now the translation into \mathcal{L}_H is immediate:

$$\forall x(M(x) \rightarrow \exists y(W(y) \wedge L(x, y))).$$

2. Likewise, we can translate “Every mother has a child whom she loves.”

$$\forall x(\exists y(x = m(y)) \rightarrow \exists z(C(z, x) \wedge L(x, z))).$$

Make an effort to understand how these translations work. Pay special attention to the use of variables and the quantification.

4.3.2. Restricted quantification

As already discussed, we often quantify not over all individuals from the domain, but over a specified family of them, e.g. “There is a *positive* number x such that ...”, “Every *woman* ...”, etc. There are two natural ways in which to formalize such a *restricted* quantification. We can either use a *many-sorted domain* (e.g. in the domain of numbers we can have sorts for integers, positive numbers, rational numbers, etc.; in the domain of humans we can have sorts for male, female, young, etc., individuals) and different stocks of variables for each sort¹² or we can use unary predicates for each sort and logical connectives to express the *restricted quantification schemes*, as we did when we formalized the sentence “Every man loves a woman.” For instance, in order to quantify over positive numbers, we can introduce a predicate $P(x)$, saying that “ x is a *positive* number” and then

$$\exists x(P(x) \wedge \dots x \dots)$$

says that there exists an object x which is a *positive number* and which satisfies $\dots x \dots$. Likewise,

$$\forall x(P(x) \rightarrow \dots x \dots)$$

says that all objects x , which are *positive numbers*, satisfy $\dots x \dots$.

Of course, in this case we need not introduce the predicate for positive numbers because it is *definable* as long as we have the symbol $<$ in our language. Thus, the formulae $\exists x(0 < x \wedge \dots x \dots)$ and $\forall x(0 < x \rightarrow \dots x \dots)$ will do the same job as those above.

While the former method is more commonly used in informal reasoning, the latter is a more universal one. In usual mathematical practice, though, the following convenient combination of the two methods is used. For the most important families of individuals, we introduce some standard notation, e.g. the set of natural numbers is usually denoted by \mathbb{N} , the set of integers by \mathbb{Z} , the set of rational numbers by \mathbb{Q} and the set of reals by \mathbb{R} . Now, if we want to quantify, for example, over integers, we can say: “for all x in \mathbb{Z} ”, respectively “there is x in \mathbb{Z} such that”, or symbolically:

$$\forall x \in \mathbb{Z}(\dots x \dots), \text{ and } \exists x \in \mathbb{Z}(\dots x \dots)$$

For instance, $\forall x \in \mathbb{Z}(x^2 \geq x)$ states (truly) that the square of every *integer* is greater than or equal to that integer, $\exists x \in \mathbb{Q}(x^3 - 2x + 2 = 0)$ states (falsely) that there is a *rational* solution of

¹² In fact, quite often we do that in mathematical discourse. It is a tradition to denote a real number by x, y, z , etc., while an integer would rather be denoted by i, j, k, n, m , a function by f, g, h , etc. Of course, there is no inherent reason, apart from the risk of confusion, not to use f for an integer or j for a real number. And, when working with complex numbers, i becomes something quite different.

the equation $x^3 - 2x + 2 = 0$ and $\forall x \in \mathbb{R} \exists z \in \mathbb{Z}(x < z)$ says that for every real number there is an integer greater than it. (True or false?)

4.3.3. Free and bound variables. Scope of a quantifier

There are two essentially different ways in which we use individual variables in first-order formulae.

1. First, we use them to denote *unknown or unspecified objects*, like in $(5 < x) \vee (x^2 + x - 2 = 0)$. We say that the variable x occurs **free** in that formula. As long as a formula contains free variables, it cannot (in general) be assigned a truth value until these free variables are assigned values. Therefore, it cannot be regarded as a proposition, for its truth may vary depending on the different possible values of the occurring free variables. For example, assigning value 1 or 6 to x in the formula above turns it into a true proposition in \mathcal{R} , while assigning value 2 to x turns it into a false one.
2. Second, we use individual variables in order to *quantify*, as in $\forall x((5 < x) \vee (x^2 + x - 2 = 0))$ and $\exists x((5 < x) \vee (x^2 + x - 2 = 0))$. In these formulae the variable x is said to be **bound** by the quantifier \forall (i.e. **universally bound**) in the first example and, respectively, by \exists (i.e. **existentially bound**) in the second one.

Note that the same variable can be *both free and bound* in a formula. For example: x in the formula $x > 0 \wedge \exists x(5 < x)$. That is why we talk about free and bound *occurrences* of a variable in a formula.

To make the notion of a bound occurrence of a variable more precise, note that every occurrence of a quantifier Q in a formula is the beginning of a *unique* subformula QxA , called the **scope** of that occurrence of the quantifier. For example, in the formula $\forall x((x > 5) \rightarrow \forall y(y < 5 \rightarrow (y < x \wedge \exists x(x < 3))))$, the scope of the first occurrence of \forall is the whole formula, while the scope of the second occurrence of \forall is $\forall y(y < 5 \rightarrow (y < x \wedge \exists x(x < 3)))$ and the scope of the occurrence of \exists is $\exists x(x < 3)$.

Now, every bound occurrence of a variable x is bound by the *innermost* occurrence of a quantifier Q over x in which scope that occurrence of x is. In other words, an occurrence of a variable x is bound by the first occurrence of the quantifier Q in a formula QxA , if and only if that occurrence of x is free in the subformula A . For example, in the formula $\forall x((x > 5) \rightarrow \forall y(y < 5 \rightarrow (y < x \wedge \exists x(x < 3))))$ the first three occurrences of x are bound by the first occurrence of \forall , while the last two are bound by the occurrence of \exists .

Definition 4.3.1 A formula with no bound variables is called an **open formula**. A formula with no free variables is called a **closed formula** or a **(first-order) sentence**.

Once interpreted in a structure, first-order sentences have a determined meaning and represent propositions about that structure.

The following important observation allows us to only consider the assignment of values to the free variables occurring in a formula in order to determine the truth of that formula.

Proposition 4.3.2 The truth of a formula in a given first-order structure under a given variable assignment only depends on the assignment of values to the free variables occurring in that formula. That

is, if v_1, v_2 are variable assignments in S such that $v_1|_{FVAR(A)} = v_2|_{FVAR(A)}$, where $FVAR(A)$ is the set of free variables in A , then

$$S, v_1 \models A \text{ iff } S, v_2 \models A.$$

In particular, if a variable x does not occur free in A , then $S, v \models A$ if and only if $S, v' \models A$ for every x -variant v' of v and, therefore, if and only if $S, v \models \forall x A$.

4.3.4. Renaming of a bound variable in a formula. Clean formulae

Note that a bound variable in a formula plays an auxiliary rôle and *does not have its own meaning* (we also call it a *dummy* variable) in the sense that we can replace it by another one throughout the formula without altering the meaning of that formula. For example, it should be intuitively clear that $\exists x(5 < x \vee x^2 + x - 2 = 0)$ means exactly the same as $\exists y(5 < y \vee y^2 + y - 2 = 0)$ (in particular, these two are equally true); likewise $\forall x(5 < x \vee x^2 + x - 2 = 0)$ means the same as $\forall y(5 < y \vee y^2 + y - 2 = 0)$. On the other hand, $5 < x \vee x^2 + x - 2 = 0$ is *essentially different* from $5 < y \vee y^2 + y - 2 = 0$; depending on the values of the free variables x and y , one of these can be true, while the other one is false.

You should be able to distinguish very well between the meaning and the use of free and bound variables. For example, the free and the bound occurrences of the variable x in the formula $(x > 5) \wedge \forall x(x < 2x)$ have *nothing* to do with each other.

Furthermore, different occurrences of the same variable can be bound by different quantifiers, and that may also be confusing. Examples are:

- $\exists x(x > 5) \vee \forall x(2x > x)$. Clearly, the occurrences of x , bound by the first quantifier, have nothing to do with those bound by the second one.
- $\exists x(x > 5) \wedge \exists x(x < 3)$. Likewise, the two x 's claimed to exist here need not (in fact, cannot) be the same, so this formula has the same meaning as $\exists y(y > 5) \wedge \exists x(x < 3)$ or $\exists x(x > 5) \wedge \exists z(z < 3)$ or $\exists y(y > 5) \wedge \exists z(z < 3)$.
- $\forall x((x > 5) \rightarrow \exists x(x < 3))$. Again, the occurrences of x in the subformula $\exists x(x < 3)$ are bound by \exists and are not related to the first two occurrences of x , bound by \forall , so this formula has the same meaning as $\forall x((x > 5) \rightarrow \exists y(y < 3))$ or $\forall z((z > 5) \rightarrow \exists x(x < 3))$ or $\forall z((z > 5) \rightarrow \exists y(y < 3))$.

The best way to avoid confusions like these in formal mathematical arguments and proofs is *always to use different variables for different purposes*. In particular, never use the same variable as both free and bound, nor as bound by two different quantifiers in the same formula or proof.

Definition 4.3.3 A formula A is *clean* if no variable occurs both free and bound in A and every two occurrences of quantifiers bind different variables.

Thus, $\exists x(x > 5) \wedge \exists y(y < z)$ is clean, while $\exists x(x > 5) \wedge \exists y(y < x)$ and $\exists x(x > 5) \wedge \exists x(y < x)$ are not.

Definition 4.3.4 The uniform replacement of all occurrences of a variable x bound by the same occurrence of a quantifier in a formula A with a variable not occurring in A is called *renaming* the variable x in A .

For instance, a correct renaming of $(x > 5) \wedge \forall x(x > 5 \rightarrow \neg \exists x(x < y))$ is the formula $(x > 5) \wedge \forall x(x > 5 \rightarrow \neg \exists z(z < y))$, but neither $(z > 5) \wedge \forall x((x > 5) \rightarrow \neg \exists x(x < y))$, nor $(x > 5) \wedge \forall z((z > 5) \rightarrow \neg \exists z(z < y))$, nor $(x > 5) \wedge \forall x(x > 5 \rightarrow \neg \exists y(y < y))$ is correct.

Every formula can be transformed into a clean formula by means of several consecutive renamings of variables. For example, the formula

$$(x > 5) \wedge \forall x((x > 5) \rightarrow \neg \exists x(x < y))$$

can be transformed into the clean formula

$$(x > 5) \wedge \forall z_1((z_1 > 5) \rightarrow \neg \exists z_2(z_2 < y)).$$

4.3.5. Substitution of a term for a variable in a formula. Capture of a variable

Substitution of a term for a variable in a formula

Given a formula A , a variable x and a term t , we can obtain a formula $A[t/x]$ by *simultaneously* substituting t for all *free* occurrences of x in A . The formula $A[t/x]$ is called **the result of substitution of t for x in A** . The semantic meaning of such substitution is to assign the value of the term t to the variable x when evaluating the truth of A . For instance, given the formula

$$A = \forall x(P(x, y) \rightarrow (\neg Q(y) \vee \exists y P(x, y))),$$

we have

$$A[f(y, z)/y] = \forall x(P(x, f(y, z)) \rightarrow (\neg Q(f(y, z)) \vee \exists y P(x, y))),$$

while

$$A[f(y, z)/x] = A,$$

because x does not occur free in A .

Capture of a variable

Intuitively, $A[t/x]$ is supposed to say about the individual denoted by t the same as what A says about the individual denoted by x . However, is that always the case? No. An unwanted “capture” effect can occur after such a substitution if we substitute a term t for a variable x in a formula A where x is in the scope of a quantifier over another variable y that occurs in t . For example, substituting $y + 1$ for x in the formula $\exists y(x < y)$, which is true in \mathcal{N} for *every* value assigned to x , will produce the false sentence $\exists y(y + 1 < y)$ since the occurrence of y in the term $y + 1$ is *captured* by the quantifier $\exists y$ because the substitution mixed free and the bound uses of y . Similarly, in the formula $\forall x \exists z(x < z)$, meaning that for every number there is a greater one, renaming x with the variable z will produce $\forall z \exists z(z < z)$, which clearly distorts the meaning of the formula. The reason is that when substituting z for x it is *captured* by the quantifier $\exists z$.

Formally, **capture** happens when new occurrences of a variable, say y , are introduced in the scope of a quantifier Qy in a formula A as a result of substitution of a term t containing y for another variable x in that formula. The following definition means to eliminate substitutions that allow capture.

Definition 4.3.5 A term t is *free for (substitution for) a variable x in a formula A* if no variable in t is captured by a quantifier as t is substituted for any free occurrence of x in A .

To avoid the confusions mentioned above we only allow a substitution t/x in a formula A when t is free for x in A .

Hereafter, when t is a constant symbol c , we will often write $A(c)$ instead of $A[c/x]$.

A note on renamings and substitutions in a formula

Note that renaming and substitution are *very different* operations: renaming always acts on bound variables, while substitution always acts on free variables. Also, as we will see later, renamings preserve the formula up to logical equivalence, while substitutions do not.

On the other hand, a suitable renaming of a formula can prepare it for a substitution, by rendering the term to be substituted free for such substitution in the renamed formula. For instance, the term $f(x, y)$ is not free for substitution for y in

$$A = \forall x(P(x, y) \wedge \exists y Q(y)),$$

but it becomes free for such substitution after renaming of A , for example, to

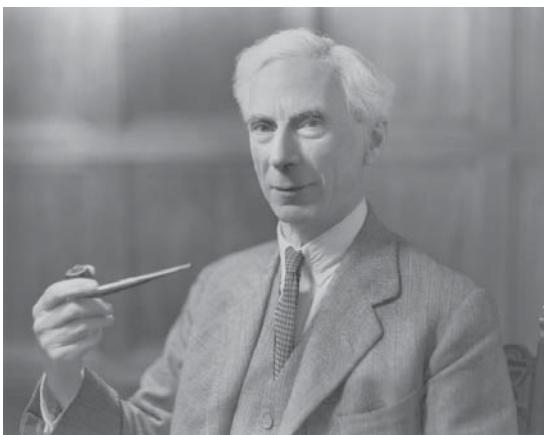
$$A' = \forall x'(P(x', y) \wedge \exists y Q(y)).$$

4.3.6. Exercises

- ❶ Using the additional predicate $I(x)$ for “ x is an integer” formalize the following sentences in the first-order language for the structure of real numbers \mathcal{R} .
 - (a) Every square of an integer is greater than 0.
 - (b) Every square of a real number that is not an integer is greater than 0.
 - (c) Some real numbers are not integers.
 - (d) Every integer is even or odd.
 - (e) No integer is both even and odd.
 - (f) For every integer there is a greater integer.
 - (g) Every positive integer is a square of some negative real number.
 - (h) Not every real number is greater than an integer.
 - (i) There is an integer such that not every real number is greater than it.
 - (j) No real number is greater than every integer.
 - (k) Every real number that is not zero has a reciprocal.
 - (l) There is a real number that, when multiplied by any real number, produces that number.
 - (m) Between every two different real numbers there is an integer.
- ❷ Using unary predicates $T(x)$ for “ x talks” and $L(x)$ for “ x listens”, formalize the following sentences in a first-order language for the domain of all humans.

- (a) Everybody talks or everybody listens.
- (b) Everybody talks or listens.
- (c) If John talks everybody listens.
- (d) If somebody talks everybody listens.
- (e) If somebody talks everybody else listens.
- (f) Nobody listens if everybody talks.
- ③ Translate the following sentences to the first-order language \mathcal{L}_H for the structure of all humans \mathcal{H} . Remember that the language \mathcal{L}_H does not have a unary predicate for "... is a child", but a binary predicate "... is a child of ...".
- (a) Some men love every woman.
- (b) Every woman loves every man who loves her.
- (c) Every man loves a woman who does not love him.
- (d) Some women love no men who love them.
- (e) Some men love only women who do not love them.
- (f) No woman loves a man who loves every woman.
- (g) Every woman loves her children.
- (h) Every child loves his/her mother.
- (i) Everyone loves a child. (*Note the ambiguity here.*)
- (j) A mother loves every child. (*... and here.*)
- (k) Every child is loved by someone.
- (l) Some woman loves every child.
- ④ Using a first-order language with $=$ only, write:
- (a) a sentence λ_n saying that the domain has at least n elements;
- (b) a sentence μ_n saying that the domain has at most n elements;
- (c) a sentence σ_n saying that the domain has exactly n elements.
- ⑤ Let R be a binary relational symbol. Write a first-order formula, in the language with R and $=$, saying that R is functional, i.e. every element of the domain is R -related to a unique element.
- ⑥ Determine the scope of each quantifier and the free and bound occurrences of variables in the following formulae (where P is a unary predicate and Q a binary one).
- (a) $\exists x \forall z(Q(z, y) \vee \neg \forall y(Q(y, z) \rightarrow P(x)))$
- (b) $\exists x \forall z(Q(z, y) \vee \neg \forall x(Q(z, z) \rightarrow P(x)))$
- (c) $\exists x(\forall z Q(z, y) \vee \neg \forall z(Q(y, z) \rightarrow P(x)))$

- (d) $\exists x(\forall z Q(z, y) \vee \neg \forall y Q(y, z)) \rightarrow P(x)$
 (e) $\exists x \forall z(Q(z, y) \vee \neg \forall y Q(x, z)) \rightarrow P(x)$
 (f) $\exists x \forall z Q(z, y) \vee \neg (\forall y Q(y, x) \rightarrow P(x))$
 (g) $\exists x(\forall z Q(z, y) \vee \neg (\forall z Q(y, z) \rightarrow P(x)))$
 (h) $\exists x(\forall x(Q(x, y) \vee \neg \forall z(Q(y, z) \rightarrow P(x))))$
 (i) $\exists x(\forall y(Q(x, y) \vee \neg \forall x Q(x, z))) \rightarrow P(x)$
- 7 Rename the bound variables in each of the formulae above to obtain a clean formula.
- 8 For each of the following formulae (where P is a unary predicate and Q a binary one) determine if the indicated term is free for substitution for the indicated variable. If so, do the substitution.
- (a) Formula: $\exists x(\forall z P(y) \vee \neg \forall y(Q(y, z) \rightarrow P(x)))$; term: $f(x)$; variable: z .
 - (b) Same formula; term: $f(z)$; variable: y .
 - (c) Same formula; term: $f(y)$; variable: y .
 - (d) Formula: $\forall x((\neg \forall y Q(x, y) \vee P(z)) \rightarrow \forall y \neg \exists z \exists x Q(z, y))$; term: $f(y)$; variable: z .
 - (e) Same formula; term: $g(x, f(z))$; variable: z .
 - (f) Formula: $\forall y(\neg(\forall x \exists z(\neg P(z) \wedge \exists y Q(z, x))) \wedge (\neg \forall x Q(x, y) \vee P(z)))$; term $f(x)$; variable: z .
 - (g) Same formula; term: $f(y)$; variable: z .
 - (h) Formula: $(\forall y \exists z \neg P(z) \wedge \forall x Q(z, x)) \rightarrow (\neg \exists y Q(x, y) \vee P(z))$; term: $g(f(z), y)$; variable: z .
 - (i) Same formula; term: $g(f(z), y)$; variable: x .
- 9* Additional exercises. Recall that a natural number is prime if it has no other positive integer divisors but 1 and itself. Express the following statements in the language \mathcal{L}_N :
- (a) The number (denoted by) x is prime.
 - (b) There are infinitely many primes. (Hint: this is equivalent to saying that for every natural number there is a greater prime.)
 - (c) (Goldbach's conjecture) Every even integer greater than 2 is the sum of two primes.
 - (d) (The prime twins conjecture) There are infinitely many pairs of primes that differ by 2.
 - (e) Every natural number n has a prime divisor not greater than \sqrt{n} . (Note: the function $\sqrt{\cdot}$ is not in the language.)
 - (f) The numbers x and y have no common prime divisor.



Bertrand Arthur William Russell, (18.5.1872–2.2.1970)

Bertrand Russell was a famous British philosopher, logician, mathematician, intellectual and a political activist with strong pacifist views, who made seminal contributions to the foundations of mathematics and to the advancement of modern formal logic. He is also regarded as one of the founders of modern analytic philosophy.

Russell studied mathematics and philosophy at Trinity College, Cambridge, from where he graduated with distinction. He later held academic positions there with several interruptions during which he worked for short periods at several universities in the United States of America.

Russell's first major work was his book *The Principles of Mathematics* published in 1903, where he presented and discussed his famous paradoxical definition of the set of all sets that are not members of themselves, which became known as *Russell's paradox* (see the box at the end of Section 2.1 in Chapter 2). That discovery dealt a devastating blow to Cantor's set theory and to Frege's formalization of logic (see Section 4.1) and played a crucial role in bringing about a major crisis in the foundations of mathematics at the beginning of the 20th century. In this book and in the following three-volume *Principia Mathematica*, written with Whitehead and published during 1910–1913, Russell advanced the theory of *logicism*. The aim of logicism was to base the entire mathematics on purely logical principles. It was one of the three main philosophical approaches to reforming the foundations of mathematics, prompted by the foundational crisis. According to the logicistic theory, first put forward by Frege, all mathematical truths can be translated into logical truths and all mathematical proofs can be transformed into logical proofs, and therefore all mathematical results are theorems of logic. In Russell's words, "The fact that all Mathematics is Symbolic Logic is one of the greatest discoveries of our age; and when this fact has been established, the remainder of the principles of mathematics consists in the analysis of Symbolic Logic itself." With these books and later philosophical works Russell became immensely influential for many prominent philosophers and mathematicians, including his genius student Wittgenstein.

Russell was a staunch pacifist and shortly after the First World War he was convicted and imprisoned for 6 months for his antiwar views and speeches, where he wrote his extremely popular *Introduction to Mathematical Philosophy*. He pursued his antiwar activities and kept a high political profile, which brought him worldwide fame well beyond academic circles, until his death in 1970 at the age of 97.

Russell also had a very eventful personal life, with four marriages and a number of intermittent affairs. His often revolutionary views and writings on marriage, morals and society were controversial and in 1940 caused his appointment at the City College New York to be revoked. However, 10 years later Russell received the Nobel Prize for literature, *inter alia*, for his book *Marriage and Morals*.

In summary, Russell was one of the most influential logicians and philosophers of the 20th century and one of the most notable intellectual personalities of his time. He said of himself: "Three passions, simple but overwhelmingly strong, have governed my life: the longing for love, the search for knowledge, and unbearable pity for the suffering of mankind."

4.4. Truth, logical validity, equivalence and consequence in first-order logic

4.4.1. More on truth of sentences in structures. Models and countermodels

Recall that a sentence is a formula with no free variables.

As we noted earlier, the truth of a sentence in a given structure *does not depend on the variable assignment*. Therefore, for a structure S and sentence A we can simply write

$$S \vDash A$$

if $S, v \vDash A$ for *any* (hence, *every*) variable assignment v .

We then say that S is a **model** of A and that defstyle A is true in S or that defstyle A is satisfied by S . Otherwise we write $S \not\vDash A$ and say that defstyle A is false in S or that S is a **countermodel for A** .

Example 4.4.1

True sentences express properties of structures, for example:

- $\mathcal{R} \vDash \forall x \forall y (x + y = y + x)$ (commutativity of the addition of real numbers).
- $\mathcal{R} \vDash \forall x \forall y \forall z ((x + y) \times z = (x \times z) + (y \times z))$ (distributivity of multiplication over addition of real numbers).
- $\mathcal{N} \vDash \forall x \exists y (x < y)$ (for every natural number there is a larger one), while $\mathcal{N} \not\vDash \forall x \exists y (y < x)$ (not for every natural number is there a smaller one).
Hence, $\mathcal{N} \vDash \neg \forall x \exists y (y < x)$.
However, $\mathbb{Z} \vDash \forall x \exists y (y < x)$.
- $\mathbb{Q} \vDash \forall x \forall y (x < y \rightarrow \exists z (x < z \wedge z < y))$ (the ordering of the rationals is dense), while $\mathbb{Z} \not\vDash \forall x \forall y (x < y \rightarrow \exists z (x < z \wedge z < y))$ (the ordering of the integers is not dense).
- $\mathbb{Q} \vDash \exists x (3x = 1)$, but $\mathbb{Z} \not\vDash \exists x (3x = 1)$ ($1/3$ is a rational number, but is not an integer).
- $\mathcal{R} \vDash \forall x (x > \mathbf{0} \rightarrow \exists y (y^2 = x))$ (every positive real number is a square of a real number), but $\mathbb{Q} \not\vDash \forall x (x > \mathbf{0} \rightarrow \exists y (y^2 = x))$ (not every positive rational number is a square of a rational number).
- $\mathcal{R} \not\vDash \forall x \forall y (xy > \mathbf{0} \rightarrow (x > \mathbf{0} \vee y > \mathbf{0}))$. (Take, for example, $x = y = -1$.)
- $\mathcal{R} \vDash \exists x \forall y (xy < \mathbf{0} \rightarrow y = \mathbf{0})$?
- $\mathcal{H} \vDash \forall x \forall y (\exists z (x = m(z) \wedge y = m(z)) \rightarrow x = y)$.
What does this sentence mean? Is it only true for this interpretation?
- $\mathcal{N} \vDash (\forall x (\forall y (y < x \rightarrow P(y)) \rightarrow P(x)) \rightarrow \forall x P(x))$, where P is any (uninterpreted) unary predicate.
Why is this true? What does this sentence say about \mathcal{N} ? Is it also true in \mathbb{Z} ?

4.4.2. Satisfiability and validity of first-order formulae

A first-order formula A is:

- **satisfiable** if $S, v \vDash A$ for some structure S and some variable assignment v in S ;
- **(logically) valid**, denoted $\models A$, if $S, v \vDash A$ for every structure S and every variable assignment v in S .

If A is a sentence, the variable assignment in these definitions is not relevant and can be omitted. Actually, validity of any formula can be reduced to a validity of a sentence, as follows.

Definition 4.4.2 Given a first-order formula A with free variables x_1, \dots, x_n , the sentence $\forall x_1 \dots \forall x_n A$ is called a *universal closure* of A .

Note that, depending on the order of the quantified variables, a formula can have several universal closures but, as we will see further, they are all essentially equivalent. The following is an easy exercise to prove.

Proposition 4.4.3 A formula A is logically valid if and only if any of its universal closures is logically valid.

Example 4.4.4

Here are some examples of logically valid and non-valid first-order formulae:

- Every first-order instance of a tautology is logically valid.
Thus, for instance, $\models \neg\neg(x > 0) \rightarrow (x > 0)$ and $\models P(x) \vee \neg P(x)$, for any unary predicate P .
- If A is a logically valid formula and x is any variable then $\forall x A$ is also valid.
Thus, $\models \forall x(P(x) \vee \neg P(x))$, as well as $\models \forall x(P(y) \vee \neg P(y))$.
- $\models \forall x(x = x)$, because of the very meaning of the equality symbol $=$.
- $\models \forall x \forall y (\exists z(x = f(z) \wedge y = f(z)) \rightarrow x = y)$, where f is any unary functional symbol, because of the very meaning of the notion of a function.
- The sentence $\exists x P(x)$ is *not valid*: take for instance $P(x)$ to be interpreted as the empty set in any nonempty domain. This will give a *countermodel* for that sentence. However, $\exists x P(x)$ is *satisfiable*; a model is any structure where the interpretation of $P(x)$ is nonempty.
- The sentence $\forall x P(x) \vee \forall x \neg P(x)$ is not valid, but it is *satisfiable*.
(Find a model and a countermodel.)
- The sentence $\exists x(P(x) \wedge \neg P(x))$ is *not satisfiable*. Why?
- $\models \exists x \forall y P(x, y) \rightarrow \forall y \exists x P(x, y)$.
It is a good exercise to check this from the definition; we will discuss it again later.
- However, $\not\models \forall y \exists x P(x, y) \rightarrow \exists x \forall y P(x, y)$. Find a countermodel!

4.4.3. Logical equivalence in first-order logic

Logical equivalence in first-order logic is based on the same idea as in propositional logic: the first-order formulae A and B are logically equivalent if always one of them is true if and only if the other is true. Formally, this is given as follows.

Definition 4.4.5 *The first-order formulae A and B are logically equivalent, denoted $A \equiv B$, if for every structure S and variable assignment v in S :*

$$S, v \models A \text{ if and only if } S, v \models B.$$

In particular, if A and B are sentences, $A \equiv B$ means that every model of A is a model of B and every model of B is a model of A .

An equivalent definition: the first-order formulae A and B are **logically equivalent**, denoted $A \equiv B$, if $A \leftrightarrow B$ is logically valid.

Some basic properties of logical equivalence:

1. $A \equiv A$.
2. If $A \equiv B$ then $B \equiv A$.
3. If $A \equiv B$ and $B \equiv C$ then $A \equiv C$.
4. If $A \equiv B$ then $\neg A \equiv \neg B$, $\forall x A \equiv \forall x B$ and $\exists x A \equiv \exists x B$.
5. If $A_1 \equiv B_1$ and $A_2 \equiv B_2$ then $A_1 \circ A_2 \equiv B_1 \circ B_2$, where \circ is any of \wedge , \vee , \rightarrow and \leftrightarrow .

Example 4.4.6

Here are some examples of logical equivalences between first-order formulae:

- Any first-order instance of a pair of equivalent propositional formulae is a pair of logically equivalent formulae. For example:

$$\begin{aligned} \neg\neg\exists x Q(x, y) &\equiv \exists x Q(x, y) \text{ (being an instance of } \neg\neg p \equiv p \text{) and} \\ \exists x P(x) \rightarrow Q(x, y) &\equiv \neg\exists x P(x) \vee Q(x, y) \text{ (an instance of } p \rightarrow q \equiv \neg p \vee q \text{).} \end{aligned}$$

- The result of renaming a variable in a formula is logically equivalent to that formula. For example:

$$\exists x (5 < x \vee x^2 + x - 2 = 0) \equiv \exists y (5 < y \vee y^2 + y - 2 = 0).$$

Therefore, *every formula can be transformed into a logically equivalent clean formula*.

- $\neg\exists x P(x)$ and $\exists x \neg P(x)$ are not logically equivalent. Indeed, "There is no student who passed the exam" is not equivalent to "There is a student who did not pass the exam".
- "The integer x is not less than 0" is *not logically equivalent* to "The integer x is greater than or equal to 0". Mathematically these mean the same, because of the mathematical property of the ordering of integers, called "*trichotomy*", which arranges all integers in a line, but *not* because of logical reasons. Likewise, $2+2=4$ is a mathematical, not a logical, truth. To put it simply: *logic does not know any mathematics*. It is important to distinguish logical from nonlogical truths and logical from nonlogical equivalences.

4.4.4. Some logical equivalences involving quantifiers

1. To begin with, the negation changes the quantifiers as follows:

$$\begin{aligned}\neg \forall x A &\equiv \exists x \neg A \\ \neg \exists x A &\equiv \forall x \neg A\end{aligned}$$

For example:

- “Not every student wrote the test” means the same as “There is a student who did not write the test”.
- “There is no natural number less than 0” means “Every natural number is not less than 0”.

2. By negating both sides of the equivalences above we find that each of the universal and existential quantifier is *definable* in terms of the other:

$$\begin{aligned}\forall x A &\equiv \neg \exists x \neg A, \\ \exists x A &\equiv \neg \forall x \neg A.\end{aligned}$$

3. Universal quantifiers distribute over conjunctions:

$$\forall x P \wedge \forall x Q \equiv \forall x(P \wedge Q).$$

4. Existential quantifiers distribute over disjunctions:

$$\exists x P \vee \exists x Q \equiv \exists x(P \vee Q).$$

5. Assuming that x does not occur free in Q , the following distributive equivalences hold:

- $\forall x P \wedge Q \equiv Q \wedge \forall x P \equiv \forall x(P \wedge Q),$
- $\forall x P \vee Q \equiv Q \vee \forall x P \equiv \forall x(P \vee Q),$
- $\exists x P \vee Q \equiv Q \vee \exists x P \equiv \exists x(P \vee Q),$
- $\exists x P \wedge Q \equiv Q \wedge \exists x P \equiv \exists x(P \wedge Q),$

6. Two nested quantifiers of the same type commute:

$$\begin{aligned}\forall x \forall y A &\equiv \forall y \forall x A, \\ \exists x \exists y A &\equiv \exists y \exists x A.\end{aligned}$$

7. Consequently, every two universal closures of a formula are logically equivalent.

8. The case of two different nested quantifiers is more delicate: while

$$\exists x \forall y A \rightarrow \forall y \exists x A$$

is valid (see that for yourself), the converse implication

$$\forall y \exists x A \rightarrow \exists x \forall y A$$

is not. For instance,

“For every integer x there is an integer y such that $x + y = 0$ ”,

which is true, certainly does not imply that

"There is an integer y such that for every integer x it holds that $x+y=0$ ",

which is false. Likewise, "Every man loves a woman" does not imply that "There is a woman whom every man loves".

Therefore, $\exists x \forall y A$ and $\forall y \exists x A$ are *not equivalent*. It is important to remember which of these implies the other, and why.

4.4.5. Negating first-order formulae

Using the first pair of equivalences in Section 4.4.4, in addition to those from propositional logic, we can now systematically negate not only propositions but any first-order sentences as well. For example, the sentence

"For every car, there is a driver who, if (s)he can start it, then (s)he can stop it"

can be formalized in a suitable language as

$$\forall x (\text{Car}(x) \rightarrow \exists y (\text{Driver}(y) \wedge (\text{Start}(x, y) \rightarrow \text{Stop}(x, y)))).$$

Now, we negate:

$$\begin{aligned} & \neg \forall x (\text{Car}(x) \rightarrow \exists y (\text{Driver}(y) \wedge (\text{Start}(x, y) \rightarrow \text{Stop}(x, y))))) \\ & \equiv \exists x \neg (\text{Car}(x) \rightarrow \exists y (\text{Driver}(y) \wedge (\text{Start}(x, y) \rightarrow \text{Stop}(x, y))))) \\ & \equiv \exists x (\text{Car}(x) \wedge \neg \exists y (\text{Driver}(y) \wedge (\text{Start}(x, y) \rightarrow \text{Stop}(x, y))))) \\ & \equiv \exists x (\text{Car}(x) \wedge \forall y \neg (\text{Driver}(y) \wedge (\text{Start}(x, y) \rightarrow \text{Stop}(x, y))))) \\ & \equiv \exists x (\text{Car}(x) \wedge \forall y (\neg \text{Driver}(y) \vee \neg (\text{Start}(x, y) \rightarrow \text{Stop}(x, y))))) \\ & \equiv \exists x (\text{Car}(x) \wedge \forall y (\neg \text{Driver}(y) \vee (\text{Start}(x, y) \wedge \neg \text{Stop}(x, y)))). \end{aligned}$$

Since $\neg A \vee B \equiv A \rightarrow B$, the last formula is equivalent to

$$\exists x (\text{Car}(x) \wedge \forall y (\text{Driver}(y) \rightarrow (\text{Start}(x, y) \wedge \neg \text{Stop}(x, y)))).$$

Thus, the negation of the sentence above is equivalent to:

"There is a car such that every driver can start it and cannot stop it."

Note that the restricted quantifiers satisfy the same equivalences (and nonequivalences) mentioned in Section 4.4.4. In particular,

$$\begin{aligned} \neg \forall x (P(x) \rightarrow A) & \equiv \exists x (P(x) \wedge \neg A), \\ \neg \exists x (P(x) \wedge A) & \equiv \forall x (P(x) \rightarrow \neg A), \end{aligned}$$

and hence:

$$\begin{aligned} \neg \forall x \in X(A) & \equiv \exists x \in X(\neg A), \\ \neg \exists x \in X(A) & \equiv \forall x \in X(\neg A). \end{aligned}$$

Thus, the negation of the definition of a limit of a function:

$$\neg(\forall \epsilon > 0 \exists \delta > 0 (\forall x < c |x - c| < \delta \rightarrow |f(x) - L| < \epsilon))$$

is logically equivalent to

$$\exists \epsilon > 0 \forall \delta > 0 (\forall x < c |x - c| < \delta \wedge \neg |f(x) - L| < \epsilon).$$

4.4.6. Logical consequence in first-order logic

The notion of logical consequence is fundamental in logic. In propositional logic we defined the logical consequence $A_1, \dots, A_n \models B$ as “preservation of the truth from the premises to the conclusion”. The same idea applies in first-order logic, but instead of truth tables we now use *truth in a structure*.

Definition 4.4.7 A first-order formula B is a *logical consequence* from the formulae A_1, \dots, A_n , denoted $A_1, \dots, A_n \models B$, if for every structure S and every variable assignment v in S for which the formulae A_1, \dots, A_n are satisfied (equivalently, $A_1 \wedge \dots \wedge A_n$ is satisfied) in S by v , the formula B is also satisfied in S by v . Formally:

$$S, v \models A_1 \wedge \dots \wedge A_n \quad \text{implies} \quad S, v \models B.$$

In particular, if A_1, \dots, A_n, B are sentences, then $A_1, \dots, A_n \models B$ means that B is true in every structure in which all A_1, \dots, A_n are true.

If $A_1, \dots, A_n \models B$, we say that B follows logically from A_1, \dots, A_n or that A_1, \dots, A_n logically imply B .

Note that:

- $\emptyset \models A$ iff $\models A$.
- $A_1, \dots, A_n \models B$ if and only if $\models A_1 \wedge \dots \wedge A_n \rightarrow B$.

Thus, the logical consequence in first-order logic is reduced to logical validity.

Example 4.4.8

Some examples and nonexamples of first-order logical consequences are:

1. If $A \equiv B$ then $A \models B$ and $B \models A$. Why?
2. If A_1, \dots, A_n, B are propositional formulae such that $A_1, \dots, A_n \models B$ and A'_1, \dots, A'_n, B' are first-order instances of A_1, \dots, A_n, B obtained by the same substitution, then $A'_1, \dots, A'_n \models B'$. For instance, $\exists x A, \exists x A \rightarrow \forall y B \models \forall y B$.
3. $\forall x A(x) \models A[t/x]$ for any formula A and term t free for x in A .

Indeed, if $\forall x A(x)$ is true in a structure S , under some assignment, then A is true for every possible value of x in S , including the value of t under that assignment.

To put it in simple words: *if everybody is mortal then, in particular, John is mortal and the mother of Mary is mortal, etc.*

4. $\forall x(P(x) \rightarrow Q(x)), \forall xP(x) \models \forall xQ(x)$. Why? Note that this is *not* an instance of a propositional logical consequence.
5. $\exists xP(x) \wedge \exists xQ(x) \not\models \exists x(P(x) \wedge Q(x))$.

Indeed, the structure \mathcal{N}' obtained from \mathcal{N} where $P(x)$ is interpreted as “ x is even” and $Q(x)$ is interpreted as “ x is odd” is a countermodel:

$\mathcal{N}' \models \exists xP(x) \wedge \exists xQ(x)$, while $\mathcal{N}' \not\models \exists x(P(x) \wedge Q(x))$.



Example 4.4.9

Here we illustrate “semantic reasoning”, based on the formal semantics of first-order formulae, for proving or disproving first-order logical consequences.

1. Show the logical validity of the following argument:

“If Charlie suffers from slyme disease and everyone who suffers from slyme has pink eyes, then somebody has pink eyes.”

Proof. Let us first formalize the argument in first-order logic by introducing predicates: $P(x)$ meaning “ x suffers from slyme disease”, $Q(x)$ meaning “ x has pink eyes”, and a constant symbol c interpreted as “Charlie”.

Now the argument can be formalized as follows:

$$(1) (P(c) \wedge \forall x(P(x) \rightarrow Q(x)) \models \exists yQ(y).$$

To show its validity take any structure S for the first-order language introduced above and any variable assignment v in S . Now, suppose

$$(2) S, v \models (P(c) \wedge \forall x(P(x) \rightarrow Q(x)).$$

We have to show that

$$(3) S, v \models \exists yQ(y).$$

From the assumption (2) we have, by the truth definition of \wedge , that

$$(4) S, v \models P(c) \text{ and}$$

$$(5) S, v \models \forall x(P(x) \rightarrow Q(x)).$$

Let v' be a variable assignment obtained from v by redefining it on x as follows: $v'(x) = c^S$. (Recall that c^S is the interpretation of the constant symbol c in S .) Then:

$$(6) S, v' \models P(x).$$

According to the truth definition of \forall , it follows from (5) that

(7) $S, v' \models P(x) \rightarrow Q(x)$.

From (6) and (7) it follows that

(8) $S, v' \models Q(x)$.

Now, let v'' be a variable assignment obtained from v by redefining it on y as follows: $v''(y) = v'(x) = c^S$. Then we have

(9) $S, v'' \models Q(y)$.

According to the truth definition of \exists , it follows from (9) that

(3) $S, v \models \exists y Q(y)$, so we are done.

2. Prove that the following argument is not logically valid:

"If everything is black or white then everything is black or everything is white."

Proof: Again, we first formalize the argument in first-order logic, by introducing predicates:

$B(x)$ meaning " x is black" and $W(x)$ meaning " x is white".

Now the argument can be formalized as follows:

$$\forall x(B(x) \vee W(x)) \models \forall xB(x) \vee \forall xW(x).$$

To falsify this logical consequence it suffices to find *any countermodel*, i.e. a structure S and a variable assignment v such that

$$S, v \models \forall x(B(x) \vee W(x)) \text{ and } S, v \not\models \forall xB(x) \vee \forall xW(x).$$

Note that the interpretations of the predicates B and W need not have anything to do with black, white or any colour in that structure. Therefore, we choose to take the structure S with the domain the set \mathbf{Z} of all integers, where the interpretation of B is the predicate Even, where $\text{Even}(n)$ means " n is even", and the interpretation of W is the predicate Odd, where $\text{Odd}(n)$ means " n is odd".

We already know that the variable assignment is irrelevant here because there are no free variables in this argument, but we will nevertheless need it in order to process the truth definitions of the quantifiers.

Consider *any* variable assignment v in S . Then take any variable assignment v' that differs from v possibly only on x . As the integer $v'(x)$ is even or odd, $S, v' \models B(x)$ or $S, v' \models W(x)$; hence (by the truth definition of \vee):

(1) $S, v' \models B(x) \vee W(x)$.

Thus, by the truth definition of \forall , it follows that

(2) $S, v \models \forall x(B(x) \vee W(x))$.

On the other hand, consider the variable assignment v_1 obtained from v by redefining it on x as follows: $v_1(x) = 1$. Then

(3) $S, v_1 \not\models B(x)$.

By the truth definition of \forall , it follows that

(4) $S, v \not\models \forall x B(x)$.

Likewise, consider the variable assignment v_2 obtained from v by redefining it on x as follows: $v_2(x) = 2$. Then

(3) $S, v_2 \not\models W(x)$.

By the truth definition of \forall , it follows that

(4) $S, v \not\models \forall x W(x)$.

By the truth definition of \vee , it follows that

(5) $S, v \not\models \forall x B(x) \vee \forall x W(x)$.

Therefore, by the truth definition of \rightarrow , it follows from (2) and (5) that

(6) $S, v \not\models \forall x(B(x) \vee W(x)) \rightarrow (\forall xB(x) \vee \forall xW(x))$.

A logical consequence in first-order logic satisfies all basic properties of propositional logical consequence. In particular, the following are equivalent:

1. $A_1, \dots, A_n \models B$.
2. $\models A_1 \wedge \dots \wedge A_n \rightarrow B$.
3. $\models A_1 \rightarrow (A_2 \rightarrow \dots (A_n \rightarrow B) \dots)$.

Besides, some important additional properties refer to the quantifiers:

- (a) If $A_1, \dots, A_n \models B$ then $\forall x A_1, \dots, \forall x A_n \models \forall x B$.
- (b) If $A_1, \dots, A_n \models B$ and A_1, \dots, A_n are sentences, then $A_1, \dots, A_n \models \forall x B$, and hence $A_1, \dots, A_n \models \bar{B}$, where \bar{B} is any universal closure of B .
- (c) If $A_1, \dots, A_n \models B[c/x]$, where c is a constant symbol not occurring in A_1, \dots, A_n , then $A_1, \dots, A_n \models \forall x B(x)$.
- (d) If $A_1, \dots, A_n, A[c/x] \models B$, where c is a constant symbol not occurring in any of A_1, \dots, A_n, A , or B , then $A_1, \dots, A_n, \exists x A \models B$.
- (e) For any term t free for substitution for x in A :
 - (a) $\forall x A \models A[t/x]$.
 - (b) $A[t/x] \models \exists x A$.

A first-order logical consequence can be established by using an extension of any of the propositional deductive systems with additional rules for the quantifiers, to be introduced in the next sections.

Finally, to summarize: logical validity, consequence and equivalence in first-order logic are related just like in propositional logic, namely the following are equivalent:

1. $A \equiv B$.
2. $\models A \leftrightarrow B$.
3. $A \models B$ and $B \models A$.

4.4.7. Exercises

In all of the exercises below “semantic reasoning” refers to reasoning based on the formal semantics introduced in the advanced track. For the basic track, read “informal reasoning” instead.

- ❶ Use semantic reasoning to decide which of the following first-order formulae are logically valid. For each of those that are not, give a *countermodel*, i.e. a structure that falsifies it.

- (a) $\forall x(P(x) \vee \neg P(x))$
- (b) $\forall xP(x) \vee \forall x\neg P(x)$
- (c) $\forall xP(x) \vee \neg\forall xP(x)$
- (d) $\exists x(P(x) \vee \exists x\neg P(x))$
- (e) $\exists xP(x) \rightarrow \forall xP(x)$
- (f) $\exists xP(x) \rightarrow \exists yP(y)$
- (g) $\exists x(P(x) \rightarrow \forall yP(y))$
- (h) $\exists x(P(x) \rightarrow \forall xP(x))$
- (i) $\forall x(P(x) \rightarrow \exists yP(y))$
- (j) $\forall x(P(x) \rightarrow \forall yP(y))$
- (k) $\forall x\exists yQ(x, y) \rightarrow \forall y\exists xQ(x, y)$
- (l) $\forall x\exists yQ(x, y) \rightarrow \forall y\exists xQ(y, x)$
- (m) $\forall x(\exists yQ(x, y) \rightarrow \exists yQ(y, x))$
- (n) $\exists x\neg\exists yP(x, y) \rightarrow \forall y\neg\forall xP(x, y)$
- (o) $(\forall x\exists yP(x, y) \wedge \forall x\forall y(P(x, y) \rightarrow P(y, x))) \rightarrow \exists xP(x, x)$

- ❷ If x does not occur free in Q , use semantic arguments to show that the following logical equivalences hold.

- (a) $\forall x(P \vee Q) \equiv \forall xP \vee Q$
- (b) $\exists x(P \wedge Q) \equiv \exists xP \wedge Q$
- (c) $\forall x(Q \rightarrow P) \equiv Q \rightarrow \forall xP$
- (d) $\forall x(P \rightarrow Q) \equiv \exists xP \rightarrow Q$

- (e) $\exists x(P \rightarrow Q) \equiv \forall xP \rightarrow Q$
 (f) $\exists x(Q \rightarrow P) \equiv Q \rightarrow \exists xP$
- ③ Use semantic reasoning to decide which of the following logical equivalences hold. For each one that does not hold give a countermodel, i.e. a structure in which one formula is true while the other is false.
- $\forall x(P(x) \wedge Q(x)) \equiv \forall xP(x) \wedge \forall xQ(x)$
 - $\forall x(P(x) \vee Q(x)) \equiv \forall xP(x) \vee \forall xQ(x)$
 - $\forall x(P(x) \rightarrow Q(x)) \equiv \forall xP(x) \rightarrow \forall xQ(x)$
 - $\exists x(P(x) \wedge Q(x)) \equiv \exists xP(x) \wedge \exists xQ(x)$
 - $\exists x(P(x) \vee Q(x)) \equiv \exists xP(x) \vee \exists xQ(x)$
 - $\exists x(P(x) \rightarrow Q(x)) \equiv \exists xP(x) \rightarrow \exists xQ(x)$
 - $\forall xQ(x, x) \equiv \forall x\exists yQ(x, y)$
 - $\forall x\exists yQ(x, y) \equiv \exists xQ(x, x)$
 - $\exists x\forall yQ(x, y) \equiv \exists yQ(y, y)$
 - $\forall x\forall yQ(x, y) \equiv \forall x\forall yQ(y, x)$
 - $\exists xP(x) \rightarrow \forall xP(x) \equiv \exists x\neg P(x) \rightarrow \forall x\neg P(x)$
- ④ Negate each of the following formulae and import the negations inside all other logical connectives.
- $\forall x(x = x^2 \rightarrow x > 0)$
 - $\forall x((x = x^2 \wedge x > 1) \rightarrow x^2 < 1)$
 - $\forall x\exists y(x > y \rightarrow x > y^2)$
 - $\forall x(x = 0 \vee \exists y\neg(xy = x))$
 - $\forall x\exists y(\neg x = y \rightarrow x > y)$
 - $\exists x\exists y(x > y \vee -x > y)$
 - $\exists x(P(x) \rightarrow \forall yP(y))$
 - $\forall x(P(x) \rightarrow Q(x)) \rightarrow (\forall xP(x) \rightarrow \forall xQ(x))$
 - $(\exists xP(x) \rightarrow \exists xQ(x)) \rightarrow \exists x(P(x) \rightarrow Q(x))$
- ⑤ Using semantic reasoning, show that the following logical consequences hold.
- $\forall xA(x) \vDash \neg\exists x\neg A(x)$
 - $\neg\exists x\neg A(x) \vDash \forall xA(x)$
 - $\exists xA(x) \vDash \neg\forall x\neg A(x)$

(d) $\neg\forall x\neg A(x) \vDash \exists xA(x)$

(e) $\exists x\exists yA(x, y) \vDash \exists y\exists xA(x, y)$

- ⑥ Suppose x is not free in Q . Show that the following logical consequences hold by giving semantic arguments.

(a) $\forall x(P(x) \vee Q) \vDash \forall xP(x) \vee Q$

(b) $\forall xP(x) \vee Q \vDash \forall x(P(x) \vee Q)$

(c) $\exists x(P(x) \wedge Q) \vDash \exists xP(x) \wedge Q$

(d) $\exists xP(x) \wedge Q \vDash \exists x(P(x) \wedge Q)$

(e) $\forall x(Q \rightarrow P(x)), Q \vDash \forall xP(x)$

(f) $\exists xP(x) \rightarrow Q \vDash \forall x(P(x) \rightarrow Q)$

(g) $\exists x(P(x) \rightarrow Q), \forall xP(x) \vDash Q$

(h) $\exists x(Q \rightarrow P(x)), Q \vDash \exists xP(x)$

- ⑦ Using semantic reasoning, determine which of the following logical consequences hold. For those that do not, construct a *countermodel*, i.e. a structure in which all premises are true, while the conclusion is false.

(a) $\forall xA(x), \forall xB(x) \vDash \forall x(A(x) \wedge B(x))$

(b) $\forall x(A(x) \wedge B(x)) \vDash \forall xA(x) \wedge \forall xB(x)$

(c) $\forall xA(x) \vee \forall xB(x) \vDash \forall x(A(x) \vee B(x))$

(d) $\forall x(A(x) \vee B(x)) \vDash \forall xA(x) \vee \forall xB(x)$

(e) $\forall xA(x) \rightarrow \forall xB(x) \vDash \forall x(A(x) \rightarrow B(x))$

(f) $\forall x(A(x) \rightarrow B(x)) \vDash \forall xA(x) \rightarrow \forall xB(x)$

(g) $\exists x(A(x) \wedge B(x)) \vDash \exists xA(x) \wedge \exists xB(x)$

(h) $\exists xA(x), \exists xB(x) \vDash \exists x(A(x) \wedge B(x))$

(i) $\exists x(A(x) \vee B(x)) \vDash \exists xA(x) \vee \exists xB(x)$

(j) $\exists xA(x) \vee \exists xB(x) \vDash \exists x(A(x) \vee B(x))$

(k) $\exists x(A(x) \rightarrow B(x)) \vDash \exists xA(x) \rightarrow \exists xB(x)$

(l) $\exists xA(x) \rightarrow \exists xB(x) \vDash \exists x(A(x) \rightarrow B(x))$

- ⑧ Check for each of the following statements whether it logically implies the other, by formalizing them in first-order logic and using semantic reasoning.

A: “Not everybody who is a lawyer is greedy.”

B: “There exists a lawyer and not everybody is greedy.”

If either of the logical consequences does not hold, give an appropriate countermodel.

- ⑨ Check for each of the following statements if it logically implies the other, by formalizing them in first-order logic and using semantic arguments.

A: "No man is happy only when he is drunk."

B: "There is a happy man and there is a man who is not drunk."

If either of the logical consequences does not hold, give an appropriate countermodel.

- ⑩ Formalize the following arguments in first-order logic (FOL), by introducing suitable predicates. Then, using semantic reasoning, determine which of them is logically correct.

All logicians are clever.

(a) All clever people are rich.

All logicians are rich.

All natural numbers are integers.

(b) Some integers are odd numbers.

Some natural numbers are odd numbers.

All penguins are birds.

(c) Some penguins are white.

Therefore, some white bird is a penguin.

No yellow plonks are qlinks.

(d) Some object is a plonk or not a qlink.

Therefore, there is an object that is not a yellow qlink.



Alfred Tarski, born Alfred Teitelbaum (14.1.1901–26.10.1983)

Alfred Tarski was a Polish-American logician and mathematician, who made seminal contributions to many areas of mathematics, including set theory, universal algebra, topology, geometry, classical and universal algebra and several branches of mathematical logic, including theories of truth, logical consequence and definability, algebraic logic, model theory and metamathematics (proof theory), as well as the methodology of science.

At the age of 22 Tarski became the youngest student to be awarded a PhD from the University of Warsaw. It was at that time that he changed his religion from Jewish to Roman Catholic and his surname from Teitelbaum to Tarski, to reaffirm his Polish identity, but also to make it possible to get an academic position which, at that time, was almost impossible for Jews in Poland.

Even though Tarski did not have a university position in Warsaw and taught mostly in high schools there until 1939, he had already made key contributions to logic and set theory, which brought him international recognition. In 1933 Tarski published his groundbreaking paper *The concept of truth in formalized languages*, where he rigorously defined the formal semantics of logical languages and the notion of truth of a logical formula. In 1936 Tarski published another seminal work, *On the concept of logical consequence*, where he precisely defined logical consequence to hold when the conclusion is true in every model in which all premises are true. Yet another work of fundamental importance was *A decision method for elementary algebra and geometry*, which he essentially completed around 1930 but only published in the United States in 1948. In it Tarski showed, using the method of quantifier elimination, that the first-order theory of the real numbers under addition and multiplication is algorithmically decidable – one of the most important results on decidability of a mathematical theory. In 1936 he also produced his very popular and influential undergraduate textbook on logic, *Introduction to Logic and to the Methodology of Deductive Sciences*.

In 1939, during Tarski's visit to the United States for a conference, the Nazi army invaded Poland and started the Second World War, so Tarski could not return to his country and remained in the United States, where he became a naturalized citizen in 1945. From 1942 Tarski worked as a lecturer and from 1949 until his retirement as a Professor of Mathematics at the University of California at Berkeley, where he organized his famous logic seminar and created a very strong research environment.

Tarski was famous for his extreme productivity and precision, both in his teaching and his scientific writings. He was an energetic and charismatic teacher, whose lectures were meticulously prepared and presented. He was an inspiring and demanding supervisor of 26 doctoral students, many of whom became distinguished mathematicians. Tarski was also notorious for his all-night working sessions with his students in his smoke-filled study in the basement of his home in Berkeley. He had a strong and colourful personality: a heavy smoker and drinker, a womanizer, extremely ambitious and self-righteous, yet very civilized and polite.

With his enormous legacy in scientific works, students, followers and general scientific influence, Tarski is widely regarded as one of the greatest logicians of all times, along with Aristotle, Frege and Gödel.

4.5. Semantic tableaux for first-order logic

Here we extend the system of propositional semantic tableaux to first-order logic by adding rules for the quantifiers. The resulting deductive system is sound and complete for the valid logical consequences of first-order logic.

Quantifier rules for semantic tableaux

$\forall x A(x) : T$ \downarrow $A[t/x] : T$ <small>for any ground term^(*) t</small>	$\exists x A(x) : T$ \downarrow $A[c/x] : T$ <small>for a new constant symbol c not yet occurring on this branch^(**)</small>
$\exists x A(x) : F$ \downarrow $A[t/x] : F$ <small>for any ground term^(*) t</small>	$\forall x A(x) : F$ \downarrow $A[c/x] : F$ <small>for a new constant symbol c not yet occurring on this branch^(**)</small>

(*) Recall that a ground term is a term with no variables. In order to always enable this rule we assume that there is at least one constant symbol in the language.

(**) This rule may be applied only once for the given formula on each branch.

The rules are quite natural and their correctness, which will be stated more formally further, follows from some semantic properties of quantifiers and logical consequence that we listed in Section 4.4.6. Still, some discussion is in order. We defer the discussion on the quantifier rules $(\forall T)$ and $(\exists F)$ to the end of this section. Now, let us explain the proviso (**). It is only needed to prevent “useless” redundant applications of the rules $(\exists T)$ and $(\forall F)$. Note that, without that proviso, each of these rules could be applied in principle infinitely many times – once for every constant symbol in the language that does not occur yet on the current branch. Intuitively, however, such applications should be redundant, because we only need *one* witness of the truth (respectively of falsity) of the formula assumed to be true in the rule $(\exists T)$ (respectively false in $(\forall F)$), if such a witness exists at all. Therefore, we only need to introduce *one name* for such a witness in any given branch of the tableau. Thus, if the assumption of the existence of such a witness can lead to a contradiction and close the branch, then one application of the rule should suffice; otherwise no repeated applications of the rule can have an effect at all, but would make it impossible to declare the branch, and the tableau construction, saturated.

The idea of derivation in first-order logic by means of the method of semantic tableaux is essentially the same as in propositional logic: in order to prove that $A_1, \dots, A_n \models C$, we search systematically for a *countermodel*, i.e. a structure where A_1, \dots, A_n and $\neg C$ are satisfied simultaneously. Closed and open branches and tableaux are defined as in propositional semantic tableaux, but with the more refined notion of saturation of a first-order tableau, as discussed above. A closed tableau with the formulae $A_1, \dots, A_n, \neg C$ at the root certifies that there cannot

be such a countermodel; that constitutes a derivation in semantic tableaux of the logical consequence $A_1, \dots, A_n \models C$, which will be denoted by $A_1, \dots, A_n \vdash_{\text{ST}} C$. On the other hand, a tableau with an open and saturated branch contains the information needed for the construction of such a countermodel.

Theorem 4.5.1 *The system of semantic tableaux for first-order logic is sound and complete, i.e. for every first-order formulae A_1, \dots, A_n, C :*

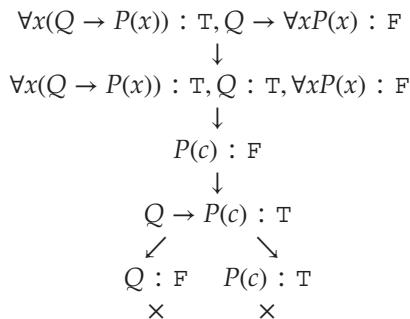
$$A_1, \dots, A_n, \vdash_{\text{ST}} C \quad \text{if and only if} \quad A_1, \dots, A_n, \models C.$$

Unlike propositional logic, where every tableau either closes or saturates and terminates as open, now a third case is also possible, where the tableau neither closes nor produces an open and saturated branch. This case will be illustrated in the last example below and termination will be discussed at the end of the section.

4.5.1. Some derivations using first-order semantic tableau

Example 4.5.2

Using semantic tableaux, check if $\forall x(Q \rightarrow P(x)) \models Q \rightarrow \forall xP(x)$, where x does not occur free in Q .



The tableau above closes, so $\forall x(Q \rightarrow P(x)) \vdash_{\text{ST}} Q \rightarrow \forall xP(x)$, and hence $\forall x(Q \rightarrow P(x)) \models Q \rightarrow \forall xP(x)$. Where was the assumption that x is not free in Q used?

Example 4.5.3

Using semantic tableaux, show that $\models \neg \forall x \forall y (P(x) \wedge \neg P(f(y)))$, where P is a unary predicate symbol and f is a unary functional symbol.

$$\begin{array}{c}
 \neg \forall x \forall y (P(x) \wedge \neg P(f(y))) : F \\
 \downarrow \\
 \forall x \forall y (P(x) \wedge \neg P(f(y))) : T \\
 \downarrow \\
 \forall y (P(f(c)) \wedge \neg P(f(y))) : T \\
 \downarrow \\
 P(f(c)) \wedge \neg P(f(c)) : T \\
 \downarrow \\
 P(f(c)) : T, \neg P(f(c)) : T \\
 \downarrow \\
 P(f(c)) : T, P(f(c)) : F \\
 \downarrow \\
 \times
 \end{array}$$

The tableau closes, so $\vdash_{ST} \neg \forall x \forall y (P(x) \wedge \neg P(f(y)))$, and hence $\models \neg \forall x \forall y (P(x) \wedge \neg P(f(y)))$. In this example we first applied the rule ($\forall T$) in a somewhat artificial way, by introducing a new constant symbol and guessing the ground term $f(c)$ to substitute for x . Such guessing would be avoided in the free variable version of the semantic tableaux method, mentioned at the end of the section.

Example 4.5.4

Using semantic tableaux, check if $\exists x (A(x) \rightarrow B(x)), \exists x A(x) \models \exists x B(x)$.

$$\begin{array}{c}
 \exists x (A(x) \rightarrow B(x)) : T, \exists x A(x) : T, \exists x B(x) : F \\
 \downarrow \\
 A(c_1) \rightarrow B(c_1) : T \\
 \downarrow \\
 A(c_2) : T \\
 \downarrow \\
 B(c_1) : F \\
 \downarrow \\
 B(c_2) : F \\
 \swarrow \quad \searrow \\
 A(c_1) : F \quad B(c_1) : T \\
 \circ \qquad \times
 \end{array}$$

Note that, due to the proviso (*), the left-hand branch in this tableau can only be extended any further by applying the rule $\exists F$ to the signed formula $\exists xB(x) : F$ to produce instances of the type $B(c) : F$ for new constant symbols c (as there are no function symbols in the formula), which clearly cannot produce a contradictory pair of formulae, so we can treat such applications of the rule as redundant. Therefore, the left-hand branch remains open. Thus, the tableau does not close, so $\exists x(A(x) \rightarrow B(x)), \exists xA(x) \vDash \exists xB(x)$ cannot be derived in ST and hence this consequence is not valid. Indeed, the necessary information for building a countermodel can be collected from the open branch of the tableau: it must have at least two elements, say a_1 and a_2 , interpreting respectively c_1 and c_2 , and these should suffice because no other names of elements are mentioned in the tableau. Then the predicate A must be false for a_1 and true for a_2 , i.e. its interpretation in the domain $\{a_1, a_2\}$ is $\{a_2\}$, while the predicate B must be false for both a_1 and a_2 , i.e. its interpretation in the domain $\{a_1, a_2\}$ is \emptyset . Indeed, it is easy to check that the resulting model falsifies the logical consequence $\exists x(A(x) \rightarrow B(x)), \exists xA(x) \vDash \exists xB(x)$.

Example 4.5.5

Using semantic tableaux, check if $\forall x \exists y A(x, y) \vDash \exists x A(x, c_0)$.

$$\begin{array}{c}
 \forall x \exists y A(x, y) : T, \exists x A(x, c_0) : F \\
 \downarrow \\
 A(c_0, c_0) : F \\
 \downarrow \\
 \exists y A(c_0, y) : T \\
 \downarrow \\
 A(c_0, c_1) : T \\
 \downarrow \\
 A(c_1, c_0) : F \\
 \downarrow \\
 \exists y A(c_1, y) : T \\
 \downarrow \\
 A(c_1, c_2) : T \\
 \downarrow \\
 A(c_2, c_0) : F \\
 \vdots
 \end{array}$$

It should now be evident that this tableau can be extended *forever* without closing. This indicates that $\forall x \exists y A(x, y) \vDash \exists x A(x, c_0)$ cannot be derived in ST and hence this consequence is not valid. (Indeed, a simple countermodel can be defined in the structure of natural numbers \mathcal{N} , where $A(x, y)$ is interpreted as $x < y$ and c_0 is interpreted as 0.) However, this fact cannot be formally established at any finite stage of the extension of the tableau. Thus, we see that *the construction of a first-order tableau does not always terminate*.

4.5.2. Semantic tableaux for first-order logic with equality

The system of Semantic Tableaux presented here did not involve rules for the equality, but such rules can be added to produce a sound and complete system for first-order logic with equality.

Semantic tableaux rules for the equality.

$$\begin{array}{ccc}
 & A[t/x] : T & \\
 & \downarrow & \\
 (=_1 T) & A[s/x] : T & \\
 & \text{for any terms } s, t \text{ free for } x \text{ in } A \text{ such that} \\
 & s = t : T \text{ or } t = s : T \text{ occurs on the branch} \\
 \\
 & A[t/x] : F & \\
 & \downarrow & \\
 (=_2 F) & A[s/x] : F & \\
 & \text{for any terms } s, t \text{ free for } x \text{ in } A \text{ such that} \\
 & s = t : F \text{ or } t = s : F \text{ occurs on the branch}
 \end{array}$$

In addition, the following condition for closing a branch is added: a branch closes if a formula $t = t : F$ for any term t occurs on it.

Example 4.5.6

Using semantic tableaux with equality, check whether $\forall x(P(x) \leftrightarrow x = c) \models P(c)$ for a constant symbol c .

Here is the run of the tableau:

$$\begin{array}{c}
 \forall x(P(x) \leftrightarrow x = c) : T, P(c) : F \\
 \downarrow \\
 P(c) \leftrightarrow c = c : T \\
 \swarrow \quad \searrow \\
 P(c) : T, c = c : T, P(c) : F \quad P(c) : F, c = c : F, P(c) : F \\
 \times \qquad \qquad \qquad \times
 \end{array}$$

4.5.3. Discussion on the quantifier rules and on termination of semantic tableaux

We end this section with some discussion on the quantifier rules ($\forall T$) and ($\exists F$) and on the termination of the semantic tableaux method for first-order logic.



1. When the language has at least one constant and one functional symbol there are infinitely many ground terms that can be instantiated in the rules $(\forall T)$ and $(\exists F)$. All but finitely many of these instantiations would be “useless” for the closure of the branch of the tableau. Therefore, the rules, as they stand, appear to be “wasteful” because of allowing many “useless” applications. As we saw in some of the examples, this could lead to unnecessary nontermination, but will not affect the soundness and completeness of the system, stated in Theorem 4.5.1. In automated theorem provers these rules can be used quite efficiently by employing suitable heuristics and strategies on how to use them.
2. On the other hand, these rules can be strengthened by allowing instantiation with *any* term t that is free for x in A . The resulting rules would still be sound, but even more “wasteful” in the sense of allowing “useless” applications.
3. The problem of identifying and preventing “useless” applications of the rules $(\forall T)$ and $(\exists F)$ is very nontrivial. As we note later, this problem is, in fact, algorithmically unsolvable in any sound and complete version of the tableaux method. Intuitively, it may seem sufficient to only apply these rules for terms that have already appeared on the branch, if a contradiction is to be reached on that branch. However, such a restriction would be too strong and can prevent the tableau from closing, e.g. for the unsatisfiable input formula $\forall x \forall y (P(x) \wedge \neg P(f(y)))$.
4. There is a variation of the quantifiers rules, leading to the so-called *free variable tableaux* version of the tableaux method, which defers the term instantiations applied with the rules $(\forall T)$ and $(\exists F)$ to a later stage, where it may become clear exactly what instantiations are needed in order to close the branch, if possible. The idea is to modify the rules $(\forall T)$ and $(\exists F)$ to instantiate not with ground terms but with new free variables and also to modify the rules $(\forall F)$ and $(\exists T)$ to instantiate not with a new constant but with a term $f(v_1, \dots, v_k)$, where f is a *new* functional symbol and v_1, \dots, v_k are free variables on the branch which may have appeared there by applications of the rules $(\forall T)$ and $(\exists F)$. The technique behind this modification of the rules is called Skolemization and will be presented and discussed in Section 4.6.2. In the case when there are no such free variables, the rules act like the present ones, by instantiating with new constant symbols.

For example, the guessing in Example 4.5.3 would be avoided in the free variable tableau version by instantiating x with a new free variable v_1 , then instantiating y with another free variable v_2 and finally instantiating v_1 with $f(v_2)$. This is a simple case of the *term unification* technique, which we will discuss in detail in Section 4.7.3.

Finally, on termination. We have seen that the system of semantic tableaux presented here does not always terminate, even in cases where a suitable application of the rules would close

the tableau. Therefore, the question arises: can the rules of the system be modified so that it remains sound and complete, but also always terminates? The answer is *No!* – at least, not if the rules are to be algorithmically implementable. More precisely, in any sound and complete system of semantic tableaux it is not possible to determine algorithmically whether the construction of the tableau for any given first-order formula can terminate (as open or closed) or not. This impossibility is a direct consequence from one of the most profound and important results in mathematical logic, proved in 1936 by Alonso Church, one of the leading logicians of the 20th century, viz. the *algorithmic undecidability of the validity in first-order logic: there is no algorithm which can determine whether a given first-order formula is valid.*

4.5.4. Exercises

For each of the following exercises use semantic tableaux.

① Prove the following logical validities and consequences.

- (a) $\models \forall x P(x) \rightarrow \forall y P(y)$
- (b) $\models \exists x P(x) \rightarrow \exists y P(y)$
- (c) $\forall x A(x) \models \neg \exists x \neg A(x)$
- (d) $\neg \exists x \neg A(x) \models \forall x A(x)$
- (e) $\exists x A(x) \models \neg \forall x \neg A(x)$
- (f) $\neg \forall x \neg A(x) \models \exists x A(x)$
- (g) $\models \exists x \exists y Q(x, y) \rightarrow \exists y \exists x Q(x, y)$

② Suppose x is not free in Q . Prove the following logical consequences.

- (a) $\forall x(P(x) \vee Q) \models \forall xP(x) \vee Q$
- (b) $\forall xP(x) \vee Q \models \forall x(P(x) \vee Q)$
- (c) $\exists x(P(x) \wedge Q) \models \exists xP(x) \wedge Q$
- (d) $\exists xP(x) \wedge Q \models \exists x(P(x) \wedge Q)$
- (e) $\forall x(Q \rightarrow P(x)), Q \models \forall xP(x)$
- (f) $\exists xP(x) \rightarrow Q \models \forall x(P(x) \rightarrow Q)$
- (g) $\exists x(P(x) \rightarrow Q), \forall xP(x) \models Q$
- (h) $\exists x(Q \rightarrow P(x)), Q \models \exists xP(x)$

③ Check which of the following first-order formulae are logically valid. For each of those which are not, construct a *countermodel*, i.e., a structure which falsifies it.

- (a) $\exists x(P(x) \rightarrow \forall y P(y))$
- (b) $\forall x(P(x) \rightarrow \exists y P(y))$
- (c) $\forall x \exists y Q(x, y) \rightarrow \forall y \exists x Q(x, y)$

- (d) $\forall x \exists y Q(x, y) \rightarrow \forall y \exists x Q(y, x)$
- (e) $\forall x \exists y Q(x, y) \rightarrow \exists y \forall x Q(x, y)$
- (f) $\forall x \exists y Q(x, y) \rightarrow \exists x Q(x, x)$
- (g) $\exists y \forall x Q(x, y) \rightarrow \exists x Q(x, x)$
- (h) $\forall x (\exists y Q(x, y) \rightarrow \exists y Q(y, x))$
- (i) $\exists x \neg \exists y P(x, y) \rightarrow \forall y \neg \forall x P(x, y)$
- (j) $(\forall x \exists y P(x, y) \wedge \forall x \forall y (P(x, y) \rightarrow P(y, x))) \rightarrow \exists x P(x, x)$

④ Check which of the following logical consequences hold. For those that do not, use the tableau to construct a countermodel, i.e. a structure and assignment in which all premises are true, while the conclusion is false.

- (a) $\forall x A(x), \forall x B(x) \vDash \forall x (A(x) \wedge B(x))$
- (b) $\forall x (A(x) \wedge B(x)) \vDash \forall x A(x) \wedge \forall x B(x)$
- (c) $\forall x A(x) \vee \forall x B(x) \vDash \forall x (A(x) \vee B(x))$
- (d) $\forall x (A(x) \vee B(x)) \vDash \forall x A(x) \vee \forall x B(x)$
- (e) $\forall x A(x) \rightarrow \forall x B(x) \vDash \forall x (A(x) \rightarrow B(x))$
- (f) $\forall x (A(x) \rightarrow B(x)) \vDash \forall x A(x) \rightarrow \forall x B(x)$
- (g) $\forall x (A(x) \rightarrow B(x)) \vDash \exists x A(x) \rightarrow \exists x B(x)$
- (h) $\exists x (A(x) \wedge B(x)) \vDash \exists x A(x) \wedge \exists x B(x)$
- (i) $\exists x A(x), \exists x B(x) \vDash \exists x (A(x) \wedge B(x))$
- (j) $\exists x (A(x) \vee B(x)) \vDash \exists x A(x) \vee \exists x B(x)$
- (k) $\exists x A(x) \vee \exists x B(x) \vDash \exists x (A(x) \vee B(x))$
- (l) $\exists x (A(x) \rightarrow B(x)) \vDash \exists x A(x) \rightarrow \exists x B(x)$
- (m) $\exists x A(x) \rightarrow \exists x B(x) \vDash \exists x (A(x) \rightarrow B(x))$

⑤ Determine which of the following logical equivalences hold. For each of those that do not hold, construct a countermodel, i.e. a structure in which one formula is true while the other false.

- (a) $\forall x (P(x) \wedge Q(x)) \equiv \forall x P(x) \wedge \forall x Q(x)$
- (b) $\forall x (P(x) \vee Q(x)) \equiv \forall x P(x) \vee \forall x Q(x)$
- (c) $\forall x (P(x) \rightarrow Q(x)) \equiv \forall x P(x) \rightarrow \forall x Q(x)$
- (d) $\exists x (P(x) \wedge Q(x)) \equiv \exists x P(x) \wedge \exists x Q(x)$
- (e) $\exists x (P(x) \vee Q(x)) \equiv \exists x P(x) \vee \exists x Q(x)$
- (f) $\exists x (P(x) \rightarrow Q(x)) \equiv \exists x P(x) \rightarrow \exists x Q(x)$

- (g) $\forall x Q(x, x) \equiv \forall x \exists y Q(x, y)$
 (h) $\forall x \exists y Q(x, y) \equiv \exists x Q(x, x)$
 (i) $\exists x \forall y Q(x, y) \equiv \exists y Q(y, y)$
 (j) $\forall x \forall y Q(x, y) \equiv \forall x \forall y Q(y, x)$
 (k) $\exists x P(x) \rightarrow \forall x P(x) \equiv \exists x \neg P(x) \rightarrow \forall x \neg P(x)$
- ⑥ In each of the following cases, formalize both statements in first-order logic and check which, if any, logically implies the other. If any of the logical consequences does not hold, construct an appropriate countermodel.
- (a) A: "Not every lawyer is greedy."
 B: "There exists a lawyer and not everybody is greedy."
 - (b) A: "Some men are happy only when they are drunk."
 B: "There is a drunk man or there is a man who is not happy."
 - (c) A: "If somebody is talking then everybody is listening."
 B: "There is somebody, such that if he is not listening then nobody is talking."
- ⑦ Formalize the following arguments in FOL, by introducing suitable predicates. Then check for each of them whether it is logically valid. If not valid, use the tableau to construct a countermodel.
- (a) All logicians are clever.
 All clever people are rich.
All logicians are rich.
- (b) All natural numbers are integers.
 Some integers are odd numbers.
Some natural numbers are odd numbers.
- (c) No driver is a pedestrian.
 Every child is a pedestrian.
 Johnnie is a child.
Therefore somebody is not a driver.
- (d) All penguins are birds.
 Some penguins are white.
 No white bird eats butterflies.
Therefore, not all penguins eat butterflies.

Everybody likes some black gmuck.

Nobody likes any white gmuck.

(e) Therefore no gmuck is black and white.

Some bachelors love every woman they know.

Every bachelor is a man.

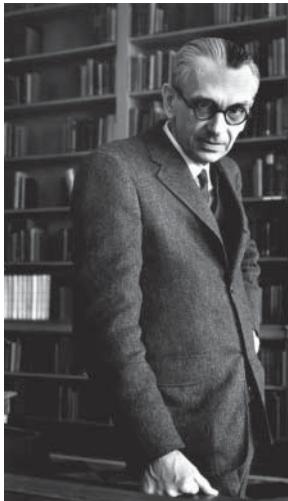
(f) Eve is a woman whom every man knows.

Some bachelor loves a woman.

(g) Replace the conclusion in the previous example by

“Every bachelor loves a woman.”

and check again whether the resulting argument is logically valid.



Kurt Friedrich Gödel (28.4.1906–14.1.1978)

Kurt Gödel was an Austrian–American logician, mathematician and philosopher, regarded as the most influential logician of the 20th century.

Gödel's first interest in logic was sparked when he attended a seminar on Russell's book *Introduction to Mathematical Philosophy*. Later, he attended a lecture by David Hilbert on the completeness and consistency of mathematical systems, where Hilbert posed the question of whether there is a consistent formal system of axioms of first-order logic that can derive every valid – that is, true in all models – statement of first-order logic. Gödel chose this problem as the topic of his doctoral work and he completed his doctoral dissertation under the supervision of Hans Hahn in 1929, at the age of 23. In his thesis he proposed an axiomatic system for the first-order predicate logic and established its completeness. This was his first famous result, known as *Gödel's completeness theorem*.

In 1931 Gödel published his most important and groundbreaking work, *On Formally Undecidable Propositions of "Principia Mathematica" and Related Systems*, where he proved his famous *incompleteness theorems*. The *first incompleteness theorem* states that any axiomatic system of first-order logic that is consistent (that is, no contradiction can be derived in it) and has an effectively recognizable (recursive) set of axioms and is expressive enough to describe the arithmetic of the natural numbers with addition and multiplication must be incomplete; i.e. there are true arithmetical statements that can be stated in the language of that system but cannot be derived from its axioms. In particular, this result applies to Peano's system of axioms for arithmetic (see Section 4.2) and to the Zermelo–Fraenkel set theory with the Axiom of choice (ZFC). The *second incompleteness theorem* states that, in particular, no such consistent system can derive a statement (formalized in the language of arithmetic) claiming its own consistency. Gödel's basic idea for the proof of the

incompleteness theorems was conceptually simple but extremely original: for any given formal axiomatic system of arithmetic with an effectively enumerable set of axioms he used a specially developed technique to encode, in the language of the arithmetic itself, the notions of formulae, axioms and derivations, and eventually constructed a formula of that language that claims, via the encoding, that it is unprovable in the given formal system. Such a formula cannot be provable in the system, for this would imply its inconsistency, and therefore what it states must be true.

Gödel's incompleteness theorems had a shattering effect on the attempts to find an effectively enumerable set of axioms sufficient to derive all true statements in mathematics, beginning with the work of Frege half a century earlier and culminating in *Principia Mathematica* and Hilbert's program for the formalization of the mathematics (see the box at the end of Section 3.3 in Chapter 3).

Gödel's theorems have also had a profound effect on mathematical logic and the foundations and philosophy of mathematics. They are one of the greatest achievements of 20th century mathematics, of fundamental importance not only for the foundations of mathematics but also for computer science, as they imply that a computer can never be programmed to derive all true mathematical statements, not even all those about the arithmetic of natural numbers.



4.6. Prenex and clausal normal forms

4.6.1. Prenex normal forms

Definition 4.6.1 Here we extend some technical concepts from propositional to first-order logic:

- A **first-order literal** is an atomic formula or a negation of an atomic formula.
- A **first-order elementary conjunction/disjunction** is a first-order literal or a conjunction/disjunction of (two or more) first-order literals.
- A **first-order disjunctive normal form (DNF)** is a first-order elementary conjunction or a disjunction of (two or more) first-order elementary conjunctions.
- Respectively, a **first-order conjunctive normal form (CNF)** is a first-order elementary disjunction or a conjunction of (two or more) first-order elementary disjunctions.

Definition 4.6.2 A first-order formula $Q_1x_1 \cdots Q_nx_n A$, where Q_1, \dots, Q_n are quantifiers and A is an open formula, is said to be in a **prenex form**. The quantifier string $Q_1x_1 \cdots Q_nx_n$ is called the **prefix** and the formula A is called the **matrix** of the prenex form.

If A is a first-order DNF (respectively CNF) then $Q_1x_1 \cdots Q_nx_n A$ is in a **prenex disjunctive normal form** (respectively a **prenex conjunctive normal form**).

Example 4.6.3

1. The formula $\forall x \exists y (\neg x > 0 \vee (y > 0 \wedge \neg x = y^2))$ is in a prenex DNF, but not a prenex CNF.
2. $\forall x \exists y (x > 0 \rightarrow (y > 0 \wedge x = y^2))$ is in a prenex form but neither in DNF nor in CNF.
3. The formula $\forall x (x > 0 \rightarrow \exists y (y > 0 \wedge x = y^2))$ is not in a prenex form.

Theorem 4.6.4 Every first-order formula is equivalent to a formula in a prenex disjunctive normal form (PDNF) and to a formula in a prenex conjunctive normal form (PCNF).

Proof: Here is an algorithm for construction of these normal forms. Given any formula A :

1. Eliminate all occurrences of \rightarrow and \leftrightarrow as in the propositional case.
2. Import all negations inside all other logical connectives, i.e. in front of atomic formulae.
3. Use the equivalences (Section 4.4.4):
 - (a) $\forall x P \wedge \forall x Q \equiv \forall x (P \wedge Q)$,
 - (b) $\exists x P \vee \exists x Q \equiv \exists x (P \vee Q)$,
to pull some quantifiers outwards and, after renaming the formula *wherever necessary*, use also the following equivalences from Section 4.4.4, where x does not occur free in Q :
 - (c) $\forall x P \wedge Q \equiv Q \wedge \forall x P \equiv \forall x (P \wedge Q)$,
 - (d) $\forall x P \vee Q \equiv Q \vee \forall x P \equiv \forall x (P \vee Q)$,
 - (e) $\exists x P \vee Q \equiv Q \vee \exists x P \equiv \exists x (P \vee Q)$,
 - (f) $\exists x P \wedge Q \equiv Q \wedge \exists x P \equiv \exists x (P \wedge Q)$,
4. Finally, transform the matrix in a DNF or CNF just like a propositional formula.

A simplified, but more inefficient procedure for getting the quantifiers out first transforms the formula into an equivalent clean one by suitable renaming of bound variables. Then it pulls out all quantifiers, by maintaining the order, tacitly using the last four equivalences above. This procedure often introduces unnecessary new variables, which makes the result not optimal for the Skolemization step of the procedure, described in Section 4.6.2.

Example 4.6.5

Let $A = \exists z(\exists xQ(x, z) \vee \exists xP(x)) \rightarrow \neg(\neg\exists xP(x) \wedge \forall x\exists zQ(z, x))$.

1. Eliminating \rightarrow : $A \equiv \neg\exists z(\exists xQ(x, z) \vee \exists xP(x)) \vee \neg(\neg\exists xP(x) \wedge \forall x\exists zQ(z, x))$.
2. Importing the negation:

$$A \equiv \forall z(\neg\exists xQ(x, z) \wedge \neg\exists xP(x)) \vee (\neg\neg\exists xP(x) \vee \neg\forall x\exists zQ(z, x)) \equiv \forall z(\forall x\neg Q(x, z) \wedge \forall x\neg P(x)) \vee (\exists xP(x) \vee \exists x\forall z\neg Q(z, x))$$
.
3. Using the equivalences (a) and (b): $A \equiv \forall z\forall x(\neg Q(x, z) \wedge \neg P(x)) \vee \exists x(P(x) \vee \forall z\neg Q(z, x))$.
4. Renaming: $A \equiv \forall z\forall x(\neg Q(x, z) \wedge \neg P(x)) \vee \exists y(P(y) \vee \forall w\neg Q(w, y))$.
5. Using the equivalences (c) to (f) and pulling the quantifiers in front:

$$A \equiv \forall z\forall x\exists y\forall w((\neg Q(x, z) \wedge \neg P(x)) \vee P(y) \vee \neg Q(w, y))$$
.
6. The resulting formula is in a prenex DNF. For a prenex CNF we need to distribute the \vee over \wedge : $A \equiv \forall z\forall x\exists y\forall w((\neg Q(x, z) \vee P(y) \vee \neg Q(w, y)) \wedge (\neg P(x) \vee P(y) \vee \neg Q(w, y)))$.

We could have renamed the formula A into a clean formula right from the beginning. That, however, would have made the resulting prenex formula much larger.

4.6.2. Skolemization

Skolemization is a procedure (first applied by the Norwegian logician Thoralf Skolem) for systematic elimination of the existential quantifiers in a first-order formula in a prenex form by way of uniform replacement of all occurrences of existentially quantified individual variables with terms, headed by new functional symbols called *Skolem functions*.

In the traditional version of Skolemization these Skolem functions take as arguments all variables (if any) that are bound by universal quantifiers in the scope of which the given existential quantifier sits. In particular, existentially quantified variables that are not in the scope of any universal quantifiers are replaced by constant symbols. We will explain the Skolemization procedure by some examples.

Example 4.6.6

1. The result of Skolemization of the formula

$$\exists x\forall y\forall z(P(x, y) \rightarrow Q(x, z))$$

- is $\forall y\forall z(P(c, y) \rightarrow Q(c, z))$, where c is a new constant, called a **Skolem constant**.
2. More generally, the result of Skolemizing the formula

$$\exists x_1 \cdots \exists x_k \forall y_1 \cdots \forall y_n A(x_1, \dots, x_k, y_1, \dots, y_n)$$

- is $\forall y_1 \cdots \forall y_n A(c_1, \dots, c_k, y_1, \dots, y_n)$, where c_1, \dots, c_k are new Skolem constants.
3. The result of Skolemization of the formula

$$\exists x\forall y\exists z(P(x, y) \rightarrow Q(x, z))$$

is $\forall y(P(c, y) \rightarrow Q(c, f(y)))$, where c is a new Skolem constant and f is a new unary function, called a **Skolem function**.

4. More generally, the result of Skolemizing the formula

$$\forall y \exists x_1 \cdots \exists x_k \forall y_1 \cdots \forall y_n A(y, x_1, \dots, x_k, y_1, \dots, y_n)$$

is $\forall y \forall y_1 \cdots \forall y_n A(y, f_1(y), \dots, f_k(y), y_1, \dots, y_n)$, where f_1, \dots, f_k are new Skolem functions.

5. The result of Skolemization of the formula

$$\forall x \exists y \forall z \exists u A(x, y, z, u)$$

is the formula $\forall x \forall z A(x, f(x), z, g(x, z))$, where f is a new unary Skolem function and g is a new binary Skolem function.

Note. The result of Skolemization of any first-order formula A is a formula that is generally *not logically equivalent* to A , but is equally satisfiable with A .¹³ Therefore, for the purpose of the theorem proving based on checking (un)satisfiability, Skolemization is an admissible procedure.

4.6.3. Clausal forms

Recall that a **literal** is an atomic formula or a negation of an atomic formula and a **clause** is a set of literals (representing their disjunction).

Example:

$$\{P(x), \neg P(f(c, g(y))), \neg Q(g(x), y), Q(f(x, g(c)), g(g(g(y))))\}.$$

Note that all variables in a clause are implicitly assumed to be universally quantified.

A **clausal form** is a set of clauses (representing their conjunction). For example:

$$\begin{aligned} & \{ \\ & \{P(x)\}, \\ & \{\neg P(f(c)), \neg Q(g(x, x), y)\}, \\ & \{\neg P(f(y)), P(f(c)), Q(y, f(x))\} \\ & \}. \end{aligned}$$

Theorem 4.6.7 Every first-order formula A can be transformed to a clausal form $\{C_1, \dots, C_k\}$ such that A is equally satisfiable with the universal closure $(\overline{C_1 \wedge \dots \wedge C_k})$ of the conjunction of all clauses considered as disjunctions.

Here is an algorithm transforming a formula into clausal form:

1. Transform A into a prenex CNF.
2. Skolemize all existential quantifiers.
3. Remove all universal quantifiers.
4. Write the matrix (which is in CNF) as a set of clauses.

¹³ Assuming the Axiom of choice.

Example 4.6.8

Consider the formula

$$A = \exists z(\exists xQ(x, z) \vee \exists xP(x)) \rightarrow \neg(\neg \exists xP(x) \wedge \forall x \exists zQ(z, x))$$

from Example 4.6.5.

1. Transforming A into a prenex CNF:

$$A \equiv \forall z \forall x \exists y \forall w ((\neg Q(x, z) \vee P(y)) \vee \neg Q(w, y)) \wedge (\neg P(x) \vee P(y) \vee \neg Q(w, y)).$$

2. Skolemizing all existential quantifiers:

$$\begin{aligned} \text{Skolem}(A) = & \forall z \forall x \forall w ((\neg Q(x, z) \vee P(f(z, x)) \vee \neg Q(w, f(z, x))) \wedge (\neg P(x) \vee P(f(z, x)) \\ & \vee \neg Q(w, f(z, x))). \end{aligned}$$

3. Removing all universal quantifiers and writing the matrix as a set of clauses:

$$\text{clausal}(A) = \{\neg Q(x, z), P(f(z, x)), \neg Q(w, f(z, x))\}, \{\neg P(x), P(f(z, x)), \neg Q(w, f(z, x))\}.$$

4.6.4. Exercises

- ① Transform each of the following formulae into a prenex DNF and a prenex CNF. Then Skolemize the resulting formula and transform it into clausal form.

- (a) $\forall x((\exists yP(y) \wedge \forall zQ(z, x)) \rightarrow \exists yQ(x, y))$
- (b) $\exists z(\exists xQ(x, z) \rightarrow (\exists xP(x) \vee \neg \exists zP(z)))$
- (c) $\forall y \neg(P(y) \leftrightarrow \exists xQ(y, x))$
- (d) $\forall z(\exists yP(y) \leftrightarrow Q(z, y))$
- (e) $(\exists xP(x) \rightarrow \neg \exists xQ(x)) \rightarrow \forall x(P(x) \rightarrow \neg Q(x))$
- (f) $\forall x(\neg \forall yQ(x, y) \wedge P(z)) \rightarrow \exists z(\forall yQ(z, y) \wedge \neg P(x))$
- (g) $\neg \exists x(\forall y(\exists zQ(y, z) \leftrightarrow P(z)) \wedge \forall zR(x, y, z))$
- (h) $\neg(\forall y(\forall zQ(y, z) \rightarrow P(z)) \rightarrow \exists z(P(z) \wedge \forall x(Q(z, y) \rightarrow Q(x, z))))$
- (i) $\neg(\neg \exists z(P(z) \wedge \forall x(Q(z, y) \rightarrow Q(x, z))) \rightarrow \neg \forall y(\forall zQ(y, z) \rightarrow P(z)))$
- (j) $\neg(\forall y(\neg \exists zQ(y, z) \rightarrow P(z)) \rightarrow \exists z((P(z) \rightarrow Q(z, y)) \wedge \neg \exists xR(x, y, z)))$



Thoralf Albert Skolem (23.5.1887–23.3.1963)

Thoralf Skolem was a Norwegian mathematician and logician who made seminal contributions to mathematical logic, set theory and algebra.

Skolem was extremely productive and published around 180 papers on Diophantine equations, mathematical logic, group theory, lattice theory and set theory. However, many of his results were not widely known and were later rediscovered by others, because they were published in not widely accessible Norwegian journals.

In 1922 Skolem refined and made precise Zermelo's axioms for set theory, essentially in the form in which they are known today. In 1923 he created a formal theory of primitive recursive arithmetic meant to avoid the paradoxes of the infinite and formally developed a great deal of number theory in it. His system can now be regarded as a programming language for defining functions on natural numbers, and thus he made an early contribution to computer science. His other work which later became very important in computer science and instrumental in resolution-based automated theorem proving was on what later became known as *Skolemization* and *Skolem normal form*.

In 1930 Skolem proved that the subsystem of Peano arithmetic (see Section 4.2) involving only axioms for multiplication but not addition, now called *Skolem arithmetic* in his honour, was complete and decidable, unlike the full system of Peano arithmetic (including both addition and multiplication), proved to be incomplete by Gödel in 1931 (see Section 4.5) and undecidable by Church in 1936 (see Section 4.7).

The completeness of first-order logic, first published by Gödel in 1930, follows from results that Skolem obtained in the early 1920s and published in 1928, but he was apparently not aware of that fact at the time, because the problem of proving completeness of an axiomatic system for first-order logic was only stated explicitly in 1928, in the first edition of Hilbert and Ackermann's *Principles of Mathematical Logic*.

The most influential and widely recognized work of Skolem was his pioneering research in *model theory* – the study of applications of logic to mathematics. His most well-known result is the *Löwenheim – Skolem theorem*, an extension of an earlier work by Löwenheim, published in 1920. It states that if a first-order theory, i.e. a set of sentences of first-order logic, has a model – a structure that satisfies them all – then it has a *countable* model. This theorem has some consequences that sound paradoxical, thus giving rise to what became known as “Skolem's paradox”: assuming that the axiomatic system ZFC of set theory is satisfied in some model then it must have one with a countable universe, despite the fact that it can prove the existence of uncountable sets. Likewise, the first-order theory of the field of reals, which is uncountable, also has a countable model. On the other hand, Skolem also showed that if a first-order theory has an infinite model then it has models of arbitrarily large infinite cardinalities. In particular, the system of Peano arithmetic, as well as the first-order theory of all true sentences of the arithmetic of natural numbers, have uncountable models. Thus, Skolem provided some of the first constructions of nonstandard models of arithmetic and set theory.

4.7. Resolution in first-order logic

4.7.1. Propositional resolution rule in first-order logic

The propositional resolution rule, extended to first-order logic reads:

$$\frac{C \vee Q(s_1, \dots, s_n), \quad D \vee \neg Q(s_1, \dots, s_n)}{C \vee D}.$$

This rule, however, is not strong enough. For example, the clause set $\{\{P(x)\}, \{\neg P(f(y))\}\}$ is not satisfiable, as it corresponds to the unsatisfiable formula $\forall x \forall y (P(x) \wedge \neg P(f(y)))$. However, the resolution rule above cannot produce an empty clause, because it *cannot unify* the two clauses in order to resolve them. Therefore, we need a stronger derivation mechanism that can handle cases like this. There are two natural solutions:

1. *Ground resolution*.¹⁴ Generate *sufficiently many ground instances* of every clause (in the so-called *Herbrand universe* of the language) and then apply the standard resolution rule above.

In the example, ground resolution would generate the ground clauses $\{P(f(c))\}$ and $\{\neg P(f(c))\}$ for some constant symbol c .

This method is sound and complete but inefficient, as it leads to the generation of too many unnecessary clauses. It will not be discussed further.

2. *Resolution with unification*. Introduce a stronger resolution rule, which first tries to match a pair of clauses by applying a suitable substitution that would enable resolving that pair.

We will present this method in detail.

4.7.2. Substitutions of terms for variables revisited

Recall that substitution of a term t for a variable x in a term s , denoted by $s[t/x]$, is the result of *simultaneous* replacements of all occurrences of x in s by t . For instance:

$$f(g(x), f(y, x))[g(a)/x] = f(g(g(a)), f(y, g(a))).$$

This can generalize to simultaneous substitutions of several terms for *different* variables, denoted $s[t_1/x_1, \dots, t_k/x_k]$. For instance:

$$f(g(x), f(y, x))[g(y)/x, f(x, b)/y] = f(g(g(y)), f(f(x, b), g(y))).$$

Note that the condition of simultaneity is important: if we first substituted $g(y)$ for x and then $f(x, b)$ for y the result would have been different (and wrong).

One can apply a substitution to several terms simultaneously, e.g. when a substitution acts on an atomic formula $P(t_1, \dots, t_n)$. For instance:

$$P(x, f(x, y), g(y))[g(y)/x, a/y] = P(g(y), f(g(y), a), g(a)).$$

Given the substitution $\sigma = [t_1/x_1, \dots, t_k/x_k]$, the set of variables $\{x_1, \dots, x_k\}$ is called the **domain** of σ , denoted $\text{dom}(\sigma)$.

¹⁴ Introduced by John Alan Robinson in: A Machine-Oriented Logic Based on the Resolution Principle, JACM, 1965.

Substitutions can be *composed* by consecutive application: the composition of substitutions τ and σ is a substitution $\tau\sigma$ obtained by applying τ followed by applying σ to the result, i.e. $\tau\sigma = (\tau\sigma)\sigma$ (note the order). For instance:

$$\begin{aligned} & f(g(x), f(y, x)) [f(x, y)/x] [g(a)/x, x/y] \\ &= f(g(f(x, y)), f(y, f(x, y))) [g(a)/x, x/y] \\ &= f(g(f(g(a), x)), f(x, f(g(a), x))). \end{aligned}$$

The composition of two substitutions $\tau = [t_1/x_1, \dots, t_k/x_k]$ and σ can be computed as follows:

1. Extend $\text{dom}(\tau)$ to cover $\text{dom}(\sigma)$ by adding to τ substitutions $[x/x]$ for every variable x in $\text{dom}(\sigma) \setminus \text{dom}(\tau)$.
2. Apply the substitution σ simultaneously to all terms $[t_1, \dots, t_k]$ to obtain the substitution $[t_1\sigma/x_1, \dots, t_k\sigma/x_k]$.
3. Remove from the result all cases x_i/x_i , if any.

For instance:

$$[f(x, y)/x, x/y][y/x, a/y, g(y)/z] = [f(y, a)/x, y/y, g(y)/z] = [f(y, a)/x, g(y)/z].$$

4.7.3. Unification of terms

A substitution σ *unifies* two words s and s' if $\sigma(s) = \sigma(s')$. A substitution σ that unifies the respective arguments of the literals $Q(s_1, \dots, s_n)$ and $Q(t_1, \dots, t_n)$, i.e. such that $\sigma(s_1) = \sigma(t_1), \dots, \sigma(s_n) = \sigma(t_n)$, is called a **unifier** of $Q(s_1, \dots, s_n)$ and $Q(t_1, \dots, t_n)$. For instance, the substitution $\sigma = [f(c)/x, c/y, c/z]$ unifies the literals $Q(x, c, f(f(z)))$ and $Q(f(y), z, f(x))$.

Two terms are **unifiable** if they have a unifier; otherwise they are **nonunifiable**.

A unifier τ of two terms (or literals) is **more general** than a unifier ρ if there is a substitution σ such that $\rho = \tau\sigma$. For instance, $\rho = [c/x, f(c)/y]$ is a unifier of the literals $P(f(x))$ and $P(y)$, but $\tau = [f(x)/y]$ is a more general unifier because $\rho = \tau\sigma$, where $\sigma = [c/x]$.

A unifier of two terms (or literals) is their **most general unifier** (MGU) if it is more general than any unifier of these terms (literals).

A most general unifier (if it exists) need not be unique. For instance, in the example above, τ is a most general unifier, as is $[z/x, f(z)/y]$ for any variable z .

Eventually, we are interested in unifying literals, i.e. (negated) atomic formulae. Note that two atomic formulae $P(t_1, \dots, t_m)$ and $Q(s_1, \dots, s_n)$ cannot be unified unless $P = Q$ and respectively $m = n$. If that is the case, in order to unify the formulae we need to unify the respective pairs of arguments, i.e. we are looking for a (most general) unifier of the system

$$\begin{aligned} t_1 &= s_1 \\ &\vdots \\ t_n &= s_n. \end{aligned}$$

Informally, we do that by computing a most general unifier (if one exists) of the first pair, then applying it to both lists of terms, then computing a most general unifier of the next pair (if there is one) in the resulting lists, applying it to both resulting lists of terms, etc. In order to

unify the current pair of terms we apply the algorithm recursively to the pair of lists of their respective arguments. The composition of all most general unifiers computed as above, if they all exist, is a most general unifier of the two input lists.

Here is a pseudocode of a simple recursive algorithm for computing a most general unifier of two lists of terms **p** and **q**, presented below. (The predicate ‘variable(t)’ is true if the term t is a variable.)

```

procedure mgu(p, q)
  θ := ε      (the empty substitution).
  Scan p and q simultaneously, left-to-right,
  and search for the first corresponding subterms
  where p and q disagree (mismatch).
  If there is no mismatch, return θ.
  Else, let s and t be the first respective subterms
  in p and q where mismatch occurs.
  If variable(s) and s ∉ Var(t) then
    θ := compose(θ, [t/s]);
    θ := compose(θ, mgu(pθ, qθ)).
  Else, if variable(t) and t ∉ Var(s) then
    θ := compose(θ, [s/t]);
    θ := compose(θ, mgu(pθ, qθ)).
  Else, return failure.
end

```

Example 4.7.1

Some examples of unification of atomic formulae using the algorithm above are:

- Literal 1: `parents(x, father(x), mother(Bill))`,
 Literal 2: `parents(Bill, father(Bill), y)`,
 Most general unifier: `[Bill/x, mother(Bill)/y]`.
- Literal 1: `parents(x, father(x), mother(Bill))`,
 Literal 2: `parents(Bill, father(y), z)`,
 Most general unifier: `[Bill/x, Bill/y, mother(Bill)/z]`.
- Literal 1: `parents(x, father(x), mother(Jane))`,
 Literal 2: `parents(Bill, father(y), mother(y))`.
 The algorithm fails; therefore, a unifier does not exist.
- Term 1: `g(x, f(x))`,
 Term 2: `g(f(y), y)`.
 The procedure **mgu** starts computing: `[f(y)/x, ...failure]`.
 MGU does not exist. Indeed, any unifier would have to unify $f(f(x))$ with x , which is impossible.

4.7.4. Resolution with unification in first-order logic

The first-order resolution rule is combined with a preceding unification of the clausal set in order to produce a complementary pair of literals in the resolving clauses:

$$\text{Res : } \frac{C \vee Q(s_1, \dots, s_n), \quad D \vee \neg Q(t_1, \dots, t_n)}{\sigma(C \vee D)}$$

where:

- (i) The two clauses have no variables in common (achieved by renaming of variables),
- (ii) σ is a most general unifier of the literals $Q(s_1, \dots, s_n)$ and $Q(t_1, \dots, t_n)$.

Note: we require that the unifier applied in the rule is a most general unifier of the respective literals in order not to weaken the resolvent unnecessarily.

Some examples of resolution with unification are:

$$\begin{aligned} & \frac{\{P(x)\}, \{\neg P(f(y))\}}{\{\}} \text{ (MGU : } [f(y)/x]), \\ & \frac{\{P(a, y), \quad Q(y)\} \{\neg P(x, b), \neg Q(x)\}}{\{Q(b), \neg Q(a)\}} \text{ (MGU : } [a/x, b/y]), \\ & \frac{\{P(a, y), \quad Q(y)\} \{\neg P(x, f(x)), \quad Q(f(x))\}}{\{Q(f(a))\}} \text{ (MGU : } [a/x, f(a)/y]). \end{aligned}$$

Note that both literals in the resolvent become equal.

$$\frac{\{P(a, y), \quad P(x, f(x))\} \{\neg P(x, f(a))\}}{\{\}} \text{ (MGU : } [a/x, f(a)/y]).$$

Note that, after unification, both literals in the first clause become $P(a, f(a))$.

Definition 4.7.2 A *resolution-based derivation* of a formula C from a list of formulae A_1, \dots, A_n , denoted $A_1, \dots, A_n \vdash_{\text{Res}} C$, is a derivation of the empty clause $\{\}$ from the set of clauses obtained from $A_1, \dots, A_n, \neg C$, by successive applications of the rule **Res**.

Theorem 4.7.3 The method of first-order resolution is sound and complete, i.e. for any first-order formulae A_1, \dots, A_n, C :

$$A_1, \dots, A_n, \vdash_{\text{Res}} C \text{ if and only if } A_1, \dots, A_n, \models C.$$

Unlike propositional resolution, first-order resolution *may run forever*, i.e. never terminate, in some cases when the conclusion does not follow logically from the premises.

It may also run forever even when the conclusion *does* follow logically from the premises, if an unbounded number of unnecessary resolvents is produced.

To avoid that, special strategies or additional mechanisms for disposal of used-up clauses need to be applied.

4.7.5. Examples of resolution-based derivations

Example 4.7.4

Prove with the method of first-order resolution that

$$\forall x(P(x) \rightarrow Q(x)) \vDash \forall xP(x) \rightarrow \forall xQ(x).$$

1. Transform

$$\{\forall x(P(x) \rightarrow Q(x)), \neg(\forall xP(x) \rightarrow \forall xQ(x))\}$$

to clausal form:

$$\{\neg P(x), Q(x)\}, \quad \{P(y)\}, \quad \{\neg Q(c)\}$$

for some Skolem constant c .

2. Successive applications of Res:

(a) Unify $P(x)$ and $P(y)$ with MGU $[y/x]$.

Then resolve $\{\neg P(x), Q(x)\}$ and $\{P(y)\}$ to obtain $\{Q(y)\}$.

(b) Unify $Q(c)$ and $Q(y)$ with MGU $[c/y]$.

Then resolve $\{\neg Q(c)\}$ and $\{Q(y)\}$ to obtain $\{\}$.

Example 4.7.5

Show that if

Everybody loves somebody and

Everybody loves any lover then

Everybody loves everybody.

1. Formalize the assumptions and the goal conclusion, using the predicate $L(x, y)$ meaning “ x loves y ”:

Everybody loves somebody: $\forall x \exists y L(x, y)$.

Everybody loves any lover: $\forall x (\exists y L(x, y) \rightarrow \forall z L(z, x))$.

Everybody loves everybody: $\forall x \forall y L(x, y)$.

2. Transform the set

$$\{\forall x \exists y L(x, y), \forall x (\exists y L(x, y) \rightarrow \forall z L(z, x)), \neg(\forall x \forall y L(x, y))\}$$

to a clausal form:

$$\{L(x, f(x))\}, \quad \{\neg L(x_1, y), L(z, x_1)\}, \quad \{\neg L(a, b)\}$$

for some Skolem constants a, b and a Skolem function f .

3. Successive applications of Res:

- (a) Unify $L(z, x_1)$ and $L(a, b)$ with MGU $[b/x_1, a/z]$ and resolve $\{\neg L(x_1, y), L(z, x_1)\}$ and $\{\neg L(a, b)\}$ to obtain $\{\neg L(b, y)\}$.
- (b) Unify $L(x, f(x))$ and $L(b, y)$ with MGU $[b/x, f(b)/y]$ and resolve $\{L(x, f(x))\}$ and $\{\neg L(b, y)\}$ to obtain $\{\}$.

Example 4.7.6

Check if

$$\forall x \exists y R(x, y), \forall x \forall y \forall z ((R(x, y) \wedge R(y, z)) \rightarrow R(x, z)) \models \exists x R(x, x).$$

1. Transform

$$\{\forall x \exists y R(x, y), \forall x \forall y \forall z ((R(x, y) \wedge R(y, z)) \rightarrow R(x, z)), \neg(\exists x R(x, x))\}$$

to a clausal form:

$$C_1 = \{R(x, f(x))\}, C_2 = \{\neg R(u, y), \neg R(y, z), R(u, z)\}, C_3 = \{\neg R(v, v)\}$$

for some unary Skolem function f .

2. Successive applications of Res to the clausal set give

$$C_1 = \{R(x, f(x))\}, C_2 = \{\neg R(u, y), \neg R(y, z), R(u, z)\}, C_3 = \{\neg R(v, v)\}.$$

(Wherever necessary, we rename variables in the derived clauses to keep all clauses with disjoint sets of variables.)

- (a) Unify $R(u, z)$ in C_2 and C_3 , with MGU $[v/u, v/z]$ and resolve:

$$C_4 = \{\neg R(v, y), \neg R(y, v)\}.$$

- (b) Unify $R(v, y)$ in C_4 and $R(x, f(x))$ in C_1 with MGU $[x/v, f(x)/y]$ and resolve:
 $C_5 = \{\neg R(f(x), x)\}.$

After renaming x : $C_5 = \{\neg R(f(x_1), x_1)\}$.

- (c) Unify $R(u, z)$ in C_2 and $R(f(x_1), x_1)$ in C_5 with MGU $[f(x)/u, x/z]$ and resolve:

$$C_6 = \{\neg R(f(x), y), \neg R(y, x)\}.$$

After renaming: $C_6 = \{\neg R(f(x'), y'), \neg R(y', x')\}$.

- (d) Unify $R(f(x'), y')$ in C_6 and $R(x, f(x))$ in C_1 with MGU $[f(x')/x, f(f(x'))/y']$ and resolve:
 $C_7 = \{\neg R(f(f(x')), f(x'))\}$. After renaming, $C_7 = \{\neg R(f(f(x_2)), f(x_2))\}$, etc.

Thus, an infinite set of clauses can be generated:

$$\{\neg R(f(x_1), x_1)\}, \{\neg R(f(f(x_2)), f(x_2))\}, \{\neg R(f(f(f(x_3))), f(f(x_3)))\}, \dots$$

but the empty clause cannot be derived.

In fact, the logical consequence does not hold. A countermodel is the set of natural numbers with $R(x, y)$ interpreted as $x < y$.

4.7.6. Resolution for first-order logic with equality

The method of resolution presented here does not involve rules for the equality. Such rules can be added to produce a sound and complete system for first-order logic with equality. The most common additional rule sufficient for that equational reasoning is called **paramodulation**:

$$\text{Par : } \frac{C \vee s_1 = s_2, \quad D \vee Q(t, t_1, \dots, t_n)}{(C \vee D \vee Q(s_2, t_1, \dots, t_n))\sigma} \quad (\sigma = \text{MGU}(s_1, t)).$$

4.7.7. Optimizations of the resolution method for first-order logic

Just like propositional resolution, the method of resolution for first-order logic is amenable to various optimizations by means of special additional rules and derivation strategies that go beyond the scope of this book. Such optimizations are much more important for first-order Resolution because if the search for a derivation of the empty clause is not cleverly organized it may not terminate, even if such a derivation exists. The techniques of *deletion of repeating literals*, *tautology deletion*, *subsumption deletion* and *removal of clauses with monopolar literals* still apply to first-order resolution. Moreover, they can be combined with each other and with suitable unification pre-processing, e.g. to produce more specific techniques, such as *factoring*:

$$\frac{\{\dots P(\bar{s}), P(\bar{t})\dots\}}{\{\dots P(\bar{s})\dots\}\sigma} \quad (\sigma = \text{MGU}(\bar{s}, \bar{t})).$$

For example:

$$\frac{\{P(x), P(c), \neg Q(x, y)\}}{\{P(c), \neg Q(c, y)\}} \quad (\text{MGU} : [c/x]).$$

Exercise 4.7.7 Prove that factoring is a logically sound rule.

In addition, widely used additional optimizing techniques are *clause ordering*, *indexing techniques*, etc., as well as strategies for application of the resolution rule, such as *unit resolution*, *set-of-support resolution*, *hyper-resolution*, *linear resolution*, etc. A special version of the latter strategy, called *selective linear definite clause resolution*, or just *selective linear definite (SLD) clause resolution*, is the derivation strategy of the logic programming language PROLOG.

4.7.8. Exercises

For each of the following exercises use first-order resolution with unification.

❶ Prove the following logical validities and consequences.

- (a) $\models \forall x P(x) \rightarrow \forall y P(y)$
- (b) $\models \exists x P(x) \rightarrow \exists y P(y)$
- (c) $\models \exists x(P(x) \rightarrow \forall y P(y))$
- (d) $\forall x A(x) \models \neg \exists x \neg A(x)$
- (e) $\neg \exists x \neg A(x) \models \forall x A(x)$
- (f) $\exists x A(x) \models \neg \forall x \neg A(x)$

(g) $\neg \forall x \neg A(x) \models \exists x A(x)$

(h) $\exists x \exists y A(x, y) \models \exists y \exists x A(x, y)$

② Suppose x is not free in Q . Prove the following logical consequences.

(a) $\forall x(P(x) \vee Q) \models \forall x P(x) \vee Q$

(b) $\forall x P(x) \vee Q \models \forall x(P(x) \vee Q)$

(c) $\exists x(P(x) \wedge Q) \models \exists x P(x) \wedge Q$

(d) $\exists x P(x) \wedge Q \models \exists x(P(x) \wedge Q)$

(e) $\forall x(Q \rightarrow P(x)), Q \models \forall x P(x)$

(f) $\exists x P(x) \rightarrow Q \models \forall x(P(x) \rightarrow Q)$

(g) $\exists x(P(x) \rightarrow Q), \forall x P(x) \models Q$

(h) $\exists x(Q \rightarrow P(x)), Q \models \exists x P(x)$

③ Check which of the following logical consequences hold. For those that do not, construct a countermodel, i.e. a structure and assignment in which all premises are true, while the conclusion is false.

(a) $\forall x A(x), \forall x B(x) \models \forall x(A(x) \wedge B(x))$

(b) $\forall x(A(x) \wedge B(x)) \models \forall x A(x) \wedge \forall x B(x)$

(c) $\forall x A(x) \vee \forall x B(x) \models \forall x(A(x) \vee B(x))$

(d) $\forall x(A(x) \vee B(x)) \models \forall x A(x) \vee \forall x B(x)$

(e) $\forall x A(x) \rightarrow \forall x B(x) \models \forall x(A(x) \rightarrow B(x))$

(f) $\forall x(A(x) \rightarrow B(x)) \models \forall x A(x) \rightarrow \forall x B(x)$

(g) $\exists x(A(x) \wedge B(x)) \models \exists x A(x) \wedge \exists x B(x)$

(h) $\exists x A(x), \exists x B(x) \models \exists x(A(x) \wedge B(x))$

(i) $\exists x(A(x) \vee B(x)) \models \exists x A(x) \vee \exists x B(x)$

(j) $\exists x A(x) \vee \exists x B(x) \models \exists x(A(x) \vee B(x))$

(k) $\exists x(A(x) \rightarrow B(x)) \models \exists x A(x) \rightarrow \exists x B(x)$

(l) $\exists x A(x) \rightarrow \exists x B(x) \models \exists x(A(x) \rightarrow B(x))$

④ Formalize the following logical consequences in first-order logic and check for each of them whether it is logically correct.

All logicians are clever.

(a) All clever people are rich.

All logicians are rich.

All natural numbers are integers.

Some integers are odd numbers.

-
- (b) Some natural numbers are odd numbers.

All penguins are birds.

Some penguins are white.

All penguins eat some fish.

-
- (c) Therefore, there is a white bird that eats some fish.

No yellow plonks are qlinks.

-
- (d) Some object is a plonk or not a qlink.

Therefore, there is an object which is not a yellow qlink.

- ⑤ Using the predicates: $F(x)$ for “ x is a fairy”, $E(x)$ for “ x is an elf”, $G(x)$ for “ x is a goblin”, $H(x)$ for “ x is a hobbit”, $T(x)$ for “ x is a troll”, $L(x, y)$ for “ x likes y ”, formalize the following argument in first-order logic. Then check whether it is logically valid. If it is not valid, give an appropriate countermodel.

All fairies are elves. Hobbits do not like any elves.

Kermit likes a goblin or a fairy. Everyone who likes any goblins likes no trolls.

Therefore, if Kermit likes any trolls then Kermit is not a hobbit.

- ⑥ Formalize the following arguments in first-order logic by using suitable predicates interpreted in suitable domains. Then check for each of them whether it is logically valid. If it is not valid, give an appropriate countermodel.

All trolls are elves or goblins. No troll can fly, unless it is an elf. All dragons can fly.

-
- (a) All ugly goblins are trolls. Some goblins are ugly.

Therefore, there is an ugly creature that is not a dragon.

Some bachelors love every woman they know.

-
- (b) Every bachelor is a man.

Eve is a woman whom every man knows.

Some bachelor loves a woman.

- (c) Replace the conclusion in the previous example by

“Every bachelor loves a woman.”

and check again whether the resulting argument is logically valid.


Alonzo Church (14.6.1903–11.8.1995)

Alonzo Church was an American mathematician and logician who made profound contributions to mathematical logic and the theoretical foundations of computer science.

Church studied and had his academic career at the Mathematics Department of Princeton University, where he worked until 1967, when he formally retired and moved to the University of California, Los Angeles, as Flint Professor of Philosophy and Mathematics for a further 23 years, until his final retirement at the age of 87, in 1990, after 63 years of a teaching career.

In the early 1930s Church created the *lambda-calculus*, as a formal system for defining functions that are effectively computable. The *lambda-calculus* is also relevant to his two-part work *A Set of Postulates for the Foundations of Logic*, published in 1932–1933, intended to develop a set of axioms that “would lead to a system of mathematical logic free of some of the complications entailed by Bertrand Russell’s theory of types, and would at the same time avoid the well-known paradoxes, in particular the Russell paradox.” While it later turned out that the original system of lambda-calculus did contain an inconsistency, the idea flourished further and later became very influential in computer science as the basis of functional programming.

In 1936 Church published two papers of profound importance for logic and computability theory. In the first, *An Unsolvable Problem of Elementary Number Theory*, he defined the notion of “effective calculability” by identifying it with the notion of recursive functions or, equivalently as it turned out, with lambda-definable functions of natural numbers. He then proved that the problem of deciding whether a given first-order formula of the language with addition and multiplication is true of the arithmetic of natural numbers is not effectively calculable. By equating effective calculability with algorithmic solvability Church put forward what was later called by Kleene “Church’s thesis”. Thus, Church essentially proved that the truth in the elementary arithmetic is algorithmically unsolvable.

In his other historic paper from 1936, *A Note on the Entscheidungsproblem* (two pages long, followed by a two-page correction), Church proved what is now called *Church’s theorem*: the problem of deciding validity of formulae in first-order logic is not effectively calculable, which means – assuming Church’s thesis – algorithmically unsolvable.

Church had an extremely long and fruitful scientific career, spanning over more than 70 years since his first publication as an undergraduate student in 1924 until his last paper, *A Theory of the Meaning of Names*, published in 1995. In 1956 he published his classic textbook *Introduction to Mathematical Logic*, which educated and influenced generations of logicians. Church was also an extremely popular and successful supervisor and had 31 doctoral students, many of whom became distinguished logicians and computer scientists, including Martin Davis, Leon Henkin, John Kemeny, Steven Kleene, Michael O. Rabin, Nicholas Rescher, Hartley Rogers, J. Barkley Rosser, Dana Scott, Reymond Smullyan and Alan Turing.



4.8. Applications of first-order logic to mathematical reasoning and proofs

In mathematics the truth of a statement is established by proving it. A proof may consist of a simple argument or of lots of complicated calculations with essentially no logical reasoning. It may also be a long and intricate argument involving a number of other, already proven, statements, conjectures, logical inferences, etc.

This section is about *logical* aspects of mathematical reasoning and proofs. We discuss strategies and tactics for proofs, essentially based on the rules of the logical system of *natural deduction* (which we will not formally present here) and illustrate them with a few examples.

4.8.1. Proof strategies: direct and indirect proofs

A typical mathematical statement looks like this:

If P_1, \dots, P_n then C .

where P_1, \dots, P_n are several (possibly none) *premises* or *assumptions* and C is a *conclusion*.

The logical structure of a proof of such a statement naturally depends on the logical structure of the premises and of the conclusion. Here we will provide some *proof tactics* telling us how to go about specific, *local* steps of the proof, but we will first discuss possible *proof strategies*, telling us how to organize proofs *globally*.

I. Direct proof

With this proof strategy one *assumes* that all premises are true and then tries to *deduce* the desired conclusion by applying a sequence of correct (logical or mathematical) inference steps. Schematically, a direct proof looks as follows:

```

Assume  $P_1, \dots, P_n$ .
:
(a sequence of valid inferences)
:
Conclude  $C$ .

```

Example 4.8.1

A direct proof of the statement

If n is an odd integer, then n^2 is an odd integer.

Proof.

Suppose n is an odd integer.

Then there is an integer k such that $n = 2k + 1$.

$$\text{Therefore, } n^2 = (2k + 1)(2k + 1) = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1.$$

Therefore, n^2 is odd.

II. Indirect proof

Also known as **proof by assumption of the contrary** or **proof by contradiction**. The idea of the indirect proof strategy is the same as the idea of semantic tableaux: to assume that *all premises are true, while the conclusion is false* (hence *the negation of the conclusion is true*), and try to reach a *contradiction*, again by applying only valid inferences. One typically obtains a contradiction by deducing a statement known to be false (like deducing that $1 + 1 = 3$) or by deducing a statement and its negation. Often one can reach a contradiction by deducing the negation of some of the premises (see the example below), which have been assumed to be true. The rationale behind the proof by contradiction is clear: if all our assumptions are true and we only apply valid inferences, then all our conclusions *must* also be true. Therefore, by deducing a false conclusion, we show that, given that all original assumptions are true, the additional one that we have made – namely, the negation of the conclusion – must have been wrong, i.e. that *the conclusion must be true*.

Thus, the scheme of a proof by contradiction is:

Assume P_1, \dots, P_n .

Assume $\neg C$.

⋮

(a sequence of valid inferences)

⋮

Derive both A and $\neg A$

for some proposition A .

Conclude that $\neg C$ cannot be true.

Therefore, conclude C .

Example 4.8.2

A proof by contradiction of the statement

If x and y are integers, x is odd and xy is even, then y is even.

Proof:

Suppose that x, y are integers, where x is odd and xy is even.

Suppose also that y is not even.

Therefore, y is odd.

Then there is an integer k such that $x = 2k + 1$, since x is odd,
and there is an integer m such that $y = 2m + 1$, since y is odd.

Therefore, $xy = (2k + 1)(2m + 1) = \dots = 2(2km + k + m) + 1$.

Therefore, xy is odd.

On the other hand, we have assumed that xy is even and, therefore, xy is not odd.

Thus, xy is odd and xy is not odd.

This is a contradiction. It implies that the additional assumption, that y is not even, is wrong.

Therefore y is even, which completes the proof.

Example 4.8.3

Euclid's proof by contradiction is that

$\sqrt{2}$ is irrational.

Proof:

Suppose it is not the case that $\sqrt{2}$ is irrational.

Therefore, $\sqrt{2}$ is rational.

Hence, $\sqrt{2}$ can be represented as an irreducible fraction, i.e. there are relatively prime integers m and n such that $\sqrt{2} = m/n$.

Therefore, $2 = m^2/n^2$ and hence $2n^2 = m^2$.

Hence m^2 is even.

Therefore, m must be even, since if it were odd, then m^2 would have been odd (see the example of a direct proof).

Thus, there is an integer a such that $m = 2a$.

Hence, $m^2 = 4a^2$.

Therefore, $2n^2 = 4a^2$ and hence $n^2 = 2a^2$.

Thus, n^2 is even.

Therefore, n must be even, by the same argument as above.

Thus, there is an integer b such that $n = 2b$.

Therefore, the fraction $m/n = 2a/2b$ is not irreducible, which contradicts the assumption that m/n is irreducible.

This contradiction completes the proof.

A variation of proof by contradiction is **proof by contraposition**. This strategy is usually applied when the statement to be proved is of the type “If P , then C ”, i.e. there is only one premise.¹⁵ This proof strategy is based on the fact that the implication “If P , then C ” is logically equivalent to its contrapositive “If $\neg C$, then $\neg P$ ”. Therefore, if we prove the latter, we have a proof of the former.

Schematically, the proof by contraposition of “If P , then C ” looks as follows:

Assume $\neg C$.

:

(a sequence of valid inferences)

:

Derive $\neg P$.

One can also think of this strategy, as obtained from the proof by contradiction strategy, by assuming P together with $\neg C$ at the beginning and then proving $\neg P$ to produce the desired contradiction.

Usually we apply the proof by contraposition when it is easier to prove the contrapositive of the original implication.

Example 4.8.4

Here is a proof by contraposition of the statement

If n is an integer such that n^2 is even, then n is even.

Proof.

Suppose n is not even.

Therefore, n is odd.

Then (see the example of a direct proof) n^2 is odd.

Therefore, n^2 is not even.

This completes the proof.

¹⁵In fact, it can be applied in the general case, since we can always replace all premises by their conjunctions.

Lastly, we note that, whereas semantic tableaux and resolution are very convenient and relatively easy to use systems of deduction, they are only suitable for proofs by contradiction, but not for direct proofs. However, reasoning by contradiction is often unnecessary and sometimes unnatural. Besides, this kind of reasoning requires that we have a specified goal (conclusion) to prove and they are of no use if we simply want to derive logical consequences from the knowledge that we have, possibly plus additional assumptions that we have made. In such cases, direct reasoning has no good alternative.

4.8.2. *Tactics for logical reasoning*

Here we discuss *tactics* of proof, that are applicable to any of the methods discussed above. These tactics can be extracted from the rules of a deductive system called Natural Deduction, not presented here. The choice of tactics depends both on the logical structure of the conclusion and on the logical structure of the premises. We look at these separately.

Tactics based on the conclusion to be proved

- If the conclusion is a **negation**, $C = \neg A$, then one can apply a proof by contradiction, by assuming A (which is equivalent to the negation of $\neg A$) and trying to reach a contradiction. Alternatively, one can import the negation in $\neg A$ inside A and try another tactic.

- If the conclusion is a **conjunction**, $C = A \wedge B$, one has to prove each of A and B .
- If the conclusion is a **disjunction**, $C = A \vee B$, one can try to prove one of A or B . This works sometimes, but not always. If it does not work, then one can try a proof by contradiction, i.e. assume $\neg C$, which is equivalent to $\neg A \wedge \neg B$. Hence we assume each of $\neg A$ and $\neg B$, and try to reach a contradiction.

Also, one can use the equivalence $P \vee Q \equiv \neg P \rightarrow Q$ and transform the disjunction into implication; then see below.

- If the conclusion is an **implication**, $C = A \rightarrow B$, one assumes A , in addition to all premises, and tries to prove B .

Alternatively, one can attempt a proof by contraposition, by tackling a conclusion $\neg B \rightarrow \neg A$ instead.

Lastly, one can try a proof by contradiction, by assuming $\neg(A \rightarrow B)$, or equivalently, A and $\neg B$.

- If the conclusion is a **biconditional**, $C = A \leftrightarrow B$, then it is equivalent to the conjunction $(A \rightarrow B) \wedge (B \rightarrow A)$ and we follow the tactics mentioned above.

- If the conclusion is a **universally quantified statement**, $C = \forall x A(x)$, one typically reasons by saying “Let c be a (name of an) arbitrary object from the domain (e.g. an arbitrary real number).” Then one tries to prove that $A(c)$ holds with a general argument, without assuming any specific properties of c . In particular, the name c must be a new one, but one that has not been mentioned in the proof yet, for the argument to be really general.

If one succeeds in proving that $A(c)$ holds in a way independent from c , then that proof will apply to *any* object c from the domain, which will produce a proof that $A(x)$ holds for *every* object x , i.e. a proof of $\forall x A(x)$.

- If the conclusion is an **existentially quantified statement**, $C = \exists x A(x)$, then we either try to find an explicit *witness* c , such that $A(c)$ holds, or we attempt a proof by contradiction, i.e. a

proof that *it cannot be the case that there does not exist x for which $A(x)$ holds*. The latter is called a *nonconstructive proof of existence*. One often comes across such nonconstructive proofs in mathematics.

Tactics based on the premises

We use each premise P according to its logical form as follows:

- If the premise is a **negation**, $P = \neg A$, then one can import that negation inside all other logical connectives in A , if any.
- If the premise is a **conjunction**, $P = A \wedge B$, one can replace it by two new premises, viz. A and B .
- If the premise is a **disjunction**, $P = A \vee B$, and it is not known which of A and B is true, then one must reason *per cases*, i.e. consider separately each of the two possible cases, viz. *Case 1*: A is true; *Case 2*: B is true.
If one succeeds to prove that in each of these cases the conclusion C follows, then this provides a proof that C follows from $A \vee B$. (Why?)
- If the premise is an **implication**, $P = A \rightarrow B$, one can try to derive A and then conclude B and use it further.
Alternatively, one can replace $A \rightarrow B$ by the equivalent disjunction $\neg A \vee B$ and reason as in the previous case.
- If the premise is a **biconditional**, $P = A \leftrightarrow B$, then one can replace it by the conjunction $(A \rightarrow B) \wedge (B \rightarrow A)$.
- If the premise is a **universally quantified statement**, $P = \forall x A(x)$, one can obtain from it new premises (possibly infinitely many) $A(c)$ for each object c from the domain. More generally, one can assume $A(t)$ for any term that is free for substitution for x in $A(x)$.
- If the premise is an **existentially quantified statement**, $P = \exists x A(x)$, then one can introduce a new name, say c , for an object that satisfies A , that is to say, “Well, if there is an object x such that $A(x)$, then let us take one and give it a name, say c .” Then one can replace the premise by $A(c)$. Note that the name c must be a new one that has not been mentioned in the proof yet nor occurs in the target conclusion, and no assumptions about c can be made other than $A(c)$.

4.8.3. Exercises

- ❶ For each of the examples of proofs given in this section determine which steps in the proof above are based on logical inferences and which on substantial (mathematical) arguments.
- ❷ Prove the following statements. Clearly indicate the method of proof and explain the steps. Distinguish the logical from the substantial steps.
 - (a) If the sum of two integers is even, then so is their difference.
 - (b) If a, b, x are real numbers, $a < b$ and $(x - a)(x - b) < 0$, then $a < x < b$.
 - (c) If a and b are rational numbers and $a < b$, then there exists a rational number r such that $a < r < b$.

- (d) For every two integers m, n , if mn is even, then m is even or n is even.
- (e) For every natural number n the number $n^2 + n$ is even.
- (f) For every two integers a, b if ab is odd, then $a + b$ is even.
- (g) No positive integers m and n exist such that $1/m + 1/n > 2$.
- (h) The number $\sqrt[3]{3}$ is irrational.
- (i) If $x + y$ is an irrational number, then x is an irrational number or y is an irrational number.



Alan Mathison Turing (23.6.1912–7.6.1954)

Alan Turing was a British mathematician, logician, computer scientist and crypto-analyst, regarded as one of the founders of Theoretical Computer Science, for his creation of the universal model of computations, now called the Turing machine, and of Artificial Intelligence, for proposing the idea of using a special test, now known as Turing's test, for deciding whether an artificial agent has intelligence. Turing was instrumental in breaking the German Enigma codes during the Second World War.

In 1935 Turing studied Gödel's incompleteness results (see Section 4.5) and Hilbert's Entscheidungsproblem (see the box at the end of Section 3.3 in Chapter 3) – the problem that asks for an algorithm that takes as input a sentence of

a first-order logic and answers whether it is valid. In the following year he wrote his groundbreaking paper *On Computable Numbers, with an Application to the Entscheidungsproblem*, where he introduced an abstract universal computing device, now called a *Turing machine*, and argued that such a machine can perform any conceivable mathematical computation representable as an algorithm. Remarkably, Turing machines described the concept of modern programmable computers before they were actually invented. Turing described in his 1936 paper the construction of a *universal Turing machine*, which can simulate the work of any Turing machine. He then showed that the *halting problem* for Turing machines is algorithmically undecidable in a sense that there cannot exist an algorithm (represented by a Turing machine) that can take the code of any Turing machine and an input for it and determine as an output whether that Turing machine will eventually halt for that input. Then Turing concluded that Hilbert's Entscheidungsproblem has no solution.

Turing completed a doctorate in mathematical logic in Princeton University under Church's supervision in 1938. Church (see Section 4.7) and Turing showed that Church's lambda-calculus and the Turing machine were equivalent as models of computation and argued that they capture precisely the intuitive notion of an algorithm. This claim became known as the *Church – Turing thesis*.

In 1938, Turing was invited to work for the British cryptanalytic department, to help with breaking the secret codes of the German encrypting machines Enigma. Turing was instrumental in developing methods and constructing special devices, called *Bombe*, for decrypting messages sent by Enigma coders, which saved many lives in the course of World War II. At the end of the war Turing was invited by the National Physical Laboratory in London to design a real, physical computer and he submitted a report proposing the Automatic Computing Engine (ACE) in 1946.

Turing was homosexual and was arrested and convicted for violating the British antihomosexuality law in 1952. To avoid going to prison he accepted to be subjected to chemical castration. Turing died in 1954 of cyanide poisoning and it is widely believed that this was a suicide, induced by his prosecution and treatment. In 2009 the British Prime Minister offered an official apology from the British Government for Turing's prosecution and in 2013 Turing was granted posthumously a royal pardon by the Queen.

In 1966 the Association for Computing Machinery established the *Turing Award* as the most prestigious award for scientific contributions in computer science, regarded as the equivalent of the Nobel Prize.

5

Number Theory

5.1.	The principle of mathematical induction revisited	220
5.2.	Divisibility	224
5.3.	Computing greatest common divisors. Least common multiples	230
5.4.	Prime numbers. The fundamental theorem of arithmetic	236
5.5.	Congruence relations	243
5.6.	Equivalence classes and residue systems modulo n	248
5.7.	Linear Diophantine equations and linear congruences	253
5.8.	Chinese remainder theorem	257
5.9.	Euler's function. Theorems of Euler and Fermat	261
5.10.	Wilson's theorem. Order of an integer	266
5.11.	Application: public key cryptography	269

Number theory is one of the oldest (over 2300 years) and most beautiful branches of mathematics, often called "the queen of mathematics". It studies the arithmetical properties of the natural numbers, in particular those related to *divisibility* and *prime numbers*. Some questions in number theory were formulated in the early times of that field, yet still remain unanswered today, like whether any odd perfect number exists. This difficulty in proving relatively simple statements prompted the great mathematician Karl Friedrich Gauss to say the following about number theory: "*It is just this which gives the higher arithmetic that magical charm which has made it the favourite science of the greatest mathematicians, not to mention its inexhaustible wealth, wherein it so greatly surpasses other parts of mathematics.*"

Since its inception during the time of Euclid, number theory was largely separate from the rest of mathematics, but towards the end of the 1800s it became more and more related with other branches of mathematics. Today number theory plays a fundamental role in the entire mathematics and is deeply connected with all other branches of it, even with seemingly distant ones such as complex analysis, where the most prominent currently open problem in mathematics, the Riemann hypothesis, is intimately related to the distribution of prime numbers.

The development of computer technology has brought about revolutionary advancement of number theory, which in turn has provided numerous important applications to information theory, coding theory, algorithmic complexity theory, cryptography and, most recently, to computational biology and bioinformatics. Among the many and various applications of number theory we will discuss here only one, viz. *public key cryptography*.

5.1. The principle of mathematical induction revisited

Here we will extend and generalize the Principle of Mathematical Induction (PMI). We begin by stating two equivalent principles from which PMI can be deduced.

A fundamental property of the set of natural numbers, which we can take as an axiom, is the following **Principle of Well-Ordering (PWO)**:

Every nonempty set of natural numbers has a least element.

This is equivalent to the following **Principle of Descending Chains (PDC)**:

There is no infinite strictly descending sequence of natural numbers.

Indeed, assuming the PWO, then every strictly descending sequence of natural numbers, considered as a set of its members, must have a least element; hence it may not be infinite. Conversely, assume the PDC and suppose PWO does not hold. Then take any nonempty set X of natural numbers with no least element. Take any $n_0 \in X$. Since it is not least in X , there must be $n_1 \in X$ such that $n_1 < n_0$. Again, n_1 is not least in X , so there must be $n_2 \in X$ such that $n_2 < n_1$, etc. Thus, we construct an infinite strictly descending sequence of natural numbers, which contradicts the PDC.

Assuming any of the principles above, we can now prove the PMI.

Theorem 5.1.1 (Principle of Mathematical Induction) *Suppose that some property P of natural numbers holds for some natural number $n_0 \geq 0$, and whenever P holds for some natural number $k \geq n_0$, it also holds for $k + 1$. Then P holds for every natural number greater than or equal to n_0 .*

Proof: The PMI can be derived from the PWO by assumption of the contrary, as follows. Suppose that there are natural numbers greater than or equal to n_0 for which P does not hold. The collection of such numbers has a least element m . It must be greater than n_0 since P holds for n_0 . Therefore $m - 1$ is a natural number less than m and greater than or equal to n_0 , hence P

must hold for $m - 1$. Then P must also hold for m since $m = (m - 1) + 1$. This is a contradiction; therefore our assumption was wrong.

Here is a seemingly stronger, but in fact equivalent, form of the principle of mathematical induction.

Theorem 5.1.2 (Principle of Complete Mathematical Induction (PCMI)) Suppose a property P of natural numbers holds for a natural number n_0 and also suppose that, for any natural number $k \geq n_0$, if P holds for all natural numbers greater than or equal to n_0 and less than k , then P holds for k . Then P holds for every natural number greater than or equal to n_0 .

Proof: Left as Exercise 4.

Thus, when using the PMCI we need to prove the following:

1. $P(n_0)$ holds for some natural number n_0 .
2. For any $k \geq n_0$, assuming that $P(k)$ holds for all natural numbers greater than or equal to n_0 and less than k then $P(k)$ also holds.

Exercise 5.1.3 Show that in the statement of Theorem 5.1.2 the base step, i.e. the assumption that the property P holds for n_0 , can be omitted.

Example 5.1.4

Every natural number $n \geq 1$ can be written as a sum of distinct powers of 2.

Solution

Let $P(n)$ be the statement “The natural number $n \geq 1$ can be written as a sum of distinct powers of 2”. We will use the PMCI to prove this statement. We start with the base step: for $n = 1$ we have that $1 = 2^0$.

Now, for any $k > 1$, we assume that P is true for all natural numbers greater than or equal to 1 and less than k (the inductive hypothesis, IH). Our goal is to prove that $P(k)$ holds, i.e. that k can be written as a sum of distinct powers of 2, using the IH. We consider the two possible cases, of k odd and k even, separately:

Case 1 Suppose k is odd. Then $k - 1$ is even and less than k and therefore by the inductive hypothesis $k - 1$ can be written as a sum of distinct powers of 2. Note that each power is at least 1 since k is even and $k > 1$. Let

$$k - 1 = 2^{p_1} + 2^{p_2} + \cdots + 2^{p_m}$$

where $1 \geq p_1 < p_2 < \cdots < p_m$. Then:

$$k = (k - 1) + 1 = (2^{p_1} + 2^{p_2} + \cdots + 2^{p_m}) + 2^0,$$

which is a sum of distinct powers of 2, as required.

Case 2 Suppose k is even. Then $\frac{k}{2}$ is a natural number less than k and by our inductive hypothesis there exist natural numbers $p_1 < p_2 < \dots < p_r$ such that

$$\frac{k}{2} = 2^{p_1} + 2^{p_2} + \dots + 2^{p_r}.$$

Now by multiplying both sides by 2 we get

$$k = 2(2^{p_1} + 2^{p_2} + \dots + 2^{p_r}) = 2^{p_1+1} + 2^{p_2+1} + \dots + 2^{p_{r+1}},$$

which is a sum of distinct powers of 2, as required.

Therefore, by the PMCI, $P(n)$ is true for all natural numbers $n \geq 1$.

5.1.1. Exercises

① Prove that for every natural number n :

$$(a) 2^0 + 2^1 + 2^2 + \dots + 2^n = 2^{n+1} - 1$$

$$(b) 1 + 3 + 5 + \dots + (2n - 1) = n^2$$

$$(c) 1^2 + 2^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$(d) 1^3 + 2^3 + \dots + n^3 = \frac{n^2(n+1)^2}{4}$$

$$(e) 1 \times 3 + 2 \times 4 + \dots + n \times (n+2) = \frac{n(n+1)(2n+7)}{6}$$

$$(f) \frac{1}{1 \times 3} + \frac{1}{2 \times 4} + \dots + \frac{1}{n \times (n+2)} = \frac{n(3n+5)}{4(n+1)(n+2)}$$

$$(g) \frac{1}{\sqrt{1}} + \frac{1}{\sqrt{2}} + \dots + \frac{1}{\sqrt{n}} \leq 2\sqrt{n}$$

② (Bernoulli's inequality) If $x \geq -1$ then $(1+x)^n \geq 1 + nx$ for every $n \in \mathbb{N}$.

③* Find closed form expressions for the following sums and prove them using the PMI:

$$(a) 1 \times 2 + 2 \times 3 + \dots + n \times (n+1)$$

$$(b) \frac{1}{1 \times 2} + \frac{1}{2 \times 3} + \dots + \frac{1}{n \times (n+1)}$$

④ Prove the PMCI, as stated in Theorem 5.1.2, using a variation of the proof of the PMI in Theorem 5.1.1.

⑤ Prove the claim in Exercise 5.1.3.

⑥* Show that the PMCI is equivalent to the PMI.

⑦* Prove that the PMCI is equivalent to the following form: "Let a property P of natural numbers be such that for any natural number k , if it holds for all natural numbers less than k then it holds for k . Then P holds for all natural numbers."

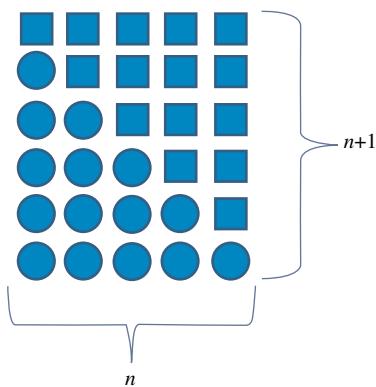


Carl Friedrich Gauss (30.4.1777–23.2.1855)

Gauss was a famous German mathematician who made fundamental contributions to many fields of mathematics and science, including number theory, algebra, analysis, differential geometry, statistics, geodesy, geophysics, electrostatics, astronomy and optics – too many even to be listed here. He is sometimes referred to as “the Prince of Mathematics” and is regarded as one of the greatest and most influential mathematicians ever.

Gauss was a child prodigy and one anecdote about his early signs of genius goes as follows. His primary school teacher tried to keep the pupils busy while he attended to his own affairs, by making them add the first 100 natural numbers. To his astonishment (and, probably, displeasure) the little Gauss gave the correct answer within seconds. How did he do it? He realized that pairwise addition of terms equally far from the opposite ends of the list of numbers always gives the same sum: $1 + 100 = 2 + 99 = 3 + 98 = \dots = 101$ which gives $50 \times 101 = 5050$.

This is how he discovered the general formula for the sum of the first n positive integers. A good lesson to learn from this story is that when solving mathematical problems a good insight can replace a lot of hard labour. The picture below gives an insightful graphical proof of the formula discovered by Gauss:



5.2. Divisibility

Divisibility is the most specific arithmetical relation between integers. While very simple to define, the study of divisibility quickly leads to the most difficult problems of arithmetic, some of which are extremely easy to formulate, yet have remained unsolved for centuries. However, let us begin with the basics.

Definition 5.2.1 A nonzero integer a divides the integer b if there is an integer q such that $b = aq$. If a divides b we write $a \mid b$ and also say that b is divisible by a , or a is a divisor of b , or b is a multiple of a . If a does not divide b we write $a \nmid b$.

For instance $3 \mid 15$ and $-3 \mid 15$, but $3 \nmid 16$, $-3 \nmid 16$ and, by definition, $0 \nmid b$ for any integer b , including $b = 0$.

Hereafter, whenever we write $a \mid b$ we will assume that $a \neq 0$.

5.2.1. Basic properties of divisibility

Theorem 5.2.2 (Basic properties of divisibility) The following hold for any integers a, b, c :

- (D1) $1 \mid b$.
- (D2) $a \mid 0$ and $a \mid a$.
- (D3) If $a \mid b$ then $-a \mid b$ and $a \mid -b$.
- (D4) If $a \mid b$ then $a \mid bc$.
- (D5) If $a \mid b$ and $b \mid c$ then $a \mid c$.
- (D6) If $a \mid b$ and $a \mid c$ then $a \mid (mb + nc)$ for any integers m, n ; in particular, $a \mid b - c$ and $a \mid b + c$.
- (D7) If $a \mid b$ and $b \mid a$ then $a = b$ or $a = -b$. In particular, if a and b are natural numbers, then $a = b$.
- (D8) If $a \mid b$ and $a, b > 0$ then $a \leq b$.
- (D9) If m is a nonzero integer then $a \mid b$ if and only if $ma \mid mb$.
- (D10) If $a \mid b$ and $c \mid d$ then $ac \mid bd$.

Proof: All these properties are immediate consequences of the definition. Let us show for instance (D6).

Let $a \mid b$ and $a \mid c$. Then $b = xa$ for some integer x and $c = ya$ for some integer y . Therefore $mb + nc = mxa + nya = (mx + ny)a$ and hence $a \mid (mb + nc)$.

We leave the proofs of the remaining properties to the reader. See Exercise 1 at the end of the section.

5.2.2. Division with a remainder

Now we can state and prove the most important basic result involving divisibility.

Theorem 5.2.3 (The fundamental theorem about division with remainder) *Given any integers $a > 0$ and b , there exist unique integers q and r such that*

$$b = qa + r$$

and

$$0 \leq r < a.$$

Furthermore, $r = 0$ if and only if $a \mid b$.

The numbers q and r are called respectively the *quotient* and the *remainder* of the division of b by a . Thus, the theorem says that every integer b can be divided with a remainder by any positive integer a in a unique way.

Proof: If $b = 0$ then $b = 0 \cdot a + b$ and $b < a$. Suppose that $b \neq 0$. Consider the arithmetic sequence $\dots, b - 2a, b - a, b, b + a, b + 2a, \dots$ Notice that 0 belongs to this sequence if and only if $b = ka$ for some integer k , i.e. if and only if $a \mid b$. Now suppose that $a \nmid b$; hence 0 does not occur in the sequence. There are positive terms in the sequence (why?). Let r be the least of them and $r = b - qa$. (Why does r exist?) We claim that $r < a$. Indeed, if $r \geq a$ then $r - a \geq 0$ and $r - a = (b - qa) - a = b - (q + 1)a$, so $r - a$ also belongs to the sequence. Furthermore, $r - a < r$, which contradicts the choice of r . Thus $0 < r < a$, so we have proved the existence of r and q with the desired properties.

Now let us show that r and q are unique with these properties. Suppose that there are two such pairs, (q_1, r_1) and (q_2, r_2) , and assume that $r_1 \geq r_2$. Then $r_1 - r_2 = (b - q_1a) - (b - q_2a) = (q_2 - q_1)a$. On the other hand, $0 \leq r_1 - r_2 < a$, which leaves the only possibility for $q_2 - q_1$ to be 0, i.e. $q_2 = q_1$ from where $r_2 = r_1$.

Finally, if $r = 0$ then $a \mid b$ by definition. Conversely, if $a \mid b$ then $a = kb + 0$ for some integer k . Then, by uniqueness of q and r , q must equal k and r must be 0.

5.2.3. Greatest common divisor

Definition 5.2.4 An integer d is a *common divisor* of the integers a and b if $d \mid a$ and $d \mid b$. The greatest integer amongst the common divisors of a and b (if it exists) is called the *greatest common divisor* of a and b , denoted $\gcd(a, b)$.

Note that $\gcd(a, b)$ is always positive: if c is also a common divisor of a and b , and c is negative, then $-c$ is also a common divisor of a and b and is greater than c .

Theorem 5.2.5 (Greatest common divisor) If a and b are integers, at least one of them nonzero, then

1. $\gcd(a, b)$ exists and is unique.
2. $\gcd(a, b) = ua + vb$ for some integers u and v .
3. Moreover, $\gcd(a, b)$ is the least positive integer that can be represented in this way.

Proof:

1. It follows from Exercise 2 at the end of this section that, unless $a = b = 0$, a and b have finitely many common divisors, so there is a greatest among them.

2. Consider the set L of all **integral linear combinations** $xa + yb$, where x and y range over all integers. There are positive numbers in L (why?). Let $d = ua + vb$ be the least positive integer in L . (Here we apply again the fundamental property of well-ordering of the natural numbers.) We will show that $d \mid a$ and $d \mid b$. Suppose $d \nmid a$. Then, by Theorem 5.2.3, there are integers q and r such that $a = dq + r$ and $0 < r < d$. Hence we have

$$r = a - qd = a - q(ua + vb) = (1 - qu)a + (qv)b$$

and thus r belongs to L , which is impossible since r is a positive integer less than d . Therefore $d \mid a$. Likewise $d \mid b$. Thus d is a common divisor of a and b . Let $g = (a, b)$. Then $a = ga'$ and $b = gb'$ and hence $d = (ga')u + (gb')v = g(a'u + b'v)$. Thus $g \mid d$ and hence $g \leq d$ by (D8) in Theorem 5.2.2 and $d \leq g$ by definition of g . Therefore $g = d$.

In the next section we will discuss the question, given a and b , of how to find coefficients u and v such that $\gcd(a, b) = ua + vb$. Hereafter, whenever we write $\gcd(a, b)$ we will assume that a and b are integers and are not both zero.

Theorem 5.2.6 (Basic properties of the greatest common divisor) *For any integers a, b, c, d, m :*

(GCD1) $\gcd(a, b) = \gcd(b, a)$.

(GCD2) *If $a \mid b$, then $\gcd(a, b) = |a|$. In particular, if $a \neq 0$ and $b = 0$ then $\gcd(a, b) = |a|$.*

(GCD3) $\gcd(a, b) = \gcd(a, -b) = \gcd(-a, b) = \gcd(-a, -b)$.

(GCD4) $\gcd(a, b) = \gcd(a, b + ac) = \gcd(a + bc, b)$.

(GCD5) *For any integer m , $\gcd(ma, mb) = m\gcd(a, b)$.*

(GCD6) *If $d \mid a$ and $d \mid b$ then $\gcd\left(\frac{a}{d}, \frac{b}{d}\right) = \frac{\gcd(a, b)}{d}$. In particular, if $d = \gcd(a, b)$ then $\gcd\left(\frac{a}{d}, \frac{b}{d}\right) = 1$.*

(GCD7) $\gcd(\gcd(a, b), c) = \gcd(a, \gcd(b, c))$.

(GCD8) *If a and b are not both 0, then every common divisor of a and b divides $\gcd(a, b)$.*

Proof: We will verify (GCD4), (GCD5) and (GCD6); the rest are left as exercises.

(GCD4) Let $g = \gcd(a, b)$, $h = \gcd(a, b + ac)$. Then $g \mid b + ac$ by Theorem 5.2.2, and hence $g \leq h$. On the other hand, for some integers u and v , $g = au + bv = a(u - cv) + (b + ac)v$. Thus, $h \mid g$ by Theorem 5.2.2, so $h \leq g$, and therefore $g = h$, since g and h are both positive. Likewise $\gcd(a, b) = \gcd(a + bc, b)$.

(GCD5) We can assume that $m > 0$; otherwise we take $-m$ and apply (GCD3). Now $\gcd(ma, mb)$ is the least positive value of $mua + mvb = m(ua + vy)$, where u, v are integers, while $\gcd(a, b)$ is the least positive value of $ua + vy$, whence the identity.

(GCD6) Again, we can assume that $d > 0$. Let $g = \gcd(\frac{a}{d}, \frac{b}{d})$ and $h = \gcd(a, b)$. Then $a = da'$, $b = db'$ and, by (GCD5), $h = d\gcd(a', b') = dg$, hence $g = \frac{h}{d}$.

The definition of greatest common divisor is readily generalized to several integers. We denote the greatest common divisor of the integers a_1, a_2, \dots, a_k by $\gcd(a_1, a_2, \dots, a_k)$.

The properties (GCD1) and (GCD7) imply that $\gcd(a_1, a_2, \dots, a_k)$ does not depend on the order in which the integers are taken, and can be computed step by step, taking only two numbers at a time.

5.2.4. Exercises

- ① Prove the remaining properties of Theorem 5.2.2.
- ② Show that every nonzero integer a has only finitely many divisors.
- ③ Divide a by b with a remainder, where
 - (a) $a = 93, b = 93$.
 - (b) $a = 93, b = 3$.
 - (c) $a = 93, b = 17$.
- ④ How many integers between 100 and 1000 are divisible by 11?
- ⑤ Prove that for any integer n the remainder of the division of n^2 by 4 is either 0 or 1.
- ⑥ Show that for any natural number n represented in a decimal system, n is divisible by
 - (a) 2 if and only if its last digit is divisible by 2.
 - (b) 4 if and only if the number formed by its last two digits is divisible by 4.
 - (c) 5 if and only if its last digit is divisible by 5.
- ⑦ Prove that for every natural number k , exactly one of any k consecutive integers is divisible by k .
- ⑧ Prove that for every natural number n ,
 - (a) if n is odd then $8 \mid n^2 - 1$.
 - (b) if n is odd then $48 \mid n^3 + 3n^2 - n - 3$.
 - (c) $8 \mid 3^{2n} - 1$.
 - (d) $9 \mid 4^n + 15n - 1$.
 - (e) $25 \mid 2^{n+2}3^n + 5n - 4$.
- ⑨ Prove that for any positive integers m, n : if $m \mid n$ then $2^m - 1 \mid 2^n - 1$.
- ⑩ Determine:
 - (a) $\gcd(24, 42)$.
 - (b) $\gcd(24, -35)$.
 - (c) $\gcd(-24, -72)$.
- ⑪ Given that $\gcd(a, 4) = 2$ and $\gcd(b, 4) = 2$ show that $\gcd(a + b, 4) = 4$.
- ⑫ Let a, b, n be positive integers such that for any positive integer k such that $k \neq b$, it holds that $k - b \mid k^n - a$. Show that $a = b^n$.
- ⑬ Prove the remaining properties from Theorem 5.2.6.

- ⑯ Let x and y be integers such that $x + y = 1000$. Is it possible that $\gcd(x, y) = 3$?
- ⑰ Using the properties (GCD1) and (GCD7) in Theorem 5.2.6 show that
- $\gcd(a_1, a_2, \dots, a_k) = \gcd(a_1, \gcd(a_2, \dots, \gcd(a_{k-1}, a_k)))$.
 - if b_1, b_2, \dots, b_k is a rearrangement of a_1, a_2, \dots, a_k then $\gcd(b_1, b_2, \dots, b_k) = \gcd(a_1, a_2, \dots, a_k)$.
- ⑱ Generalize Theorem 5.2.5 for greatest common divisors of several integers by showing that if a_1, a_2, \dots, a_k are not all zeros then $\gcd(a_1, a_2, \dots, a_k)$ exists and equals the least positive integer, which can be represented as an integer linear combination of a_1, a_2, \dots, a_k , i.e. the least positive integer d such that

$$d = u_1 a_1 + u_2 a_2 + \cdots + u_k a_k$$

for some integers u_1, u_2, \dots, u_k .

- ⑲ Show that every common divisor of a_1, a_2, \dots, a_k divides $\gcd(a_1, a_2, \dots, a_k)$.
- ⑳ Generalization of (GCD6): prove that if $\gcd(a_1, a_2, \dots, a_k) = d$ then $\gcd\left(\frac{a_1}{d}, \frac{a_2}{d}, \dots, \frac{a_k}{d}\right) = 1$.
- ㉑ Prove that for any integers a and b ,

$$\gcd(a, b) = \gcd(3a + 5b, 11a + 18b).$$

- ㉒ If a, b and c are odd integers prove that

$$\gcd(a, b, c) = \gcd\left(\frac{a+b}{2}, \frac{b+c}{2}, \frac{c+a}{2}\right).$$

- ㉓ Prove that for any integers a, b and c ,

$$\gcd(a, b) = \gcd(a + bc, a + b(c - 1)).$$

- ㉔ Prove that the fraction $\frac{21n+4}{14n+3}$ is irreducible for any natural number n .¹

- ㉕ Show that the fraction $\frac{a^3+2a}{a^4+3a^2+1}$ is irreducible.

- ㉖* A right-angled triangle has sides with integer lengths. Show that:

- (a) The length of one of the two shorter sides of the triangle is divisible by 3.

- (b) The length of one of the sides of the triangle is divisible by 5.

- ㉗* Find all three-digit numbers n such that the number formed by the last three digits of any power of n equals n .

- ㉘* Show that for any natural numbers n and m the following sums are *not* integers.

(a) $N_1 = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

(b) $N_2 = \frac{1}{n} + \frac{1}{n+1} + \cdots + \frac{1}{n+m}$

(c) $N_3 = \frac{1}{3} + \frac{1}{5} + \cdots + \frac{1}{2n+1}$

- ㉙* Prove that if $m \mid ab^n + cn + d$ for every integer $n \geq 0$ then $m \mid c^2$.

- ㉚* Prove that $m \mid ab^n + cn + d$ for every integer $n \geq 0$ if and only if $m \mid a + d$, $m \mid (b - 1)c$ and $m \mid a(b - 1) + c$.

¹ This was Problem 1 from the First International Mathematical Olympiad (IMO) in Romania, 1959.

- ㉙* Show that the number $m^4 + m^3 + m^2 + m + 1$, where m is a natural number, is a square only when $m = 3$.
- ㉚* If m is a natural number greater than 1, show that every natural number N can be uniquely represented in the form

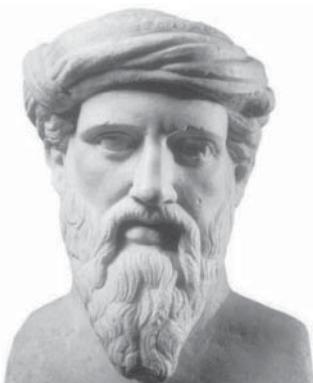
$$N = c_0 + c_1m + c_2m^2 + \cdots + c_km^k$$

(when $k = 0, N = c_0$), where the coefficients c_i are integers satisfying the conditions $0 \leq c_i < m$ ($i = 0, 1, \dots, k - 1$) and $0 < c_k < m$.

This statement means that every natural number can be uniquely represented in a number system with a base m for every natural $m > 1$.

- ㉛* Let a, b, c, A, B, C be integers such that $\gcd(a, b) = c$ and $\gcd(A, B) = C$. Show that $\gcd(aA, ab, Ab, AB) = cC$.
- ㉜* Let k be any natural number and $a_n = k^{2^n} + 1$. Show that if $m \neq n$ then $\gcd(a_m, a_n) = 1$ if k is even; otherwise $\gcd(a_m, a_n) = 2$.
- ㉝* Let $a > 1$ and m be natural numbers. Show that

$$\gcd\left(\frac{a^m - 1}{a - 1}, a - 1\right) = \gcd(a - 1, m).$$



Pythagoras of Samos (c. 570 BC–c. 495 BC)

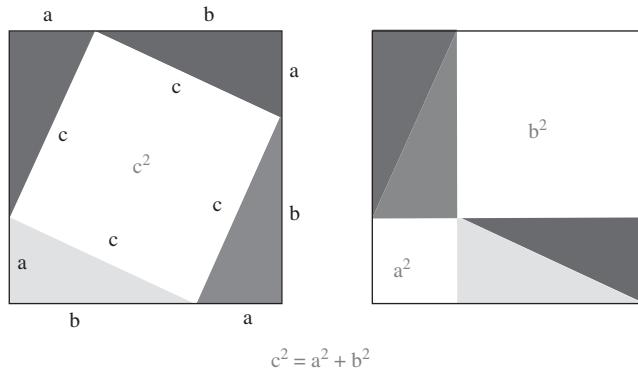
Pythagoras was an ancient Greek mathematician and philosopher, founder of the religious movement called Pythagoreanism. He made important contributions to mathematics, astronomy, religion and music.

Pythagoras left Samos at the age of 40 and, after years of travelling and religious studies in Egypt and Babilon, settled in Crotona, a Greek colony in southern Italy, where he founded a philosophical and religious school and a secret society, engaging in a mix of mathematics, religion and mysticism. Pythagoras believed that all things are (positive) integers and proportions between them, that mathematics is the basis for everything and the physical world can

be understood through mathematics, and that geometry is the highest form of mathematical studies. He also believed that numbers have personalities, characteristics, strengths and weaknesses. However, in his time Pythagoras was famous not so much as a mathematician but rather as a magician and religious leader, who claimed that the soul resides in the brain, is immortal and moves through a series of reincarnations, sometimes from humans to animals.

Pythagoras and his followers made many discoveries in geometry and arithmetic, including possibly the discovery of irrational numbers, as the first proof that the length of the diagonal of a square is not commensurable with the length of its side is attributed to the Pythagorean school, even though it contradicts their belief that only ratios between integers, i.e. rational numbers, existed. It is believed that Pythagoras knew the five regular (also known as Platonic) solids: tetrahedron, cube, octahedron, icosahedron, dodecahedron, and that he knew how to construct the first three of them. Pythagoras also related music to mathematics and discovered that vibrating strings sounded harmoniously when the lengths of the strings were proportional to whole numbers, e.g. 2:1, 3:2, 4:3.

However, Pythagoras is probably most famous with proving, perhaps the first ever, geometric result – now called *Pythagoras theorem* but known to the Babylonians more than a 1000 years earlier – that in a right-angled triangle the square of the hypotenuse is equal to the sum of the squares of the other two sides. Here is a purely pictorial proof of Pythagoras theorem:



5.3. Computing greatest common divisors. Least common multiples

5.3.1. Euclid's algorithm for computing greatest common divisors

Now we know many properties of the greatest common divisor of two integers, but still the main question remains: *how to compute it?* Due to properties (GCD1), (GCD2) and (GCD3) in Theorem 5.2.6 we see that it is sufficient to be able to compute the greatest common divisor for *positive* integers; then we can do so for any integers. For instance, $\gcd(-123, -321) = \gcd(123, 321)$.

We are going to present a simple and elegant recursive procedure for computing the greatest common divisor of two integers, going back to Euclid over 2300 years ago, based on the following property.

Theorem 5.3.1 *Let $a > b > 0$ be integers and r be the remainder of the division of a by b . Then $\gcd(a, b) = \gcd(r, b)$.*

Proof: The proof is left as an exercise.

Theorem 5.3.2 (The Euclidean algorithm) *Let a and b be positive integers. The following procedure computes $\gcd(a, b)$.*

1. Set $X := \max(a, b)$ and $Y := \min(a, b)$.
2. Divide X by Y with a remainder: $X = QY + R$, where $0 \leq R < Y$.
3. If $R = 0$ then $\gcd(a, b) = Y$ and the procedure ends.
4. If $R > 0$ put $X := Y$, then $Y := R$, and go to step 2.

Before we show *why* this algorithm works, let us see *how* it works on a particular example.

Example 5.3.3

To find $\gcd(30, 102)$ using the Euclidean algorithm we follow the procedure:

Set $X := 102$ and $Y := 30$.

Divide X by Y with a remainder: $102 = 3 \cdot 30 + 12$.

The remainder is not 0; hence $X := 30$ and $Y := 12$.

Divide X by Y with a remainder: $30 = 2 \cdot 12 + 6$.

The remainder is not 0; hence $X := 12$ and $Y := 6$.

Divide X by Y with a remainder: $12 = 2 \cdot 6 + 0$.

The remainder is 0; hence $\gcd(30, 102) = Y = 6$.

End of the procedure.

Now let us prove that the Euclidean algorithm works as it should, i.e. to verify its *correctness*. We must check two things:

1. **Partial correctness:** for any two positive integers, if the procedure ends the result is indeed their greatest common divisor.

Let us first observe that at any stage of the execution of the algorithm, $\gcd(X, Y)$ remains equal to $\gcd(a, b)$.² At the beginning this is trivially so. The only changes of the values of X and Y occur at step 4. There the pair (X, Y) is replaced by (Y, R) , where $R = X - QY$ is the remainder of the division of X by Y . Hence we only need to check that

$$\gcd(X, Y) = \gcd(Y, R),$$

which follows from Theorem 5.3.1 and (GCD1) in Theorem 5.2.6.

Thus, $\gcd(X, Y)$ is always equal to $\gcd(a, b)$. It remains to show that whenever the procedure ends, the result is what we need. The procedure ends only if the remainder R becomes 0, i.e. the values of X and Y become such that $Y \mid X$. Then the result is Y , which is indeed $\gcd(X, Y)$, due to (GCD2).

2. **Termination:** for any two natural numbers a, b the procedure ends.

Suppose that for some natural numbers a, b the procedure never ends. This can only happen if step 4 (the only one that sends the execution back) is executed infinitely many times. Let us notice two things:

- (a) Each time when step 4 is executed, the value of X strictly decreases, being replaced by the value of Y . Indeed, $X \geq Y$ is always the case, and if $X = Y$ then R becomes 0.

² In computer science terms, $\gcd(X, Y)$ is an *invariant* of the algorithm.

- (b) The values of X and Y always remain natural numbers. Thus, the consecutive values of X form an infinite, strictly decreasing sequence of positive integers. Due to the principle of descending chains of natural numbers, this is impossible because the set of elements in such a sequence has no least element. Therefore, the procedure must terminate.

Finally, we note that the Euclidean algorithm also produces coefficients u and v such that $\gcd(a, b) = ua + vb$. Indeed, suppose $a > b > 0$ and the algorithm goes through the steps

$$(1) \quad a = q_0b + r_1,$$

$$(2) \quad b = q_1r_1 + r_2,$$

...

$$(k) \quad r_{k-1} = q_kr_k + r_{k+1},$$

$$(k+1) \quad r_k = q_{k+1}r_{k+1},$$

so that $\gcd(a, b) = r_{k+1}$.

From equation (k) we get $r_{k+1} = r_{k-1} - q_kr_k$, i.e. obtain $\gcd(a, b)$ as a linear combination of r_{k-1} and r_k . Then, from equation $(k-1)$ we find $r_k = r_{k-2} - q_{k-1}r_{k-1}$ and, eliminating r_k , we obtain $\gcd(a, b)$ as a linear combination of r_{k-2} and r_{k-1} , etc. Thus, working backwards we eventually obtain $\gcd(a, b)$ in the form $ua + vb$ for some u and v .

For instance, from Example 5.3.3 we find that $\gcd(30, 102) = 6 = 30 - 2 \times 12 = 30 - 2(102 - 3 \times 30) = 7 \times 30 - 2 \times 102$.

5.3.2. Least common multiple

Definition 5.3.4 An integer m is a **common multiple** of the integers a and b if $a \mid m$ and $b \mid m$. The least positive number amongst the common multiples of a and b , if it exists, is called the **least common multiple** of a and b , denoted $\text{lcm}(a, b)$.

Note that if a and b are nonzero then $\text{lcm}(a, b)$ always exists. Indeed, the collection of positive common multiples of a and b is nonempty, since $|ab|$ belongs to it. Therefore, by the principle of well-ordering of natural numbers, there is a least number in that collection.

Theorem 5.3.5 (Basic properties of the least common multiple) For any integers a, b, c, n :

$$(LCM1) \quad \text{lcm}(a, b) = \text{lcm}(b, a).$$

$$(LCM2) \quad \text{If } a \mid b \text{ then } \text{lcm}(a, b) = |b|.$$

$$(LCM3) \quad \text{lcm}(a, b) = \text{lcm}(a, -b) = \text{lcm}(-a, b) = \text{lcm}(-a, -b).$$

$$(LCM4) \quad \text{lcm}(na, nb) = |n| \text{lcm}(a, b).$$

$$(LCM5) \quad \text{lcm}(\text{lcm}(a, b), c) = \text{lcm}(a, \text{lcm}(b, c)).$$

Proof: Verifying these properties is routine and left as an exercise.

One can likewise introduce a least common multiple of several integers a_1, a_2, \dots, a_k , denoted by $\text{lcm}(a_1, a_2, \dots, a_k)$.

Theorem 5.3.6 *For any nonzero integers a_1, a_2, \dots, a_k , $\text{lcm}(a_1, a_2, \dots, a_k)$ divides each of their common multiples.*

Proof: Let $m = \text{lcm}(a_1, a_2, \dots, a_k)$ and n be a common multiple of a_1, a_2, \dots, a_k . Let us divide n by m with a remainder: $n = mq + r$, $0 \leq r < m$. Then $r = n - mq = 1 \cdot n + (-q)m$. Hence, by (D5) in Theorem 5.2.2, r must be a common multiple of a_1, a_2, \dots, a_k that is less than m . Therefore $r = 0$ and hence $m \mid n$.

Theorem 5.3.7 *For every pair of positive integers a and b :*

$$\gcd(a, b) = \frac{ab}{\text{lcm}(a, b)}.$$

Proof: Let $g = \gcd(a, b)$, $m = \text{lcm}(a, b)$ and $d = \frac{ab}{m}$. Then $a = \frac{m}{b}d$ and $b = \frac{m}{a}d$, where $\frac{m}{b}$ and $\frac{m}{a}$ are integers. Therefore d is a common divisor of a and b . By property (GCD8) in Theorem 5.2.6, $d \mid g$, i.e. $g = dk$ for some positive integer k . Then

$$\frac{m}{k} = \frac{md}{g} = \frac{ab}{g},$$

which is an integer. Moreover, $\frac{m}{k}$ is a positive common multiple of a and b :

$$\frac{m}{k} = \frac{a}{g}b = \frac{b}{g}a,$$

where $\frac{a}{g}$ and $\frac{b}{g}$ are integers. Since m is the least positive common multiple of a and b , it follows that $m \leq \frac{m}{k}$, which is possible only if $k = 1$. Thus $g = d$.

Example 5.3.8

Find $\text{lcm}(28, 42)$.

Solution

Using the Euclidean algorithm we find $\gcd(28, 42) = 14$; hence $\text{lcm}(28, 42) = \frac{28 \times 42}{14} = 84$.

5.3.3. Exercises

- ① Prove Theorem 5.3.1.
- ② Using the Euclidean algorithm, find
 - (a) $\gcd(70, 112)$.
 - (b) $\gcd(258, 801)$.

- (c) $\gcd(7469, 2464)$.
- (d) $\gcd(2689, 4001)$.
- ③ Show that the product of every
 - (a) 3 consecutive integers is divisible by 6.
 - (b) 4 consecutive integers is divisible by 24.
 - (c) 5 consecutive integers is divisible by 120.
- ④ Prove that for every integer n :
 - (a) $6 \mid n^3 - n$.
 - (b) $30 \mid n^5 - n$.
 - (c) $120 \mid n^5 - 5n^3 + 4n$.
- ⑤ Verify the basic properties of the least common multiple listed in Theorem 5.3.5.
- ⑥ Find:
 - (a) $\text{lcm}(70, 71)$.
 - (b) $\text{lcm}(630, 1344)$.
- ⑦ Let n be a natural number. Find $\text{lcm}(n, n+1)$.
- ⑧ Let m and n be natural numbers such that $\gcd(m, n) = \text{lcm}(m, n)$. Show that $m = n$.
- ⑨ Show that $\text{lcm}(a_1, a_2, \dots, a_k) = \text{lcm}(a_1, \text{lcm}(a_2, \dots, a_k))$.
- ⑩ Find all pairs of natural numbers m, n for which $\gcd(m, n) = 10$ and $\text{lcm}(m, n) = 100$.
- ⑪* Prove that for any natural number n ,

$$\text{lcm}(1, 2, 3, \dots, 2n) = \text{lcm}(n+1, n+2, \dots, 2n).$$

- ⑫* Prove that for any natural numbers m and n ,

$$\text{lcm}(1, 2, \dots, m, n, n+1, \dots, n+m-1) = \text{lcm}(n, n+1, \dots, n+m-1).$$

- ⑬* Let m and n be natural numbers. Show that

$$\gcd(m, n) = 1 \text{ iff } \gcd(2^m - 1, 2^n - 1) = 1.$$

- ⑭* Let a, b, m, n be natural numbers such that $a > 1$ and $\gcd(a, b) = 1$. Prove that if $a^m + b^m \mid a^n + b^n$ then $m \mid n$.
- ⑮* Find all pairs of natural numbers $\langle x, y \rangle$ satisfying the equation $2^x = 3^y + 5$.


Euclid of Alexandria (c. 325 BC–c. 265 BC)

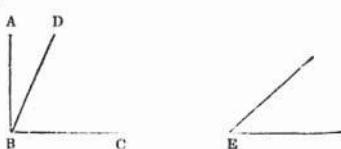
Euclid was the most prominent and influential mathematician of ancient Greece. He wrote the monumental 13-book work, known as the *Elements*, in which he systematically laid the sound foundations of both geometry and arithmetic. The *Elements* encompassed the system of postulates and theorems for what is now called Euclidean geometry and is probably the most influential book written in the history of mathematics, still used as the classic textbook for the study of geometry until the end of the 19th century, when David Hilbert revised and modernized it.

**THE
ELEMENTS OF EUCLID.**

BOOK I.

DEFINITIONS.

- I.** A point is that which hath no parts, or which hath no magnitude.
- II.** A line is length without breadth.
- III.** The extremities of a line are points.
- IV.** A straight line is that which lies evenly between its extreme points.
- V.** A superficies is that which hath only length and breadth.
- VI.** The extremities of a superficies are lines.
- VII.** A plane superficies is that in which any two points being taken, the straight line between them lies wholly in that superficies.
- VIII.** "A plane angle is the inclination of two lines to one another* in a plane, which meet together, but are not in the same direction."
- IX.** A plane rectilineal angle is the inclination of two straight lines to one another, which meet together, but are not in the same straight line.



* See Notes.

POSTULATES.

- I.** Let it be granted that a straight line may be drawn from any one point to any other point.
- II.** That a terminated straight line may be produced to any length in a straight line.
- III.** And that a circle may be described from any centre, at any distance from that centre.

AXIOMS.

- I.** THINGS which are equal to the same are equal to one another.
- II.** If equals be added to equals, the wholes are equals.

The first book of the *Elements* contains 5 postulates for points and lines in the plane. The fifth postulate is the famous *Parallel Postulate*, stating that for every line and for every point that does not lie on that line there exists a unique line through the point that is parallel to the given line. Since it was not as simple to state and obvious as the other postulates, Euclid himself and many mathematicians after him tried for two millennia to derive it from the other four postulates. However, all these attempts were futile and only in the early 19th century did Gauss, Lobachevski and Bolyai show independently that negations of this postulate lead to the development of consistent, alternative *non-Euclidean geometries*, like elliptic and hyperbolic geometries.

The many fundamental contributions to number theory in the *Elements* include the study of prime numbers and Euclid's celebrated proof of the infinitude of prime numbers, Euclid's lemma on factorization and, of course, the Euclidean procedure for computing the greatest common divisor of two integers, given in Theorem 5.3.2. This procedure was one of the first and most famous instances of what is nowadays called an *algorithm*.

Here is a copy of the first few definitions and postulates from an 1838 edition of Euclid's *Elements*, published by Robert Simson in Philadelphia, USA:

5.4. Prime numbers. The fundamental theorem of arithmetic

5.4.1. Relatively prime numbers

The integers a and b are called **relatively prime** if

$$\gcd(a, b) = 1.$$

For instance 8 and 15 are relatively prime, while 9 and 15 are not.

The following theorem will summarize those properties of relatively prime numbers which will be used further on.

Theorem 5.4.1 (Basic properties of relatively prime numbers) *For all integers a, b and c :*

- (RP1) *If $c \mid a$ and $\gcd(a, b) = 1$ then $\gcd(c, b) = 1$.*
- (RP2) *If $c \mid ab$ and $\gcd(c, a) = 1$ then $c \mid b$.*
- (RP3) *If $\gcd(a, b) = 1$ then $\gcd(a, bc) = \gcd(a, c)$.*
- (RP4) *$\gcd(a, bc) = 1$ if and only if $\gcd(a, b) = 1$ and $\gcd(a, c) = 1$.*
- (RP5) *If $\gcd(a, b) = 1$ then $ab \mid c$ if and only if $a \mid c$ and $b \mid c$.*

Proof:

- (RP1) Every common divisor of c and b is also a common divisor of a and b .
- (RP2) Assume that $c \mid ab$ and $\gcd(c, a) = 1$. Then $ab = kc$ for some integer k and $1 = uc + va$ for some integers u and v . Multiplying the latter equality by b we get

$$b = ubc + vab = ubc + vkc = (ub + vk)c;$$

hence $c \mid b$.

- (RP3) Let $\gcd(a, b) = 1$, $\gcd(a, bc) = d_1$ and $\gcd(a, c) = d_2$. Then $d_2 \mid d_1$ because $d_2 \mid a$ and $d_2 \mid bc$. On the other hand, d_1 must be relatively prime to b because $d_1 \mid a$. Hence, by (RP1), $d_1 \mid c$ since $d_1 \mid bc$. Therefore $d_1 \mid d_2$ and hence $d_1 = d_2$ because $d_1, d_2 > 0$.
- (RP4) If $\gcd(a, bc) = 1$ then $\gcd(a, b) = 1$ and $\gcd(a, c) = 1$ since every divisor of $\gcd(a, b)$ or $\gcd(a, c)$ is a divisor of $\gcd(a, bc)$. Conversely, if $\gcd(a, b) = 1$ and $\gcd(a, c) = 1$ then, by (RP2), $\gcd(a, bc) = \gcd(a, c) = 1$.
- (RP5) If $ab \mid c$ then $a \mid c$ and $b \mid c$. Now, let $\gcd(a, b) = 1$ and suppose that $a \mid c$ and $b \mid c$. Then $c = ak$ for some integer k and $b \mid ak$. Therefore, by (RP2), $b \mid k$ and hence $ab \mid ak$, i.e. $ab \mid c$.

Definition 5.4.2 *Natural numbers a_1, \dots, a_n are said to be:*

- *relatively prime* if $\gcd(a_1, \dots, a_n) = 1$;
- *relatively prime in pairs* if every two of them are relatively prime.

Obviously, if the integers a_1, \dots, a_n are relatively prime in pairs then they are relatively prime. The converse, however, is not generally true (see Exercise 1 at the end of this section).

5.4.2. Prime numbers

Definition 5.4.3 A natural number $p > 1$ is called a **prime number** if it has exactly two different positive divisors: 1 and p (called the **trivial divisors** of p). Otherwise p is called a **composite number**.

For instance, 2, 3, 5, 37, 101, 2011 are primes; 4, 189, 1001, 2013 are composite (check that). Note that 1 is neither prime nor composite by this definition.

Theorem 5.4.4 Every natural number greater than 1 has a prime divisor.

Proof: Assume the contrary: there are natural numbers greater than 1 without any prime divisors. Let m be the least of them. Since $m \mid m$, m cannot be prime; hence it is composite, i.e. has a positive divisor n , different from 1 and from m . Then $1 < n < m$ and hence n must have a prime divisor p . Thus $p \mid n$ and $n \mid m$ and hence $p \mid m$, which is a contradiction. Therefore there is no such m .

Remark 4 In fact, every composite number n has a prime divisor not greater than \sqrt{n} . Indeed, let $n = n_1 n_2$ where $n_1, n_2 > 1$. Then at least one of n_1 and n_2 , assume n_1 , is not greater than \sqrt{n} . Now, n_1 has a prime divisor, which is of course not greater than n_1 and is also a divisor of n .

Theorem 5.4.5 (Euclid, about 2350 years ago) There are infinitely many prime numbers.

Proof: Assume the contrary: there are only finitely many primes and let p_1, p_2, \dots, p_k be a list of all of them. Now, consider the number

$$N = p_1 p_2 \dots p_k + 1.$$

Since N is greater than each of p_1, p_2, \dots, p_k it must be composite. By the previous theorem, N has a prime divisor p , which must be some of p_1, p_2, \dots, p_k , for instance $p = p_i$. Then $p_i \mid N$ and $p_i \mid p_1 p_2 \dots p_k$ since p_i occurs as a factor in $p_1 p_2 \dots p_k$. Hence, by (D5) in Theorem 5.2.2, $p_i \mid (N - p_1 p_2 \dots p_k)$, i.e. $p_i \mid 1$, which is impossible since every prime is greater than 1. Therefore we must conclude that there are infinitely many primes.

Theorem 5.4.6 For all positive integers a and b and a prime p :

1. Either $p \mid a$ or $\gcd(p, a) = 1$.
2. If $p \mid ab$ then p divides at least one of a and b .

Proof:

1. $\gcd(p, a)$ can only be either p or 1.
2. If $p \nmid a$ then $(p, a) = 1$ and hence, by (RP2) in Theorem 5.4.1, $p \mid b$.

The last property can easily be generalized by induction to the following theorem.

Theorem 5.4.7 For all natural numbers a_1, \dots, a_k and a prime p , if $p \mid a_1 \dots a_k$ then p divides at least one of a_1, \dots, a_k .

Proof: The proof is left as an exercise.

5.4.3. The fundamental theorem of arithmetic

The notion of prime number is pivotal in number theory. As we will see from the next theorem, the prime numbers are like “elementary particles” from which all natural numbers are built.

Theorem 5.4.8 (Fundamental theorem of arithmetic) *Every natural number $n > 1$ can be decomposed into a product of primes:*

$$n = p_1 \cdots p_k, \quad k \geq 1.$$

Moreover, this decomposition is unique, up to the order of the factors.

Proof: We can apply the principle of (complete) mathematical induction. Assume that the theorem holds for all natural numbers m such that $m < n$. We must show that it also holds for n . If $n = 1$ nothing is claimed, so that this case is trivially true. Now, let $n > 1$. Then n has prime divisors. Let p_1 be the least of them. Then $n = p_1 n_1$ and $n_1 < n$. If $n_1 = 1$ then $n = p_1$ is prime and this is the decomposition of n ; it is unique since p_1 cannot be expressed as a product of two or more primes. Now, if $1 < n_1$ then, according to the inductive assumption, n_1 is decomposed into a product of primes: $n_1 = p_2 \cdots p_k$. Then we have a decomposition of n :

$$n = p_1 p_2 \cdots p_k.$$

Now, for the uniqueness. Suppose that besides this decomposition of n there is another one: $n = q_1 \cdots q_m$. Then

$$p_1 \cdots p_k = q_1 \cdots q_m.$$

Thus $p_1 \mid q_1 \cdots q_m$, and hence p_1 divides some of the factors q_1, \dots, q_m , by Theorem 5.4.7. We may assume (possibly after reordering) that $p_1 \mid q_1$. This means that $p_1 = q_1$ since q_1 is prime. Then, after cancelling both sides by p_1 we obtain:

$$p_2 \cdots p_k = q_2 \cdots q_m.$$

Again, the inductive assumption holds for the product $p_2 \cdots p_k$, so q_2, \dots, q_m must be a reordering of $p_2 \cdots p_k$. Hence $k = m$ and q_1, \dots, q_m is a reordering of $p_1 \cdots p_k$.

This completes the induction; hence the theorem is valid for every $n > 1$.

The above theorem can also be reformulated as follows.

Corollary 5.4.9 *Every natural number $n > 1$ can be decomposed into a product of powers of different primes:*

$$n = p_1^{\alpha_1} \cdots p_k^{\alpha_k}, \quad \alpha_1, \dots, \alpha_k, \quad k \geq 1.$$

Furthermore, this decomposition is unique up to the order of the factors.

Theorem 5.4.10 *Let $n = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$ and $m = p_1^{\beta_1} \cdots p_k^{\beta_k}$ be decompositions of m and n in products of primes, where p_1, p_2, \dots, p_k are different and some of the α 's and β 's can be 0. Then:*

1. $n \mid m$ iff $\alpha_1 \leq \beta_1, \dots, \alpha_k \leq \beta_k$.
2. $\gcd(m, n) = p_1^{\gamma_1} \cdots p_k^{\gamma_k}$ and $\text{lcm}(m, n) = p_1^{\delta_1} \cdots p_k^{\delta_k}$, where $\gamma_i = \min(\alpha_i, \beta_i)$ and $\delta_i = \max(\alpha_i, \beta_i)$, for $i = 1, \dots, k$.

Proof:

1. If $n \mid m$ then $p_i^{\alpha_i} \mid m$, i.e. $p_i^{\alpha_i} \mid p_1^{\beta_1} \cdots p_i^{\beta_i} \cdots p_k^{\beta_k}$. However, $\gcd(p_i^{\alpha_i}, p_j^{\beta_j}) = 1$ for every $j \neq i$ and hence $p_i^{\alpha_i} \mid p_i^{\beta_i}$, by repeated application of Theorem 5.4.1 (RP2). Thus, $\alpha_i \leq \beta_i$. Conversely, if $\alpha_i \leq \beta_i$ for every $i = 1, \dots, k$, then $m = nq$ where $q = p_1^{\beta_1 - \alpha_1} \cdots p_k^{\beta_k - \alpha_k}$; hence $n \mid m$.
2. This follows easily from (1) and is left as an exercise.

Example 5.4.11

If $n = 9800 = 2^3 \times 5^2 \times 7^2$ and $m = 14\,742 = 2 \times 3^4 \times 7 \times 13$ then we can represent them as $n = 2^3 \times 3^0 \times 5^2 \times 7^2 \times 13^0$ and $m = 2^1 \times 3^4 \times 5^0 \times 7^1 \times 13^1$. Then, using Theorem 5.4.10, we obtain $\gcd(m, n) = 2^1 \times 3^0 \times 5^0 \times 7^1 \times 13^0 = 14$ and $\text{lcm}(m, n) = 2^3 \times 3^4 \times 5^2 \times 7^2 \times 13^1 = 10\,319\,400$.

5.4.4. On the distribution of prime numbers

Looking at the sequence of primes one can see that they are rather irregularly scattered amidst the natural numbers. On the one hand, there are many (a hypothesis says *infinitely many*) so-called **prime twins**, i.e. pairs of primes that differ by 2. On the other hand, there are arbitrarily large “gaps”, i.e. sequences of natural numbers without any primes.

Proposition 5.4.12 *For any positive integer k there is a sequence of k consecutive composite natural numbers.*

Proof: We leave as an exercise to show that for any positive integer k , all numbers $(k+1)! + 2, \dots, (k+1)! + k, (k+1)! + (k+1)$ are composite.

Finally, we state without proofs (which are too complex to be presented here) some important and very useful facts about the distribution of primes. The first one was conjectured by Bertrand in 1845 and proved by Tchebychef in 1850.

Theorem 5.4.13 *For every integer $n \geq 4$ there is at least one prime p such that $n < p < 2n - 2$.*

The second one is credited to Dirichlet.

Theorem 5.4.14 *Every arithmetic progression $a, a+d, a+2d, \dots$, where $d > 0$ and $\gcd(a, d) = 1$, contains infinitely many primes.*

The last one, known as “the prime number theorem” and proved independently by Hadamard and de la Vallée Poussin in 1896, gives an asymptotic estimate of the density of the primes in the sequence of all natural numbers.

Theorem 5.4.15 (The prime number theorem) Let $\pi(n)$ be the number of primes amongst $\{1, \dots, n\}$. Then $\pi(n)$ is asymptotically equal to $n/\ln n$, i.e.:

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n/\ln n} = 1.$$

Consequently, the probability of a randomly chosen number n to be prime is approximately $\frac{1}{\ln n}$.

The proof of the prime number theorem uses Riemann's ζ function and methods of complex analysis, and this links the distribution of the primes with one of the most famous still unsolved problems in mathematics, the Riemann hypothesis.

5.4.5. Exercises

- ① (a) Find three integers that are relatively prime but no two of them are relatively prime.
 (b) Find four integers that are relatively prime but no three of them are relatively prime.
 (c) Find four integers such that every three but no two of them are relatively prime.
- ② Show that amongst any 5 consecutive integers there is at least one that is relatively prime to each of the others.
- ③ Factor by hand each of the following integers into products of primes:
 - (a) 128
 - (b) 129
 - (c) 728
 - (d) 1599 (Hint: $1599 = 1600 - 1$)
 - (e) 8532
- ④ Prove Claim 2 of Theorem 5.4.10.
- ⑤ Using Theorem 5.4.10 determine the following:
 - (a) $\gcd(60, 144)$ and $\text{lcm}(60, 144)$.
 - (b) $\gcd(728, 8532)$ and $\text{lcm}(728, 8532)$.
- ⑥ If $p \geq q \geq 5$ are primes, show that:
 - (a) $3 \mid p+q$ or $3 \mid p-q$.
 - (b) $24 \mid p^2 - q^2$.
- ⑦ Prove that if p is prime and $0 < k < p$ then $p \mid \binom{p}{k}$.
- ⑧ Without using Dirichlet's Theorem 5.4.14, show that there are infinitely many primes of the form
 - (a) $4n+3$.
 - (b) $6n+5$.

Hint: assume the contrary and apply an argument similar to the proof of Theorem 5.4.5.

- ⑨ Show that if a prime is divided by 30 the remainder is either 1 or a prime. Is this true if 30 is replaced by 60?
- ⑩ For any prime p other than 2 and 5 prove that p divides infinitely many of the integers 11, 111,
- ⑪ If p_n denotes the n th prime in the sequence of natural numbers, prove the inequality

$$p_n < 2^{2^n}.$$

Hint: use the idea of the proof of Theorem 5.4.8.

- ⑫ Prove that for every $n > 2$ there is a prime p such that $n < p < n!$.
- ⑬* If $\gcd(a, b) = 1$ show that $\gcd(a + b, a^2 - ab + b^2)$ is either 1 or 3.
- ⑭ Show that if the number $2^p - 1$ is a prime then p is a prime.
Prime numbers of the kind $2^p - 1$ are called **Mersenne primes**.
- ⑮ Show that if the number $2^n + 1$ is a prime then n is a power of 2.
Prime numbers of the kind $2^{2^k} + 1$ are called **Fermat primes**.
- ⑯ Show that for any integers a and b ,

$$\gcd(a, b) = \gcd(a + b, \text{lcm}(a, b)).$$

- ⑰ Using Theorem 5.4.10 prove the following identities for any natural numbers a, b, c .
- (a) $\gcd(a, \text{lcm}(b, c)) = \text{lcm}(\gcd(a, b), \gcd(a, c))$
- (b) $\text{lcm}(a, \gcd(b, c)) = \gcd(\text{lcm}(a, b), \text{lcm}(a, c))$
- (c) $\text{lcm}(\gcd(a, b), \gcd(b, c), \gcd(c, a)) = \gcd(\text{lcm}(a, b), \text{lcm}(b, c), \text{lcm}(c, a))$
- (d) $\text{lcm}(a, b, c) = \frac{abc}{\gcd(ab, bc, ca)}$
- (e) $\frac{(\text{lcm}(a, b, c))^2}{\text{lcm}(a, b)\text{lcm}(b, c)\text{lcm}(c, a)} = \frac{(\gcd(a, b, c))^2}{\gcd(a, b)\gcd(b, c)\gcd(c, a)}$
- ⑱ Let m_1, \dots, m_n be natural numbers, $M = m_1 \cdots m_n$ and $M_i = \frac{M}{m_i}$, for $i = 1, 2, \dots, n$. Prove that m_1, \dots, m_n are relatively prime in pairs if and only if $\gcd(M_1, \dots, M_n) = 1$.
- ⑲ A natural number n is a **perfect k th power** if $n = a^k$ for some natural number a . Prove that if m and n are natural numbers such that $\gcd(m, n) = 1$ and mn is a perfect k th power for some $k \geq 2$ then m and n are perfect k th powers.
- ⑳* Show that the sequence $\{2^n - 3 \mid n = 2, 3, 4, \dots\}$ contains infinitely many numbers relatively prime in pairs.³
- ㉑* A sequence of natural numbers is defined by the following recurrent equation: $t_1 = 2$ and $t_{n+1} = t_n^2 - t_n + 1$. Show that if $m \neq n$ then $\gcd(t_m, t_n) = 1$.
- ㉒* Prove that amongst any 16 consecutive integers there is one that is relatively prime to each of the others.
- ㉓* A natural number m is called **perfect** if it equals the sum of its proper positive divisors (i.e. all positive divisors of m excluding m itself). For example, $6 = 1 + 2 + 3$ and $28 = 1 + 2 + 4 + 7 + 14$ are perfect numbers.
Prove the following theorem (Euclid, *Elementa*, Book 9):
If $2^m - 1$ is a prime then $2^{m-1}(2^m - 1)$ is a perfect number.

³ This was Problem 3 from the 13th IMO in Czechoslovakia, 1971.

24* If a, m, n are natural numbers, $a > 1$ and $\gcd(m, n) = d$, show that

$$\gcd(a^m - 1, a^n - 1) = a^d - 1.$$

25 Let n and $k \geq 2$ be natural numbers. Show that there exist n consecutive natural numbers each of which can be factorized in a product of at least k (not necessarily different) prime divisors.

Primality testing

An important problem in number theory is the detection of primes amongst the natural numbers. It was perhaps the Greek mathematician Eratosthenes (circa 200 BC) who first introduced a general method to single out the primes in the sequence of natural numbers, called now the *Sieve of Eratosthenes*. It is very simple and works as follows: in order to list the prime numbers between 1 and n we take the sequence of all natural numbers from 1 to n and cross out all multiples of 2 except 2, then all multiples of the next noncrossed number, viz. 3, except 3 itself, and so on, until we have deleted all numbers divisible by any natural number not greater than \sqrt{n} (recall the remark after Theorem 5.4.4). The remaining numbers are precisely the primes between 1 and n .

Since Eratosthenes, a number of increasingly more efficient methods and algorithms have been discovered for testing primality. One of the most popular and practically useful is the Miller – Rabin Randomized Primality Test. Being a probabilistic algorithm, it always gives the correct answer when the input number is prime, but with a probability that can be made arbitrarily small it may return an incorrect answer when the input number is not prime. The number theory behind the algorithm can be understood by everyone who has learned the material covered in this chapter, and is very short and easy to grasp. Here is a pseudocode of the algorithm (copied from the Wikipedia article):

```

Input:  $n > 3$ , an odd integer to be tested for primality;
Input:  $k$ , a parameter that determines the accuracy of the test
Output: composite if  $n$  is composite, otherwise probably prime
write  $n - 1$  as  $2^s \cdot d$  with  $d$  odd by factoring powers of 2 from  $n - 1$ 
WitnessLoop: repeat  $k$  times:
    pick a random integer  $a$  in the range  $[2, n - 2]$ 
     $x \leftarrow a^d \bmod n$ 
    if  $x = 1$  or  $x = n - 1$  then do next WitnessLoop
    repeat  $s - 1$  times:
         $x \leftarrow x^2 \bmod n$ 
        if  $x = 1$  then return composite
        if  $x = n - 1$  then do next WitnessLoop
    return composite
return probably prime

```

Until the early 2000s it was not known whether there is a deterministic algorithm for primality testing running in polynomial time (i.e. in the number of steps bounded by a polynomial on the number of digits of the input) and many believed that such an algorithm did not exist. Therefore, it came as somewhat of a surprise when the first such deterministic algorithm for primality testing was discovered in 2002 by Manindra Agrawal, Neeraj Kayal and Nitin Saxena. The result was announced in the paper: PRIMES is in P, *Annals of Mathematics*, 2004, 160 (2): 781–793. It is only 9 pages long and most of it is accessible with the knowledge acquired in this chapter. While this method, being deterministic, is theoretically superior to probabilistic primality testing, as it always gives the correct result in a time that is considered reasonable, the Miller – Rabin test is still practically more efficient.

5.5. Congruence relations

Here we will develop a very convenient and powerful method for expressing and solving problems in number theory.

Definition 5.5.1 Let a, b, n be integers and $n > 0$. If $n \mid a - b$ then a and b are said to be **congruent modulo n** , denoted

$$a \equiv b \pmod{n}.$$

An expression like this is called a **congruence** (modulo n). We also say that b is a **residue of a modulo n** .

Lemma 5.5.2 $a \equiv b \pmod{n}$ iff a and b leave the same remainder when divided by n .

Proof: Let $a = q_a n + r_a$ and $b = q_b n + r_b$ where $0 \leq r_a, r_b < n$. Then $a - b = (q_a - q_b)n + (r_a - r_b)$. Therefore, if $r_a = r_b$ then $n \mid a - b$, so $a \equiv b \pmod{n}$. Conversely, if $a \equiv b \pmod{n}$ then $n \mid a - b$; hence $n \mid r_a - r_b$, but $|r_a - r_b| < n$, and thus $r_a - r_b = 0$, i.e. $r_a = r_b$.

In particular, if $a \equiv b \pmod{n}$ and $0 \leq b < n$ then b is the remainder of the division of a by n , hereafter called the **remainder of a modulo n** , also denoted $a \text{ rem } n$.

As can be seen, using congruences is just another way to speak about divisibility. Thus many properties of divisibility can easily be translated into properties of congruences.

Theorem 5.5.3 (Basic properties of congruences) Let a, b, c, d, k, m, n be integers and $n > 0$. Then:

- (CON1) $a \equiv b \pmod{n}$ iff $a - b \equiv 0 \pmod{n}$.
- (CON2) $a \equiv a \pmod{n}$.
- (CON3) If $a \equiv b \pmod{n}$ then $b \equiv a \pmod{n}$.
- (CON4) If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ then $a \equiv c \pmod{n}$.
- (CON5) If $a \equiv b \pmod{n}$ then $a + c \equiv b + c \pmod{n}$, $a - c \equiv b - c \pmod{n}$ and $ac \equiv bc \pmod{n}$.
- (CON6) If $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$ then $ak + cm \equiv bk + dm \pmod{n}$.
In particular, $a + c \equiv b + d \pmod{n}$ and $a - c \equiv b - d \pmod{n}$.
- (CON7) If $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$ then $ac \equiv bd \pmod{n}$.
- (CON8) If $a \equiv b \pmod{n}$ and $n \geq 0$ then $a^k \equiv b^k \pmod{n}$.
- (CON9) If $a \equiv b \pmod{n}$ and $d \mid n$, $d > 0$ then $a \equiv b \pmod{d}$.
- (CON10) If $a \equiv b \pmod{n}$ and $c > 0$ then $ac \equiv bc \pmod{nc}$.

Proof: All these properties are easy consequences from the definition of a congruence and the basic properties of divisibility. Let us verify just two of them:

- (CON6) Let $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$. Then $n \mid a - b$ and $n \mid c - d$. Therefore $n \mid ak - bk$ and $n \mid cm - dm$ and hence $n \mid (ak - bk) + (cm - dm)$, i.e. $n \mid (ak + cm) - (bk + dm)$, which means that $ak + cm \equiv bk + dm \pmod{n}$.
- (CON7) Again $n \mid a - b$ and $n \mid c - d$ and hence $n \mid (a - b)c$ and $n \mid (c - d)b$. Therefore $n \mid (a - b)c - (c - d)b$, i.e. $n \mid ac - bd$, so $ac \equiv bd \pmod{n}$.

The proof of the remaining properties are left as an exercise.

Here are some examples demonstrating the power of congruences.

Example 5.5.4

Find the remainder 7 000 000 000 rem 11.

Solution

$7 000 000 000 = 7 \times 10^9$. Given that $10 \equiv -1 \pmod{11}$ we apply (CON8) and obtain $10^9 \equiv (-1)^9 = -1 \pmod{11}$; hence $7 \times 10^9 \equiv 7 \times (-1) = -7 \equiv 4 \pmod{11}$. Thus, the answer is 4.

Example 5.5.5

Find the last digit of 1234^{4321} .

Solution

Firstly, notice that the last digit (in decimal notation) of *any* natural number is the remainder of that number modulo 10. Therefore we have to solve for x the congruence

$$1234^{4321} \equiv x \pmod{10}, \quad 0 \leq x < 10.$$

Since $1234 \equiv 4 \pmod{10}$, by (CON8) that is equivalent to

$$4^{4321} \equiv x \pmod{10}.$$

Let us list the first few powers of 4 modulo 10 :

$4^1 \equiv 4 \pmod{10}$, $4^2 = 16 \equiv 6 \pmod{10}$, $4^3 = 64 \equiv 4 \pmod{10}$, $4^4 = 4^3 \times 4 \equiv 4 \times 4 = 6 \pmod{10}$, etc.

We see that the odd powers of 4 leave a remainder 4, while the even powers of 4 leave a remainder 6 modulo 10. Therefore, $4^{4321} \equiv 4 \pmod{10}$.

Example 5.5.6

Find the last two digits of 7^{7^7} .

Solution

The last two digits in decimal notation of *any* natural number form the remainder of that number modulo 100. Therefore we have to solve for x the congruence

$$7^{7^7} \equiv x \pmod{100}, \quad 0 \leq x < 100.$$

The idea is to start with some power of 7 that has a small remainder modulo 100, preferably 1 or -1 , and then to raise that congruence to a power close to 7^{7^7} . We are lucky to find that

$$7^4 = 2401 \equiv 1 \pmod{100}.$$

Now we want to raise this congruence to some power x such that $(7^4)^x = 7^{4x}$ is as close as possible to 7^{7^7} . Then we would get $7^{4x} \equiv 1^x = 1 \pmod{100}$ and, supposing that $7^{7^7} = 4x + y$, then $7^{7^7} = 7^{4x} \times 7^y$, from where

$$7^{7^7} \equiv 7^y \pmod{100}.$$

Thus we can reduce our problem to finding the remainder $y = (7^{7^7} \text{ rem } 4)$, i.e. solving for y the congruence

$$7^{7^7} \equiv y \pmod{4}, \quad 0 \leq y < 4.$$

We have $7 \equiv -1 \pmod{4}$ and hence $7^{7^7} \equiv (-1)^{7^7} = -1 \equiv 3 \pmod{4}$. Therefore

$$7^{7^7} \equiv 7^3 = 343 \equiv 43 \pmod{100}.$$

Thus, the last two digits of 7^{7^7} are 43.

We see that in a way congruences behave like equalities with respect to addition, subtraction and multiplication. With division, however, the matter is more delicate. Instead of the usual cancellation law, the following holds.

Theorem 5.5.7 *For all integers a, x, y and $n > 0$,*

$$ax \equiv ay \pmod{n} \quad \text{if and only if} \quad x \equiv y \left(\pmod{\frac{n}{\gcd(a, n)}} \right).$$

Proof: We may assume that $a \neq 0$ and $x \neq y$; otherwise the statement is trivial. Now, if $ax \equiv ay \pmod{n}$ then $ax - ay = kn$ for some integer $k \neq 0$. Therefore, $\frac{a}{(a, n)}(x - y) = \frac{n}{(a, n)}k$, and thus $\frac{n}{(a, n)} \mid \frac{a}{(a, n)}(x - y)$. However, $\gcd\left(\frac{n}{(a, n)}, \frac{a}{(a, n)}\right) = 1$ by (GCD6). Therefore $\frac{n}{(a, n)} \mid (x - y)$, so $x \equiv y \left(\pmod{\frac{n}{\gcd(a, n)}} \right)$.

Conversely, if $x \equiv y \pmod{\frac{n}{\gcd(a, n)}}$ then $ax \equiv ay \pmod{\frac{an}{\gcd(a, n)}}$ by (CON10) in Theorem 5.5.3. The number $\frac{a}{\gcd(a, n)}$ is an integer and hence $n \mid \frac{an}{\gcd(a, n)}$. Therefore $ax \equiv ay \pmod{n}$ by (CON9).

For instance, $12x \equiv 12y \pmod{30}$ if and only if $x \equiv y \pmod{5}$.

As a consequence of Theorem 5.5.7 we obtain the following frequently used fact.

Corollary 5.5.8 *If $ax \equiv ay \pmod{n}$ and $(a, n) = 1$, then $x \equiv y \pmod{n}$.*

For example, if $45x \equiv 45y \pmod{14}$ then $x \equiv y \pmod{14}$.

Theorem 5.5.9 *For any integers x, y and positive integers n_1, n_2, \dots, n_k :*

$$x \equiv y \pmod{n_i} \text{ for } i = 1, 2, \dots, k \text{ if and only if } x \equiv y \pmod{\text{lcm}(n_1, n_2, \dots, n_k)}.$$

Proof: If $x \equiv y \pmod{n_i}$ for $i = 1, 2, \dots, k$ then $n_i \mid (x - y)$ for $i = 1, 2, \dots, k$. Thus $x - y$ is a common multiple of n_1, n_2, \dots, n_k and hence $\text{lcm}(n_1, n_2, \dots, n_k) \mid x - y$ by Theorem 5.3.6, which implies $x \equiv y \pmod{\text{lcm}(n_1, n_2, \dots, n_k)}$. Conversely, if $x \equiv y \pmod{\text{lcm}(n_1, n_2, \dots, n_k)}$ then $x \equiv y \pmod{n_i}$ for $i = 1, 2, \dots, k$, by (CON9).

5.5.1. Exercises

- ① Prove all properties of congruences listed in Theorem 5.5.3.
- ② Show that:
 - (a) Every decimal number that is a square has one of the following for its units digit: 0, 1, 4, 5, 6, 9.
 - (b) Every decimal number that is a fourth power has one of the following for its units digit: 0, 1, 5, 6.
- ③ Find
 - (a) the possible remainders modulo 8 of any square of an integer.
 - (b) the possible remainders modulo 7 of any cube of an integer.
- ④ Find the last decimal digit of
 - (a) 2^{222} .
 - (b) 3^{333} .
 - (c) 7^{777} .
- ⑤ Prove that the number $3n^2 - 5$, where n is an integer, can never be a perfect square. (Hint: consider the remainder modulo 4.)
- ⑥ Show that if $x \equiv y \pmod{n}$ then $\gcd(x, n) = \gcd(y, n)$. Is the converse true?
- ⑦ Show that any natural number n in decimal representation is divisible by

- (a) 3 if and only if the sum of its digits is divisible by 3.
 (b) 9 if and only if the sum of its digits is divisible by 9.
- ⑧* Prove that if p is prime and a, b are integers, then

$$(a+b)^p \equiv a^p + b^p \pmod{p}.$$

- ⑨* Let A be the sum of the digits of the number 4444^{4444} and B be the sum of the digits of A . Find the sum of the digits of B . (All numbers are decimal.)⁴
- ⑩* N is a sum of the eighth powers of 100 consecutive natural numbers. Find the last digit of N .
- ⑪* Prove that $7 \mid (2222^{5555} + 5555^{2222})$.
- ⑫* Find the last two digits of $2^{999} + 3^{999}$.
- ⑬* Find the remainder modulo 7 of

$$10^{10} + 10^{(10^2)} + \cdots + 10^{(10^{10})}.$$



Julius Wilhelm Richard Dedekind (6.10.1831–12.2.1916)

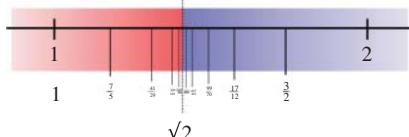
Richard Dedekind was a German mathematician who made important contributions to abstract algebra, number theory and the foundations of the real numbers.

He studied and did his doctoral work under Gauss' supervision in Göttingen and submitted a thesis on the theory of Eulerian integrals, for which he received his doctorate in 1852.

Dedekind made many important contributions to mathematics and to the way that it is presented and studied today. One remarkable piece of work was his redefinition of irrational numbers by means of *Dedekind cuts* on the real line. Another seminal contribution, to abstract algebra, was his introduction of the fundamental notion of an ideal in ring theory.

Dedekind's mathematical heritage consists not only in the concepts he introduced and the results he proved, but also in the new style of precision and rigour in mathematics that has strongly influenced mathematicians ever since.

Here are two different impressions on how Dedekind cuts may look:



⁴ This was Problem 4 from the 17th IMO in Bulgaria, 1975.

5.6. Equivalence classes and residue systems modulo n

5.6.1. Equivalence relations and partitions

The properties (CON2), (CON3) and (CON4) in Theorem 5.5.3 respectively say that the relation “congruence” is:

- *reflexive*: $a \equiv a \pmod{n}$,
- *symmetric*: if $a \equiv b \pmod{n}$ then $b \equiv a \pmod{n}$,
- *transitive*: if $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ then $a \equiv c \pmod{n}$.

Recall from Section 2.5 that a binary relation on a set X which is reflexive, symmetric and transitive is called an *equivalence relation* on X . Most of what follows here is a particular case of the general relationship between equivalence relations and partitions developed in Section 2.5, but we nevertheless present it here to make this subsection self-contained.

The most common example of an equivalence relation is the equality. Another example, as the properties (CON1), (CON2) and (CON3) show, is the “congruence modulo n ” for any positive integer n , which is an equivalence relation in \mathbb{Z} . We will come back to these equivalence relations later. Before that we need some basic knowledge of equivalence relations.

Definition 5.6.1 A *partition* of a set X is a family of nonempty and pairwise disjoint subsets of X , the union of which is X . The sets in a partition are called *clusters* of the partition.

For instance, $\{\{1\}, \{2, 4, 6, 7\}, \{3, 5, 9\}, \{8\}\}$ is a partition of the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$; the sets of even and odd integers form a partition of the set of all integers.

Note (recall from Section 2.5) that every partition P of a set X naturally defines an equivalence relation \sim_P on X as follows: for every $a, b \in X$, $a \sim_P b$ iff a and b belong to the same cluster of P . On the other hand, we will show that every equivalence relation \sim in a set X defines a partition of X .

Definition 5.6.2 Let \sim be an equivalence relation in a set X and $x \in X$. The set

$$[[x]]_{\sim} = \{y \in X \mid x \sim y\}$$

is called the *equivalence class* of x with respect to \sim .

Lemma 5.6.3 Let \sim be an equivalence relation in a set X . Then:

1. If $x, y \in X$ and $y \in [[x]]_{\sim}$ then $[[x]]_{\sim} = [[y]]_{\sim}$.
2. The family $\{[[x]]_{\sim} \mid x \in X\}$ is a partition of X .

Proof:

1. Let $y \in [[x]]_{\sim}$, so $x \sim y$, and hence $y \sim x$ by symmetry. Moreover, for every $z \in X$, if $z \in [[x]]_{\sim}$ then $x \sim z$, and hence $y \sim z$ by transitivity, so $z \in [[y]]_{\sim}$; likewise, if $z \in [[y]]_{\sim}$ then $y \sim z$, and hence $x \sim z$ by transitivity, so $z \in [[x]]_{\sim}$. Thus, $[[x]]_{\sim} = [[y]]_{\sim}$.

2. $[[x]]_\sim$ is nonempty because $x \in [[x]]_\sim$. For the same reason, the union of $\{[[x]]_\sim \mid x \in X\}$ is the whole X . It remains to show that every two equivalence classes $[[x]]_\sim$ and $[[y]]_\sim$ are either disjoint or equal. Indeed, suppose that $z \in [[x]]_\sim \cap [[y]]_\sim$. Then $[[z]]_\sim = [[x]]_\sim$ and $[[z]]_\sim = [[y]]_\sim$; hence $[[x]]_\sim = [[y]]_\sim$.

Thus, equivalence relations and partitions are two sides of the same coin.

5.6.2. Equivalence classes modulo n . Modular arithmetic

We now return to the equivalence relation “congruence modulo n ” in \mathbb{Z} , which we hereafter denote by \equiv_n . The equivalence class of x with respect to \equiv_n will be denoted by $[[x]]_n$ and will be called the **equivalence class of x modulo n** . For instance, $[[3]]_5 = \{\dots, -7, -2, 3, 8, 13, \dots\}$. Then, the set

$$\mathbb{Z}_n = \{[[x]]_n \mid x \in \mathbb{Z}\}$$

is called **the set of integers modulo n** .

By Lemma 5.5.2, $[[x]]_n = [[y]]_n$ iff x and y leave the same remainders modulo n . There are precisely n possible remainders modulo n : $0, 1, \dots, n - 1$. Therefore, \mathbb{Z}_n has precisely n different elements: $[[0]]_n, [[1]]_n, \dots, [[n - 1]]_n$. Hereafter we will denote them by $[0]_n, [1]_n, \dots, [n - 1]_n$, i.e.

$$\mathbb{Z}_n = \{[0]_n, [1]_n, \dots, [n - 1]_n\}.$$

Thus, $[[x]]_n = [a]_n$ iff $a = x \text{ rem } n$. For example, $[[- 7]]_5 = [3]_5$ and $[[42]]_7 = [0]_7$.

Furthermore, we can define the operations of *addition* \oplus_n and *multiplication* \odot_n in \mathbb{Z}_n as follows:

$$[[x]]_n \oplus_n [[y]]_n = [[x + y]]_n$$

and

$$[[x]]_n \odot_n [[y]]_n = [[x \cdot y]]_n.$$

For instance, in \mathbb{Z}_6 : $[2]_6 \oplus_6 [5]_6 = [[2 + 5]]_6 = [[7]]_6 = [1]_6$; $[3]_6 \odot_6 [4]_6 = [[3 \cdot 4]]_6 = [[12]]_6 = [0]_6$.

Hereafter we will omit the index n in \oplus_n and \odot_n whenever clear from the context.

Lemma 5.6.4 *In every \mathbb{Z}_n :*

1. \oplus is commutative and associative.
2. \odot is commutative and associative.
3. \odot is distributive over \oplus , i.e. $[a]_n \odot ([b]_n \oplus [c]_n) = ([a]_n \odot [b]_n) \oplus ([a]_n \odot [c]_n)$.
4. $[a]_n \oplus [0]_n = [a]_n$.
5. $[a]_n \odot [0]_n = [0]_n$.
6. $[a]_n \odot [1]_n = [a]_n$.

Proof: The proofs are left as exercises.

Remark 5 The arithmetic of \mathbb{Z}_n with \oplus and \odot is called **arithmetic modulo n** , or simply **modular arithmetic**.⁵ Lemma 5.6.4 shows that \oplus and \odot satisfy most of the usual properties of addition and multiplication of integers, while $[0]_n$ and $[1]_n$ behave with respect to these operations much like 0 and 1 in \mathbb{Z} . On the other hand, unlike \mathbb{Z} , in \mathbb{Z}_n the product of two nonzero elements can be zero, e.g. $[2]_4 \odot [2]_4 = [0]_4$.

5.6.3. Residue systems

As already discussed, given a positive integer n , the set of all possible remainders modulo n is $\{0, 1, 2, \dots, n - 1\}$, i.e. every integer is congruent modulo n to exactly one number from this set. The sets $\{1, 2, \dots, n\}$, $\{1, 3, 4, \dots, n, n + 2\}$, $\{-1, 0, \dots, n - 2\}$, and infinitely many others, have the same property. This observation leads us to the following definition.

Definition 5.6.5 Given a positive integer n , a set of integers $\{x_1, \dots, x_n\}$ is called a **complete residue system modulo n** if for every integer y there is exactly one x_i such that x_i is a residue of y modulo n , i.e. $y \equiv x_i \pmod{n}$.

The complete residue system modulo n $\{0, 1, 2, \dots, n - 1\}$ is called the **primary residue system modulo n** .

Thus, a complete residue system modulo n contains exactly one representative of every equivalence class modulo n . Of course the numbers x_1, \dots, x_n in a complete residue system need not be consecutive. For instance, $\{-25, 13, 134, -1359, 2\}$ is a complete residue system modulo 5.

Lemma 5.6.6 For any positive integer n :

1. A set of n integers forms a complete residue system modulo n if and only if no two integers in the set are congruent modulo n .
2. If $\{x_1, \dots, x_n\}$ is a complete residue system modulo n then for any integer a , the set $\{x_1 + a, \dots, x_n + a\}$ is also a complete residue system modulo n .
3. If $\{x_1, \dots, x_n\}$ is a complete residue system modulo n then for any integer a , $\{ax_1, \dots, ax_n\}$ is a complete residue system modulo n if and only if $\gcd(a, n) = 1$.

Proof: The first two claims are quite easy and are left as exercises. For the last one, first suppose that $\{ax_1, \dots, ax_n\}$ is not a complete residue system modulo n ; hence $ax_i \equiv ax_j \pmod{n}$ for some $i \neq j$. Then $n \mid a(x_i - x_j)$ and since $\gcd(a, n) = 1$, it follows by property (RP2) in Theorem 5.4.1 that $n \mid (x_i - x_j)$, which contradicts the assumption that $\{x_1, \dots, x_n\}$ is a complete residue system modulo n .

Conversely, let $\gcd(a, n) = d > 1$. Then $d \mid ax_i$ for every $i = 1, \dots, n$; hence none of ax_1, \dots, ax_n has a residue 1 modulo n , so $\{ax_1, \dots, ax_n\}$ is not a complete residue system modulo n .

Definition 5.6.7 Given a positive integer n , a **reduced residue system modulo n** is the subset of any complete residue system modulo n , consisting only of those integers that are relatively prime to n .

⁵ Sometimes also called “clock arithmetic”, for one can think of the clock as presenting the time modulo 12 or 24 hours.

As we will see further, every two reduced residue systems modulo n have the same number of elements.

5.6.4. Multiplicative inverses in \mathbb{Z}_n

Note that for element $[a]_n \in \mathbb{Z}_n$, $[a]_n \oplus [n-a]_n = [n-a]_n \oplus [a]_n = [0]_n$, i.e. $[a]_n$ has an inverse with respect to the addition in \mathbb{Z}_n , called an **additive inverse** of $[a]_n$. The question of inverses with respect to \odot is less trivial.

Definition 5.6.8 The element $[b]_n$ is a **multiplicative inverse** of $[a]_n$ in \mathbb{Z}_n if $[a]_n \odot [b]_n = [1]_n$.

Note that not every element of \mathbb{Z}_n has an inverse with respect to \odot . For instance, $[2]_4$ does not have one. (Why?)

Theorem 5.6.9 $[a]_n$ has a multiplicative inverse in \mathbb{Z}_n iff $\gcd(a, n) = 1$.

Proof: If $[a]_n \odot [b]_n = [1]_n$ then $ab \equiv 1 \pmod{n}$ and hence $ab - 1 = qn$ for some integer q , i.e. $1 = ab + (-q)n$. Therefore $\gcd(a, n) = 1$.

Conversely, if $\gcd(a, n) = 1$ then $1 = au + nv \equiv au \pmod{n}$ for some integers u, v . Then $[[u]]_n$ is the multiplicative inverse of $[a]_n$ in \mathbb{Z}_n .

Corollary 5.6.10 The integer $n > 1$ is prime iff for every a such that $0 < a < n$, the element $[a]_n \in \mathbb{Z}_n$ has a multiplicative inverse.

Proof: If n is prime and $0 < a < n$ then $\gcd(a, n) = 1$; hence $0, a, 2a, \dots, (n-1)a$ is a complete residue system modulo n , by Lemma 5.6.6, and thus, for some ca , $[[ca]]_n = [1]_n$. Then $[c]_n$ is a multiplicative inverse of $[a]_n$.

Conversely, if n is not prime then $n = ab$ for some $a, b \in \{2, \dots, n-1\}$. Then $[b]_n$ cannot have a multiplicative inverse, for if $[b]_n \odot [c]_n = [1]_n$ then $[a]_n \odot ([b]_n \odot [c]_n) = [a]_n \odot [1]_n = [a]_n$, while, on the other hand, $[a]_n \odot ([b]_n \odot [c]_n) = ([a]_n \odot [b]_n) \odot [c]_n = [0]_n \odot [c]_n = [0]_n$ and hence $[a]_n = [0]_n$, which is a contradiction.

5.6.5. Exercises

- ① Using the properties (CON6) and (CON7) show that the operations \oplus_n and \odot_n in \mathbb{Z}_n are correctly defined, i.e. the definitions do not depend on the representatives x, y of the equivalence classes.
- ② Prove the properties listed in Theorem 5.6.4.
- ③ Evaluate:
 - (a) $[[29]]_8$
 - (b) $[[182]]_{13}$
 - (c) $[7]_9 \oplus [8]_9$
 - (d) $[7]_9 \odot [8]_9$

- ④ Compute the addition and multiplication tables of \mathbb{Z}_4 and \mathbb{Z}_5 and use them to solve the following equations:
- $x \oplus [3]_4 = [1]_4$
 - $x \oplus [4]_5 = [3]_5$
 - $x \odot [4]_5 = [3]_5$
 - $x \odot [2]_5 = [0]_5$
 - $x \odot [2]_4 = [0]_4$
- ⑤ For which n is it possible that the product of two nonzero elements in \mathbb{Z}_n is $[0]_n$?
- ⑥ Prove the first two properties of Lemma 5.6.6.
- ⑦* Show that if $m > 2$ then $\{1^2, 2^2, \dots, m^2\}$ is not a complete residue system modulo m .

Babylonian mathematics

The civilizations of ancient Mesopotamia – first the Sumerians (2300–2000 BC) and later the Babylonians (2000–1600 BC) – had advanced systems of mathematics, which would later exert a significant influence on ancient Greek mathematics. A remarkable amount of Sumerian and Babylonian mathematics has been preserved, thanks mainly to the practice of writing on wet clay tablets that would be baked hard in the fire or in the sun. Thousands of these tablets have been found by archeologists. They are written in a *cuneiform script*, which is made by indenting the wet clay with the end of a stylus.

In contrast to our modern base 10 number system, the Babylonians employed a sexagesimal (base 60) positional number system. We have inherited the division of the day into 24 hours, the hour into 60 minutes and the minute into 60 seconds from the Babylonians. When we write $5\text{ h }30'\text{ }24''$ (5 hours, 30 minutes and 24 seconds) we are in fact writing $5 \times 60^0 + 30 \times 60^{-1} + 24 \times 60^{-2} = 5 + 30/60 + 24/3600 = 5 + 1/2 + 1/150$. The great advantage of this system is the fact that 60 has so many divisors. In fact, 60 is, what is known as, a *superior highly composite number*. This facilitates division and working with fractions, e.g. as in the simplification above.

The Babylonians made use of tables of squares to facilitate the computation of products, much in the same way that tables of logarithms were used for this purpose before digital calculators became widely available. This process was based on the following identities, which reduce multiplication to addition, subtraction and looking up squares in a table:

$$ab = \frac{(a+b)^2 - (a-b)^2}{4} \quad \text{or} \quad ab = \frac{(a+b)^2 - a^2 - b^2}{2}.$$

The Babylonians seem to have been aware of Pythagoras's theorem and had formulae for calculating the areas and volumes of many common geometric shapes and solids. They calculated the circumference of a circle as three times the diameter, which implies that they were using 3 as an approximation of π . Astronomical phenomena, like eclipses, the solstices and equinoxes, played an extremely important role in their culture and a lot of their mathematics was developed for the sake of astronomical calculations.

5.7. Linear Diophantine equations and linear congruences

5.7.1. Linear Diophantine equations

Definition 5.7.1 Algebraic equations with integer coefficients, for which only integral solutions are considered, are called **Diophantine equations**.

In particular, an equation of the type

$$ax + by = c, \quad (5.1)$$

where a, b, c are integers such that neither a nor b are 0, and for which only integral solutions are considered, is called a **linear Diophantine equation of two unknowns**.

Theorem 5.7.2 A necessary and sufficient condition for the equation (5.1) to have an integral solution for x and y is that $\gcd(a, b) \mid c$.

Proof: The condition is necessary because $\gcd(a, b) \mid ax + by$ for any integers x and y . Now, suppose $\gcd(a, b) \mid c$. Recall (Theorem 5.2.5) that there are integers u and v such that $\gcd(a, b) = ua + vb$. We will show that the pair

$$x_0 = \frac{uc}{\gcd(a, b)}, \quad y_0 = \frac{vc}{\gcd(a, b)}$$

is a solution of the Definition 5.7.1 equation for any u, v such that $\gcd(a, b) = ua + vb$. Indeed, x_0 and y_0 are integers and

$$a \frac{uc}{\gcd(a, b)} + b \frac{vc}{\gcd(a, b)} = \left(\frac{au}{\gcd(a, b)} + \frac{bv}{\gcd(a, b)} \right) c = \frac{au + bv}{\gcd(a, b)} c = c.$$

Example 5.7.3

1. The equation $21x - 12y = 5$ has no integral solutions because $\gcd(21, 12) = 3$ and $3 \nmid 5$.
2. The equation $21x - 12y = 6$ has integral solutions since $3 \mid 6$. Since $3 = (-1) \times 21 + (-2) \times (-12)$, one solution is

$$x_0 = \frac{(-1) \times 6}{3} = -2, \quad y_0 = \frac{(-2) \times 6}{3} = -4.$$

Theorem 5.7.4 If (x_0, y_0) is a solution of the equation in Definition 5.7.1 then all other integral solution of that equation can be obtained as

$$x = x_0 + \frac{b}{\gcd(a, b)} t, \quad y = y_0 - \frac{a}{\gcd(a, b)} t,$$

where t is an arbitrary integer parameter.

Proof: If $ax_0 + by_0 = c$ and $ax + by = c$, then $(ax + by) - (ax_0 + by_0) = 0$, i.e. $a(x - x_0) + b(y - y_0) = 0$, so $a(x - x_0) = b(y_0 - y)$ and hence $\frac{a}{\gcd(a, b)}(x - x_0) = \frac{b}{\gcd(a, b)}(y_0 - y)$. Then $\frac{a}{\gcd(a, b)} \mid \frac{b}{\gcd(a, b)}(y_0 - y)$

y), but $\gcd\left(\frac{a}{\gcd(a,b)}, \frac{b}{\gcd(a,b)}\right) = 1$ and hence $\frac{a}{\gcd(a,b)} \mid y_0 - y$. Let $y_0 - y = \frac{a}{\gcd(a,b)}t$ for some integer t . Then $y = y_0 - \frac{a}{\gcd(a,b)}t$. Likewise $x = x_0 + \frac{b}{\gcd(a,b)}t$.

Conversely, if $x = x_0 + \frac{b}{\gcd(a,b)}t$, $y = y_0 - \frac{a}{\gcd(a,b)}t$. Then:

$$ax + by = a\left(x_0 + \frac{b}{\gcd(a,b)}t\right) + b\left(y_0 - \frac{a}{\gcd(a,b)}t\right) = ax_0 + \frac{ab}{\gcd(a,b)}t + by_0 - \frac{ab}{\gcd(a,b)}t = ax_0 + by_0 = c.$$

Example 5.7.5

The general solution of the equation $21x - 12y = 6$ is given by

$$\begin{aligned} x &= -2 + \frac{-12}{3}t = -2 - 4t, \\ y &= -4 - \frac{21}{3}t = -4 - 7t. \end{aligned}$$

For instance, when $t = 1$ we obtain the solution $(-6, -11)$; when $t = -1$ we obtain $(2, 3)$, etc.

5.7.2. Linear congruences

Definition 5.7.6 A linear congruence is a congruence of the type

$$ax \equiv c \pmod{n},$$

where a, c, n are integers and $n > 0$.

We are interested in the question of whether this congruence has solutions for x and how to find them.

First, let us observe that if the congruence $ax \equiv c \pmod{n}$ has one solution x_0 , then it has infinitely many solutions: $x_0 + kn$, for any integer k . Moreover, note the following theorem.

Theorem 5.7.7 Consider the congruence

$$ax \equiv c \pmod{n}. \quad (5.2)$$

1. This equation has a solution if and only if $\gcd(a, n) \mid c$. Then, one solution is $x_0 = \frac{uc}{\gcd(a,n)}$, where u is such that $\gcd(a, n) = ua + vn$ for some integer v .
2. If x_0 is a solution of that equation then an integer x is a solution if and only if it satisfies the congruence

$$ax \equiv ax_0 \pmod{n},$$

and hence, by Theorem 5.5.7, the congruence

$$x \equiv x_0 \pmod{\frac{n}{(a,n)}}.$$

3. All solutions of the equation are given by the formula

$$x = x_0 + \frac{n}{(a,n)}k,$$

where x_0 is any solution and k is an integer.

Proof:

1. Note that $ax \equiv c \pmod{n}$ if and only if $n | ax - c$ iff $ax - c = ny$ for some integer y , if and only if the equation $ax - ny = c$ has an integer solution iff $\gcd(a, n) | c$, by Theorem 5.7.2.
2. Exercise.
3. Follows from Theorem 5.7.4.

For instance, the congruence $6x \equiv 2 \pmod{8}$ has a solution $x = 3$ and hence all solutions are the numbers $3 + 4k$, for $k \in \mathbb{Z}$.

Often a congruence can be simplified by cancelling according to Theorem 5.5.7.

Example 5.7.8

Solve the congruence

$$9965x \equiv 19955 \pmod{4950}.$$

Solution

First we reduce the congruence by dividing by 5, using Theorem 5.5.7, to

$$1993x \equiv 3991 \pmod{990}.$$

Let us compute $\gcd(1993, 990)$:

$$1993 = 2 \times 990 + 13.$$

$$990 = 76 \times 13 + 2.$$

$$13 = 6 \times 2 + 1.$$

Thus, $\gcd(1993, 990) = 1$, so the congruence has solutions.

Now we look for integers u and v such that $1 = 1993u + 990v$:

$$\begin{aligned} 1 &= 1 \times 13 - 6 \times 2 = 1 \times 13 - 6 \times (990 - 76 \times 13) = 457 \times 13 - 6 \times 990 = 457(1993 - 2 \times 990) - 6 \times 990 = 457 \times 1993 - 920 \times 990. \end{aligned}$$

Therefore, one solution is $x_0 = 457 \cdot 3991 = 1\,823\,887$.

We can obtain a smaller solution by taking the remainder of the division of x_0 by 990: $1\,823\,887 = 1842 \times 990 + 307$.

Then, the general solution is $x = 307 + 990k$.

5.7.3. Exercises

1 Solve the Diophantine equations:

- (a) $81x - 24y = 18$
 - (b) $28x + 91y = 146$
 - (c) $429x + 154y = 121$
- 2** Solve the congruences:
- (a) $27x \equiv 12 \pmod{15}$
 - (b) $25x \equiv 5 \pmod{16}$
 - (c) $166x \equiv 18 \pmod{38}$
 - (d) $84x \equiv 24 \pmod{35}$
 - (e) $28x \equiv 42 \pmod{49}$
 - (f) $1001x \equiv 91 \pmod{104}$
 - (g) $3700x \equiv 11 \pmod{111}$

- 3*** Every two of n arithmetic progressions have a common term. Show that all progressions have a common term.
- 4*** Show that for every natural number n , in every arithmetic progression of natural numbers there are n consecutive terms that are composite numbers.
- 5*** Prove that for every positive integer k there exists a prime p such that each of the numbers $p - 1$, $p + 1$ and $p + 2$ has at least k different prime divisors.

⁶ Translation taken from <http://mathworld.wolfram.com/DiophantussRiddle.html>.



Diophantus of Alexandria (c. AD 200–c. AD 285)

Diophantus was an Alexandrian Greek mathematician, sometimes called “the father of algebra”, for his fundamental contributions to arithmetic and algebra, especially on finding integer solutions to algebraic equations with integer coefficients, now called *Diophantine equations*.

Diophantus’ magnum opus was the collection of 13 books called *Arithmetica*, of which only six have survived. They contain a collection of 130 problems requiring integer solutions of Diophantine equations. The methods for solving such equations, which may have many solutions, are now called Diophantine analysis. Diophantus’ *Arithmetica* greatly stimulated the advances in number theory for centuries after him and strongly influenced many generations of mathematicians, including the French lawyer and amateur mathematician Pierre de Fermat (see the end of Section 5.10). Fermat famously claimed that certain Diophantine equations have no solutions – a claim now called *Fermat’s Last Theorem* – and just noted in the margin of his copy of *Arithmetica* that he had found a “truly marvellous proof of this proposition”. No one could find such a proof for over 300 years, until 1994.

Diophantus was the first Greek mathematician who recognized fractions as numbers. He also made significant contributions to the mathematical notation.

Here is a famous riddle⁶, a poem from *Anthologia Palatina*, a collection of old Greek poems and epigrams dating from the 5th to 6th centuries, which encodes a mathematical problem, the solution of which will give you how many years Diophantus lived (even though the precise years of his birth and death are not known):

Here lies Diophantus, the wonder behold. Through art algebraic, the stone tells how old: ‘God gave him his boyhood one-sixth of his life, One twelfth more as youth while whiskers grew rife; And then yet one-seventh ere marriage begun; In five years there came a bouncing new son. Alas, the dear child of master and sage after attaining half the measure of his father’s life chill fate took him. After consoling his fate by the science of numbers for four years, he ended his life.’

How long did Diophantus live?

5.8. Chinese remainder theorem

Having solved the problem for *one* linear congruence, we now proceed to *systems* of linear congruences. What follows might appear to be a piece of modern, though elementary, mathematics, but in fact the method for solving such systems was essentially known in China some 2000 years back! The following problem was posed by Sun Tsu Suan-Ching in the 4th century AD: *There are certain things whose number is unknown. Repeatedly divided by 3, the remainder is 2; by 5 the remainder is 3; and by 7 the remainder is 2. What will be the number?*

The method for solving problems of this kind is given by the following theorem.

Theorem 5.8.1 (Chinese remainder theorem) *Let m_1, \dots, m_k be positive integers, relatively prime in pairs, and b_1, \dots, b_k be any integers. Then:*

1. *The system of congruences*

$$\begin{aligned}x &\equiv b_1 \pmod{m_1} \\&\vdots \\x &\equiv b_k \pmod{m_k}\end{aligned}$$

has a solution, which is unique modulo $M = m_1 \cdots m_k$.

2. *The least non-negative solution is*

$$x_0 = (M_1 y_1 + \cdots + M_k y_k) \text{ rem } M$$

where $M_i = M/m_i$ and y_i is a solution of the congruence

$$M_i y_i \equiv b_i \pmod{m_i}$$

for $i = 1, 2, \dots, k$.

3. Moreover, every solution can be obtained as $x_0 + nM$ for some integer n .

Proof: Since $x_0 \equiv M_1 y_1 + \cdots + M_k y_k \pmod{M}$, it follows by (CON9) that $x_0 \equiv M_1 y_1 + \cdots + M_k y_k \pmod{m_i}$ for every $i = 1, 2, \dots, k$. However, $M_i y_i \equiv b_i \pmod{m_i}$ and $M_j y_j \equiv 0 \pmod{m_i}$ for every $j \neq i$ because $m_i \mid M_j$. Therefore, $M_1 y_1 + \cdots + M_k y_k \equiv b_i \pmod{m_i}$ and hence $x_0 \equiv b_i \pmod{m_i}$ for every $i = 1, 2, \dots, k$.

Furthermore, $x_0 + nM$ for every integer n is also a solution.

Finally, if x_1 and x_2 are solutions, then $x_1 - x_2 \equiv 0 \pmod{m_j}$ for $j = 1, \dots, k$. Hence $x_1 - x_2 \equiv 0 \pmod{M}$, i.e. $x_1 \equiv x_2 \pmod{M}$, by Theorem 5.3.6, because $\text{lcm}(m_1, \dots, m_k) = M$.

The Chinese remainder theorem 5.8.1 gives a direct method of finding a solution of the system of congruences.

Example 5.8.2

Find a positive integer x that leaves remainders 1, 2, 3, 4 when divided by 2, 3, 5, 7, respectively.

Solution

The problem boils down to solving the system

$$\begin{aligned}x &\equiv 1 \pmod{2} \\x &\equiv 2 \pmod{3} \\x &\equiv 3 \pmod{5} \\x &\equiv 4 \pmod{7}\end{aligned}$$

We apply the Chinese remainder theorem:

$$M = 2 \times 3 \times 5 \times 7 = 210, M_1 = 105, M_2 = 70, M_3 = 42, M_4 = 30.$$

We look for integers y_1, y_2, y_3 and y_4 such that

$$105y_1 \equiv 1 \pmod{2},$$

$$70y_2 \equiv 2 \pmod{3},$$

$$42y_3 \equiv 3 \pmod{5},$$

$$30y_4 \equiv 4 \pmod{7}.$$

We find $y_1 \equiv 1 \pmod{2}$, $y_2 \equiv 2 \pmod{3}$, $y_3 \equiv 4 \pmod{5}$ and $y_4 \equiv 2 \pmod{7}$. Possible values are $y_1 = 1$, $y_2 = 2$, $y_3 = 4$ and $y_4 = 2$, from where $x_0 = (105 \times 1 + 70 \times 2 + 42 \times 4 + 30 \times 2) \text{ rem } 210 = 473 \text{ rem } 210 = 53$. Thus, the least positive solution is $x_0 = 53$. All other solutions are obtained by the formula $x = 53 + 210n$.

5.8.1. Exercises

- ① Solve the following systems of congruences:

- (a) $x \equiv 2 \pmod{4}$, $x \equiv 3 \pmod{9}$.
- (b) $x \equiv 13 \pmod{14}$, $x \equiv 9 \pmod{15}$.
- (c) $x \equiv 3 \pmod{4}$, $x \equiv 5 \pmod{9}$, $x \equiv 10 \pmod{35}$.
- (d) $x \equiv 2 \pmod{3}$, $x \equiv 3 \pmod{7}$, $x \equiv 0 \pmod{10}$.
- (e) $x \equiv 4 \pmod{5}$, $x \equiv 3 \pmod{4}$, $x \equiv 2 \pmod{7}$, $x \equiv 6 \pmod{9}$.

- ② The following story comes from *Brahma-Sphuta-Siddhanta* (Brahma's Correct System) written by Brahmagupta (born 598 AD). An old woman goes to market and a horse steps on her basket and crashes the eggs. The rider offers to pay for the damages and asks her how many eggs she had brought. She does not remember the exact number, but when she had taken them out two at a time, there was one egg left. The same happened when she picked them out three, four, five and six at a time, but when she took them out seven at a time they came out even. What is the smallest number of eggs she could have had?

- ③* Find a necessary and sufficient condition for the system of congruences

$$x \equiv b_1 \pmod{m_1}$$

...

$$x \equiv b_k \pmod{m_k}$$

to have a solution.

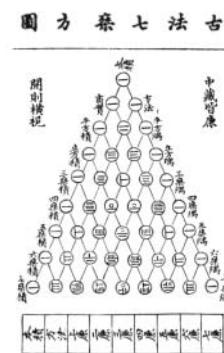
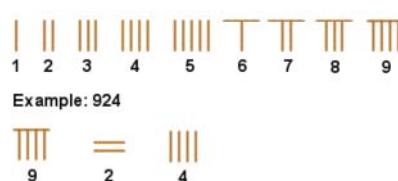
Ancient Chinese mathematics

The Chinese remainder theorem is only one impressive testimony for the sophistication of the mathematics of ancient China. Its development started around the 11th century BC. Unlike the formal and axiomatic style of ancient Greek mathematics, the Chinese mathematics was mainly oriented towards solving practical problems, related to trade, taxation, payments and accounting, engineering, astronomy and agriculture. The positional (value place) decimal system was known in China at least as far back as the 2nd millennium BC, over a thousand years before it was adopted in Europe. The digits 1 to 9 were represented with especially arranged small bamboo rods, positioned in columns representing units, tens, hundreds, thousands, etc. Much of the mathematical knowledge of ancient China was collected in a textbook called *Jiuzhang Suanshu*, or *Nine Chapters on the Mathematical Art*, gradually written by many authors over a period of time starting around 200 BC. It contained many geometric facts and constructions, including Pythagoras' theorem, methods for solving some quadratic and cubic equations, for computing approximate values of π , as well as a version of the method of Gaussian elimination for solving systems of linear equations with several unknowns.

The Chinese mathematicians were fascinated with mathematical patterns, such as *magic squares* of numbers where each row, column and diagonal added up to the same sum. They were regarded as having religious and spiritual meaning.

The Golden Age of Chinese mathematics was around the 13th century AD, when dozens of strong mathematical schools flourished across China. Perhaps the most famous Chinese mathematician of that time was Qin Jiushao, who wrote a book on applied maths, *Shiushu Jiuzhang* (*Mathematical Treatise in Nine Sections*) where, among other things, he presented in detail the Chinese Remainder Theorem and developed an early version of the method of repeated approximations, only discovered in Europe by Isaac Newton in the 17th century, for computing approximate solutions of algebraic equations of degree as high as 10.

Below on the left is an illustration of the ancient Chinese positional decimal system and on the right a Chinese version of Pascal's triangle, known by Shen Kuo some 300 years before Pascal was born.



5.9. Euler's function. Theorems of Euler and Fermat

Now we will introduce a function defined on the set of positive integers which plays a very important role in number theory. It is called **Euler's function**, denoted by ϕ and defined for all positive integers as follows.

Definition 5.9.1 $\phi(n)$ is the number of positive integers less than or equal to n and relatively prime to n .

For instance, $\phi(1) = 1$, $\phi(11) = 10$, $\phi(12) = 4$.

Obviously for large values of n the definition is not convenient for practical computation of $\phi(n)$ and we need a more efficient method. In order to find such a method we will make several observations about Euler's function.

Theorem 5.9.2

1. If n is prime then $\phi(n) = n - 1$.
2. If $n = p^k$, where p is prime, then $\phi(n) = p^k - p^{k-1}$.
3. $\phi(n)$ has the following multiplicative property: for every two relatively prime natural numbers m and n ,

$$\phi(mn) = \phi(m)\phi(n).$$

Proof:

1. All $n - 1$ positive integers less than n are relatively prime to n .
2. The numbers between 1 and p^k that are not relatively prime to p^k are exactly the p^{k-1} multiples of p : $1 \times p, 2 \times p, \dots, p^{k-1} \times p$ and the remaining $p^k - p^{k-1}$ numbers are relatively prime to p^k .
3. In order to verify that property let us arrange the integers from 1 to mn in a rectangular table with m rows and n columns as follows:

1	2	...	j	...	n
$n + 1$	$n + 2$...	$n + j$...	$2n$
...
$(i - 1)n + 1$	$(i - 1)n + 2$...	$(i - 1)n + j$...	in
...
$(m - 1)n + 1$	$(m - 1)n + 2$...	$(m - 1)n + j$...	mn

We see that the number in the i th row and the j th column of this table is $(i - 1)n + j$. Note that this number is relatively prime to n if and only if j is relatively prime to n , by (GCD4) in Theorem 5.2.6. Thus, the numbers from the table that are relatively prime to n are precisely the elements of those $\phi(n)$ columns that correspond to the numbers j relatively prime to n . Next, $\{0 \times n, 1 \times n, \dots, (m - 1)n\}$ is a complete residue system modulo m by Lemma 5.6.6, since $\gcd(m, n) = 1$. Therefore $\{0 \times n + j, 1 \times n + j, \dots, (m - 1)n + j\}$ is a complete residue system modulo m for every integer j . Thus, there are $\phi(m)$ numbers from these, which are relatively prime to m .

Now, the numbers from the table that are relatively prime to mn are those that are relatively prime both to m and to n , by Theorem 5.4.1. Hence, these are precisely the $\phi(m)$ numbers relatively prime to m from each of the $\phi(n)$ columns with numbers relatively prime to n . Therefore, there are exactly $\phi(m).\phi(n)$ numbers in the table that are relatively prime to mn , i.e. $\phi(mn) = \phi(m)\phi(n)$.

Applying Theorem 5.9.2 several times we can generalize the multiplicative property of $\phi(n)$ as follows.

Corollary 5.9.3 *If n_1, \dots, n_k are relatively prime in pairs then $\phi(n_1, \dots, n_k) = \phi(n_1) \cdots \phi(n_k)$.*

Now we are ready to compute $\phi(n)$ for any natural number n . Let $n = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$ where $p_1^{\alpha_1}, \dots, p_k^{\alpha_k}$ are powers of different primes. Then every two of these powers are relatively prime. Applying the properties of $\phi(n)$ we find:

$$\phi(n) = \phi(p_1^{\alpha_1}) \cdots \phi(p_k^{\alpha_k}) = (p_1^{\alpha_1} - p_1^{\alpha_1-1}) \cdots (p_k^{\alpha_k} - p_k^{\alpha_k-1}).$$

In a slightly more concise form:

$$\phi(n) = n \left(1 - \frac{1}{p_1}\right) \cdots \left(1 - \frac{1}{p_k}\right).$$

Example 5.9.4

1. $\phi(2646) = \phi(2 \times 3^3 \times 7^2) = (2^1 - 2^0)(3^3 - 3^2)(7^2 - 7^1) = 1512.$
2. $\phi(1000) = \phi(2^3 \cdot 5^3) = 1000 \times \frac{1}{2} \times \frac{4}{5} = 400.$

5.9.1. Theorems of Euler and Fermat

Theorem 5.9.5 (Euler's theorem) *For every natural number n and integer a , relatively prime to n , the following congruence holds:*

$$a^{\phi(n)} \equiv 1 \pmod{n},$$

that is

$$n \mid a^{\phi(n)} - 1.$$

Proof: Let $x_1, \dots, x_{\phi(n)}$ be the positive integers less than n and relatively prime to n . Now consider $ax_1, \dots, ax_{\phi(n)}$. These are $\phi(n)$ numbers that are all relatively prime to n , by Theorem 5.4.1, and no two of them are congruent modulo n (why?), so they form a *reduced residue system modulo n*. Therefore, if $y_i = ax_i \text{ rem } n$ for $i = 1, \dots, \phi(n)$, then $y_1, \dots, y_{\phi(n)}$ are $\phi(n)$ different natural

numbers less than n and relatively prime to n , so they must be exactly the numbers $x_1, \dots, x_{\phi(n)}$, possibly in another order. Therefore, their products are equal:

$$x_1 \cdots x_{\phi(n)} = y_1 \cdots y_{\phi(n)}.$$

Now

$$ax_i \equiv y_i \pmod{n}$$

for $i = 1, \dots, \phi(n)$, hence

$$(ax_1) \cdots (ax_{\phi(n)}) \equiv y_1 \cdots y_{\phi(n)} \pmod{n}.$$

However,

$$(ax_1) \cdots (ax_{\phi(n)}) = a^{\phi(n)} x_1 \cdots x_{\phi(n)} = a^{\phi(n)} y_1 \cdots y_{\phi(n)}.$$

Thus,

$$a^{\phi(n)} y_1 \cdots y_{\phi(n)} \equiv y_1 \cdots y_{\phi(n)} \pmod{n}.$$

Furthermore, the product $y_1 \cdots y_{\phi(n)}$ is relatively prime to n by Theorem 5.4.1. Therefore, by Corollary 5.5.8, we may cancel the last congruence by $y_1 \cdots y_{\phi(n)}$ and thus we obtain

$$a^{\phi(n)} \equiv 1 \pmod{n}.$$

Example 5.9.6

Since $\phi(1000) = 400$ and $\gcd(2003, 1000) = 1$, it follows that

$$2003^{400} \equiv 1 \pmod{1000},$$

i.e. the last three digits of the number 2003^{400} are 001.

Example 5.9.7

Find the last two digits of $77^{77^{77}}$.

Solution

We can approach this problem similarly to the one about 7^{7^7} in Example 5.5.6, but it could take a lot of time until we get an appropriate “starting point”. Euler’s theorem comes to help here: $\phi(100) = 40$, hence $77^{40} \equiv 1 \pmod{100}$, etc. The rest is left as an exercise.

The particular case of Euler’s theorem when n is prime is known as *Fermat’s little theorem*⁷:

Theorem 5.9.8 *For every prime number p and an integer a not divisible by p the following congruence holds:*

$$a^{p-1} \equiv 1 \pmod{p},$$

that is $p \mid a^{p-1} - 1$.

⁷ This to distinguish it from the famous Fermat’s last theorem.

5.9.2. Exercises

- ① Evaluate $\phi(n)$ for $n = 1, 2, \dots, 16$.
- ② Find a complete residue system modulo 7 consisting of multiples of 3.
- ③ Calculate $13^{242} \pmod{56}$ without using a calculator (or computer).
- ④ Using Fermat's little theorem, prove that $a^p \equiv a \pmod{p}$ for every prime number p and integer a .
- ⑤ Show that $42 \mid n^7 - n$ for every integer n .
- ⑥ Show that if $\gcd(n, 10) = 1$ then $10 \mid n^{12} - 1$.
- ⑦ Show that if $\gcd(n, 30) = 1$ then $240 \mid n^4 - 1$.
- ⑧* Find all natural numbers n such that $\phi(n) = 24$.
- ⑨* Find all natural numbers n such that $\phi(n) \mid n$.
- ⑩* If $d \mid n$ and $0 < d < n$ prove that $n - \phi(n) > d - \phi(d)$.
- ⑪* For every natural number $n > 1$ show that

$$\sum m = \frac{n\phi(n)}{2},$$

where the sum ranges over all natural numbers m less than n and relatively prime to n .

- ⑫* Show that $56\ 786\ 730 \mid mn(m^{60} - n^{60})$ for any integers m and n .

- ⑬* Prove the congruences:

- (a) $2^{11 \cdot 31} \equiv 2 \pmod{11 \cdot 31}$.
- (b) $2^{19 \cdot 73} \equiv 2 \pmod{19 \cdot 73}$.

¹⁴ 8

- (a) Find all positive integers n for which $7 \nmid 2^n - 1$.
- (b) Show that for every positive integer n , $7 \mid 2^n + 1$.

- ⑯* Let m and n be natural numbers such that $n > m \geq 1$. The last three digits in the decimal expansion of the number 1978^m coincide with the last three digits in the decimal expansion of the number 1978^n . Find m and n such that the sum $m + n$ is minimal.⁹

- ⑰* Prove that for any integer a and natural number m :

$$a^m \equiv a^{m-\phi(m)} \pmod{m}.$$

(This is a generalization of Euler's theorem.)

⁸ This was Problem 1 at the 6th IMO in Moscow, 1964.

⁹ This was Problem 1 at the 20th IMO in Romania, 1978.



Pierre de Fermat (17.8.1601–12.1.1665)

Pierre de Fermat was a French lawyer at the Parliament of Toulouse, who pursued research in mathematics in his spare time, mostly for intellectual pleasure. He made contributions to several fields of mathematics, including number theory, analytic geometry, probability and optics, which earned him the title *the prince of amateurs*.

Independently of Descartes, Fermat discovered the basic idea of analytic geometry and developed methods for finding tangents to curves and their points of local maximum and minimum, which essentially made him an inventor of the differential calculus. However, Fermat's main mathematical contributions were in number theory, and include his "little" theorem (Theorem 5.9.8).

Strangely, Fermat published only one mathematical paper in his life, anonymously as an appendix to someone else's book. Most of his results were only found after his death on loose sheets of paper or in the margins of books that he had read and annotated, mostly without any proofs, though for most of his claims it was known that he had proofs. However, Fermat's immortality was sealed by one such claim, that he wrote in the margin of his copy of Diophantus' *Arithmetica* (see Section 5.7): he claimed that he had a proof that the equation $a^n + b^n = c^n$, for any natural number $n > 2$, has no integer solutions with a, b, c all nonzero. He wrote there "I have discovered a truly marvellous proof of this proposition, which this margin is too small to contain". This claim was only discovered after his death and the statement became known as *Fermat's last theorem*. Although the claim is so easy to state, and Fermat did have proofs for the cases $n = 3$ and $n = 4$, the proof in the general case had alluded mathematicians for over 350 years. Many famous and unknown mathematicians attempted to prove it, especially after some substantial monetary prizes were offered for it, but with no success. The proof was only completed in 1994 by the British mathematician Andrew Wiles, who devoted about 30 years of his life to this work. Wiles' proof, in a highly condensed form, was over 80 pages long, involving very complex mathematics and containing many brilliant new ideas in advanced fields like elliptic curves and the theory of ideals. The mathematics used in the proof did not exist in the time of Fermat, so it is hard to believe that he really had a correct proof, though this cannot be known for sure.

Here is a photo of Andrew Wiles after concluding his historical lecture in Cambridge in 1993 where he presented the proof of a result implying as a corollary Fermat's last theorem, and thus bringing this tantalizing story to an end (after completing a gap in the proof together with his student Richard Taylor in 1994).



5.10. Wilson's theorem. Order of an integer

5.10.1. Wilson's theorem

The following theorem gives an interesting, albeit mostly theoretical, criterion for primality of a natural number.

Theorem 5.10.1 (Wilson's theorem) *For every positive integer $n > 1$, n is a prime if and only if*

$$(n-1)! + 1 \equiv 0 \pmod{n}.$$

Proof: For $n = 2, 3, 4$ the statement is verified by a direct inspection. Now, suppose $n > 4$.

If $(n-1)! + 1 \equiv 0 \pmod{n}$ then n must be prime; otherwise n has a prime divisor $p < n$. Then $p \mid (n-1)!$ and hence $p \mid 1$ – a contradiction. Conversely, let n be prime. For every k between 2 and $n-2$, $\gcd(k, n) = 1$; hence $\{k \times 0, k \times 1, \dots, k \times (n-1)\}$ is a complete residue system modulo n . Therefore, there is exactly one m between 0 and $n-1$ such that $k \cdot m \equiv 1 \pmod{n}$. This m cannot be 0, 1 or $n-1$, otherwise $k \cdot m$ would have been congruent respectively to 0, k and $n-k \neq 1$ modulo n . Moreover, if k_1 and k_2 are two different numbers between 2 and $n-2$ and $1 \leq m \leq n-1$ then $mk_1 \not\equiv mk_2 \pmod{n}$; otherwise $n \mid m(k_1 - k_2)$, which is impossible since $\gcd(n, m) = 1$ and $\gcd(n, k_1 - k_2) = 1$. Thus, the numbers $2, 3, \dots, n-2$ are grouped into pairs (k, m) such that $km \equiv 1 \pmod{n}$. Then $2 \cdot 3 \cdots (n-2) \equiv 1 \pmod{n}$ and hence $(n-1)! = 1 \cdot 2 \cdot 3 \cdots (n-2)(n-1) \equiv n-1 \equiv -1 \pmod{n}$.

5.10.2. Order of an integer

Note that, for any positive integer n and an integer a such that $\gcd(n, a) = 1$, the set S of all positive integers k for which $a^k \equiv 1 \pmod{n}$ is nonempty, since – by Euler's theorem – it contains $\phi(n)$. Then, by the well-ordering property of natural numbers, there is a least natural number in S .

Theorem 5.10.2 *Let n be a positive integer and $\gcd(n, a) = 1$. The least positive integer k such that $a^k \equiv 1 \pmod{n}$ is called the order of a modulo n , denoted by $\text{ord}_n a$.*

Theorem 5.10.3 (Properties of the order of an integer) *Let n be a positive integer, $\gcd(n, a) = 1$, and $k = \text{ord}_n a$. Then:*

1. Every positive integer m for which $a^m \equiv 1 \pmod{n}$ is a multiple of k .
2. In particular, $k \mid \phi(n)$.
3. For any integers u, v : $a^u \equiv a^v \pmod{n}$ iff $u \equiv v \pmod{k}$.
4. No two of the integers a, a^2, \dots, a^k are congruent modulo n .
5. For any positive integer m , $\text{ord}_n a^m = \frac{k}{\gcd(k, m)}$.

Proof:

1. Let m be any *positive* integer such that $a^m \equiv 1 \pmod{n}$. By dividing m by k with a remainder we get

$$m = kq + r, \quad 0 \leq r < k.$$

Then

$$1 \equiv a^{kq+r} \equiv a^{kq} \times a^r \equiv (a^k)^q \times a^r \equiv 1^q \times a^r \equiv a^r \pmod{n}.$$

Thus, r satisfies $a^r \equiv 1 \pmod{n}$ and is less than k and hence $r = 0$.

2. Follows immediately from 1 and Euler's theorem.
3. Let $u \geq v$. Since $\gcd(n, a) = 1$, $a^u \equiv a^v \pmod{n}$ iff $a^{u-v} \equiv 1 \pmod{n}$ iff $k \mid u - v$.
4. Follows immediately from 3.
5. Left as a (challenging) exercise.

5.10.3. Exercises

- ❶ If $n > 4$, show that n is composite if and only if

$$(n-1)! \equiv 0 \pmod{n}.$$

- ❷ Prove that $343 \mid 2^{147} - 1$.

- ❸* Prove Claim 5 of Theorem 5.10.3.

- ❹* If p is a prime and $h, k \geq 0$ are integers such that $h+k = p-1$, show that $h!k! + (-1)^h \equiv 1 \pmod{p}$.

- ❺* If a and n are natural numbers and p is a prime such that $\gcd(a, p) = 1$ and $\gcd(n, p-1) = 1$, show that there is exactly one integer x between 1 and p such that $x^n \equiv a \pmod{p}$.

- ❻* Let a and $n > 1$ be integers such that $a^{n-1} \equiv 1 \pmod{n}$ but $a^x \not\equiv 1 \pmod{n}$ for every proper divisor x of $n-1$. Show that n is a prime.

Ancient and Medieval Indian and Arabic mathematics

The numeral system used today is known as the *Hindu–Arabic numeral system*. It started evolving during the first centuries AD in India and was gradually transmitted to the West through trade and via Arabic mathematics.

The oldest preserved mathematical records from India are the *Sulba Sutras*, dated between the 8th century BC and the 2nd century AD. They are appendices to religious texts and give simple rules for constructing altars of various shapes, thus including some geometry and trigonometry. The Sulba Sutras also give methods for constructing a circle with approximately the same area as a given square, using different approximations of the value of π , compute $\sqrt{2}$ to several decimal places, list many Pythagorean triples (triples of integer sides of right triangles) and state the Pythagorean theorem. Much of this knowledge probably came from Babylon.

In the 5th century AD, the Indian mathematician–astronomer Aryabhata wrote the *Aryabhatiya*, where the decimal place-value system first appears in Indian mathematics and later made its way to the West. The use of the the Hindu–Arabic numeral system, and in particular the number 0, both a placeholder and decimal digit, were first explained in the 7th century by another famous Indian mathematician–astronomer Brahmagupta, in his main work *Brahma-sphuta-siddhanta* (*Correctly Established Doctrine of Brahma*). In that book, Brahmagupta explained the arithmetic operations on negative integers and on fractions, assuming that they were well known on positive integers. Brahmagupta also gave rules for computing the solution of the linear equation of one unknown $ax + b = cx + d$ and of the quadratic equation $ax^2 + bx = c$. Moreover, he also showed how to solve systems of simultaneous indeterminate equations by elimination and presented methods for solving some Diophantine equations, such as $x^2 + y^2 = z^2$, producing Pythagorean triples and (as it was called later) Pell's equation $kx^2 + 1 = y^2$, by using the Euclidean algorithm. Brahmagupta's book also contains some involved results from geometry and trigonometry and applications to astronomy.

In the 14th century Madhava of Sangamagrama founded the famous *Kerala school of astronomy and mathematics* in Kerala, India, which flourished for over 200 years. Madhava and his followers developed very advanced mathematics, including power series and series expansion for trigonometric functions and essentially used, without making them precise, methods of differentiation and integration, two centuries before that calculus was systematically developed by Newton and Leibniz.

The written history of the Arabic maths goes back to the work of the great mathematician named Muhammad ibn Musa Al-Khwarizmi (c. 780–850 AD). In his fundamental work, *Al-Kitab al-mukhtasar fi hisab al-jabr wa'l-muqabala* (*Compendious Book on Calculation by Completion and Balancing*), he first systematically presented the methods for solving linear and quadratic equations. The word “algebra” derives from *al-jabr* in the title of the book, referring to one of the two operations he used to solve quadratic equations. These methods came from older Indian or Greek sources but he was the first to present them as clear sets of mathematical instructions, giving rise to the modern notion of an *algorithm*. The word algorithm itself is derived from the name Al-Khwarizmi, via the Latin translation of his earlier book, *Algoritmi de numero Indorum*, in which he explained the Hindu–Arabic numeral system. Both books were extremely influential in medieval Europe, where Al-Khwarizmi was regarded as the inventor of algebra. Thanks to him, much of the quite advanced mathematical knowledge of the Ancient and Medieval Indian and Arabic cultures was made widely accessible in Europe.

5.11. Application: public key cryptography

5.11.1. About cryptography

Cryptography deals with methods for exchange of *secret messages*, i.e. messages that can only be written and read by authorized agents. Such exchange involves two processes:

- **encryption** of the text of the original message M , by using an **encryption function (key)** E , and
- **decryption** of the encrypted message $E(M)$, by using a **decryption function (key)** D , so that the decrypted text $D(E(M))$ is identical with M .

Note that every message, being represented by a binary string, can be considered as an integer.

Most of the traditional cryptographic methods use *secret keys* for both encryption and decryption. That creates the *Catch-22 problem* of how to exchange in a secure way the secret keys needed for the encryption and decryption. After all, the purpose of cryptography is precisely to provide such a secure channel for exchange of information, and if it already exists then the encryption becomes unnecessary; otherwise the secret keys themselves can be intercepted and the whole process compromised.

A solution to that problem is provided by the method of **Public key cryptography**, introduced in 1976 by W. Diffie and M. Hellman.¹⁰

5.11.2. The idea of public key cryptography

Public key cryptography uses *a pair of keys for every user U*: a **public key** E_U , i.e. one that is publicly available, and a **private key** D_U , which is only known to the user. These keys essentially are mutually inverse mappings on (e.g. binary) strings of symbols, which serve as encryption and decryption functions, D_U and E_U , i.e.:

$$D_U(E_U(M)) = E_U(D_U(M)) = M$$

for any (sufficiently short) string M .

The exchange of secret messages using public key cryptography between two agents, e.g. Alice and Bob, is organized as follows. Suppose Alice wants to send the secret message M to Bob. For that she uses Bob's public key E_B to encrypt the message and then sends $E_B(M)$ to Bob. Bob then uses his private key D_B to decrypt it and thus obtains $D_B(E_B(M)) = M$. If the message is longer than a predefined length (see further), then it is partitioned into blocks, which are encrypted and sent separately.

Furthermore, Alice can encrypt the message in such a way that Bob can be sure that *only* Alice could have sent it, i.e. the message will bear Alice's **digital signature**. The idea is, again, very simple and elegant: Alice first applies *her* private key D_A to the message M and then Bob's public key E_B to the result, and sends $E_B(D_A(M))$ to Bob. On receipt Bob applies *his* private key D_B to the encrypted message, and then Alice's public key to the result, and thus obtains

$$E_A(D_B(E_B(D_A(M)))) = E_A(D_A(M)) = M.$$

¹⁰ W. Diffie and M. Hellman, New Directions in Cryptography, *IEEE Transactions of Information Theory*, IT-22(6): 644-654, 1976.

Clearly, only Bob (or a person knowing Bob's private key) can obtain the string M , and if it is a meaningful message, then Bob can be practically sure that it could only have been encrypted by Alice (or a person knowing Alice's private key).

Theoretically, the method of public key cryptography can be trivially attacked: after all, the private key D_U of a user U is just the inverse of her public key E_U . Therefore, for the method to be practically secure, the public key E_U must be a **trapdoor function**: one that is easy to compute while its inverse is practically impossible to compute.

5.11.3. The method RSA

The currently most popular and widely used method for public key cryptography is RSA, introduced in 1978 by R. Rivest, A. Shamir and L. Adleman.¹¹

The idea of RSA is based on elementary number theory. The pair of keys (E, D) is determined by a pair of very large prime numbers p and q , and a pair of integers d and e such that

$$de \equiv 1 \pmod{(p-1)(q-1)}.$$

Note that for any chosen d (respectively e) that is relatively prime to $(p-1)(q-1)$, there is an e (respectively d) satisfying the congruence above.

Let $n = pq$. Then the public key E consists in the pair (n, e) and the private key D consists in the pair (n, d) .

Now the encryption function is

$$E(M) = M^e \text{ rem } n$$

and the decryption function is

$$D(P) = P^d \text{ rem } n.$$

Note that $\phi(n) = (p-1)(q-1)$ and hence $de \equiv 1 \pmod{\phi(n)}$.

Therefore, for any integer M , $(M^e)^d = M^{ed} \equiv M \pmod{n}$ by Euler's theorem. Therefore, if $M < n$ then

$$D(E(M)) = (M^e \text{ rem } n)^d \text{ rem } n = M^{ed} \text{ rem } n = M \text{ rem } n = M,$$

and likewise

$$E(D(M)) = (M^d \text{ rem } n)^e \text{ rem } n = M^{de} \text{ rem } n = M \text{ rem } n = M.$$

Thus, E and D are mutually inverse on all $M < n$.

Messages M for which $M \geq n$ are partitioned into blocks: $M = M_1 \cdots M_k$ such that $M_i < n$ for each $i = 1, \dots, k$, and the encrypted message is $E(M_1) \cdots E(M_k)$.

Furthermore, notice that, since (n, e) is the public key, $E(M)$ is easily computable (using a computer, of course). To compute $D(M)$, however, one needs to know d , which is not public knowledge. In order to find d one would have to find p and q , i.e. factorize n , and then solve the congruence

$$ex \equiv 1 \pmod{(p-1)(q-1)}.$$

¹¹ R. Rivest, A. Shamir and L. Adleman, A Method for Obtaining Digital Signatures and Public key Crypto-systems, *Communication of the ACM*, 21(2): 120–126, 1978.

Now the point is that p and q are *very large*, so the factorization of n is prohibitively long and expensive. Thus, E is a trapdoor function.

Remark 6 All currently known methods for factorization take time exponential in the size (length) of the input number n , which, measured in bits, is $\log_2 n$. Thus, the time needed to factorize n is roughly proportional to n . Therefore, for the currently known methods,¹² a 2048-bit long RSA key is safe, because if p and q are each 1024 bits long, our universe will extinguish before the most powerful computers presently existing can factor pq with the currently known methods.

On the other hand, however, factorization only requires nondeterministic polynomial time, i.e. its algorithmic complexity is in the class NP, because once the factors are guessed, the verification is quick. One of the most important, and perhaps the most popular, open problem in computer science nowadays is whether there are algorithms that can solve all NP problems in deterministic polynomial time (See the box at the end of Chapter 3.). It is widely believed that the answer is “NO”, but if such an algorithm is found, then the method RSA will be endangered, if not compromised.

Example 5.11.1

We want to create an RSA crypto-system for the 8-bit ASCII encoding of characters, i.e. the block size is 8 bits. Then we need $n > 2^8 = 256$. Take $p = 13$, $q = 31$, so $n = pq = 403$, and $\phi(n) = 12 \cdot 30 = 360 = 2^3 \cdot 3^2 \cdot 5$.

Choose $e = 77$ ($= 7 \times 11$). Solving the congruence $77d \equiv 1 \pmod{360}$ gives a minimal positive solution $d = 173$. Thus, the public key is $(403, 77)$ and the private key is $(403, 173)$. Then, for example, the word “ONE” represented in ASCII code as $(79, 78, 69)$ is encrypted as $(79^{77} \text{ rem } 403, 78^{77} \text{ rem } 403, 69^{77} \text{ rem } 403) = (300, 39, 49)$.

Remark 7 In virtually all public key systems, the encryption and decryption times are very lengthy compared to other block-oriented algorithms. Therefore in most implementations of public key systems, a temporary, random “session key” of much smaller length than the message is generated for each message and alone encrypted by the public key algorithm. The message is then actually encrypted using a faster private key algorithm with the session key. At the receiver side, the session key is decrypted using the public key algorithms and the recovered “plaintext” key is used to decrypt the message. Thus, the session key approach blurs the distinction between “keys” and “messages” – in the scheme, the message includes the key, and the key itself is treated as an encryptable “message”.

5.11.4. Exercises

- ❶ Use the crypto-system created in Example 5.11.1 above to:
 - (a) Encrypt the word “HELP”, encoded in ASCII as $(72, 69, 76, 80)$.
 - (b) Decrypt the ASCII code encrypted as $(45, 208, 148, 148)$ and determine the encoded word.

¹² At the time of writing this book.

- ② Consider a crypto-system using the public key $(143, 11)$.
 - (a) Find the minimal positive d such that $(143, d)$ is a private key for that crypto-system.
 - (b) Encrypt the ASCII code of the word “STOP”.
 - (c) Decrypt the word with the ASCII code encrypted as the string of integers $(111, 4, 88, 57, 116, 67)$.
- ③ Consider a crypto-system using the public key $(43 \times 61, 143)$.
 - (a) Find the minimal positive d such that $(43 \times 61, d)$ is a private key for that crypto-system.
 - (b) Encrypt the ASCII code of the word “ONE”.
 - (c) Decrypt the string of integers encrypted as $(2, 3, 4, 5)$.
- ④ Consider a crypto-system using the private key $(2701, 125)$.
 - (a) Find the minimal positive e such that $(2701, e)$ is a public key for that crypto-system.
 - (b) Encrypt the ASCII code of the word “SECRET”.
 - (c) Decrypt the string of integers encrypted as $(1, 11, 111, 1111)$.
- ⑤ Knowing that $\phi(8927) = 8736$, compute the private key $(8927, d)$ of the crypto-system using a public key $(8927, 55)$.

History of codes and ciphers

The word *cryptography* comes from Greek and means literally “the science of secret writing”. The use of secret codes and ciphers goes back to ancient times, as the secret exchange of messages was of crucial importance during wars. Accordingly, code breaking and the decryption of encrypted messages became extremely important, which is how *cryptanalysis* emerged.

What is the difference between a *code* and a *cipher*? A code is like a dictionary that translates complete words or phrases into other words or phrases. Reading an encoded text is done with a *codebook*. A *cipher* replaces single letters by other symbols or strings of symbols – letters, numbers, etc. Ciphers are short and can be memorized, so a codebook is not needed. Often the word *code* is used to mean either a code or a cipher.

The ancient Greeks invented a simple cipher that changed letters into numbers, using a 5×5 table, where A is written as 11, B is 12, and so on. In this way, the word CODE is encrypted into 13 34 14 15. A version of this code was still used during the First World War.

The Roman Emperor Julius Caesar also invented his own code, by moving each letter of the alphabet along three places, so that A became D, B became E and so on. Thus, CAESAR in encrypted into FDHVFU.

In 1838 the American, Samuel Morse, created an entirely new way of sending messages quickly over long distances, by encoding every letter into a sequence of long and short electromagnetic signals of “dots” and “dashes” and then encrypting secret messages.

Before the Second World War the Germans designed a special cipher machine called *Enigma* to encode their military messages. The Enigma machine could encode a message in more than 150×10^{18} different ways. It looked like a typewriter in a wooden box. The original message was typed in and the encrypted message came out, by sending electric currents from the keyboard

through a set of rotors and a plugboard to light up the “code” alphabet. The Germans changed the order of the rotors, their starting positions and the plugboard connections every day. Already before the war, Polish cipher experts secretly began to try to crack the code. When Poland was invaded, that task was taken over by the British Government Code and Cypher School at Bletchley Park, where thousands of very clever mathematicians and linguists worked, one of them being Alan Turing (see the box at the end of Section 4.8 in Chapter 4), who was instrumental in breaking the code of Enigma. For that purpose Turing and his collaborators designed and used a special decrypting device called Bombe. It did help a lot that the British naval forces captured a German submarine with an Enigma machine and codebooks on board in 1941, but when the Germans started using more modern, four-rotor Enigma machines in February 1942, the code breaking task became enormous. For that, the British team constructed the huge room-sized supercomputer Colossus, which can now be put into a microprocessor the size of a thumb.

Source: <http://archive.iwm.org.uk/upload/package/10/enigma>.

6

Combinatorics

6.1.	Two basic counting principles	275
6.2.	Combinations. The binomial theorem	284
6.3.	The principle of inclusion–exclusion	293
6.4.	The Pigeonhole Principle	299
6.5.	Generalized permutations, distributions and the multinomial theorem	304
6.6.	Selections and arrangements with repetition; distributions of identical objects . .	312
6.7.	Recurrence relations and their solution	318
6.8.	Generating functions	329
6.9.	Recurrence relations and generating functions	337
6.10.	Application: classical discrete probability	343

Combinatorics is a branch of mathematics that studies finite, discrete objects. In this chapter we will mostly concern ourselves with enumerative combinatorics, which studies ways to count sets of finite discrete objects that satisfy given constraints. This typically involves counting the number of ways in which certain patterns can be formed, and enables us to answer questions like:

- *How many possible different outcomes are there in a lottery draw and how many tickets do I need to buy to stand at least a 1% chance of winning?*
- *In how many different ways can an amount of 5 euros be made up by coins?*
- *How many different DNA strands consisting of a hundred nucleotides are there?*
- *In how many different ways can a telephone call from Sydney to Moscow be routed through the telephone network?*

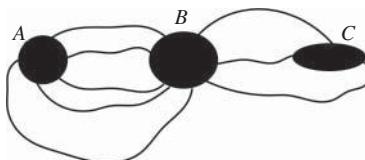
As this list of questions might suggest, combinatorics has a wide variety of applications, both inside and outside mathematics.

Apart from learning some fundamental and very useful counting techniques, the reader who masters the content of this chapter will also gain some valuable experience in problem solving and mathematical modelling: the two skills that mathematicians have that are probably most valuable for society. This chapter does not contain a lot of theoretical material; it rather provides some basic principles and methods and shows how they can be applied to solve a wide variety of problems. The problems are typically “word problems”, i.e. problems formulated in everyday language, rather than in mathematical symbols. Solving a problem like this requires first modelling it in some appropriate mathematical way and then finding a suitable strategy based on combinatorial principles to solve it. For these reasons, solving combinatorial problems involves quite a bit of “art”. This is something that comes with experience, and the exercises therefore take on a central role in this chapter.

6.1. Two basic counting principles

In this section we first review two principles that lie at the heart of combinatorics, namely the **multiplicative** and the **additive counting principles**. These two principles are so simple and intuitive that you might, upon first acquaintance, be tempted to consider them rather trivial. However, as the chapter progresses, it will become clear that almost every counting strategy and technique we develop is reducible to these two principles in some way.

We first introduce the multiplicative counting principle by way of an example. Consider the three towns A , B and C pictured below:



There are five roads from A to B and three roads from B to C . In how many ways can I drive from A to C via B ? Each of the five roads from A to B can be combined with each of the three roads from B to C . Therefore, every such combination corresponds to an ordered pair (i, j) where $1 \leq i \leq 5$ and $1 \leq j \leq 3$. Thus there are $5 \times 3 = 15$ ways to go from A via B to C .

Here is the general counting principle behind this argument.

The multiplicative counting principle (MCP)

Consider a sequence of $m > 1$ events $E_1, E_2, E_3, \dots, E_m$ where E_1 can occur in n_1 different ways; then E_2 can occur in n_2 different ways, e.g. for each $2 \leq k \leq m$, E_k can occur in n_k different ways after the preceding events E_1, E_2, \dots, E_{k-1} have occurred. Then the entire sequence of events $E_1, E_2, E_3, \dots, E_m$ can occur in $n_1 \times n_2 \times n_3 \times \cdots \times n_m$ different ways.

It is important to note that in the formulation of the multiplicative counting principle we use the number n_j of ways in which event E_j can occur given that E_1 to E_{j-1} have already occurred. This is because the number of ways in which E_j can occur might depend on how the events E_1 to E_{j-1} have occurred. If for each event E_i , $1 \leq i \leq m$, the number of ways in which E_i can occur is independent of the ways in which E_1 to E_{j-1} occurred, we call the events $E_1, E_2, E_3, \dots, E_m$ **commuting**. We now have the following easy corollary of the multiplicative principle.

The multiplicative counting principle for commuting events

Given a sequence of m commuting events $E_1, E_2, E_3, \dots, E_m$ such that E_i can occur in n_i ways for $1 \leq i \leq m$, then the entire sequence of events $E_1, E_2, E_3, \dots, E_m$ can occur in $n_1 \times n_2 \times n_3 \times \dots \times n_m$ ways.

Using the term “event” is just one way of formulating the MCP. When applying the principle it sometimes seems more appropriate to think in terms of a *process* that consists of m *steps* or *tasks* $E_1, E_2, E_3, \dots, E_m$, where the step or task E_i can be completed in n_i ways, $1 \leq i \leq m$. Thus, in the example of routes between town A to C via B we want to know the number of different outcomes of the process of putting together such a route. This process consists of two steps, namely firstly choosing a road from A to B and secondly choosing a road from B to C .

Here are a few simple applications of the multiplication principle.

Example 6.1.1

How many different four-digit natural numbers containing only odd digits are there?

Solution

To answer this question we rephrase it as follows: in how many different ways can we choose odd digits to fill each of the four positions in the four-digit integer? This choice process is naturally broken down into four steps corresponding to choosing the digit for the thousands, hundreds, tens and units positions, respectively. Each of these steps can be completed in 5 different ways, as there are 5 odd digits to choose from, namely 1, 3, 5, 7 and 9:

<i>Thousands</i>	<i>Hundreds</i>	<i>Tens</i>	<i>units</i>
5 <i>choices</i>	5 <i>choices</i>	5 <i>choices</i>	5 <i>choices</i>

Moreover, these steps are independent, since choosing a digit for any position does not affect the available choices for any other position. Therefore, by the MCP, the number of different four-digit natural numbers containing only odd digits is $5 \times 5 \times 5 \times 5 = 5^4$.

Example 6.1.2

How many different arrangements of the five letters a, b, c, d, e can be made?

Solution

To make an arrangement we need to fill five positions with letters from a, b, c, d, e , without repeating a letter. Starting from the left, this gives five steps that are *not* independent: the first position can be filled in five ways as any one of the letters can be used. Any one of the *remaining* four letters can be placed second. The third letter is chosen from the remaining three and the fourth letter from the remaining two. There is only one left to fill the fifth place:

Position 1	Position 2	Position 3	Position 4	Position 5
5 choices	4 choices	3 choices	2 choices	1 choice

Therefore, by the multiplication principle, there are $5 \times 4 \times 3 \times 2 \times 1 = 5! = 120$ different arrangements in total.

Example 6.1.3

How many arrangements of n different objects are there?

Solution

This is a direct generalization of the question in Example 6.1.2. Reasoning in the same way, there are n ways to place the first object, $n - 1$ ways to place the second object, $n - 2$ ways to place the third object, ..., three ways to place the $(n - 2)$ nd object, two ways to place the $(n - 1)$ st object and only one way to place the last object. Thus, in total there are $n \times (n - 1) \times (n - 2) \times \dots \times 3 \times 2 \times 1 = n!$ different arrangements.

Example 6.1.4

In how many ways can 10 different books be arranged on a shelf so that two specific books are next to each other?

Solution

To make sure that these two books end up together we could tie them together with string. Thus, considering these two books as one, the question becomes: In how many ways can nine different books be arranged? We know the answer is $9!$. However, the two books can be tied together on the shelf in two ways (depending on which one we put on the left and which on the right) so that there are $2 \times 9!$ possible arrangements with the two always next to each other.

Permutations

The type of counting problems in Examples 6.1.2–6.1.4 is very common and there is a special name for it. A **permutation** of n distinct objects is an arrangement, or ordering, of all n objects. The number of permutations of n distinct objects is denoted by $P(n)$ and is, as we have seen, equal to $n!$.

Example 6.1.5

In how many ways can a first, second and third prize be awarded in a class of 22 students?

Solution

By the multiplicative principle, there are $22 \times 21 \times 20$ ways.

Note that the answer can be written using factorial notation as follows:

$$\begin{aligned} 22 \times 21 \times 20 &= \frac{22 \times 21 \times 20 \times 19 \times \cdots \times 3 \times 2 \times 1}{19 \times \cdots \times 3 \times 2 \times 1} \\ &= \frac{22!}{19!} = \frac{22!}{(22 - 3)!}. \end{aligned}$$

r -Permutations and arrangements

The type of problem in Example 6.1.5 occurs very often and is a generalization of the permutation problems in the previous examples. It leads to the following notion. An **r -permutation of n distinct objects** is an (ordered) arrangement of r of these n objects. The number of r -permutations of n distinct objects is denoted by $P(n, r)$ and, by the MCP, is equal to

$$P(n, r) = n \times (n - 1) \times \cdots \times (n - r + 1) = \frac{n!}{(n - r)!}.$$

In particular, note that $P(n) = P(n, n) = n!$.

We also refer to r -permutations as **arrangements**. Sometimes they are also called **variations**.

Example 6.1.6

How many ways are there to select a committee consisting of a chairperson, a secretary and a treasurer from a group of 12 people?

Solution

The problem reduces to selecting three people out of a possible 12 and assigning them the three portfolios of chairperson, secretary and treasurer. If we fix these three portfolios in a certain order, say chairperson first, secretary second and treasurer third, assigning them to our chosen three candidates is equivalent to ordering the three candidates, with each candidate then being assigned to the portfolio corresponding to his/her position in the ordering. So it becomes clear that we are after ordering 3 objects chosen out of 12. By the discussion above, the number of ways to do this is given by

$$P(12, 3) = \frac{12!}{(12 - 3)!} = \frac{12!}{9!} = 12 \times 11 \times 10 = 1320.$$

Example 6.1.7

Let A and B be sets of cardinality n and m , respectively, for $n, m \in \mathbb{N}$.

1. How many functions are there from A into B ?
2. How many injective (one-to-one) functions are there from A into B ?

Solution

1. We will count the number of ways in which a function from A into B can be constructed.

Suppose a_1, a_2, \dots, a_n are the elements of A . For each a_i we need to pick an image in B . As there are m elements in B this can be done in m ways for each a_i . Now, by the MCP (noting the independence of the events) we have $\underbrace{(m \times m \times m \times \cdots \times m)}_{n \text{ times}} = m^n$ functions

from A into B .

2. We begin by noting that if $n > m$ then there can be no injective function from A to B (more on this later, when we discuss the Pigeonhole Principle in Section 6.4), so we can assume that $n \leq m$. We need to restrict the construction in (1) so that no two elements in A are assigned the same image in B . The number of ways in which an image can be picked now depends on previous choices for images of elements of A , viz. for a_1 there are m possible choices, then there remain $(m - 1)$ choices for a_2 , $(m - 2)$ choices for a_3 , etc. Applying the MCP we obtain $m(m - 1)(m - 2) \cdots (m - n + 1) = P(m, n)$ injective functions from A into B .

When counting a set of objects, it is sometimes convenient to first divide it into subsets, which are easier to count, and then to add up the results. For this to work, the subsets must form a partition (recall Definition 2.5.7). This is the gist of the following counting principle.

The additive counting principle (ACP)

Let S_1, S_2, \dots, S_m be a partition of a set S , i.e. $S = S_1 \cup S_2 \cup \dots \cup S_m$ and $S_i \cap S_j = \emptyset$ for all $i \neq j$. Further, suppose that $|S_i| = n_i$ for $1 \leq i \leq m$. Then

$$|S| = |S_1| + |S_2| + |S_3| + \dots + |S_m|.$$

Example 6.1.8

Suppose you have five different English books, six different Afrikaans books and eight different Zulu books. In how many ways can you pick an (unordered) pair of books in different languages?

Solution

To count the set of all pairs of books in different languages, a good strategy is to partition this set into subsets according to the pair of languages involved, count these separately and then apply the ACP. Thus, we consider the three subsets, consisting of all English–Afrikaans pairs, all English–Zulu pairs and all Afrikaans–Zulu pairs, respectively. These clearly form a partition of the set we want to count. To count the English–Afrikaans book pairs we now apply the MCP: there are 5 options for the English book and 6 for the Afrikaans, giving $5 \times 6 = 30$ pairs. Similarly there are $5 \times 8 = 40$ English–Zulu pairs and $6 \times 8 = 48$ Afrikaans–Zulu pairs. Thus, by the ACP, there are $30 + 40 + 48 = 118$ ways to choose the pair of books.

Example 6.1.9

How many four-digit integers containing exactly three odd digits are there?

Solution

These integers contain three odd digits and one even digit. The number of possibilities for the even digit depends on the position in which it occurs: in the thousands position there are 9 options, as 0 is excluded there, while in each of the other three positions there are 10 options. Consequently a natural way to break the problem up is to partition the integers we want to count into 4 subsets depending on the position in which the even integer occurs. Denoting the set of four-digit integers containing exactly three odd digits and one even digit in the i th position by S_i we have

$$|S_1| = 4 \times 5^3 \text{ and } |S_2| = |S_3| = |S_4| = 5^4.$$

By the ACP, the total number is $4 \times 5^3 + 5^4 + 5^4 + 5^4 = 4 \times 5^3 + 3 \times 5^4 = 2375$.

Counting the complement

Directly counting the number of objects that have a certain property is sometimes more difficult than counting those that lack the property. Then we can apply a kind of “back door” approach: instead of counting the number of objects that have the desired property, we count the number of objects that *do not have* that property and then subtract them from the total number of objects under consideration. This strategy is often called “*counting the complement*”.

Example 6.1.10

How many four-digit natural numbers are there that contain the digit 7 at least once?

Solution

The four-digit natural numbers are those between 1000 and 9999, in total 9000. We can easily count those that *do not* contain 7. They are $8 \times 9 \times 9 \times 9 = 5832$; hence the ones that do contain 7 are $9000 - 5832 = 3168$.

Addition or multiplication?

You might sometimes find that you are not sure whether you need to apply the addition principle or the multiplication principle. In such cases the following rule of thumb is often useful: when the numbers represent the outcomes of the steps used to construct the objects you are counting, multiply them; when the numbers represent the sizes of subsets of complete objects, add them. As a further aid in making this distinction, you can ask yourself the question: “Am I constructing the object or am I dealing with sets of complete objects?”

6.1.1. Exercises

- ① An airline has three flights between New York and London and five flights between London and Moscow. How many flights can it offer from New York to Moscow passing through London?
- ② A restaurant offers 25 different beverages, 6 different starters, 7 different main courses and 4 different desserts.
 - (a) How many ways are there to order a three course meal with a beverage?
 - (b) How many three course meals are there if chocolate mousse is selected as a dessert?
 - (c) How many ways are there to order a three course meal if either fillet steak or grilled salmon is ordered as a main course?
- ③ How many four-digit integers have no equal consecutive digits?
- ④ How many different ways are there for 10 people to form a queue? How many if Rob, Alice and Lee want to stay together?
 - (a) How many ways are there to seat 12 people along one side of a long table?
 - (b) How many ways are there to seat 12 people around a round table, if all rotations of a given seating arrangement are regarded as the same?

- ⑥ How many ways are there to arrange the letters in the word ISOGRAM so that:
- G is not immediately followed by R?
 - G is immediately followed by either S or M?
- ⑦ How many ways are there to seat 5 men and 5 women along one side of a long table with 10 seats? How many ways if men and women must alternate seats?
- ⑧ You can travel between the airport and the soccer stadium directly using either the bus, tram, metro or taxi; between the soccer stadium and the opera house directly using the bus, tram or taxi; and between the opera house and airport directly using the ferry or taxi.
- How many different ways altogether are there to travel from the airport to the opera house?
 - How many different ways are there to travel from the airport to the opera house and back?
 - How many different ways are there to travel from the airport to the opera house if any form of transport may be used at most once?
 - How many different ways are there to travel from the airport to the soccer stadium if at most two different forms of transport may be used?
- ⑨ There are 12 different books and 9 different CDs. In how many ways can Alice pick a book *or* a CD and then Bob pick a book *and* a CD?
- ⑩ How many ways are there to roll a pair of dice, one red and one blue, to obtain a sum divisible by 3?
- ⑪ How many different license plates are there if any license plate must consist of three consecutive letters (and no vowels (a, e, i, o, u) are allowed) followed by three consecutive digits?
- ⑫ (a) How many four-letter words (i.e. any sequences of four letters) can be made using only letters from *a, b, c, d, e* and *f*, at most once each?
 (b) How many of them contain the letter *a*?
 (c) How many of them contain at least one of the letters *a* and *b*?
 (d) How many of them contain *a* but not *b*?
 (e) How many of them contain exactly one of the letters *a* and *b*?
 (f) How many of them contain both letters *a* and *b*?
- ⑬ Answer the questions from the previous problem if every letter can be used repeatedly.
- ⑭ How many times does the digit 4 occur in a list consisting of all the numbers from 1 to 10 000?
- ⑮ How many different numbers can be formed by the product of two or more of the numbers 2, 2, 2, 3, 5, 5, 7?
- ⑯ A binary sequence is a sequence consisting of 1's and 0's.
- How many different *n*-digit binary sequences are there?
 - How many of them begin and end with the same digit?
 - How many begin and end with different digits?
 - How many have no two consecutive digits the same?

- 17 A ternary sequence is a sequence consisting of 0's, 1's and 2's.
- How many different n -digit ternary sequences are there?
 - How many of them begin and end with the same digit?
 - How many begin and end with different digits?
 - How many have no two consecutive digits the same?
- 18 Six computers on a network are infected by a virus. Every minute each infected computer spreads the virus to six uninfected computers and then crashes. After ten minutes, how many computers have crashed?

Stirling's approximation of $n!$

Working out a particular value of $n!$ can require a lot of multiplication. In applications it is often sufficient to have a good approximation. Stirling's approximation formula

$$S(n) = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

gives a good *asymptotic* approximation of $n!$, meaning that

$$\lim_{n \rightarrow \infty} \frac{n!}{S(n)} = 1.$$

The following table lists the values of $n!$ together with those of $S(n)$ and $n!/S(n)$ for $n = 1, \dots, 18$, rounded to two decimal places.

Table 6.1 The first 18 factorials and Stirling numbers

n	$n!$	$S(n)$	$\frac{n!}{S(n)}$
1	1	0.92	1.08
2	2	1.92	71.04
3	6	5.84	1.03
4	24	23.51	1.02
5	120	118.02	1.02
6	720	710.08	1.01
7	5 040	4 980.40	1.01
8	40 320	39 902.40	1.01
9	362 880	359 536.87	1.01
10	3 628 800	3 598 695.62	1.01
11	39 916 800	39 615 625.05	1.01
12	479 001 600	475 687 486.47	1.01
13	6 227 020 800	6 187 239 475.19	1.01
14	87 178 291 200	86 661 001 740.60	1.01
15	1 307 674 368 000	1 300 430 722 199.47	1.01
16	20 922 789 888 000	20 814 114 415 223.10	1.01
17	355 687 428 096 000	353 948 328 666 101.00	1.00
18	6 402 373 705 728 000	6 372 804 626 194 310.00	1.00

6.2. Combinations. The binomial theorem

In the previous section we considered the number of ways to choose and order objects from a given set. Very often the order of the selected objects is irrelevant. Such selections, also known as **combinations**, are the topic of this section. The numbers of such combinations crop up in various places in mathematics and, in particular, as the so-called binomial coefficients, which we will also treat here.

6.2.1. Counting sheep and combinations

Suppose we have five books A, B, C, D, E , of which we want to choose three without taking into account the order of choices: that is, the choices ABC and BAC are the same as the choice ACB , etc. Applying the MCP directly does not work here because selecting the books one by one automatically yields an ordering. To deal with the problem we will use a simple trick, sometimes called the **shepherd's principle**. Here is the story behind its name: a certain shepherd had a very large flock of sheep, which he had to count every evening after returning from pasture. Once someone asked him "How do you manage to count so many sheep?" "Oh, that is easier than it seems", replied the shepherd, "for I simply count their legs and then divide by 4." Though this sounds like a silly joke, sometimes it is easier, indeed, to count the "legs" than the "sheep". In our case, let the "sheep" be the possible three-books selections from the given five books and let the "legs" of each "sheep" be the possible permutations of the selection. Consider any choice of "sheep", say $\{A, B, C\}$. For this choice there are $3!$ permutations, namely ABC, ACB, BAC, BCA, CAB and CBA . So each "sheep" has $3! = 6$ "legs". We know that the total number of "legs" is the number of arrangements of 3 books out of 5, which is $P(5, 3)$. Therefore, applying the shepherd's trick, we find that the number of all "sheep", i.e. all combinations of 3 books out of 5, is

$$\begin{aligned} C(5, 3) &= \frac{P(5, 3)}{3!} \\ &= \frac{5!}{(5 - 3)!3!} \\ &= \frac{5 \cdot 4}{2 \cdot 1} = 10 \end{aligned}$$

We can easily generalize the argument above by replacing 5 with n and 3 with r , to obtain the following theorem.

Theorem 6.2.1 *The number of combinations, i.e. selections where the order is not taken into account, of r objects out of n objects is*

$$C(n, r) = \frac{P(n, r)}{r!} = \frac{n!}{(n - r)!r!}.$$

Example 6.2.2

How many different committees of four people can be selected out of a meeting of seven people?

Solution

Note that, unlike Example 6.1.6, the order of people in the committee is not important now as there are no distinguished roles within the committee. So the answer is

$$\begin{aligned} C(7, 4) &= \frac{7!}{(7 - 4)!4!} \\ &= 7 \times 5 = 35. \end{aligned}$$

Example 6.2.3

From a meeting of 6 mathematicians and 5 computer scientists, a committee of 5 people has to be chosen, with the restriction that no more than 3 committee members may be computer scientists. How many such committees are there?

Solution

The most efficient approach is to count the complement and subtract from the total number of possible committees.

The total number of possible five-member committees in the meeting is $C(11, 5) = 462$. Now the committees we do not want (those in the complement) are those that contain exactly 4 or 5 computer scientists. We compute the number of committees with 4 computer scientists as follows: first pick 4 of the 5 computer scientists (in $C(5, 4)$ ways) and then pick one mathematician for the fifth member (in $C(6, 1)$ ways). Thus, by the multiplication principle, there are $C(5, 4) \times C(6, 1)$ committees that contain exactly 4 computer scientists. Likewise, there is $C(5, 5) = 1$ committee containing all 5 computer scientists. Thus, by the addition principle, there are $C(5, 4) \times C(6, 1) + C(5, 5) = 31$ committees in the complement. Hence the number of committees we want is $462 - 31 = 431$.

Another approach would be to count the number of committees we are interested in directly, i.e. count the committees containing exactly 0, 1, 2 and 3 computer scientists, respectively, and add the results. There are $C(6, 5) \times C(5, 0)$, $C(6, 4) \times C(5, 1)$, $C(6, 3) \times C(5, 2)$ and $C(6, 2) \times C(5, 3)$ such committees, respectively, and hence there are $C(6, 5) \times C(5, 0) + C(6, 4) \times C(5, 1) + C(6, 3) \times C(5, 2) + C(6, 2) \times C(5, 3) = 6 \times 1 + 15 \times 5 + 20 \times 10 + 15 \times 10 = 431$.

Yet another approach one might be tempted to suggest would be the following. The requirement that there should be no more than 3 computer scientists on the committee is clearly equivalent to the requirement that there must be at least 2 mathematicians on that committee. Therefore, begin by picking 2 mathematicians, there are $C(6, 2)$ ways, and then pick the other 3 committee members from the remaining $11 - 2 = 9$ people, there are $C(9, 3)$ ways. This gives us a total number of $C(6, 2) \times C(9, 3) = 1260$ committees satisfying the requirements. This argument, however, is *wrong!* It counts some committees more than once. (Think about it!)

6.2.2. Some important properties

Now we will establish some important properties of the numbers $C(n, r)$.

Theorem 6.2.4 *If n, r and s are non-negative integers with $r, s \leq n$ then the following hold:*

1. $C(n, 0) = C(n, n) = 1$.
2. $C(n, r) = C(n, n - r)$.
3. $C(n, r) + C(n, r - 1) = C(n + 1, r)$.
4. If $C(n, r) = C(n, s)$ and $r \neq s$, then $n = r + s$.

Proof: All these follow easily using the definition of $C(n, r)$ and some algebraic manipulations, but we will give more enlightening alternative proofs of Properties 1, 2 and 3, using combinatorial arguments.

1. There is only one way to choose no objects out of n and there is also only one way to choose all the objects.
2. $C(n, r)$ is equal to the number of ways of choosing a committee of r people from a meeting of n people. This is the same as choosing the $n - r$ people who will not serve on the committee, which can be done in $C(n, n - r)$ ways.
3. Suppose a committee of r people is to be chosen from a meeting of $n + 1$ people; there are $C(n + 1, r)$ ways to do this. Further suppose that in the meeting is a certain Mrs Smith. Now, either Mrs Smith serves on the committee or she does not. So the total number of possible committees will be the number on which she serves, plus the number of those on which she does not serve. The number of committees on which she serves is equal to $C(n, r - 1)$ —we pick Mrs Smith and then the other $r - 1$ members. The number of those on which she does not serve is equal to $C(n, r)$ —we select the committee of r from the remaining n people in the meeting. So $C(n + 1, r) = C(n, r) + C(n, r - 1)$.
4. If $r \neq s$ then either $r < s$ or $s < r$. Suppose that $r < s$. Here we have

$$\frac{n!}{(n - r)!r!} = \frac{n!}{(n - s)!s!}$$

and hence

$$\frac{s!}{r!} = \frac{(n - r)!}{(n - s)!}.$$

Thus (recall the assumption that $r < s$)

$$s(s - 1)(s - 2) \cdots (r + 1) = (n - r)(n - r - 1) \cdots (n - s + 1).$$

The left-hand side and the right-hand side above are both products of $s - r$ factors of consecutively decreasing integers. Therefore it must be the case that $s = n - r$, i.e. $n = s + r$. The case $s < r$ is completely symmetric.

6.2.3. Pascal's triangle

An often used alternative notation for the number $C(n, r)$ is $\binom{n}{r}$, read as “ n choose r ”.

Properties 1 and 2 of the forgoing theorem illustrate some of the symmetries in the sequence $\binom{n}{0}, \binom{n}{1}, \dots, \binom{n}{n-1}, \binom{n}{n}$, and number 3 indicates the relationship of this sequence with $\binom{n+1}{0}, \binom{n+1}{1}, \dots, \binom{n+1}{n}, \binom{n+1}{n+1}$. These, and many other relationships, become visually clear when these sequences are arranged to form the triangular configuration known as Pascal's triangle.¹ It is formed by listing all numbers:

$$\begin{array}{ccccccc}
 & & & & \binom{0}{0} & & \\
 & & & & \binom{1}{0} & \binom{1}{1} & \\
 & & & & \binom{2}{0} & \binom{2}{1} & \binom{2}{2} \\
 & & & & \binom{3}{0} & \binom{3}{1} & \binom{3}{2} & \binom{3}{3} \\
 & & & & \binom{4}{0} & \binom{4}{1} & \binom{4}{2} & \binom{4}{3} & \binom{4}{4} \\
 & & & & \binom{5}{0} & \binom{5}{1} & \binom{5}{2} & \binom{5}{3} & \binom{5}{4} & \binom{5}{5} \\
 & & & & \binom{6}{0} & \binom{6}{1} & \binom{6}{2} & \binom{6}{3} & \binom{6}{4} & \binom{6}{5} & \binom{6}{6} \\
 & & & & \vdots & & \vdots & & & & \\
 \end{array}$$

Calculating the respective numbers we obtain:

$$\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & & 1 & 1 & & & \\
 & & & & 1 & 2 & 1 & & \\
 & & & & 1 & 3 & 3 & 1 & \\
 & & & & 1 & 4 & 6 & 4 & 1 \\
 & & & & 1 & 5 & 10 & 10 & 5 & 1 \\
 & & & & 1 & 6 & 15 & 20 & 15 & 6 & 1 \\
 & & & & \vdots & & \vdots & & \\
 \end{array}$$

According to the properties in Theorem 6.2.4, this triangle can be constructed as follows: first, place 1's on the two sides of the triangle and then compute row-by-row the entries inside by using Property 3, saying that every number inside the triangle is the sum of the two diagonally above it.

6.2.4. The binomial theorem

The expansion $(x + a)^2 = x^2 + 2xa + a^2$ is well known to all of us. Here we will generalize it to an expansion of $(x + a)^n$ for any n , to obtain the so-called **binomial theorem** – an algebraic identity which we will derive through purely combinatorial arguments using combinations. We will start by taking a closer look at the expansion of $(x + a)^3$ by simply multiplying out the

¹ See the box at the end of this section for more on Pascal's triangle.

three equal sums:

$$\begin{aligned}(x+a)^3 &= (x+a)(x+a)(x+a) \\ &= xxx + xxa + xxa + xxa + xaa + xaa + xaa + aaa\end{aligned}$$

Note that each term in the expansion above arises as the product of three factors, one picked from each pair of brackets. There is only one term xxx , as there is only one way in which this can arise, namely by multiplying the x 's from each of the three pairs of brackets. There are $C(3, 2) = 3$ products xxa as these arise by selecting the x 's from any two of the three pairs of brackets of a from the other. Likewise, there are $C(3, 2) = 3$ terms xaa and $C(3, 3) = 1$ term aaa . Thus, when collecting the equal terms together, we find that

$$(x+a)^3 = C(3, 0)x^3 + C(3, 1)x^2a + C(3, 2)xa^2 + C(3, 3)a^3.$$

We can now apply the very same reasoning to the expansion of $(x+a)^n$. This expansion will be a sum of products of n x 's and a 's, one picked from each of the n factors $x+a$ in $(x+a)^n$. Thus, all these products are of the type $x^{n-r}a^r$ for $0 \leq r \leq n$. Now, note that $x^{n-r}a^r$ is obtained by selecting r of the a 's and then (automatically) $n-r$ of the x 's from the n factors in $(x+a)^n$. This can be done in $C(n, r)$ ways, so the coefficient in front of $x^{n-r}a^r$ is $C(n, r)$. This calculation, applied for each r , gives the explicit formula of the expansion.

Theorem 6.2.5 Binomial theorem *For every pair of real numbers x and a and every natural number n the following holds:*

$$\begin{aligned}(x+a)^n &= C(n, 0)x^n + C(n, 1)x^{n-1}a + C(n, 2)x^{n-2}a^2 + \cdots + C(n, n)a^n \\ &= \sum_{r=0}^n C(n, r)x^{n-r}a^r.\end{aligned}$$

Example 6.2.6

Calculate $(11)^3$ using the binomial theorem.

Solution

$$\begin{aligned}(11)^3 &= (10+1)^3 = 10^3 + C(3, 1)(1)(10)^2 + C(3, 2)(1)^210^1 + 1^3 \\ &= 1000 + 3(100) + 3(10) + 1 \\ &= 1331.\end{aligned}$$

By replacing a with $-a$ in the binomial formula above we obtain

$$(x-a)^n = \sum_{r=0}^n (-1)^r C(n, r)x^{n-r}a^r = C(n, 0)x^n - C(n, 1)x^{n-1}a + \cdots + (-1)^na^n.$$

Example 6.2.7

Expand $(x^3 - 2)^4$ in powers of x .

Solution

Using the formula above we find

$$\begin{aligned}(x^3 - 2)^4 &= C(4, 0)(x^3)^4 + C(4, 1)(x^3)^3(-2)^1 + C(4, 2)(x^3)^2(-2)^2 \\ &\quad + C(4, 3)(x^3)^1(-2)^3 + C(4, 4)(-2)^4 \\ &= x^{12} - 2 \cdot 4x^9 + 6 \cdot 4x^6 - 4 \cdot 8x^3 + 16 \\ &= x^{12} - 8x^9 + 24x^6 - 32x^3 + 16.\end{aligned}$$

Example 6.2.8

Determine the term that is independent of x and the coefficient of x^6 in the expansion of $\left[x - \frac{2}{x}\right]^{10}$.

Solution

The general term is

$$t_{r+1} = C(10, r)x^{10-r}\left[-\frac{2}{x}\right]^r = C(10, r)(-2)^r x^{10-2r}.$$

In the term that is independent of x , $x^{10-2r} = x^0$, so that $10 - 2r = 0$, $r = 5$ and the required term is $t_6 = (-2)^5 C(10, 5)$.

Now, given that $t_{r+1} = C(10, r)(-2)^r x^{10-2r}$, for the coefficient of x^6 we obtain the equation $10 - 2r = 6$, with the solution $r = 2$. The term is therefore $t_3 = C(10, 2)(-2)^2 x^6$ and the coefficient of x^6 is $C(10, 2)(-2)^2 = 180$.

6.2.5. Exercises

- ① (a) A certain engineering firm employs 12 mechanical engineers, 7 civil engineers and 3 electrical engineers. A committee is to be appointed from among the 22 engineers.
 (b) How many five-member committees can be appointed?
 (c) How many five-member committees containing at least one electrical engineer can be formed?
 (d) How many five-member committees containing at least one engineer of each type can be formed?
 (e) How many committees can be formed in which there are more mechanical engineers than civil engineers and more civil engineers than electrical engineers?

- ②** In how many ways can a committee of 3 ladies and 4 men be chosen from 8 ladies and 7 men? What is the number of committees if Mrs Smith refuses to serve on the committee with Mr Jones?
- ③** In how many ways can
- a female chairperson and a male secretary be chosen from a meeting of 6 men and 5 women?
 - a chairman, vice-chairman, secretary and treasurer be chosen from a meeting of 15 people?
 - 3 out of 8 given books be arranged on a shelf?
 - I make a 4-letter “word” (with or without meaning) with four different letters of the 26-letter Latin alphabet?
 - I hang my 6 shirts and 4 suits in the cupboard?
 - I hang my 6 shirts and 4 suits in the cupboard if every shirt must hang on the left of every suit?
 - I arrange my 8 math books, 5 physics books and 7 chemistry books on my bookshelf if all books of each subject must be together?
- ④** In how many ways can
- 11 books be chosen from 13 different books?
 - I prepare for a trip by choosing 3 of my 6 shirts and 5 of my 7 pairs of socks?
 - a girl fill her doll house by choosing 3 dolls from 6, 8 pieces of furniture from 9, 2 carpets from 4 and 2 plants from 5?
 - a pack of 52 cards be dealt into 4 hands of 13 cards each?
- ⑤** How many ways are there to arrange 7 identical copies of *Pride and Prejudice* and 8 identical copies of *Sense and Sensibility* on a shelf?
- ⑥** How many ways are there to place 8 identical black pawns and 8 identical white pawns on a standard chess board?
- ⑦** How many different subsets (including the empty set) does an n -element set have?
Compare this with the number of binary (0–1) sequences of n terms.
- ⑧** Let $\Omega = \{\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta\}$.
- How many four-element subsets does Ω have?
 - How many five-element subsets of Ω can be formed if β must be in the set?
 - How many five-element subsets of Ω can be formed if γ may not be in the set?
 - How many five-element subsets of Ω can be formed containing γ but not containing ζ or α ?
 - How many subsets of Ω are there that contain at least 4 elements?
 - How many subsets of Ω are there that contain fewer than 7 elements?

- ⑨ A mathematics department receives 60 applications for 5 vacant PhD positions. Supposing that every applicant meets the requirements for each of the positions, in how many different ways can the department allocate the positions if:
- the positions are identical?
 - the positions are all different?
 - the five positions consist of three identical positions in Functional Analysis and two identical positions in Universal Algebra?
- ⑩* If a set S has at least a 100 different subsets of size 5, what is the smallest size S can be?
- ⑪* If one quarter of all subsets of size 5 of the set $\{1, 2, 3, \dots, m\}$ contains the element 2, determine m .
- ⑫ Expand in powers of x :
- $(x + 1)^7$
 - $(x - 2)^4$
 - $\left(3x - \frac{1}{2}\right)^3$
 - $\left[\frac{x}{4} - \frac{4}{x}\right]^5$
- ⑬ Expand in powers of x and y :
- $(3x - 2y)^4$
 - $(x^2 + 3y^2)^5$
- ⑭ Determine the coefficient of:
- x^6 in $(x + 1)^8$
 - x^{-4} in $\left[2 - \frac{1}{x}\right]^5$
 - x^8y^5 in $(3x - 4y)^{13}$
 - x^9y^{14} in $(x^3 - 3y^2)^{10}$

Pascal's triangle

			1						
			1	1	2	1			
			1	3	3	1			
			1	4	6	4	1		
			1	5	10	10	5	1	
			1	6	15	20	15	6	1
			1	7	21	35	35	21	7
			1	8	28	56	70	56	8
1									1

The table above gives the first nine rows of a simple but remarkable construction of numbers arranged in a triangle, where all entries on the 2 sides are 1 and all other entries in every row are obtained by adding the entries in the previous row that are placed diagonally above that entry. This construction has roots going back more than two thousand years, but is known in the West as *Pascal's triangle* after the French mathematician, physicist and philosopher Blaise Pascal (1623–1662) (see the box at the end of Section 6.6), who published the treatise *Trait du Triangle Arithmtique* studying this triangle. The triangle appears in Ancient Greek mathematics and also in the *Chandahsastra*, a Sanskrit text composed around 200 BC in India. In Persia (now Iran) it was studied by the poet, astronomer and mathematician Omar Khayyám (1048–1131), so in Iran it is referred to as the Khayyám–Pascal triangle. Pascal's triangle appears in the work of the Chinese mathematician Jia Xian (1010–1070) and again later in that of Yang Hui (1238–1298). In China Pascal's triangle is known as Yang Hui's triangle.

If we start counting at the 0th row, then the n th row consists of the binomial coefficients $\binom{n}{0}, \dots, \binom{n}{n}$. The rule of the construction corresponds to the fact that $\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$. We mention just three more of the many fascinating properties of Pascal's triangle:

- The sum of the entries in row n is equal to 2^n , e.g. the entries in row 5 sum to $1 + 5 + 10 + 10 + 5 + 1 = 32 = 2^5$. This corresponds to the fact that a set of n elements has 2^n different subsets and the binomial coefficients appearing in row n count the number of subsets of size 0, size 1, …, size n .
- Taking the numbers in row n as the digits of a number and carrying numbers greater than 9 appropriately, the number obtained is always equal to 11^n . For example, row 1 gives 11, row 2 gives 121, which is 11^2 , row 3 gives 1331 = 11^3 and row 5 gives $1 \times 10^5 + 5 \times 10^4 + 10 \times 10^3 + 10 \times 10^2 + 5 \times 10^1 + 1 \times 10^0 = 161\,051 = 11^5$.
- If n is a prime number, then all entries in row n are multiples of n .

6.3. The principle of inclusion–exclusion

The addition principle enables us to count the elements of a set S by counting the elements of its subsets, *if those subsets form a partition of S* . However, this is often not the case because the subsets are not disjoint. In this section we will derive the so-called Inclusion–Exclusion Principle (IEP), which will help us to deal with such cases. We will need the following preliminary result.

Lemma 6.3.1 *Let A and B be subsets of some finite set U . Then*

1. $|A \setminus B| = |A| - |A \cap B|$.
2. $|A^c| = |U| - |A|$.

Proof: These two facts follow from the additive counting principle, by noting, respectively, that $A \setminus B$ and $A \cap B$ form a partition of A and that A and A^c form a partition of U .

Proposition 6.3.2 (Inclusion – exclusion principle for two sets) *Let A and B be finite sets (see Figure 6.1). Then*

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

Proof: We note that

$$A \cup B = (A \setminus B) \cup (B \setminus A) \cup (A \cap B),$$

where $(A \setminus B)$, $(B \setminus A)$ and $(A \cap B)$ are clearly pairwise disjoint sets. Hence, by the addition principle,

$$|A \cup B| = |A \setminus B| + |B \setminus A| + |A \cap B|.$$

By Lemma 6.3.1 we have $|A \setminus B| = |A| - |A \cap B|$ and $|B \setminus A| = |B| - |A \cap B|$. Applying this to the equation above we get, as desired:

$$|A \cup B| = |A| - |A \cap B| + |B| - |A \cap B| + |A \cap B| = |A| + |B| - |A \cap B|.$$

Here is a typical and straightforward application.

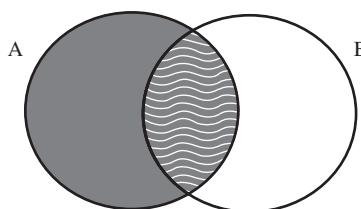


Figure 6.1: The inclusion–exclusion principle for two sets

Example 6.3.3

In a sample of 100 university students 75 take mathematics, 50 take physics and 45 take both.

1. How many take at least one of these subjects?
2. How many take neither subject?

Solution

Let S be the sample set of students, M the subset of those taking mathematics and P the subset of those taking physics. We are given that $|S| = 100$, $|M| = 75$, $|P| = 50$ and $|M \cap P| = 45$.

1. We have to determine $|M \cup P|$, which we do by the IEP:

$$|M \cup P| = |M| + |P| - |M \cap P| = 75 + 50 - 45 = 80.$$

2. The number of students taking neither subject will be the total number less the number taking either, i.e.

$$100 - 80 = 20.$$

Intuitively, we can think of the IEP for two sets as follows: in order to count the elements in the union of sets A and B we add the number of elements in A and the number of elements in B . However, if these sets intersect, we have counted the elements in the intersection twice, so we must subtract the number of those elements. Generalizing this reasoning to three sets we obtain the three-set IEP:

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|.$$

Explanation: after we add the sizes of the sets we have counted twice the elements in the two-set intersections, so we subtract them, but then we have subtracted 3 times the size of the three-set intersection, so we have to add it back.

The next example uses the formula above and again illustrates that it can sometimes be easier to count the complement of the set that one is interested in, rather than to count the set itself.

Example 6.3.4

We want to count the number of ternary sequences (sequences consisting of 0's, 1's and 2's) of length n that contain at least one 0, at least one 1 and at least one 2. Let U be the set of all ternary sequences of length n . Note that $|U| = 3^n$. Let A_0 , A_1 and A_2 be the subsets of U containing all ternary sequences containing no 0's, 1's or 2's, respectively. Clearly, $|A_0| = |A_1| = |A_2| = 2^n$. We also have $|A_0 \cap A_1| = |A_0 \cap A_2| = |A_1 \cap A_2| = 1$ and $|A_0 \cap A_1 \cap A_2| = 0$. Thus the complement in U of our set consists of all ternary sequences that miss a 0, a 1 or a 2, i.e. $A_0 \cup A_1 \cup A_2$, the size of which we obtain by the three-set IEP:

$$\begin{aligned} |A_0 \cup A_1 \cup A_2| &= |A_0| + |A_1| + |A_2| - |A_0 \cap A_1| - |A_0 \cap A_2| - |A_1 \cap A_2| + |A_0 \cap A_1 \cap A_2| \\ &= 3(2^n) - 3(1) + 0 = 3(2^n) - 3. \end{aligned}$$

So the number of ternary sequences that contain at least one 0, at least one 1 and at least one 2 is $3^n - (3(2^n) - 3)$.

Example 6.3.5

A car dealer has 30 cars in stock. Of these cars 20 have immobilizers, 8 have air conditioners and 25 have fuel injection; 20 have at least two of these features and 6 have all three.

1. How many cars have at least one of the features?
2. How many have none of these features?
3. How many have exactly one?

Solution

Let C be the set of all cars in stock, I the set of cars with immobilizers, A the set of cars with air conditioners and F the set of cars with fuel injection. The information we are given amounts to the following: $|C| = 30$, $|I| = 20$, $|A| = 8$, $|F| = 25$, $|(I \cap A) \cup (I \cap F) \cup (A \cap F)| = 20$, $|I \cap A \cap F| = 6$.

By the three-set IEP:

$$\begin{aligned} 20 &= |(I \cap A) \cup (I \cap F) \cup (A \cap F)| \\ &= |(I \cap A)| + |(I \cap F)| + |(A \cap F)| - |(I \cap A) \cap (I \cap F)| - |(I \cap A) \cap (A \cap F)| \\ &\quad - |(I \cap F) \cap (A \cap F)| + |(I \cap A) \cap (I \cap F) \cap (A \cap F)|. \end{aligned}$$

Each of the last four terms is equal to $(I \cap A \cap F)$, so we get

$$20 = |(I \cap A)| + |(I \cap F)| + |(A \cap F)| - 2|(I \cap A \cap F)|$$

and hence

$$|(I \cap A)| + |(I \cap F)| + |(A \cap F)| = 20 + 2(6) = 32.$$

We are now in a position to answer questions 1 to 3:

1. The number of cars with at least one feature is

$$\begin{aligned} |I \cup A \cup F| &= |I| + |A| + |F| - |I \cap A| - |I \cap F| - |A \cap F| + |I \cap A \cap F| \\ &= |I| + |A| + |F| - (|I \cap A| + |I \cap F| + |A \cap F|) + |I \cap A \cap F| \\ &= 20 + 8 + 25 - 32 + 6 = 27. \end{aligned}$$

2. The number of cars having none of these features is the total number of cars in stock minus the number that have at least one, i.e. $30 - 27 = 3$.
3. The number of cars with exactly one feature will be the number with at least one minus the number with at least two, i.e. $27 - 20 = 7$.

Following the pattern of reasoning for computing the formula of the three-set IEP we can extend it to any finite family of finite sets. Roughly speaking, to determine the size of the union $A_1 \cup A_2 \cup \dots \cup A_n$, we add the sizes of all possible intersections of an odd number of sets among A_1, A_2, \dots, A_n and subtract the sizes of all possible intersections of an even number of sets among A_1, A_2, \dots, A_n . Here are the formal details.

Theorem 6.3.6 Inclusion-exclusion for n sets Let A_1, A_2, \dots, A_n be finite sets. Then

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{\substack{I \subseteq \{1, 2, \dots, n\} \\ I \neq \emptyset}} (-1)^{|I|+1} \left| \bigcap_{i \in I} A_i \right|.$$

Proof: The proof is by induction on n . The base case for $n = 2$ was proven in Proposition 6.3.2. The induction hypothesis is that the theorem holds for at most $n - 1$ sets. We must now prove the induction step for n sets. Let A_1, A_2, \dots, A_n be finite sets. In the chain of equations below, the first equality follows by the associativity of union and the second by the induction hypothesis applied to the two sets $\bigcup_{i=1}^{n-1} A_i$ and A_n . The third equality follows by the induction hypothesis for the $n - 1$ sets A_1, \dots, A_{n-1} and by the distribution of intersection over union. The last equality follows by a third application of the induction hypothesis, this time to the $n - 1$ sets $(A_1 \cap A_n), (A_2 \cap A_n), \dots, (A_{n-1} \cap A_n)$.

$$\begin{aligned} \left| \bigcup_{i=1}^n A_i \right| &= \left| \left(\bigcup_{i=1}^{n-1} A_i \right) \cup A_n \right| \\ &= \left| \bigcup_{i=1}^{n-1} A_i \right| + |A_n| - \left| \left(\bigcup_{i=1}^{n-1} A_i \right) \cap A_n \right| \\ &= \sum_{\substack{I \subseteq \{1, 2, \dots, n-1\} \\ I \neq \emptyset}} (-1)^{|I|+1} \left| \bigcap_{i \in I} A_i \right| + |A_n| - \left| \bigcup_{i=1}^{n-1} (A_i \cap A_n) \right| \\ &= \sum_{\substack{I \subseteq \{1, 2, \dots, n-1\} \\ I \neq \emptyset}} (-1)^{|I|+1} \left| \bigcap_{i \in I} A_i \right| + |A_n| - \sum_{\substack{J \subseteq \{1, 2, \dots, n-1\} \\ J \neq \emptyset}} (-1)^{|J|+1} \left| \bigcap_{j \in J} (A_j \cap A_n) \right| \end{aligned}$$

The required equality now follows by grouping together the terms that involve the intersection of the same numbers of sets.

6.3.1. Exercises

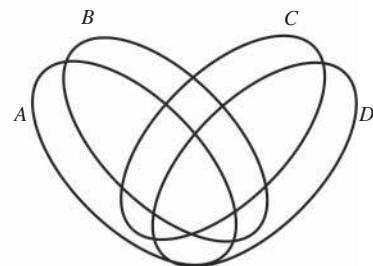
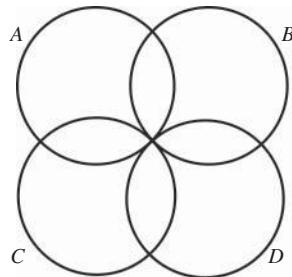
- ① In a sample of 100 households, it was found that 83 subscribe to *Time Magazine*, 41 subscribe to *The Economist* and 32 subscribe to both. Find the number of families that
 - (a) subscribe to neither.
 - (b) subscribe to precisely one of the two.
- ② In a class of 200 students, 75 take mathematics, 70 take statistics, 75 take zoology, 35 take mathematics and zoology, 20 take statistics and zoology, 40 take mathematics and statistics and 15 take all three subjects.
 - (a) How many take at least one of the three subjects?
 - (b) How many take exactly one of the three subjects?
 - (c) How many take statistics or mathematics but not zoology?

- (d) How many do not take exactly two of the three subjects?
 - (e) How many take neither statistics nor mathematics?
- ③ (a) How many positive four-digit integers miss at least one of the digits 0 or 3?
- (b) How many four digit-integers contain at least one of the digits 0 and 3?
- ④ In a survey of 250 movie goers it was found that 95 liked action films, 125 liked comedies, 125 liked dramas, 25 liked action and drama, 45 liked comedy and drama, 35 liked action and comedy and 5 liked all three.
- (a) How many liked to watch action but not comedy?
 - (b) How many liked to watch action or comedy, but not drama?
 - (c) How many liked to watch action or comedy?
 - (d) How many do not watch comedy only?
 - (e) How many watch comedy and drama but not action?
- ⑤ Out of a meeting of mathematicians, 22 like Combinatorics, 14 like Logic and 25 like Number Theory. Everyone who likes Logic likes Number Theory. There are 9 mathematicians who like Combinatorics and Number Theory. Lastly, 5 like all three fields. How many mathematicians at the meeting like at least one of these three subjects?
- ⑥ How many 6-digit integers (with no leading zeros) have 2, 4 or 6 as a digit?
- ⑦ How many positive integers less than or equal to 2013 are divisible by 3 or 4?
- ⑧ How many positive integers less than or equal to 2013 are divisible by 3 or 5, but not by 4?
- ⑨* A basic computer sold by a computer firm may be supplied with any combination of five additional features. Of the computers in stock each additional feature is possessed by 10 computers, each pair of additional features by 5 computers, any given triple of additional features by 3 computers, any given quadruple of additional features by 2 computers and all 5 additional features by 1 computer. Of the computers in stock 6 computers have no additional features. How many computers does the company have in stock?
- ⑩* How many arrangements of the letters in POSTAL are there in which no letter is in its original position?
- ⑪* How many permutations of the 26 alphabet letters are there that contain none of the sequences SO, SHE or WINS?
- ⑫* How many permutations of the 26 alphabet letters are there that contain none of the sequences HE, WAS, SO or BAD?
- ⑬* On a construction line there are 5 workers and 10 different tasks to perform. Every worker is assigned two tasks and the assignment of the tasks is changed every six months. How many ways are there to assign two tasks to each worker such that no worker has the same two tasks during the second six months of the year as he/she had during the first six months of the year?
- ⑭* If two identical dice are rolled n successive times, how many sequences of outcomes contain all doubles?

Venn diagrams

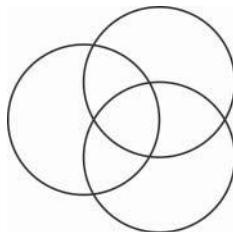
Diagrams of intersecting circles or, more generally, closed curves, have been used as graphical representations of the relationships between sets for a long time. One of the first mathematicians to use them was Euler (see the box at the end of Section 7.1 in Chapter 7). The best known method, however, is due to the British logician and philosopher John Venn (1834–1923). While in Euler's diagrams the curves are allowed to intersect in any way so as to show the relationships between particular sets, in the Venn diagram for n sets the curves must intersect in such a way as to divide the plane into the 2^n possible regions corresponding to relations between the sets.

Here are an Euler diagram and a Venn diagram for four sets:



The Euler diagram (on the left) is not a Venn diagram, as it does not contain all $2^4 = 16$ regions. For example, the possibility of a nonempty intersection between sets A and D is excluded by this diagram, so although four sets *could* stand in the relationship depicted, it does not cover *all* logical possibilities. On the other hand, the Venn diagram (on the right) divides the plane into the required 16 regions.

Notice that the Euler diagram exhibits fourfold rotational symmetry, while the Venn diagram does not. Of course we can easily draw a Venn diagram for three sets exhibits threefold rotational symmetry, as follows:



Can this be done for four sets? In fact, it cannot: it has been proven that there exists a Venn diagram for n sets exhibiting n -fold rotational symmetry, if and only if n is prime.

6.4. The Pigeonhole Principle

Do you know that there are at least two people in Hong Kong with *exactly the same* number of hairs on their heads? You would ask *how one can know that?* To establish this fact empirically one would have to count the number of hairs on the head of each inhabitant of Hong Kong – a rather absurd and very costly procedure. Then *how?* Very simple: a mixture of little bits of biology, statistics and trivial mathematics suffices:

- It is known that the number of hairs on the head of any human being is *less than* 200 000 (biology).
- More than 7 000 000 people live in Hong Kong (population statistics).
- If every two of them had different numbers of hairs on their heads then somebody should have more than 200 000 hairs on his or her head, because the number of non-negative integers less than 200 000 is, of course, only 200 000 (trivial mathematics).

Isn't that convincing?

The argument above is a typical example of the application of an extremely simple mathematical principle, which almost magically implies lots of interesting and deep results that otherwise would require enormous, if not impossible, technical efforts to obtain (as in our example). This principle is often called the **Pigeonhole Principle**,² because a popular version of it reads:

If more than n pigeons fly into n pigeonholes then at least two pigeons will end up in the same pigeonhole.

Obvious, isn't it? So obvious that if you have to prove it you would wonder what actually is there to be proved. Of course you could object that this is not a precise mathematical statement, and therefore no mathematical (i.e. formal) proof is applicable. Here is a more formal expression of the same idea, which we will hereafter call the Pigeonhole Principle (PHP for short).

The Pigeonhole Principle (PHP)

Let P (for pigeons) and H (for pigeonholes) be finite sets such that P has more elements than H . Then there is no injective mapping from P to H , i.e. every mapping from P to H must send (at least) two elements of P to the same element of H .

The PHP can be proved by mathematical induction. The question, however, is whether the principle of mathematical induction is something *more obvious* or *more basic* than the PHP, and whether it itself requires proof. In fact, it turns out that the two principles are equivalent, in a precise mathematical sense.

Here we will apply the PHP in various situations. The art of successful application of the PHP consists in an appropriate choice of the “pigeons” and “pigeonholes”. Let us start

² Also known as **Dirichlet's Principle**, after the famous German mathematician Peter Gustav Lejeune Dirichlet, who first obtained serious mathematical results (in number theory) by applying this principle. See the box at the end of this section for more on this mathematician.

with a paraphrase, using our principle, of the opening argument of this section. Consider the inhabitants of Hong Kong as the “pigeons” and 200 000 pigeonholes labelled by the positive integers from 0 to 199 999. Then we let every “pigeon” fly in the pigeonhole labelled by the number of hairs on the “pigeon’s” head. Due to the fact that the number of hairs of every human is less than 200 000 there will be no “pigeon” without a “pigeonhole”. Then, since the “pigeons” are more than the “pigeonholes”, according to the PHP at least two “pigeons” will inevitably end up in the same “pigeonhole”, i.e. two people will turn out to have the same number of hairs on their heads.

Example 6.4.1

There are 370 students in a course. Show that two of them have birthdays on the same day of the year.

Solution

Let P (pigeons) be the set of students and let H (pigeonholes) be the days of the year. However, then $|P| = 370 > 366 = |H|$, so by the PHP any function from P to H (i.e. assigning each student to his/her birthday) will associate at least two elements of $|P|$ with the same element of $|H|$.

Example 6.4.2

Prove that for every positive integer n , amongst any $n + 1$ integers m_1, \dots, m_{n+1} there are two numbers, the difference between which is a multiple of n .

Solution

Let $P = \{m_1, \dots, m_{n+1}\}$ and let H be the n possible remainders modulo n , i.e. $H = \{0, 1, 2, \dots, n - 1\}$. The PHP implies that two numbers must have the same remainder modulo n , and hence n will divide the difference between them.

Example 6.4.3

Each competitor in a piano competition has to choose and perform two of Beethoven’s 32 piano sonatas. What is the least number of entries that need to be received for the judges to be sure that at least 2 competitors will play the same selection of two sonatas?

Solution

The competitors are the “pigeons”, who we want to classify according to their choice of Beethoven sonatas. To be able to apply the PHP there needs to be more competitors than possible choices of sonatas, i.e. $|P| > |H| = C(32, 2) = 496$. Thus there must be at least 497 entries.

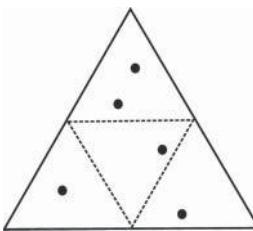


Figure 6.2: Five points in an equilateral triangle. By the Pigeonhole Principle, one of the smaller triangles must contain at least two points

Example 6.4.4

Five points are placed in an equilateral triangle with sides of length 1. Show that the distance between two of them is not greater than 0.5.

Solution

Partition the triangle, by connecting the midpoints of its sides, into 4 equilateral triangles with sides of length 0.5, as shown in Figure 6.2. Then, since there are 5 points and only 4 small triangles, the PHP implies that at least 2 of the points must be in the same small triangle. Moreover, the maximum distance between any two points in any of the small triangles is 0.5.

A generalization of the PHP, which we call the **Congested Pigeonhole Principle**, is as follows:

If more than kn pigeons fly into n pigeonholes then some pigeonhole will have to accommodate more than k pigeons.

For instance, if 16 pigeons fly into 5 pigeonholes, then at least 4 pigeons must share the same pigeonhole. Once again the principle is intuitively clear – if no more than k pigeons fly into each of the n pigeonholes then the total number of pigeons in the pigeonholes would be no more than nk . We can state this formally as follows.

The Congested Pigeonhole Principle (CPHP)

Let P and H be finite sets with respectively m and n elements and k a positive integer such that $m > kn$. Then any mapping from P to H must associate at least $k + 1$ elements of P with some element of H .

Note that the PHP is a special case of the CPHP, for $k = 1$.

Example 6.4.5

Show that in a class of 61 students there are 6 with birthdays in the same month.

Solution

Let P be the set of students and H the set of months. Then, since $|P| = 61 > 5 \times 12 = 5 \times |H|$, the claim follows by the CPHP.

Example 6.4.6

A thousand men are asked to choose and order their 3 favourite beers from a list of n beers. What is the largest value of n for which we can be sure that at least 5 men will choose the same 3 beers in the same order?

Solution

Let P be the set of men and let H be the set of possible ordered selections of 3 beers. In order to be sure that at least 5 men will make the same ordered selection, we need $1\ 000 > 4 \times P(n, 3)$, i.e. $250 > P(n, 3)$. The greatest value of n for which this inequality will hold is 7. So there can be at most 7 beers on the list.

Example 6.4.7

145 points are placed in a rectangle of size $6\text{ cm} \times 16\text{ cm}$. Show that there are 7 of these points all within a distance of less than 3 cm from each other.

Solution

Partition the rectangle into $3 \times 8 = 24$ squares with sides of length 2 cm. One of these squares must contain at least 7 of these points, because $145 > 24 \times 6$. The distance between any two points in the same square is at most $2\sqrt{2}\text{ cm} < 3\text{ cm}$.

6.4.1. Exercises

- ① How many socks must be chosen from 19 different pairs of socks to ensure that at least one pair is selected?
- ② How many positive integers not greater than 100 must be selected at random to be sure that at least one of them is divisible by 7?
- ③ Prove that in any group of people there are two who have the same number of friends in the group.
- ④ How many numbers from $\{1, 2, \dots, 99\}$ must be chosen to ensure that the sum of two of them is 100?
- ⑤ Given that more than 4 million people live in (greater) Johannesburg, are there at least 21 people in the city with exactly the same number of hairs on their heads? We may assume that no person can have more than 200 000 hairs on his or her head.
- ⑥ How many people from a group of people of 7 nationalities must be selected to ensure that at least 4 of those selected are compatriots?
- ⑦ In a school there are 393 students in 17 classes. Show that either there is a class with at least 25 students or there are at least two classes with at least 24 students each.
- ⑧ How many words in the (26-letter) Latin alphabet must be chosen, to ensure that at least 3 of them begin with the same letter and end with the same letter?

- ⑨ 606 points are placed in a square with a side 1. Show that at least 6 of them can be covered by a circle of radius $1/15$. (Hint: consider partitioning the square into smaller squares that can be inscribed in a circle of radius $1/15$.)
- ⑩ There are 13 blue, 10 red, 8 green and 6 yellow balls in a box. How many of them must be selected at random so that amongst them there will be:
- at least 5 of the same colour;
 - at least 3 green;
 - at least one of each colour?
- ⑪ In a group of 6 people every two of them either like or hate each other. Show that in that group there are either three who like one another or three who hate one another.
- ⑫* Every two of 17 points are connected by a line segment coloured either in red, blue or green. Prove that there is a triangle with vertices among these points and all sides of the same colour.
- ⑬* 50 points are placed in a circle of diameter 12. Show that there are two points at a distance not greater than 2 apart.
- ⑭* Let P be an infinite set and H be a finite set. Then show that any mapping from P to H must associate infinitely many elements of P with some element of H .
- ⑮* Show that amongst any $mn + 1$ integers there are either $n + 1$ equal or $m + 1$ pairwise different.
- ⑯* Prove that for every positive integer n , any set of n integers $\{m_1, \dots, m_n\}$ contains a nonempty subset, the sum of whose elements is a multiple of n .
- ⑰* Prove that if m and n are relatively prime positive integers then there is a positive integer k such that n divides $mk - 1$.
- ⑱** Is there a power of 7 that, when written in decimal notation, ends with 00000001?
- ⑲** Show that there is a number of the form 20142014...20140...0 that is divisible by 2013.
- ⑳** Prove that amongst $n + 1$ different integers m_1, \dots, m_{n+1} , where $1 \leq m_1 < m_2 < \dots < m_{n+1} \leq 2n$, there are three of which one is equal to the sum of the other two.
- ㉑** Let M be an arbitrary set of 10 positive integers not greater than 100. Show that there are two disjoint (i.e. having no common elements) subsets of M that have the same sum of their elements.³
- ㉒** Let A be a set of 101 positive integers, each of which is not greater than 200. Prove that there are two numbers in A , one of which is a divisor of the other.
- ㉓** Show that in every set of 201 positive integers, each less than 301, there are two numbers, the ratio of which is a power of 3.
- ㉔** Let M be a set of 75 positive integers not greater than 100. Show that for each positive integer $k \leq 49$ there are two elements of M that differ by k .
- ㉕** Let $k \geq 1$ and $n \geq 1$ be positive integers and A be a set of $(k - 1)n + 1$ integers each of which is not greater than kn . Prove that some element of A can be represented as a sum of k (not necessarily different) elements of A .

³ This problem is from the XIV International Mathematical Olympiad in Poland, 1972.



Johann Peter Gustav Lejeune Dirichlet (1805–1859)

Dirichlet was a German mathematician, today remembered for a range of outstanding contributions. He was the first to apply techniques from analysis to number theory, thereby founding the field known as *analytic number theory*. In the course of his work on number theory he introduced what we know today as the *Pigeonhole Principle*, often called *Dirichlet's Principle*, under the name of *Drawer Principle* (*Schubfachprinzip* in German). He proved, in his first published paper, some special cases of Fermat's last theorem (see the box at the end of Section 5.9 in Chapter 5), namely that for $n = 5$ and $n = 14$ the equation $x^n + y^n = z^n$ has no nonzero integer solutions. The cases for $n = 3$ and $n = 4$ had already been proved by Euler and Fermat. In analysis Dirichlet is mostly remembered for his pioneering work on Fourier analysis. He is credited as being one of the first mathematicians to introduce the modern definition of a function.

Dirichlet was born in the town of Düren in 1805, where his father was postmaster. From an early age he had a passion for mathematics and history and was an exemplary, if rather shy, student. After some initial schooling in Bonn his parents sent him to the Jesuit College in Cologne. Here he was taught by Georg Ohm, best remembered today for *Ohm's law* regarding resistance in electrical circuits. For his university education Dirichlet went to Paris where he came in contact with some of the leading mathematicians of the day, including Fourier, Laplace, Legendre and Poisson. After returning to Germany in 1825 he taught for a while at the university of Breslau, after which he moved to Berlin where he taught at the university and the military college until 1855. In 1855 he accepted the Chair of Mathematics at the University of Göttingen left vacant by the death of Gauss. Dirichlet died in Göttingen in 1859, after having suffered a heart attack the year before. His brain is preserved, along with that of Gauss, in the department of physiology at the University of Göttingen.

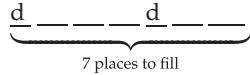
6.5. Generalized permutations, distributions and the multinomial theorem

The permutation formula $P(n, r)$, given in Section 6.1 after Example 6.1.5, allows us to count the number of ways to select and arrange *different* or *distinguishable* objects. If, however, some of these objects are indistinguishable from each other, e.g. identical copies of the same book, the formula is no longer applicable. In this section we will generalize the notion of a permutation to cover the cases where indistinguishable objects are to be arranged and will develop appropriate counting techniques for these generalized permutations. We will also show how to use the reasoning behind these techniques in an apparently different situation, namely when we want to count the number of ways to distribute different objects to different recipients, with specified numbers of objects going to each recipient. Using these techniques we will then generalize the binomial theorem.

6.5.1. Arranging nondistinct objects

Consider the problem: determine the number of different arrangements of the letters in the word *address*. Had we been asked to do this for the word *adres* (the Dutch or Afrikaans

equivalent of address) the answer would have been immediate: $5!$. However, the presence of 2 d's and 2 s's causes a problem. One possible approach to solve it is to proceed as follows. Think that we need to fill 7 places with the letters from *address*. We first select 2 places for the d's:



There are $\binom{7}{2}$ ways to do that. Then pick 2 places for the s's from the remaining 5 places in $\binom{5}{2}$ ways. Then we can place the a in $\binom{3}{1}$ ways, then the r in $\binom{2}{1}$ ways and finally we place the e in $\binom{1}{1}$ way. Hence, there are

$$\binom{7}{2} \binom{5}{2} \binom{3}{1} \binom{2}{1} \binom{1}{1} = 21 \times 10 \times 3 \times 2 \times 1 = 1260$$

ways to perform the whole procedure. As expected this is significantly less than the number of ways to arrange 7 distinct letters, which is of course $7! = 5040$.

We can now generalize the argument above to obtain the following theorem.

Theorem 6.5.1 *Given a collection of n objects, consisting of r_i indistinguishable objects of type i , for each $1 \leq i \leq m$, such that $r_1 + r_2 + \dots + r_m = n$, the number of different arrangements of these n objects is*

$$P(n; r_1, r_2, \dots, r_m) = \binom{n}{r_1} \binom{n - r_1}{r_2} \binom{n - r_1 - r_2}{r_3} \dots \binom{n - r_1 - r_2 - \dots - r_{m-1}}{r_m}.$$

The above formula is equivalent to

$$P(n; r_1, r_2, \dots, r_m) = \frac{n!}{r_1! r_2! r_3! \dots r_m!}.$$

Proof: There are n places to be filled. First we select r_1 places for the objects of type 1, which can be done in $C(n, r_1)$ ways. Now $(n - r_1)$ places remain open. Out of these we select r_2 for the objects of type 2, which can be done in $C(n - r_1, r_2)$ ways, etc. Similarly, there are $C(n - r_1 - \dots - r_{i-1}, r_i)$ ways to select places for the objects of type i , for each $1 \leq i \leq m$. Then we apply the multiplication principle and conclude that there are

$$\binom{n}{r_1} \times \binom{n - r_1}{r_2} \times \binom{n - r_1 - r_2}{r_3} \times \dots \times \binom{n - r_1 - r_2 - \dots - r_{m-1}}{r_m}$$

ways to perform the entire procedure.

To obtain the second formula in Theorem 6.5.1 we expand the binomial coefficients, bearing in mind that $n - r_1 - r_2 - \dots - r_m = 0$, and cancelling factors to simplify as follows:

$$\begin{aligned}
 & \binom{n}{r_1} \times \binom{n-r_1}{r_2} \times \binom{n-r_1-r_2}{r_3} \times \cdots \times \binom{n-r_1-r_2-\cdots-r_{m-1}}{r_m} \\
 &= \frac{n!}{r_1!(n-r_1)!} \times \frac{(n-r_1)!}{r_2!(n-r_1-r_2)!} \times \frac{(n-r_1-r_2)!}{r_3!(n-r_1-r_2-r_3)!} \times \cdots \\
 &\quad \cdots \times \frac{(n-r_1-r_2-\cdots-r_{m-1})!}{r_m!(n-r_1-r_2-r_3-\cdots-r_m)!} \\
 &= \frac{n!}{r_1!r_2!r_3!\cdots r_m!}.
 \end{aligned}$$

Example 6.5.2

How many ways are there to roll a die 7 times to obtain a sequence of outcomes that contains 3 ones, 2 fours and 2 sixes?

Solution

$$P(7; 3, 2, 2) = \frac{7!}{3!2!2!} = 210 \text{ ways.}$$

Example 6.5.3

How many ways are there to arrange the letters in the word BANANA so that the two N's are not consecutive?

Solution

It is easier to count the complement of what we want, namely the arrangements in which the N's are consecutive. We simply "glue" the two N's together, i.e. we consider NN as one object and arrange the objects B, A, A, A, NN. There are $P(5; 1, 3, 1)$ ways to do this. Subtracting this from the total number of arrangements of the letters in BANANA we find that there are

$$P(6; 1, 3, 2) - P(5; 1, 3, 1) = \frac{6!}{1!3!2!} - \frac{5!}{1!3!1!} = 60 - 20 = 40$$

ways to arrange the letters in BANANA so that the two N's are not consecutive.

6.5.2. Distributions

Suppose n distinct objects are to be distributed into m distinct boxes. How many ways are there to do this? We can simply line up the n objects and then decide for each one into which box it is to go. For each object there are m choices, so there are altogether m^n ways of completing this process.

Now let us add the restriction that r_i objects must go into box i , for each $1 \leq i \leq m$, where $r_1 + r_2 + \dots + r_m = n$. Then we begin by selecting r_1 objects for box 1 (in $\binom{n}{r_1}$ ways), then selecting r_2 objects for box 2 (in $\binom{n-r_1}{r_2}$ ways) and so on. Eventually, we find that there are

$$\binom{n}{r_1} \binom{n-r_1}{r_2} \binom{n-r_1-r_2}{r_3} \dots \binom{n-r_1-r_2-\dots-r_{m-1}}{r_m}$$

ways in which to complete this procedure. Notice the similarity with our reasoning in the previous subsection when we were calculating generalized permutations. Then we were selecting prescribed numbers of distinguishable positions in which to place objects of the same type, whereas now we are selecting prescribed numbers of distinct objects to be put in a given box. Thus, we have proved the following result.

Theorem 6.5.4 *The number of ways to distribute n distinct objects into m distinct boxes if r_i objects are to go into box i , for $1 \leq i \leq m$, where $r_1 + r_2 + \dots + r_m = n$, is equal to*

$$C(k; r_1, r_2, r_3, \dots, r_m)$$

$$= \binom{n}{r_1} \binom{n-r_1}{r_2} \binom{n-r_1-r_2}{r_3} \dots \binom{n-r_1-r_2-\dots-r_{m-1}}{r_m} = \frac{n!}{r_1!r_2!r_3!\dots r_m!}.$$

Note that it is a consequence of Theorem 6.5.4 is that $C(k; r_1, r_2, r_3, \dots, r_m) = P(k; r_1, r_2, r_3, \dots, r_m)$. We will use both these notations because the situations in which we apply them (namely distributions and arrangements of nondistinct objects, respectively) are conceptually different, even though the solutions are calculated using the same formula.

Example 6.5.5

In how many ways can a city council of 30 be divided into 5 committees, with each member serving on exactly one committee?

In addition, how many ways if each committee must consist of exactly 6 members?

Solution

As stated first, the problem poses no restrictions on the size of the committees, i.e. a committee may also be empty. Hence, there are 5^{30} ways to distribute the members between the committees.

With the restriction that each committee must consist of exactly 6 members, according to Theorem 6.5.4 there are $P(30; 6, 6, 6, 6, 6) = 30!/(6!)^5$ ways to distribute the members into the 5 committees.

Example 6.5.6

In how many ways can 7 different books be distributed to 5 people if

1. no restrictions apply?
2. each person must receive at least one book?

Solution

1. For each of the 7 books there are 5 choices as to who gets it. Thus there are 5^7 ways to distribute the books.
2. The easiest way to count the distributions in which each person receives at least one book is to break it down into cases as follows: we can either have one person receiving three books and the others receiving one each, or we could have two people receiving two books each and the others receiving one each. Since these cases are disjoint and exhaustive we may apply the addition principle. In the first case we must first pick the person who will receive the three books and in the second case we must first pick the two people who will receive two books each. Thus, the number of ways to do the distribution is equal to

$$\begin{aligned} & \binom{5}{1} \times C(7; 3, 1, 1, 1, 1) + \binom{5}{2} \times C(7; 2, 2, 1, 1, 1) \\ &= 5 \times \frac{7!}{3!(1!)^4} + 10 \times \frac{7!}{(2!)^2(1!)^3} = 16\,800. \end{aligned}$$

(An alternative “solution” one might be tempted to try is the following: give each person one book to ensure that the condition is met, and then distribute the remaining 2 books arbitrarily. This gives $\left(\binom{7}{1} \times \binom{6}{1} \times \dots \times \binom{3}{1}\right) \times 5^2 = \frac{7!}{2!} \times 5^2 = 63\,000$, which is much larger than the answer obtained above. This is because we are counting distributions more than once.)

6.5.3. The multinomial theorem

The binomial theorem tells us how to expand expressions of the form $(x_1 + x_2)^n$. We now want to generalize it to powers of sums of more than two terms, e.g. $(x_1 + x_2 + x_3)^n$. The technique we developed in the previous subsection for counting distributions now comes to our aid. Let us begin by considering the particular case of $(x_1 + x_2 + x_3)^4$. Naively multiplying this out we will get:

$$\begin{aligned} & (x_1 + x_2 + x_3)^4 \\ &= (x_1 + x_2 + x_3) \times (x_1 + x_2 + x_3) \times (x_1 + x_2 + x_3) \times (x_1 + x_2 + x_3) \\ &= x_1 x_1 x_1 x_1 + x_1 x_1 x_1 x_2 + x_1 x_1 x_1 x_3 + x_1 x_1 x_2 x_1 + x_1 x_1 x_2 x_2 + x_1 x_1 x_2 x_3 + \dots \end{aligned}$$

Each term in the last line comes about as the product of exactly one term from each pair of brackets. Therefore, there are $3^4 = 81$ such four-factor terms. The next step is to collect like

terms together. To do that effectively we need to decide how many four-factor terms equal any given four-factor term that occurs in this sum. For example, the terms equal to $x_1x_1x_2x_3$ will be exactly those containing two x_1 's, one x_2 and one x_3 . The number of ways in which we can create such a term is equal to the number of ways of selecting the x_1 from two of the four pairs of brackets, then selecting the x_2 from one of the two remaining pairs of brackets and then taking the x_3 from the only remaining pair of brackets. This gives

$$\binom{4}{2} \times \binom{2}{1} \times \binom{1}{1} = C(4; 2, 1, 1).$$

Thus, the coefficient of $x_1^2x_2x_3$ in the expansion of $(x_1 + x_2 + x_3)^4$ is equal to $C(4; 2, 1, 1)$. By similar reasoning the coefficient of $x_1^{n_1}x_2^{n_2}x_3^{n_3}$ in the expansion is given by $C(4; n_1, n_2, n_3)$ for any $n_1, n_2, n_3 \in \mathbb{N}$ with $n_1 + n_2 + n_3 = 4$.

Thus, we see the number of distributions $C(n; r_1, r_n, \dots, r_n)$, also equal to the number of generalized permutations $P(n; r_1, r_n, \dots, r_n)$, surfaces once again now. By analogy with the coefficients in the binomial theorem, for the coefficients in its generalization, called the **multinomial theorem**, we use the notation

$$\binom{n}{r_1, r_n, \dots, r_n} = \frac{n!}{r_1!r_n!\cdots r_n!},$$

known as a **multinomial coefficient**. We will henceforth use this notation interchangeably with $C(n; r_1, r_n, \dots, r_n)$ and $P(n; r_1, r_n, \dots, r_n)$.

We are now ready to state and prove the multinomial theorem in its full generality.

Theorem 6.5.7 *Let n and m be positive integers. Then*

$$\begin{aligned} (x_1 + x_2 + \cdots + x_m)^n &= \sum_{r_1+r_2+\cdots+r_m=n} \binom{n}{r_1, r_2, \dots, r_m} x_1^{r_1} x_2^{r_2} \cdots x_m^{r_m} \\ &= \sum_{r_1+r_2+\cdots+r_m=n} \frac{n!}{r_1!r_2!\cdots r_m!} x_1^{r_1} x_2^{r_2} \cdots x_m^{r_m}. \end{aligned}$$

Here the notation $\sum_{r_1+r_2+\cdots+r_m=n}$ means that the sum is taken over all non-negative integers r_1, r_2, \dots, r_m such that $r_1 + r_2 + \cdots + r_m = n$.

Proof: We give an informal proof that generalizes our argument above for computing the coefficients of $(x_1 + x_2 + x_3)^4$. A more rigorous proof, using induction on m and applying the binomial theorem, is requested in Exercise 10.

$$\begin{aligned} (x_1 + x_2 + \cdots + x_m)^n &= (x_1 + x_2 + \cdots + x_m) \times \underbrace{(x_1 + x_2 + \cdots + x_m) \times \cdots \times (x_1 + x_2 + \cdots + x_m)}_{n \text{ times}} \\ &= (x_1 + x_2 + \cdots + x_m)^1 \times (x_1 + x_2 + \cdots + x_m)^1 \times \cdots \times (x_1 + x_2 + \cdots + x_m)^1 \end{aligned}$$

When multiplying out, each term that arises comes about as the product of exactly one term from each of the n pairs of brackets. Therefore, there are m^n such n -factor terms. After collecting like terms together, each term will be of the form $x_1^{r_1}x_2^{r_2}\cdots x_m^{r_m}$ for some non-negative integers r_1, r_2, \dots, r_m such that $r_1 + r_2 + \cdots + r_m = n$. Hence, the coefficient of the term containing $x_1^{r_1}x_2^{r_2}\cdots x_m^{r_m}$ is equal to the number of ways of selecting the x_1 from r_1 of the n pairs of

brackets, then selecting the x_2 from r_2 of the $n - r_1$ remaining pairs of brackets, and so on, until we end by selecting the x_m from the remaining $n - (r_1 + r_2 + \dots + r_{m-1}) = r_m$ pairs of brackets. This gives

$$\binom{n}{r_1} \times \binom{n - r_1}{r_2} \times \binom{n - r_1 - r_2}{r_3} \times \dots \times \binom{n - r_1 - r_2 - \dots - r_{m-1}}{r_m}$$

$$= \frac{n!}{r_1! r_2! \cdots r_m!} = \binom{n}{r_1, r_2, \dots, r_m}.$$

Example 6.5.8

Let us find the coefficient of x^2yz^3 in the expansion of $(x + y + z)^6$. According to Theorem 6.5.7, the term in which x^2yz^3 appears is equal to $C(6; 2, 1, 3)x^2yz^3 = 60x^2yz^3$.

Now let us find the coefficient of x^2yz^3 in the expansion of $(3x + 4y - 7z)^6$. The term in which x^2yz^3 appears is equal to $C(6; 2, 1, 3)(3x)^2(4y)(-7z)^3 = 60 \cdot 9 \cdot 4 \cdot (-343)x^2yz^3 = -740\,880x^2yz^3$.

6.5.4. Summary

We have seen that the multinomial coefficients provide solution to at least three different counting problems, namely permutations involving indistinguishable objects, distributions of distinct objects to distinct recipients in prescribed numbers and in computing expansions of expressions of the form $(x_1 + x_2 + \dots + x_m)^n$. This is summarized in Table 6.2.

Table 6.2 Summary

Counting problem	Solution
Number of arrangements of n objects, consisting of r_i identical objects of type i , for each $1 \leq i \leq m$, where $\sum_{i=1}^m r_i = n$	$P(n; r_1, r_2, r_3, \dots, r_m)$
Number of distributions of n distinct objects to m distinct recipients	m^n
Number of distributions of n distinct objects to m distinct recipients, with r_i going to recipient i , for each $1 \leq i \leq m$, where $\sum_{i=1}^m r_i = n$	$C(n; r_1, r_2, r_3, \dots, r_m)$
Number of partitions of a set of cardinality n into m subsets S_i , with $ S_i = r_i$, for $1 \leq i \leq n$ and $\sum_{i=1}^m r_i = n$	$C(n; r_1, r_2, r_3, \dots, r_m)$
Coefficient of $x_1^{r_1} x_2^{r_2} \cdots x_m^{r_m}$, with $\sum_{i=1}^m r_i = n$ in the expansion of $(x_1 + x_2 + \cdots + x_m)^n$	$\binom{n}{r_1, r_2, r_3, \dots, r_m}$

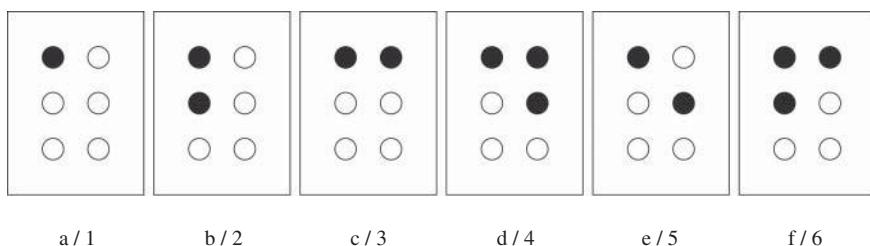
6.5.5. Exercises

- ① How many different arrangements are there of the letters in
 - (a) COMMITTEE?
 - (b) MISSISSIPPI?
- ② Suppose a bookshop has 18 books, consisting of 6 identical calculus books, 5 identical algebra books, 3 identical logic books and 4 identical combinatorics books. In how many ways can the 18 books be arranged on a shelf?
- ③ During lunch hour 12 people walk into a bakery and each orders a pie. Of these 12 people, 4 always buy steak and kidney pies, 3 always buy cornish pies, 3 always buy chicken pies and the other two buy any of the three types of pies.
 - (a) How many different sequences of pies – orders are possible?
 - (b) How many different (unordered) collections of pies are possible?
- ④ How many ways are there to distribute 20 different books to 5 people if
 - (a) there are no restrictions?
 - (b) 3 specific people get 6 books each and the other 2 people get 1 book each?
 - (c) each person gets 4 books?
- ⑤ An urn contains 5 identical purple balls, 7 identical orange balls and 3 identical magenta balls. In how many ways can the balls be drawn, one at a time, from the urn?
- ⑥ In how many ways can we divide a collection of 40 different CDs into unordered piles if
 - (a) we make four piles of ten CDs each?
 - (b) we make four piles of eight CDs each and two piles of four CDs each?
- ⑦* Suppose we distribute n distinct objects into n distinct boxes. In how many of all possible such distributions is
 - (a) no box empty?
 - (b) exactly one box empty?
 - (c) at least one box empty?
 - (d) two boxes empty?
- ⑧ What is the coefficient of $x^3y^6z^3$ in the expansion of $(x + 2y + 3z)^{12}$?
- ⑨ What is the coefficient of $x^3y^6z^{12}$ in the expansion of $(x + 2y^2 + 4z^3)^{10}$?
- ⑩* Prove the multinomial theorem for $(x_1 + x_2 + \dots + x_m)^n$, using the binomial theorem. (Hint: use induction on m . Note that the binomial theorem is just the special case of the multinomial theorem when $m = 2$.)

Braille

Braille is a tactile writing system that is used by blind or visually impaired people. It is embossed on the page with configurations of zero to six raised dots making up a Braille character, which is read by feeling it with the fingertips. It can be traced back to a system of *night writing* developed by Charles Barbier for Napoleon, who wanted a system that would enable soldiers to communicate silently and without the need for light. Barbier's original system, however, was too cumbersome to be of practical use. This system was later improved by Louis Braille after whom the current system is named.

Each Braille "cell" consists of six possible dots, of which zero or more can be embossed, arranged in two columns of three. Braille is thus a binary, six-bit system. In fact, it was historically the first writing system with a binary encoding and thus a precursor of the binary system used in modern computers. Here are the first six alphabet letters, which also double up as the numerals 1 to 6, in Braille. The black circles indicate embossed dots while the blank ones indicated the positions of dots that have not been embossed:



By the multiplication principle there are $2^6 = 64$ different Braille cells, which is more than enough to represent the 26 alphabet letters as well as the ten digits and punctuation marks. In practice, however, Braille is more than just a one-to-one correspondence between alphabet letters and cells, but rather more like shorthand with various abbreviations and contractions employed in order to speed up the reading process.

6.6. Selections and arrangements with repetition; distributions of identical objects

In this section we continue to generalize counting techniques to situations where objects might be indistinguishable.

6.6.1. Selections with repetition

Suppose that you want to order 10 ice creams and that three flavours are available, say vanilla, strawberry and chocolate. An order could be for 5 vanilla, 2 strawberry and 3 chocolate ice creams, or perhaps for 4 vanilla, 0 strawberry and 6 chocolate ice creams. How many different such orders are possible? To answer this question, suppose you write down the order on a form as follows: first you write down a number of X's representing the number of vanilla

ice creams, then put a vertical line, followed by a number of X's representing the number of strawberry ice creams, followed by a vertical line again, and finally a number of X's for the chocolate ice creams. The order forms for the two orders mentioned above would look like

$$\text{XXXXX} | \text{XX} | \text{XXX}$$

and

$$\text{XXXX} | | \text{XXXX},$$

respectively.

Clearly there is a one-to-one correspondence between such order forms and the different orders that can be placed. Therefore, all we have to do is count the number of possible order forms. Think of any such form as a 12 place sequence that consists of two |'s and 10 X's. Picking 10 of the 12 places for the X's entirely determines the sequence. Clearly there are $C(3 + 10 - 1, 10) = C(12, 10) = 66$ ways to do that.

This type of selection, which involves choosing k objects from n types with no restriction on the number of objects of a certain type is called a **k -combination with repetition from n types**. The word "repetition" indicates that objects of the same type may be selected repeatedly.

The reasoning above is generalized in the next theorem.

Theorem 6.6.1 *The number of selections with repetition of k objects (k -combinations) chosen from n types of objects is equal to $C(k + n - 1, k)$.*

Proof: This number is equal to the number of sequences of length $k + n - 1$ consisting of k X's and $n - 1$ |'s (order forms!). The number of such sequences is equal to the number of ways in which k places can be selected for the X's from the $k + n - 1$ places in all. This, in turn, is equal to $C(k + n - 1, k)$.

Corollary 6.6.2 *The number of selections with repetition of k objects chosen from n types of objects, where at least one object of each type is to be selected, is equal to $C(k - 1, k - n)$.*

Proof: Begin by selecting one object of each type. Because the objects of a given type are indistinguishable, there is only one way to do this. Now we still need to select the remaining $k - n$ objects out of the n types. By Theorem 6.6.1 there are $C((k - n) + n - 1, k - n) = C(k - 1, k - n)$ ways to do that.

Example 6.6.3

How many ways are there to select 12 pieces of fruit if apples, pears and peaches are available? How many if at least 1 fruit of each type must be selected? If at least 3 peaches have to be selected? If at least 1 fruit of each type and at least 4 pears have to be selected?

Solution

1. We are selecting 12 objects with repetition from 3 types. Therefore, by Theorem 6.6.1 there are $C(12 + 3 - 1, 12) = C(14, 12) = 91$ ways to do that.

2. Adding the restriction that at least one fruit of each type needs to be selected, we appeal to Corollary 6.6.2 to conclude that there are $C(12 - 1, 12 - 3) = C(11, 9) = 55$ ways to do that.
3. Under the restriction that at least 3 peaches are to be selected we reason as follows. First begin by selecting 3 peaches – there is one way to do this. Now we still have to pick 9 pieces of fruit out of 3 types. This can be done in $C(9 + 3 - 1, 9) = C(11, 9) = 55$ ways. Note that this is the same answer that we got in the previous case. Why is this no coincidence?
4. If at least one fruit of each type and at least 4 pears have to be selected, we can start by selecting 4 pears, 1 apple and 1 peach, which can be done in only 1 way. Selecting the remaining $12 - 4 - 1 - 1 = 6$ pieces of fruit can be done in $C(6 + 3 - 1, 6) = C(8, 6) = 28$ ways.

6.6.2. Distributions of identical objects

In Section 6.5.2 we considered the problem of distributing distinct objects to distinct recipients, either with no restrictions or with predetermined numbers going to specific recipients. Now we consider what happens when the objects to be distributed are identical. Particularly, we ask the question “*How many ways are there to distribute k identical objects to n different recipients?*” If we once again think of this in terms of “order forms”, representing a distribution with a sequence of k X’s and $n - 1$ |’s, then it is clear that the answer must be $C(k + n - 1, k)$. If we added the requirement that each recipient must receive at least one of the objects, we reason as we did in the proof of Corollary 6.6.2 to conclude that there are $C(k - 1, k - n)$ ways to do that.

Let us enshrine this as a theorem.

Theorem 6.6.4 *There are $C(k + n - 1, k)$ ways to distribute k identical objects to n distinct recipients. If, in addition, each recipient must receive at least one object, then there are $C(k - 1, k - n)$ ways to distribute the objects.*

Example 6.6.5

How many distinct non-negative integer solutions does the equation $x_1 + x_2 + x_3 + x_4 = 12$ have?

Solution

Think of this problem as follows: we have 12 objects (the 12 units) to distribute to 4 distinct recipients (the variables x_1, x_2, x_3 , and x_4). By Theorem 6.6.4 there are $C(12 + 4 - 1, 12) = P(15, 12) = 455$ ways in which this can be done. If we were asked for the number of *positive* integer solutions to the equation, we would be distributing identical objects to distinct recipients, with each recipient receiving at least 1 object. By the second part of Theorem 6.6.1 there are $C(12 - 1, 12 - 4) = C(11, 8) = 165$ ways to do this.

In general we have the following theorem.

Theorem 6.6.6 Let n and k be positive integers. Then the number of non-negative integer solutions of the equation $x_1 + x_2 + \cdots + x_n = k$ is given by $C(k + n - 1, k)$, and the number of positive integer solutions is given by $C(k - 1, k - n)$.

Example 6.6.7

How many ways are there to distribute 10 identical bones to 4 dogs if each dog must get at least 1 bone and Fifi may not receive more than 3 bones?

Solution

By Theorem 6.6.4 the number of ways to distribute 10 identical objects (the bones) to 4 distinct recipients (the dogs) with each recipient receiving at least 1 is $C(10 - 1, 10 - 4) = 84$. This includes distributions in which Fifi receives 4 or more bones, so we need to subtract the number of such distributions. We count the number of distributions in which each dog gets at least 1 bone and Fifi receives at least 4 as follows: give 4 bones to Fifi and 1 bone to each of the other dogs – since the bones are identical there is only one way in which to do this. Now distribute the remaining 3 bones to the 4 dogs – there are $C(3 + 4 - 1, 3) = C(6, 3) = 20$ ways to do this. Thus our final answer is $C(10 - 1, 10 - 4) - C(3 + 4 - 1, 3) = 84 - 20 = 64$.

6.6.3. Arrangements with repetition

Several times already we have encountered the problem of determining the number of different sequences of length r that can be formed using n different symbols, e.g. the number of binary sequences of length 3. Order is clearly important here since, for example, the sequence 010 is different from the sequence 001. However, this is not an ordinary permutation, as repetition of symbols is allowed. We reason as follows: we have three positions to fill and there are two choices for filling each of the positions. Thus, by the multiplication principle there are $2 \times 2 \times 2 = 2^3$ sequences.

More formally, an **r -permutations with repetition of n distinct objects** or an **r -permutation with replacement of n distinct objects** is an arrangement of r objects chosen from n objects where each object may be used repeatedly. The proof of the next theorem is essentially an easy application of the multiplication principle.

Theorem 6.6.8 The number of r -permutations with repetition of n distinct objects is equal to n^r .

Example 6.6.9

How many possible 7-digit telephone numbers are available?

Solution

We are dealing with a 7-permutation of 10 distinct objects – the 10 digits. By Theorem 6.6.8 there are 10^7 possible telephone numbers.

6.6.4. Summary

Table 6.3 Summary

Counting problem	Solution
<ul style="list-style-type: none"> The number of ways to select k objects from n types The number of ways to distribute k identical objects to n distinct recipients The number of non-negative integer solutions to $x_1 + x_2 + \dots + x_n = k$ 	$C(k + n - 1, k)$
<ul style="list-style-type: none"> The number of ways to select k objects from n types with at least one object of each type The number of ways to distribute k identical objects to n distinct recipients with each recipient receiving at least 1 The number of positive integer solutions to $x_1 + x_2 + \dots + x_n = k$ 	$C(k - 1, k - n)$
The number of r -permutations with repetition of n distinct objects	n^r

6.6.5. Exercises

- ① If we play with a standard deck of 52 cards, how many 13 card hands are there containing 5 hearts, 3 spades, 4 diamonds and a club?
- ② How many ways are there to distribute 26 identical sweets to 12 children if:
 - (a) there are no restrictions?
 - (b) each child must get at least one sweet?
 - (c) each child must get at least one sweet but no child may get more than three sweets?
- ③ How many non-negative integer solutions are there to the equation $z_1 + z_2 + z_3 + z_4 + z_5 = 19$?
- ④ How many positive integer solutions are there to the equation $z_1 + z_2 + z_3 + z_4 = 17$?
- ⑤ How many non-negative integer solutions are there to the equation $x_1 + x_2 + x_3 + x_4 = 21$? How many if $x_1 \geq 1, x_2 \geq 2, x_3 \geq 3$ and $x_4 \geq 4$?
- ⑥ How many ways are there to compose a 36 member choir out of sopranos, altos, tenors and bases? How many, if at least 6 singers of each group are required?
- ⑦ An urn contains 12 identical green balls and 12 identical red balls and 12 identical golden balls. How many different selections of 10 balls from the urn are there?
- ⑧ An urn contains 6 identical green balls, 6 identical red balls and 6 identical golden balls. How many ways are there to select 6 balls from the urn? How many selections of 10 balls? (Hint: count what you don't want.)

- ⑨ I want to wear one of my 3 ties every day for 6 consecutive days. In how many ways can I do this without wearing any tie more than 3 times?
- ⑩* How many 10-symbol sequences are there involving exactly 6 different symbols chosen from $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$. (The base-16 number system that uses these symbols is known as hexadecimal and is widely used in computer science.)
- ⑪* How many arrangements are possible with 4 letters chosen from SUSURRUS?
- ⑫* A bucket contains $2n$ identical maroon balls, $2n$ identical pink balls and $2n$ identical lilac balls. How many ways are there to divide the balls into two buckets, each bucket containing half, i.e. $3n$, of the balls?
- ⑬* Consider the functions from the set $\{1, 2, \dots, m\}$ to the set $\{1, 2, \dots, n\}$.
- How many are they altogether?
 - How many of them are one-to-one (injective)?
 - How many map 1 to 1 and 2 to 2?
 - How many have at least one fixed point (x is a fixed point of f if $f(x) = x$)?
 - How many have exactly one fixed point?
 - How many have exactly k fixed points, for $k > ?$
 - How many have no fixed points?



Blaise Pascal (1623–1662)

Blaise Pascal was a French mathematician, physicist, inventor, theologian and philosopher, recognized as one of the founders of the theory of probability.

Pascal was born in Clermont-Ferrand in France, where his father, Étienne Pascal, was a judge. His mother, Antoinette Begon, died when he was three years old, leaving him and his two sisters to be raised by their father. Pascal was a sickly child and was educated at home by his father, who quickly realized that he had an exceptionally gifted son. So as not to cause strain to his delicate constitution, Pascal senior omitted mathematics from the curriculum at first. Once Pascal got his first taste

of mathematics at the age of twelve, however, he had found his true passion and spent all his time on it. He took to Euclidean geometry like a fish to water and discovered and proved many of the basic theorems for himself. At sixteen he wrote a paper on conic section which he sent to Mersenne in Paris. Descartes could reportedly not believe that it was written by a sixteen year old.

In 1639 the family relocated to Rouen where Étienne had been appointed as commissioner of taxes by Cardinal Richleau. In order to help his father with the onerous tax calculations, Pascal constructed the first mechanical calculator, known as the *Pascaline*, which could perform addition and subtraction.

In 1653 he published the *Trait du Triangle Arithmétique* on what is now known as *Pascal's triangle* (see Section 6.2). In 1654 Pascal was consulted by a friend, the Chevalier de Méré, a professional gambler, regarding a fair method for dividing the stakes in game of dice that had been interrupted

before the end. This promoted a correspondence with Fermat (see the box at the end of Section 5.9 in Chapter 5) in the course of which Pascal and Fermat founded the field probability theory.

In 1654 Pascal experienced a religious conversion. From this time until his death in 1662 he did little mathematics. He had become convinced that it was sinful to spend time on, what seemed to him, the idle pursuits of science and mathematics, when there were much more urgent religious matters to attend to. He wrote on theological and philosophical subjects and published the now famous *Lettres Provinciales* and *Pensées*, regarded as masterpieces of French literature.

6.7. Recurrence relations and their solution

Solutions to instances of a counting problem can often be given very simply in terms of solutions for instances involving fewer objects. For example, suppose we wanted to know how many ways there are of forming a (ordered) row of 1 cent and 2 cent coins, to give a total of n cents. Let us denote the solution to this problem by a_n . In the case that $n = 0$, there is only one row of coins that meets the requirements, i.e. the row consisting of no coins whatsoever, and hence $a_0 = 1$. In the case that $n = 1$, again there is exactly one row of coins that meets the requirements, namely the row consisting of exactly one 1 cent coin. So $a_1 = 1$. As for a_2 , there are two possible rows of coins that add up to 2 cents, namely the row consisting of two 1 cent coins and the row consisting of one 2 cent coin. Now, for $n \geq 2$, we can construct a row adding up to n cents in two ways: either construct a row adding up to $n - 1$ cents and then add a 1 cent coin at the end or construct a row adding up to $n - 2$ cents and then add a 2 cent coin at the end. The number of such rows is thus the number of rows adding up to $n - 1$ plus the number of rows adding up to $n - 2$ or, in symbols, $a_n = a_{n-1} + a_{n-2}$. What we have done is to give a **recursive formula**, or **recurrence relation**, for a_n . In this section we will be studying recurrence relations and ways of transforming recursive formulas for a_n into nonrecursive ones – a process known as **solving recurrence relations**.

6.7.1. Recurrence relations. Fibonacci numbers

Recall that a sequence $a_0, a_1, a_2, \dots, a_n, \dots$ is a function with the set of non-negative or positive integers as domain, and which assigns to every such number n the respective term a_n . The solutions to a counting problem with an input parameter n form a sequence indexed with the parameter n , but the terms of that sequence may not be given explicitly, but only defined **recurrently**, in terms of the previous terms, as in the problem discussed above. This leads to the following notion.

Definition 6.7.1 A **recurrence relation** for a sequence a_0, a_1, a_2, \dots is a formula giving a_n in terms of one or more of the terms preceding it (i.e. in terms of $a_0, a_1, a_2, \dots, a_{n-1}$), for any n greater than some initial integer k . The values of the first few terms of the sequence needed to start computing with the recurrence relation are called the **initial values** or **initial conditions**.

The idea of a recurrence relation is that it enables the computation of terms of the sequence one after another, starting with the given first few, by repeatedly applying the recurrence relation. For instance, in the example above, where $a_n = a_{n-1} + a_{n-2}$, we only need to know the

first two terms of the sequence: $a_0 = a_1 = 1$. Thereafter, the recurrence relation yields: $a_2 = a_1 + a_0 = 2$, $a_3 = a_2 + a_1 = 3$, $a_4 = a_3 + a_2 = 5$, etc. This is actually a very famous recursively defined sequence, as we will see in the next example.

Example 6.7.2 (The Fibonacci numbers)

The following story comes from the medieval Italian mathematician Leonardo Fibonacci from Pisa (see the box at the end of this section) in his most popular book *Liber Abaci*:

A certain man put a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the second month on becomes productive?⁴

Let us denote that number by F_n , for $n \geq 0$. We find that $F_0 = 1$, $F_1 = 1$, $F_2 = 2$, $F_3 = 3$, $F_4 = 5$, $F_5 = 8$, etc. Furthermore, note that at the end of the n th month new pairs of rabbits will be born to those and only those F_{n-2} pairs that are at least $n - 2$ months old, and there will be F_{n-1} previously born rabbits. Therefore, we find that

$$F_n = F_{n-1} + F_{n-2}$$

for all $n \geq 2$. The necessary initial values are $F_0 = 1$, $F_1 = 1$.

This sequence is known as *the Fibonacci sequence* and the terms of the sequence are known as *Fibonacci numbers*.

Example 6.7.3 (The Lucas numbers)

The sequence of Lucas⁵ numbers has initial values $L_1 = 2$ and $L_2 = 1$ and recurrence relation $L_n = L_{n-1} + L_{n-2}$. Note that this is the same recursion relation as for the Fibonacci numbers, but now it generates a different sequence:

$$2, 1, 3, 4, 7, 11, \dots$$

6.7.2. Catalan numbers

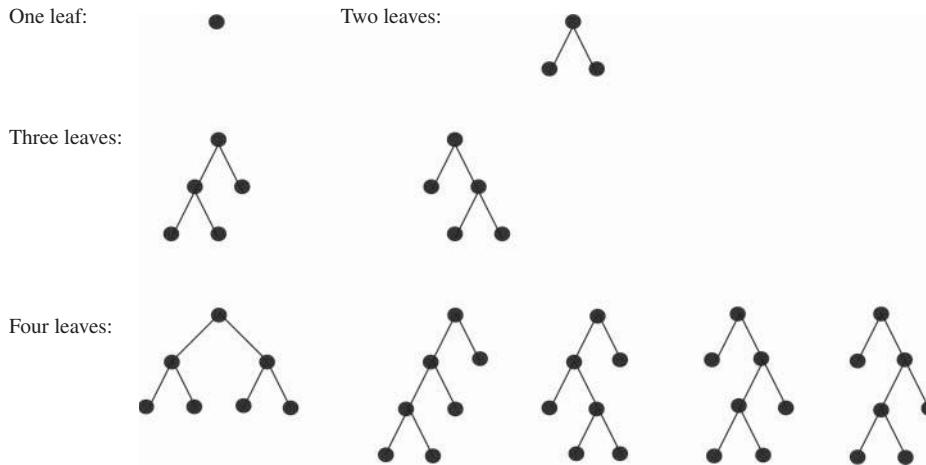
Consider the following two problems.

Complete binary trees

A **complete binary tree** is a tree in which each node has either zero or two children. (For more details on trees, see Section 7.4.) All the possible complete binary trees with 1, 2, 3 or 4 leaves are given in Figure 6.3. Denoting the number of complete binary trees with n leaves by b_n , we see that $b_1 = 1$, $b_2 = 1$, $b_3 = 2$ and $b_4 = 5$. Now suppose we know the values of b_1 up to

⁴ Encyclopdia Britannica Online Academic Edition. Encyclopdia Britannica Inc., 2013.

⁵ After the French mathematician Francois-Édouard-Anatole Lucas (1842–1891).

**Figure 6.3:** Complete binary trees with one, two, three and four leaves

b_n , and consider the root of a complete binary tree with $n + 1$ leaves. If the subtree rooted at the left child has i leaves, then the subtree rooted at the right child must have $n + 1 - i$ leaves, $1 \leq i \leq n$. Now, in this case, the left subtree can be constructed in b_i ways and the right in b_{n+1-i} ways. Therefore $b_{n+1} = b_1 b_n + b_2 b_{n-1} + \cdots + b_i b_{n+1-i} + \cdots + b_n b_1$.

Inserting brackets into associative products

Consider a product $x_1 x_2 \cdots x_n$ of n numbers. The multiplication operation is applied $(n - 1)$ times here. Accordingly, we could insert $(n - 1)$ pairs of brackets to write $x_1 x_2 \cdots x_n$ as $n - 1$ consecutive products. For two factors there is only one way to do this: $(x_1 x_2)$. For three factors, there are two ways: $(x_1 (x_2 x_3))$ and $((x_1 x_2) x_3)$. For four factors, five ways: $(x_1 (x_2 (x_3 x_4)))$, $(x_1 ((x_2 x_3) x_4))$, $((x_1 (x_2 x_3)) x_4)$, $(((x_1 x_2) x_3) x_4)$ and $((x_1 x_2) (x_3 x_4))$. Let p_n denote the number of ways to insert brackets into a product with n factors. We thus have $p_2 = 1$, $p_3 = 2$ and $p_4 = 5$. For convenience we include $p_1 = 1$: there is only one way to write the trivial product x_1 , i.e. by inserting no brackets. Any product with $n + 1$ factors can be written as a product of $n - i$ factors multiplied with one of $n + 1 - i$ factors, $1 \leq i \leq n$. Therefore $p_{n+1} = p_1 p_n + p_2 p_{n-1} + \cdots + p_i p_{n+1-i} + \cdots + p_n p_1$.

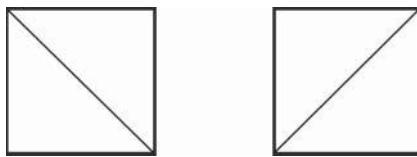
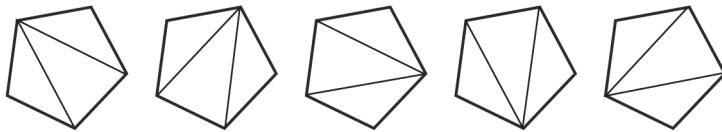
We see that the two quite different problems above have the same solution, i.e. the same sequence of numbers arises in both cases:

$$1, 1, 2, 5, 14, 42, 132, \dots$$

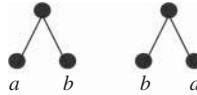
These numbers are known as the **Catalan numbers**, after the Belgian mathematician Eugène Charles Catalan (1814–1894). We will write C_n to denote the n th Catalan number starting at $n = 0$. As we have seen, C_{n+1} is given recursively in terms of C_1, \dots, C_n by the formula

$$C_{n+1} = C_1 C_n + C_2 C_{n-1} + \cdots + C_i C_{n+1-i} + \cdots + C_n C_1.$$

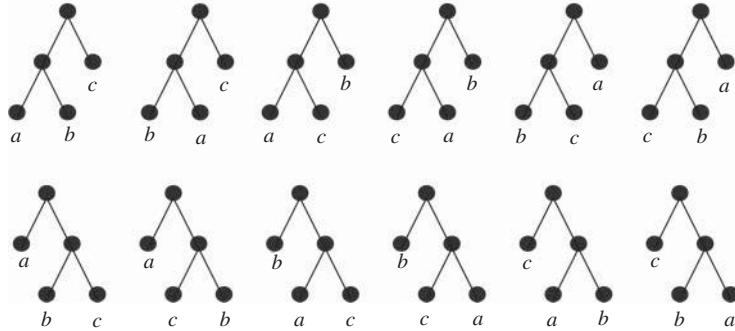
Note that $C_n = b_{n+1} = p_{n+1}$. The Catalan numbers also crop up in a variety of other settings. Another example is the number of ways to cut an $(n + 2)$ -sided polygon into triangles by

**Figure 6.4:** Dividing the square**Figure 6.5:** Dividing the pentagon

Two leaves:



Three leaves:

**Figure 6.6:** Complete binary trees with two and three named leaves

connecting its corners with $n - 1$ nonintersecting straight line segments equal to C_n . Figures 6.4 and 6.5 illustrate this for four- and five-sided polygons, respectively.

The disadvantage of the recursive formula for C_n is that we need all values of C_1, \dots, C_{n-1} to compute it. If we need to compute all the Catalan numbers one after the other, this is not a problem. However, if we only want to find C_{50} , say, it would be much more convenient to have a nonrecursive formula. To find such a formula consider a variation on the theme of computing the number of complete binary trees. Suppose we were to give names to the leaves of our complete binary trees: in that case there would be 2 different complete binary trees with 2 leaves and 12 with 3 leaves, as illustrated in Figure 6.6. How many such trees are there with $n + 1$ named leaves? If we have a tree with n leaves named x_1, x_2, \dots, x_n , we can transform it into a tree with $n + 1$ leaves named x_1, x_2, \dots, x_{n+1} as follows: take any vertex (node) in the tree and replace the subtree rooted there by inserting a new subtree consisting of a root with x_{n+1} as the right child and the subtree being replaced rooted at the left child. For example, in Figure 6.6 the first 3-leaf tree is obtained from the first 2-leaf tree by performing this operation

at the root, with $x_3 = c$. Of course, this can also be done with x_{n+1} as the right child. Since the number of leaves in a complete binary tree is always one more than the number of internal vertices, a tree with n leaves has $n + n - 1 = 2n - 1$ vertices in total. Thus, a given complete binary tree with n named leaves can be transformed into $2(2n - 1)$ complete binary trees with $n + 1$ named leaves. Now, iterating this reasoning, we see that the total number of complete binary trees with $n + 1$ named leaves is given by

$$\begin{aligned} & 2(2n - 1) \cdot 2(2n - 3) \cdot 2(2n - 5) \cdots \cdot 2(3) \cdot 2(1) \\ &= 2^n(2n - 1) \cdot (2n - 3) \cdot (2n - 5) \cdots \cdot 3 \cdot 1 \\ &= 2^n \frac{(2n)!}{2n(2n - 2)(2n - 4) \cdots 4 \cdot 2} \\ &= \frac{2^n(2n)!}{2^n n!} = \frac{(2n)!}{n!} = n! \binom{2n}{n}. \end{aligned}$$

This is not yet the value of b_{n+1} we are after, as b_{n+1} is equal to the number of complete binary trees with $n + 1$ *unnamed* leaves. However, there are $(n + 1)!$ ways to assign names to the leaves of an $(n + 1)$ -leaf tree, so we simply need to factor out the number of ways to name leaves by dividing the equation above by $(n + 1)!$, thus obtaining

$$C_n = b_{n+1} = \frac{(2n)!}{(n + 1)!(n!)^2} = \frac{1}{n + 1} \binom{2n}{n}, \quad n = 0, 1, 2, \dots.$$

6.7.3. Solving homogeneous linear recurrence relations

The example with the Catalan numbers illustrates an important fact: a function defined recursively may also sometimes be obtained in an explicit “closed form”, i.e. as a function not referring to previous terms but only involving n , as in the formula for the Catalan numbers given above. This suggests the following question: *Given a recursively defined sequence, determine a closed form of this sequence.* This is known as *solving the recurrence relation*. We will study the problem of solving a restricted class of recursive relations here. Being able to choose between recursive and closed forms is very useful as one of these forms may be more desirable than the other, depending on the context.

Definition 6.7.4 (Linear recurrence relation) *A recurrence relation of the form*

$$a_n = c_1(n)a_{n-1} + c_2(n)a_{n-2} + c_3(n)a_{n-3} + \cdots + c_m(n)a_{n-m} + f(n)$$

where c_1, c_2, \dots, c_m and f are functions of n , is called a **linear recurrence relation of order m** . In the case when $f(n) = 0$ for all n the relation is said to be **homogeneous**. In the case when the c_i 's are all constant and $c_m \neq 0$ the relation is said to have **constant coefficients and order m** .

In this section we develop methods for solving **homogeneous linear recurrence relations with constant coefficients of order m** , i.e. recurrence relations that have the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + c_3 a_{n-3} + \cdots + c_m a_{n-m}, \quad c_m \neq 0.$$

Definition 6.7.5 (Characteristic equation) *The polynomial equation*

$$x^m - c_1 x^{m-1} - c_2 x^{m-2} - \cdots - c_m = 0$$

is called the **characteristic equation** of the recurrence relation shown above.

The characteristic equation has m (not necessarily distinct) roots, $r_1, r_2, r_3, \dots, r_m$, henceforth referred to as **characteristic roots** of the recurrence relation above. These roots may be real or complex. Note, however, that since $c_m \neq 0$, they are all different from 0.

Theorem 6.7.6 *The sequence $a_n = r^n$ is a solution of the recurrence relation above if and only if r is a root of the characteristic equation in Definition 6.7.5.*

Proof: $a_n = r^n$ satisfies the recurrence relation above iff $r^n = c_1r^{n-1} + c_2r^{n-2} + \dots + c_mr^{n-m}$ (we simply substituted r^i for a_i in the recurrence relation) iff $r^m = c_1r^{m-1} + c_2r^{m-2} + \dots + c_m$ (we divided both sides by r^{m-n}) iff $r^m - c_1r^{m-1} - c_2r^{m-2} - \dots - c_m = 0$ iff r is a root of the characteristic equation.

Let (a_n) and (b_n) be sequences. We define the **addition** of the sequences component-wise: $(a_n) + (b_n)$ is $(a_n + b_n)$, i.e. $(a_1, a_2, a_3, \dots) + (b_1, b_2, b_3, \dots) = (a_1 + b_1, a_2 + b_2, a_3 + b_3, \dots)$. We also define a **constant multiple** of a sequence (a_n) to be the sequence $k(a_n) = (ka_1, ka_2, ka_3, \dots)$. These generalize to a **linear combination** of the sequences (a_n) and (b_n) : any sequence of the form $k_1(a_n) + k_2(b_n)$.

Theorem 6.7.7 *If (a_n) and (b_n) satisfy a homogeneous linear recurrence relation with constant coefficients, then every linear combination $k_1(a_n) + k_2(b_n)$ satisfies the same recurrence relation.*

Proof: It suffices to prove the claim for $(a_n) + (b_n)$ and for $k(a_n)$. See Exercise 10.

Note that the set of all linear combinations of sequences is closed under the operations of sequence addition and constant multiples as defined above, so they form a linear vector space. Theorem 6.7.7 says that the set of all solutions to a homogeneous linear recurrence relation with constant coefficients forms a subspace of that space.

Let us now consider again the m roots, $r_1, r_2, r_3, \dots, r_m$, of the characteristic equation in Definition 6.7.5. By Theorem 6.7.6, $a_n = r_i^n$ is a solution of the recurrence relation for each r_i . These solutions are called **fundamental solutions** of the recurrence relation. By repeated application of Theorem 6.7.7,

$$a_n = k_1r_1^n + k_2r_2^n + k_3r_3^n + \dots + k_mr_m^n$$

is a solution of the recurrence relation, for arbitrary constants $k_1, k_2, k_3, \dots, k_m$. Using this fact, together with the initial conditions given for the recurrence relation, we are now ready in principle to solve that recurrence relation, but just how this is done depends on whether the characteristic roots are distinct or whether some roots occur repeatedly. We consider these cases in the following two subsections.

Distinct roots

Theorem 6.7.8 *Let the roots $r_1, r_2, r_3, \dots, r_m$ of the characteristic equation in Definition 6.7.5 be distinct. Then $a_n = k_1r_1^n + k_2r_2^n + k_3r_3^n + \dots + k_mr_m^n$ is a general solution of the homogeneous linear recurrence relation, i.e. every solution of which can be expressed in this form for some choice of constants $k_1, k_2, k_3, \dots, k_m$.*

Proof: It is sufficient to show that for some choice of constants $k_1, k_2, k_3, \dots, k_m$ the following system of equations can be satisfied:

$$\begin{array}{ccccccccc} k_1 & + & k_2 & + & \cdots & + & k_m & = & b_0 \\ k_1 r_1 & + & k_2 r_2 & + & \cdots & + & k_m r_m & = & b_1 \\ k_1 r_1^2 & + & k_2 r_2^2 & + & \cdots & + & k_m r_m^2 & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ k_1 r_1^{m-1} & + & k_2 r_2^{m-1} & + & \cdots & + & k_m r_m^{m-1} & = & b_{m-1} \end{array}$$

where $a_i = b_i$, $1 \leq i \leq m$, are the initial conditions associated with the recurrence relation. It is known from linear algebra that the system has a unique solution if and only if the associated matrix of coefficients has a nonzero determinant. The matrix of coefficients of this system is known as the Vandermonde matrix and its determinant is nonzero as long as the r_i are all distinct, which is the case by assumption. Therefore, the system has a unique solution.

Furthermore, it follows from well-known facts in linear algebra that the fundamental solutions of a recurrence relation form a basis of the vector space of solutions of that relation. This completes our description of all these solutions.

Let us now illustrate the use of the method we have developed with a few examples.

Example 6.7.9

Suppose that every year the population of cats in Gotham City doubles and that at the beginning of 2000 there were 30 000 cats. How many cats were there at the beginning of 2005? How many cats will there be after n years?

Solution

Of course, this is a simple geometric progression, which we know how to solve directly, but let us do it following our general method. Regard 2000 as the 0th year and let a_n be the number of cats after n years. Then $a_0 = 30\,000$ and $a_n = 2a_{n-1}$. The characteristic equation for this recurrence relation of order 1 is $x - 2 = 0$. This has one root, namely $r = 2$. Hence we obtain one fundamental solution, $a_n = 2^n$, and the general solution $a_n = k_1 2^n$. Using the initial condition $a_0 = 30\,000$ we obtain $30\,000 = k_1 2^0$ and conclude that $k_1 = 30\,000$. Thus we have solved the recurrence relation and we find that the number of cats after n years is $a_n = 30\,000(2^n)$.

Therefore, at the beginning of 2005 there must have been $a_5 = 30\,000(2^5) = 960\,000$ cats in Gotham City!

Example 6.7.10

Recall that Fibonacci numbers were given by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ with the initial conditions $F_0 = 1$ and $F_1 = 1$. Let us solve this recurrence relation. The characteristic equation is $x^2 - x - 1 = 0$, which has two distinct roots, namely

$$\left(\frac{1}{2} + \frac{\sqrt{5}}{2}\right) \text{ and } \left(\frac{1}{2} - \frac{\sqrt{5}}{2}\right).$$

We obtain the two fundamental solutions

$$F_n = \left(\frac{1}{2} + \frac{\sqrt{5}}{2}\right)^n \text{ and } F_n = \left(\frac{1}{2} - \frac{\sqrt{5}}{2}\right)^n,$$

and the general solution

$$F_n = k_1 \left(\frac{1}{2} + \frac{\sqrt{5}}{2}\right)^n + k_2 \left(\frac{1}{2} - \frac{\sqrt{5}}{2}\right)^n.$$

Using the initial conditions we produce and solve the system of equations

$$k_1 + k_2 = 1$$

$$k_1 \left(\frac{1}{2} + \frac{\sqrt{5}}{2}\right) + k_2 \left(\frac{1}{2} - \frac{\sqrt{5}}{2}\right) = 1$$

to obtain $k_1 = 1/(\sqrt{5})$ and $k_2 = -1/(\sqrt{5})$. Hence we have

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1}{2} + \frac{\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{1}{2} - \frac{\sqrt{5}}{2}\right)^n.$$

The roots of the characteristic equation of a real recurrent sequence may turn out to be complex numbers. Recall that any complex number $a + bi$ can be written in the form $\rho(\cos \theta + i \sin \theta)$. The following theorem, which we state without proof, is often useful.

Theorem 6.7.11 *If the roots of a characteristic equation of a homogeneous linear recurrence relation of degree 2 are complex numbers $a + bi$ and $a - bi$ (i.e. the roots are complex conjugates) then the general solution of the recurrence relation is given by*

$$a_n = \rho^n (k_1 \cos n\theta + k_2 \sin n\theta),$$

where $\rho = \sqrt{a^2 + b^2}$, $\cos \theta = a/\rho$ and $\sin \theta = b/\rho$.

Proof: Left as Exercise 11.

Example 6.7.12

Consider the recurrence relation given by $a_n = 2a_{n-1} - 4a_{n-2}$ with $a_0 = 1$ and $a_1 = 2$. This has the characteristic equation $x^2 - 2x + 4 = 0$. Using the quadratic formula we find that the roots of this are $1 \pm \sqrt{3}i$. Rewriting this in polar form we find $\rho = \sqrt{1+3} = 2$ while $\sin \theta = \frac{\sqrt{3}}{2}$ and $\cos \theta = \frac{1}{2}$, which gives $\theta = \pi/3$. Hence, applying Theorem 6.7.11 we get the general solution

$$a_n = 2^n \left(k_1 \cos \frac{n\pi}{3} + k_2 \sin \frac{n\pi}{3} \right).$$

Now we have

$$a_0 = 1 = 2^0 \left(k_1 \cos \left(0 \cdot \frac{\pi}{3} \right) + k_2 \sin \left(0 \cdot \frac{\pi}{3} \right) \right) = k_1$$

and

$$\begin{aligned} a_1 &= 2 = 2^1 \left(k_1 \cos \left(1 \cdot \frac{\pi}{3} \right) + k_2 \sin \left(1 \cdot \frac{\pi}{3} \right) \right) \\ &= 2^1 \left((1) \cos \frac{\pi}{3} + k_2 \sin \frac{\pi}{3} \right) \\ &= 2 \left(\frac{1}{2} + k_2 \frac{\sqrt{3}}{2} \right) = 1 + k_2 \sqrt{3}. \end{aligned}$$

Solving we get

$$k_2 = \frac{1}{\sqrt{3}}$$

and hence

$$a_n = 2^n \left(\cos \frac{n\pi}{3} + \frac{1}{\sqrt{3}} \sin \frac{n\pi}{3} \right).$$

Repeated roots

We now turn our attention to the case in which the characteristic equation in Definition 6.7.5 has repeated roots, i.e. the m roots of the equation are not all distinct. We say that a characteristic root r has **multiplicity** j if it appears as a root of the characteristic equation $x^m - c_1x^{m-1} - c_2x^{m-2} - \dots - c_m = 0$ exactly j times. This is equivalent to $(x - r)^j$ being a divisor of the polynomial $x^m - c_1x^{m-1} - c_2x^{m-2} - \dots - c_m$ but with $(x - r)^{j+1}$ not being a divisor.

The obvious difficulty that the presence of repeated roots poses with the application of the method developed so far is that we now do not have m different fundamental solutions. There is a remedy, however. It can be shown (we will omit the proof) that if r is a characteristic root of multiplicity j , then $a_n = r^n, a_n = nr^n, a_n = n^2r^n, \dots, a_n = n^{j-1}r^n$ are linearly independent solutions, and hence can be taken as j fundamental solutions corresponding to r .

Example 6.7.13

Consider the recurrence relation $a_n = 10a_{n-1} - 25a_{n-2}$ with initial values $a_0 = 2$ and $a_1 = 5$. The characteristic equation is $x^2 - 10x + 25 = 0$. Factoring we obtain $(x - 5)^2 = 0$. This has one root of multiplicity 2, namely 5. Hence, the fundamental solutions are $a_n = 5^n$ and $a_n = n5^n$ and the general solution is $a_n = k_15^n + k_2n5^n$. Using the initial conditions we obtain the system of equations:

$$\begin{aligned} k_1 &= 2, \\ 5k_1 + 5k_2 &= 5. \end{aligned}$$

Solving we obtain $k_1 = 2$ and $k_2 = -1$ and hence the solution $a_n = 2 \cdot 5^n - n \cdot 5^n$.

6.7.4. Exercises

- ① Find a recurrence relation for the number of ways to distribute n distinct objects to 4 distinct recipients. Also, state the initial conditions.
- ② Find and solve a recurrence relation for the number of ways to count out coins, with order taken into account, until n cents are counted, if:
 - (a) 1 and 2 cent coins are used.
 - (b) 1, 2 and 5 cent coins are used.
- ③ Solve the recurrence relation for the Lucas numbers (see Example 6.7.3).
- ④ Find and solve a recurrence relation for the number of ways to make an n centimetre long row of electronic components on a circuit board. Components of types A , B and C are available. Components of types A and C take up 1 centimetre each, while those of type B take up 2 centimetres each.
- ⑤ A ternary sequence is a sequence consisting of the 0's, 1's and 2's.
 - (a) Find and solve a recurrence relation for the number of n digit ternary sequences with no equal consecutive digits.
 - (b) Find and solve a recurrence relation for the number of n digit ternary sequences where 1 does not appear directly to the left of any 2.
- ⑥ Find and solve a recurrence relation for the size a_n of an insect population in year n , which obeys the law that the year on year change is always 3 times the previous year on year change, given that $a_1 = 1000$ and $a_2 = 1200$.
- ⑦ Solve the following recurrence relations:
 - (a) $a_n = -a_{n-1} + 6a_{n-2}$ for $n > 1$, with $a_0 = 1$ and $a_1 = -2$.
 - (b) $a_n = a_{n-2}$ for $n > 1$, with $a_0 = a_1 = 1$.
 - (c) $a_n = 3a_{n-1} + 10a_{n-2}$ for $n > 2$, with $a_1 = 0$ and $a_2 = 3$.
 - (d) $a_n = 6a_{n-1} - 9a_{n-2}$ for $n > 2$, with $a_1 = 2$ and $a_2 = -2$.

(e) $a_n = 3a_{n-1} - 3a_{n-2} + a_{n-3}$ for $n > 2$, with $a_0 = a_1 = 1$ and $a_2 = 2$.

(f) $a_n = 11a_{n-1} - 39a_{n-2} + 45a_{n-3}$ for $n > 2$, with $a_0 = a_1 = a_2 = 1$.

8 Solve the following recurrence relations (the solutions involve complex numbers):

(a) $a_n = \sqrt{2}a_{n-1} - a_{n-2}$ for $n > 1$, with $a_0 = -1$, $a_1 = 1$.

(b) $a_n = -4a_{n-2}$ for $n \geq 2$, with $a_0 = 3$ and $a_1 = 4$.

(c) $a_n = \sqrt{3}a_{n-1} - a_{n-2}$ for $n > 1$, with $a_0 = 0$, $a_1 = 3$.

(d) $a_n = a_{n-1} - a_{n-2}$ for $n \geq 2$, with $a_0 = 1$ and $a_1 = 2$.

(e) $a_n = -16a_{n-2}$ for $n \geq 2$, with $a_0 = 4$ and $a_1 = 3$.

(f) $a_n = -2\sqrt{2}a_{n-1} - 4a_{n-2}$ for $n \geq 2$, with $a_0 = 1$ and $a_1 = 0$.

9* Prove that

$$C_{n+1} = \frac{2(2n+1)}{n+2} C_n$$

together with the initial condition $C_0 = 1$, which recursively defines the Catalan numbers.

10* Prove Theorem 6.7.7.

11* Prove Theorem 6.7.11.



Fibonacci (c. 1170–c. 1250)

Leonardo Pisano, known as Fibonacci, was a medieval Italian mathematician. When his father was appointed consul over the community of Pisan merchants in the North African port of Bugia, now the city of Bejaïa in Algeria, Leonardo had the opportunity to study mathematics under an Arab master. He went on to pursue his mathematical studies in various cities around the Mediterranean world. In his book *Liber Abaci* Fibonacci introduced in Europe the position-based Hindu – Arabian decimal numerical system, which we are using nowadays, instead of the awkward Roman numeral system. The book goes on to explore the advantages of this system both for practical everyday calculations and for pure mathematics. Very importantly, the Hindu – Arabian system contains a numeral 0, to denote the number zero, unlike the Roman system. The book *Liber Abaci* also famously introduces the sequence 1, 1, 2, 3, 5, 8, ... of Fibonacci numbers to the West. Fibonacci's achievements drew the attention of the Holy Roman Emperor, Frederick II, and his entourage of scholars. As a result Fibonacci was invited to appear before the imperial court. Fibonacci's book *Liber Quadratorum*, which he dedicated to Frederick, is devoted to quadratic diophantine equations, and established him, in the estimation of some, as the most important number theorist in the one and a half millennium long interval between Diophantus and Fermat.

6.8. Generating functions

Generating functions represent a very versatile, if rather abstract, counting technique. Once mastered they can be applied in various situations and can simplify the solution of many counting problems considerably. Part of the convenience and beauty of modelling counting problems with generating functions is the way in which combinatorial reasoning is replaced, and almost automated, by algebraic manipulation.

6.8.1. Introducing generating functions

One often wants to solve a family of similar counting problems, for instance the number of selections of i objects chosen in a certain way, for a whole range of possible values of i . If we denote the solution of the problem for case $i = n$ by a_n , we obtain the sequence $A = a_0, a_1, a_2, \dots$ of solutions for that family of problems.

Definition 6.8.1 *With each sequence*

$$A = a_0, a_1, a_2, \dots$$

we associate the power series

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$$

$A(x)$ is called the **generating function** of the sequence A .

Note that, in the generating function $A(x)$, a_i is the coefficient of x^i for each i . The expansion of the generating function is finite, i.e. a polynomial, just in the case that $a_i = 0$ for all $i \geq N$ for some number N . We regard generating functions as formal algebraic expressions and do not consider notions typically of importance to calculus like convergence. Indeed, for our purposes, the role of x in $A(x)$ is not that of a variable ranging over some set of numbers, but rather that of a place holder: we identify a_n by looking for the coefficient of x^n . By arranging the terms of the sequence A as the coefficients in a power series in this way, a range of algebraic manipulations are made available that, as hinted above, will mechanize some reasoning about the combinatorial problem associated with A .

By way of motivation consider $(x+1)^n$. From the binomial theorem we know that $(x+1)^n = 1 + \binom{n}{1}x + \binom{n}{2}x^2 + \dots + \binom{n}{n}x^n$. Therefore, in the expansion of $(x+1)^n$, the coefficient of x^i is $\binom{n}{i}$, that is to say, the number of ways to select i objects out of a possible n . This means that $(x+1)^n = 1 + \binom{n}{1}x + \binom{n}{2}x^2 + \dots + \binom{n}{n}x^n$ is the generating function for the sequence $A(x) = (\binom{n}{0}, \binom{n}{1}, \dots, \binom{n}{n})$ of solutions to the counting problem of selecting i objects out of a possible n .

We can add and multiply generating functions by extending the corresponding operations on polynomials as follows: if

$$A(x) = a_0 + a_1x + a_2x^2 + \dots + a_ix^i + \dots$$

and

$$B(x) = b_0 + b_1x + b_2x^2 + \dots + b_ix^i + \dots$$

we define their sum as

$$A + B = (a_0 + b_0) + (a_1 + b_1)x + (a_2 + b_2)x^2 + \dots + (a_i + b_i)x^i + \dots$$

Likewise, by analogy with multiplication of polynomials, the **Cauchy product**, $A(x) \times B(x)$, of $A(x)$ and $B(x)$ is the power series in which the coefficient of x_i is $a_0b_i + a_1b_{i-1} + a_2b_{i-2} + \cdots + a_ib_0$.

Example 6.8.2

Suppose $A(x) = 1 + 2x + 4x^2 + 8x^3 + \cdots$ and $B(x) = x + x^3 + x^5 + x^7 + \cdots$. Clearly, the coefficient a_i of x_i in $A(x)$ is equal to 2^i and the coefficient b_i of x_i in $B(x)$ is equal to 1 if i is odd and equal to 0 when i is even. From the definition of the Cauchy product, we can compute the coefficient of any power of x in $A(x) \times B(x)$. For instance, the coefficient of x^5 in $A(x) \times B(x)$ is

$$\begin{aligned} & a_0b_5 + a_1b_4 + a_2b_3 + a_3b_2 + a_4b_1 + a_5b_0 \\ &= 1 \cdot 1 + 2 \cdot 0 + 2^2 \cdot 1 + 2^3 \cdot 0 + 2^4 \cdot 1 + 2^5 \cdot 0 \\ &= 21. \end{aligned}$$

We now give a version of the multiplication principle, which we will need in order to combine generating functions when solving counting problems.

The multiplication principle for generating functions

Let us consider a procedure that consists of m different steps. Suppose that the number of ways in which step j can be performed has generating function $A_j(x)$, $1 \leq j \leq m$. If the order of the outcomes is unimportant, the generating function for the number of composite outcomes is given by $A_1(x) \times A_2(x) \times \cdots \times A_m(x)$.

We illustrate the use of generating functions for solving counting problems with several examples.

Example 6.8.3

A bowl contains 5 identical yellow, 4 identical green and 7 identical white balls. Balls are to be selected from the bowl subject to the constraints that the number of selected yellow balls is odd, the number of selected green balls is even and the number of selected white balls is at least 5. How many different selections of 10 balls from the bowl are there, subject to these constraints?

Solution

Note that there is 1 way to select 1 yellow ball, 1 way to select 3 yellow balls, 1 way to select 5 yellow balls and 0 ways to select any other number of yellow balls (only the numbers 1, 3 and 5 are allowed). Hence, the generating function for the numbers of ways to select yellow balls is $A_1(x) = x + x^3 + x^5$. Similarly, the generating function for the number of ways to select green balls is $A_2(x) = x^2 + x^4$ and the generating function for the number of ways to select white balls is $A_3(x) = x^5 + x^6 + x^7$. Thus the generating function for the number of composite outcomes is $A(x) = A_1(x)A_2(x)A_3(x) = (x + x^3 + x^5)(x^2 + x^4)(x^5 + x^6 + x^7)$. The number of ways to select 10 balls will then be the coefficient of x^{10} in $A(x)$, which, as one can easily verify, is 3.

Example 6.8.4

If 25 identical jelly beans are to be distributed among 5 children, how many ways are there to distribute the jelly beans with

1. each child receiving at least 3 jelly beans?
2. each child receiving at least 3 jelly beans and no child receiving more than 7 jelly beans?
3. each child receiving an even number of jelly beans?
4. the first child receiving either 4 or 5 jelly beans and the other four each receiving an odd number of jelly beans?

Solution

For now we will only construct a generating function for solving each of these problems. The final solutions, which will be given by the coefficient of x^{25} in each case, will be postponed until the next subsection, where we discuss methods for calculating coefficients in generating functions.

1. For each of the five children the generating function for the number of ways to receive jelly beans is $x^3 + x^4 + x^5 + \dots$. Indeed, there are 0 ways to receive 0, 1 or 2 jelly beans, as this is explicitly forbidden, and, since the jelly beans are identical, there is one way to receive 3 or any other higher number of jelly beans. Hence, by the multiplication principle for generating functions, the combined generating function is $A(x) = (x^3 + x^4 + x^5 + \dots)^5$. Note that our generating function does not stop at x^{25} . This does not matter and, as we will see, actually simplifies the calculation of coefficients.
2. In this case the generating function for the number of ways to distribute jelly beans to any particular child is $x^3 + x^4 + x^5 + x^6 + x^7$, and hence the generating function for the composite counting problem becomes $A(x) = (x^3 + x^4 + x^5 + x^6 + x^7)^5$.
3. $A(x) = (1 + x^2 + x^4 + \dots)^5$. Notice, however, that there are 0 ways in which this can be done: giving an even number of beans to each child we cannot possibly have an odd sum (namely 25). In the next section we will see that the generating function indeed reflects this fact, i.e. the coefficient of x^{25} in $A(x)$ will be 0.
4. $A(x) = (x^4 + x^5)(x + x^3 + x^5 + \dots)^4$.

Example 6.8.5

How many non-negative integer solutions are there to the equation $c_1 + c_2 + c_3 + c_4 = 25$?

Solution

We have to distribute 25 identical units to the four distinct variables c_1, c_2, c_3 and c_4 . Since the units are identical, there is exactly one way to distribute n units to variable c_i , for every $n \in \mathbb{N}$. Hence, the generating function for the number of ways to distribute n units to variable c_i is $A_i(x) = 1 + x + x^2 + \dots$. By the multiplication principle for generating functions, the generating function for the number of non-negative integer solutions of $c_1 + c_2 + c_3 + c_4 = n$ is $A(x) = (1 + x + x^2 + \dots)^4$. Specifically, the number of non-negative integer solutions to the equation $c_1 + c_2 + c_3 + c_4 = 25$ will be the coefficient of x^{25} in $A(x)$.

Notice that we could also have set $A_i(x) = 1 + x + x^2 + \dots + x^{25}$, since we do not intend to distribute more than 25 units to any variable. However, $A_i(x) = 1 + x + x^2 + \dots$ solves the more general problem of distributing n units to the variables. Moreover, as we will see below, using infinite power series rather than polynomials typically simplifies the calculation of the coefficients.

6.8.2. Computing coefficients of generating functions

In the preceding examples we mostly restricted ourselves to the construction of a generating function, but did not complete the solutions by actually calculating the required coefficient in the expansion of the constructed generating function. Here we develop methods for calculating these coefficients. The following polynomial expansions are useful for computing the coefficients of generating functions.

Theorem 6.8.6

$$\frac{1 - x^{m+1}}{1 - x} = 1 + x + x^2 + \dots + x^m$$

$$\frac{1}{1 - x} = 1 + x + x^2 + \dots$$

$$(1 + x)^n = 1 + \binom{n}{1} x + \binom{n}{2} x^2 + \dots + \binom{n}{r} x^r + \dots \binom{n}{n} x^n$$

$$(1 + x^m)^n = 1 + \binom{n}{1} x^m + \binom{n}{2} x^{2m} + \dots + \binom{n}{r} x^{rm} + \dots \binom{n}{n} x^{nm}$$

$$\frac{1}{(1 - x)^n} = 1 + \binom{1+n-1}{1} x + \binom{2+n-1}{2} x^2 + \dots + \binom{r+n-1}{r} x^r + \dots$$

Proof: It is easy (and left to the reader) to verify by multiplication that

$$(1 - x)(1 + x + x^2 + \dots + x^m) = (1 - x^{m+1})$$

and

$$(1-x)(1+x+x^2+\dots)=1,$$

from which the first and second equations immediately follow. The next equation follows from the binomial theorem, as does the following (second last) one.

It remains to show that the last equation holds. By the second equation we have

$$\frac{1}{(1-x)^n} = \left(\frac{1}{1-x}\right)^n = (1+x+x^2+\dots)^n.$$

We have to determine the coefficient of x^r in $(1+x+x^2+\dots)^n$. This is the same as the number of formal products

$$x^{e_1}x^{e_2}x^{e_3}\dots x^{e_n}$$

whose exponents add up to r . This in turn is the same as the number of non-negative integer solutions of

$$e_1 + e_2 + e_3 + \dots + e_n = r,$$

which is equal to $C(r+n-1, r)$. This proves the last equation.

Example 6.8.7

We are now able to return to Example 6.8.4 and calculate the coefficient of x^{25} in the following generating functions:

1. $A(x) = (x^3 + x^4 + x^5 + \dots)^5$
2. $A(x) = (x^3 + x^4 + x^5 + x^6 + x^7)^5$
3. $A(x) = (1 + x^2 + x^4 + \dots)^5$
4. $A(x) = (x^4 + x^5)(x + x^3 + x^5 + \dots)^4$

Solution

$$1. (x^3 + x^4 + x^5 + \dots)^5$$

$$\begin{aligned} &= x^{15}(1+x+x^2+\dots)^5 \\ &= x^{15} \frac{1}{(1-x)^5} \\ &= x^{15} \left(1 + \binom{1+5-1}{1} x + \dots + \binom{r+5-1}{r} x^r + \dots \right) \end{aligned}$$

Here the second inequality is justified by the second equation in Theorem 6.8.6 and the third by the last equation. If we multiply out in the last expression, the term containing x^{25} will be obtained by multiplying x^{15} with the term containing x^{10} in $\left(1 + \binom{1+5-1}{1} x + \dots + \binom{r+5-1}{r} x^r + \dots \right)$. Its coefficient is $1 \times C(10+5-1, 10) = C(14, 10) = 1001$.

2. $(x^3 + x^4 + \dots + x^7)^5$

$$\begin{aligned} &= x^{15}(1 + x + x^2 + \dots + x^4)^5 \\ &= x^{15} \left(\frac{1 - x^5}{1 - x} \right)^5 \\ &= x^{15}(1 - x^5)^5 \frac{1}{(1 - x)^5} \\ &= x^{15}(1 - 5x^5 + 10x^{10} - 10x^{15} + 5x^{20} - x^{25})(1 + \dots + C(r+5-1, r)x^r + \dots) \end{aligned}$$

Here the second equality follows from the first equation in Theorem 6.8.6 while the last follows from the last two equations. When multiplying out the last expression, and before collecting like terms, each term will consist of x^{15} , multiplied by one term from $(1 - 5x^5 + 10x^{10} - 10x^{15} + 5x^{20} - x^{25})$, multiplied by a term from $(1 + \dots + C(r+5-1, r)x^r + \dots)$. Terms like these in which the exponents of the x 's add up to 15 will be exactly $x^{15} \cdot 1 \cdot C(10+5-1, 10)x^{10}$, $x^{15} \cdot (-5x^5) \cdot C(5+5-1, 5)x^5$ and $x^{15} \cdot (10x^{10}) \cdot C(0+5-1, 0)x^0$. Thus, finally, the coefficient of x^{25} is equal to $C(10+5-1, 10) + (-5)C(5+5-1, 5) + 10C(0+5-1, 0) = 1001 - 5 \cdot 126 + 10 \cdot 1 = 381$.

3. $(1 + x^2 + x^4 + \dots)^5$

$$\begin{aligned} &= \left(\frac{1}{1 - x^2} \right)^5 \\ &= \frac{1}{(1 - x^2)^5} \\ &= 1 + \binom{1+5-1}{1} (x^2)^1 + \binom{2+5-1}{2} (x^2)^2 + \dots + \binom{r+5-1}{r} x^{2r} + \dots \end{aligned}$$

The first equality is justified by the first equation in Theorem 6.8.6 and the last equality by the last equation in Theorem 6.8.6. Notice that, as predicted in Example 6.8.4, the coefficient of x^{25} is 0.

4. $(x^4 + x^5)(x + x^3 + x^5 + \dots)^4$

$$\begin{aligned} &= (x^4 + x^5)x^4(1 + x^2 + x^4 + \dots)^4 \\ &= (x^8 + x^9) \frac{1}{(1 - x^2)^4} \\ &= (x^8 + x^9) \left(1 + \binom{1+4-1}{1} x^2 + \dots + \binom{r+4-1}{r} x^{2r} + \dots \right) \end{aligned}$$

The third equality is justified by the second equation in Theorem 6.8.6 and the last equality by the last equation of Theorem 6.8.6. When multiplying out the last expression, a term containing x^{25} arises only when x^9 from the first pair of brackets is multiplied with $C(8+4-1, 8)x^{24}$ from the second. Hence the coefficient of x^{25} is $C(8+4-1, 8) = 165$.

6.8.3. Exercises

- ① Find the coefficient of x^9 in $(1 + x + x^2 + \dots)^5$.
- ② Find the coefficient of x^7 in $(1 + x)^m(1 + x^2 + x^4 + x^6)$.
- ③ Find the coefficient of x^{12} in $(1 + x^2)^m$.
- ④ Find the coefficient of x^k in $(x^3 + x^4 + x^5 + \dots)^6$, $k \geq 18$.
- ⑤ Find the coefficient of x^k in $(x + x^3 + x^5)(1 + x)^n$, $k \geq 5$.
- ⑥ Find the coefficient of x^k in

$$(1 + x + x^2 + x^3 + \dots)^4 \cdot \frac{1 - x^7}{1 - x}.$$

- ⑦ Find a generating function for the number of different selections of r pencils from a pencil case containing 3 blue, 3 black, 3 red and 3 green pencils. How many different selections of 5 pencils are there?
- ⑧ Use generating functions to find the number of ways to select r objects from a collection consisting of
 - (a) 3 cricket balls, 4 rugby balls and 4 tennis balls. How many ways are there to select 5 balls?
 - (b) 3 wine glasses, 5 beer glasses and eight liqueur glasses, if at least one of each type of glass must be selected. How many ways are there to select 4 glasses?
 - (c) seven types of transistors (in unlimited supply), if an odd number of the first type and an odd number of the seventh type has to be selected. How many ways are there to select ten transistors?
- ⑨ Use generating functions to determine the number a_r of ways to distribute r identical balls into:
 - (a) 10 different boxes with at least one ball in each box.
 - (b) 5 different boxes with at most 4 balls in each box.
 - (c) 4 different boxes with between 2 and 7 balls in each box.
 - (d) 4 different boxes with fewer than 6 balls in each box.
- ⑩* A board of directors consists of 30 members. Of them 4 members are nominated for the position of chairperson. Each member must cast one anonymous vote.
 - (a) Use a generating function to determine the number of possible election outcomes.
 - (b) If each nominated member votes for himself/herself, use a generating function to determine the number of possible outcomes.
 - (c) Use a generating function to determine the number of possible election outcomes if members are allowed to abstain from voting.
- ⑪ Use a generating function to determine the number of ways to buy n pieces of fruit if avocados, bananas and papayas are available, with avocados sold in packs of 2, bananas in bunches of 6 and papayas are sold loose.
- ⑫* Use a generating function to determine the number of integers, the digits sum of which is n , if we consider only
 - (a) integers between 0 and 99 999.

- (b) 5-digit positive integers.
- ⑩ Use a generating function to find the number of non-negative integer solutions to the following equations:
- $z_1 + z_2 + z_3 = n$.
 - $z_1 + z_2 + 2z_3 = n$.
- ⑪ Find a generating function for the number of non-negative integer solutions to the equation $x_1 + 2x_2 + 3x_3 = n$.

Combinatorial games: Nim

Combinatorial games are two-player games in which players make alternate moves. Unlike poker, for example, where a player does not know the hands of others and cards are dealt randomly, players have complete information in a combinatorial game and there is no chance. In a normal play of such a game the last player to be able to make a legal move is the winner, while in a so-called misère play of the game the player forced to make the last move loses. Chess, checkers and Go are well-known examples of combinatorial games. In the theory of combinatorial games a game is called “solved” if it can be proved that, in any given play of the game, one of the players has a winning strategy.

The game of *Nim* is a simple combinatorial game. It consists of a finite number of finite piles of identical objects. A move consists of removing any number of objects from any one pile. The player who makes the last move wins. The simplest version of Nim is played with two heaps of objects. In this case, if the piles have unequal size, the first player to move has a winning strategy: in her first move she removes the number of objects from the bigger heap required to make the heaps equal. Thereafter she simply keeps on equalizing the heaps after every move by her opponent by copying his last move on the other heap. This will ensure that the game always comes to a point where there is only one nonempty heap left, whereupon she simply removes all the remaining objects in that pile and wins. On the other hand, if the game starts with two heaps of equal size, the second player has a similar winning strategy. In other words, the game is solved for two heaps.

Nim has been solved for any number of heaps, but with three or more heaps the strategy is a bit more complicated. Consider a Nim game with three heaps, consisting of two, three and four objects:



The Nim sum of a game is equal to the numbers of objects in the heaps, expressed in binary, summed using Nim addition. Nim addition, which we will denote by \oplus , is binary addition without carrying. Thus the Nim sum of the game above is $10 \oplus 11 \oplus 100 = 101$. The winning strategy is to make the Nim sum 0 after each of your moves. This is possible iff the Nim sum is not already 0. Thus if the game starts with a Nim sum unequal to zero, the first player has a winning strategy, otherwise the second player has one. A player with a winning strategy can calculate her next move as follows: add the Nim sum of the game to the number of objects in each heap using Nim addition. Take a heap the size of which decreases under this operation and decrease it to that size. In our example we get $10 \oplus 101 = 111$, $11 \oplus 101 = 110$ and $100 \oplus 101 = 1$. Only the size of the heap with four objects decreases, namely to one, so the next move is to take three from that heap.

6.9. Recurrence relations and generating functions

Now we apply generating functions to solve recurrence relations. The theory of generating functions was treated in the previous section, so the exposition here will consist mainly of a number of examples. The method we will illustrate can be used to solve any linear homogeneous recurrence relations with constant coefficients, even if the technique in Section 6.7 is quicker, but we will see that the method of recurrence relations can also be used to solve certain inhomogeneous recurrence relations.

Example 6.9.1

Consider the recurrence relation $a_n = 2a_{n-1} - 3$ for $n > 0$, with the initial condition $a_0 = 4$. This yields a sequence a_0, a_1, a_2, \dots . Let

$$f(x) = a_0 + a_1x + a_2x^2 + \dots$$

be the generating function for this sequence. Notice that we have $a_n - 2a_{n-1} + 3 = 0$ for all $n > 0$. We will now use this to rewrite the generating function $f(x)$ into a form from which we can easily read off a formula for the coefficient a_n of x^n , thus solving the recurrence relation. From the above equation we obtain

$$\begin{aligned} f(x) &= a_0 + a_1x + a_2x^2 + \dots \\ -2xf(x) &= -2a_0x - 2a_1x^2 - 2a_2x^3 - \dots \\ \frac{3}{1-x} &= 3 + 3x + 3x^2 + \dots \end{aligned}$$

Adding the left-hand sides and the right-hand sides, we obtain

$$f(x) - 2xf(x) + \frac{3}{1-x} = (a_0 + 3) + (a_1 - 2a_0 + 3)x + (a_2 - 2a_1 + 3)x^2 + \dots$$

Using the fact that $a_n - 2a_{n-1} + 3 = 0$ for all $n > 0$ and that $a_0 = 4$, we have that

$$f(x) - 2xf(x) + \frac{3}{1-x} = 4 + 3 = 7,$$

i.e.

$$f(x)(1 - 2x) = 7 - \frac{3}{1-x},$$

and so

$$f(x) = \frac{7}{1-2x} - \frac{3}{(1-x)(1-2x)}.$$

Now, using partial fraction decomposition, we obtain

$$\frac{3}{(1-x)(1-2x)} = \frac{A}{1-x} + \frac{B}{1-2x},$$

so

$$3 = A(1-2x) + B(1-x).$$

Setting $x = 1$ we find $A = -3$ and setting $x = 1/2$ we find $B = 6$. Thus we obtain

$$\begin{aligned} f(x) &= \frac{7}{1-2x} - \left[\frac{6}{1-2x} - \frac{3}{1-x} \right] \\ &= \frac{1}{1-2x} + \frac{3}{1-x} \\ &= (1 + 2x + 2^2x^2 + \dots) + 3(1 + x + x^2 + \dots) \\ &= (2^0 + 3) + (2^1 + 3)x + (2^2 + 3)x^2 + \dots \end{aligned}$$

Thus the coefficient of x^n in $f(x)$ is $2^n + 3$. Since $f(x)$ is the generating function of the sequence (a_n) , this means that $a_n = 2^n + 3$. Thus we have solved the recurrence relation. You can test the correctness of this solution by checking that it produces the same values of a_n as the recursive definition, for a few values of n .

Example 6.9.2

Consider the recurrence relation $a_n = 3a_{n-1} + n$ for $n > 0$, with $a_0 = 1$. Let $f(x) = a_0 + a_1x + a_2x^2 + \dots$ be the generating function for (a_n) . We want to exploit the fact that $a_n - 3a_{n-1} - n = 0$ for all $n > 0$ to find a closed form for $f(x)$ and solve the recurrence, like we did in Example 6.9.1. Therefore, we begin to manipulate $f(x)$:

$$\begin{array}{rcl} f(x) &=& a_0 + a_1x + a_2x^2 + \dots \\ -3xf(x) &=& -3a_0x - 3a_1x^2 - \dots \\ ? &=& 0 - 1x - 2x^2 - \dots \end{array}$$

Adding the right-hand sides above we get exactly what we want: from the second column onwards the sum of the coefficients is $a_n - 3a_{n-1} - n$, which is equal to 0. However, we do not yet have a closed form in the left-hand side for the power series $-1x - 2x^2 - 3x^3 - \dots$. We can find it by using the identity in the last of the equations in Theorem 6.8.6 in Section 6.8 for $n = 2$:

$$\begin{aligned} \frac{1}{(1-x)^2} &= 1 + \binom{1+2-1}{1}x + \binom{2+2-1}{2}x^2 + \dots + \binom{r+2-1}{r}x^r + \dots \\ &= 1 + \binom{2}{1}x + \binom{3}{2}x^2 + \dots + \binom{r+1}{r}x^r + \dots \\ &= 1 + 2x + 3x^2 + \dots + (r+1)x^r + \dots \end{aligned}$$

This is almost what we want, but the coefficients and exponents are out of sync and we have +'s where we want -'s. To fix this we simply multiply both sides by $-x$:

$$\frac{-x}{(1-x)^2} = -1x - 2x^2 - 3x^3 - \dots - (r+1)x^{r+1} - \dots$$

Now we can add both sides of the equations above at the beginning of this example to obtain

$$\begin{aligned} f(x) - 3xf(x) - \frac{x}{(1-x)^2} &= a_0 + (a_1 - 3a_0 - 1)x + (a_2 - 3a_1 - 2)x^2 + \dots \\ &= a_0 + 0x + 0x^2 + \dots \\ &= 1. \end{aligned}$$

Thus

$$\begin{aligned} f(x)(1-3x) &= 1 + \frac{x}{(1-x)^2} \\ f(x) &= \frac{1}{1-3x} + \frac{x}{(1-x)^2(1-3x)}. \end{aligned}$$

In order to find an expansion for the last term of the right-hand side above we decompose it using partial fractions:

$$\frac{x}{(1-x)^2(1-3x)} = \frac{A}{1-3x} + \frac{B}{1-x} + \frac{C}{(1-x)^2},$$

which gives

$$x = A(1-x)^2 + B(1-3x)(1-x) + C(1-3x).$$

First setting $x = 1$ we find $C = -1/2$, then setting $x = 1/3$ we find $A = 3/4$ and finally using these values and setting $x = 0$ we get $B = -1/4$. Hence

$$\begin{aligned} f(x) &= \frac{1}{1-3x} + \frac{3}{4}\frac{1}{1-3x} - \frac{1}{4}\frac{1}{1-x} - \frac{1}{2}\frac{1}{(1-x)^2} \\ &= \frac{7}{4}\frac{1}{1-3x} - \frac{1}{4}\frac{1}{1-x} - \frac{1}{2}\frac{1}{(1-x)^2}. \end{aligned}$$

Now we know that

$$\begin{aligned} \frac{1}{1-3x} &= 1 + 3x + 3^2x^2 + \dots + 3^n x^n + \dots, \\ \frac{1}{1-x} &= 1 + x + x^2 + \dots + x^n + \dots \text{ and} \\ \frac{1}{(1-x)^2} &= 1 + 2x + 3x^2 + \dots + (n+1)x^n + \dots. \end{aligned}$$

Hence, from the pair of equations above we find that the coefficient of x^n in $f(x)$ is $a_n = \frac{7}{4}(3^n) - \frac{1}{4}(1) - \frac{1}{2}(n+1) = \frac{7}{4}3^n - \frac{1}{2}n - \frac{3}{4}$.

Example 6.9.3

Consider the recurrence relation $a_k = 3a_{k-1} + 4^k$ for $k > 0$, with $a_0 = 1$. We will be guided by the fact that $a_k - 3a_{k-1} - 4^k = 0$ for $k > 0$. Let $f(x)$ be the generating function of (a_n) . Then we have

$$\begin{aligned} f(x) &= a_0 + a_1x + a_2x^2 + \dots \\ 3xf(x) &= \quad \quad 3a_0x + 3a_1x^2 + \dots \\ \frac{1}{1-4x} &= 1 + 4x + 4^2x^2 + \dots \end{aligned}$$

Adding these equations we get

$$\begin{aligned} f(x) - 3xf(x) - \frac{1}{1-4x} &= (a_0 - 1) + (a_1 - 3a_0 - 4)x + (a_2 - 3a_1 - 4^2)x^2 + \dots \\ &= a_0 - 1 = 1 - 1 = 0, \end{aligned}$$

and thus

$$\begin{aligned} f(x)(1-3x) &= \frac{1}{1-4x} \\ f(x) &= \frac{1}{(1-4x)(1-3x)}. \end{aligned}$$

Using partial fractions we decompose the right-hand side of the last equation:

$$\begin{aligned} \frac{1}{(1-4x)(1-3x)} &= \frac{A}{1-3x} + \frac{B}{1-4x} \\ 1 &= A(1-4x) + B(1-3x) \end{aligned}$$

Setting $x = 1/3$ we find $A = -3$ and setting $x = 1/4$ we find $B = 4$. Hence:

$$\begin{aligned} f(x) &= \frac{4}{1-4x} - \frac{3}{1-3x} \\ &= 4(1+4x+4^2x^2+\dots+4^nx^n+\dots) - 3(1+3x+3^2x^2+\dots+3^nx^n+\dots). \end{aligned}$$

Hence we find that the coefficient of x^n in $f(x)$ is $a_n = 4(4^n) - 3(3^n) = 4^{n+1} - 3^{n+1}$.

Lastly, let us now solve a second-order recurrence relation using generating functions.

Example 6.9.4

Suppose $a_n = 2a_{n-1} + 3a_{n-2}$ for all $n > 1$ and that $a_0 = 8$ and $a_1 = 16$. Thus, $a_n - 2a_{n-1} - 3a_{n-2} = 0$ for all $n > 1$. Let $f(x)$ be the generating function for (a_n) . Then we have

$$\begin{aligned} f(x) &= a_0 + a_1 x + a_2 x^2 + \dots \\ 2x f(x) &= 2a_0 x + 2a_1 x^2 + 2a_2 x^3 + \dots \\ 3x^2 f(x) &= 3a_0 x^2 + 3a_1 x^3 + 3a_2 x^4 + \dots \end{aligned}$$

and hence

$$\begin{aligned} f(x) - 2x f(x) - 3x^2 f(x) &= a_0 + (a_1 - 2a_0)x + (a_2 - 2a_1 - 3a_0)x^2 + (a_3 - 2a_2 - 3a_1)x^3 + \dots \\ f(x)(1 - 2x - 3x^2) &= a_0 + (a_1 - 2a_0)x \\ &= 8 + (16 - 2(8))x = 8 \\ f(x) &= \frac{8}{1 - 2x - 3x^2} \\ &= \frac{8}{(1 - 3x)(1 + x)}. \end{aligned}$$

Decomposing with partial fractions:

$$\begin{aligned} \frac{8}{(1 - 3x)(1 + x)} &= \frac{A}{1 - 3x} + \frac{B}{1 + x} \\ 8 &= A(1 + x) + B(1 - 3x). \end{aligned}$$

Setting $x = 1/3$ we find $A = 6$, and $x = -1$, $B = 2$; hence

$$\begin{aligned} f(x) &= \frac{6}{1 - 3x} + \frac{2}{1 - (-x)} \\ &= 6(1 + 3x + 3^2 x^2 + \dots + 3^n x^n + \dots) + 2(1 - x + x^2 - \dots + (-1)^n x^n + \dots). \end{aligned}$$

Hence we have $a_n = 6(3^n) + 2(-1)^n$.

6.9.1. Exercises

Solve the following recurrence relations using generating functions.

- ① $a_k = a_{k-1} + 3$ for $k > 0$ and $a_0 = 5$.
- ② $a_k = a_{k-1} + 6a_{k-2}$ for $k > 1$, $a_0 = 1$ and $a_1 = 4$.
- ③ $a_k = 2a_{k-1} + k$ for $k > 0$ and $a_0 = 3$.
- ④ $a_k = 2a_{k-1} + 3^k$ for $k > 0$ and $a_0 = 1$.
- ⑤ $a_{n+1} - a_n = 3^n$ for $n \geq 0$ and $a_0 = 0$.
- ⑥ $a_n - 3a_{n-1} = 5^{n-1}$ for $n > 0$ and $a_0 = 1$.
- ⑦ $a_n = 4a_{n-1} + 2n$ for $n \geq 1$ and $a_0 = 3$.
- ⑧ $a_n = 7a_{n-1} - 10a_{n-2}$ for $n \geq 2$, $a_0 = 1$ and $a_1 = 3$.
- ⑨ $a_{n+2} - 2a_{n+1} + a_n = 2^n$ for $n \geq 0$, $a_0 = 1$ and $a_1 = 2$.
- ⑩ $a_k = 6a_{k-1} - 9a_{k-2} + 2^k$ for $k \geq 2$, $a_0 = 1$ and $a_1 = 8$.

Fibonacci numbers and the golden ratio

The golden ratio is the ratio $\phi = a/b$ such that

$$\frac{a}{b} = \frac{a+b}{a}.$$

Thus

$$\phi = \frac{a+b}{a} = 1 + \frac{b}{a} = 1 + \frac{1}{\phi}$$

and therefore

$$\phi^2 - \phi - 1 = 0.$$

Using the quadratic formula we find that

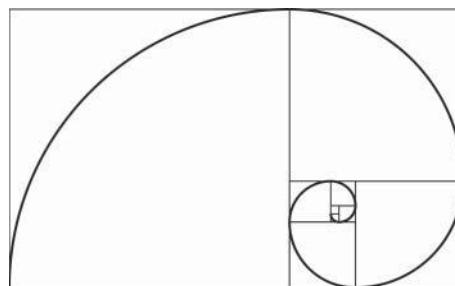
$$\phi = \frac{1 + \sqrt{5}}{2}.$$

The golden ratio appears throughout mathematics and nature, and it has been observed that humans find geometric objects exhibiting this ratio particularly pleasing. For example, if people are shown different rectangles and asked to choose those that they like the best, they tend to pick ones in which the side length stands in the golden ratio. The golden ratio can be found in the measurements of classical architecture and sculptures.

The golden ratio appears in the solution of the recurrence relation for the Fibonacci numbers:

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n.$$

The golden spiral is a logarithmic spiral given by a polar equation of the form $r = ae^{b\theta}$, where a is an arbitrary constant and $b = 2 \ln \phi / \pi$. The golden spiral is seen in nature in the shapes of various things from Nautilus shells to spiral galaxies. It is approximated by the Fibonacci spiral, illustrated below, which is obtained by arranging squares with side lengths given by the Fibonacci numbers, as shown, and drawing the inscribed quarter circles.



6.10. Application: classical discrete probability

Throughout this chapter we have developed various counting techniques. Counting objects with a given property is at the heart of computing discrete probabilities. In this section we will introduce some of the fundamental concepts of discrete probability theory and will demonstrate how the counting techniques from the previous sections apply for solving probability problems.

6.10.1. Common sense probability

Suppose we toss a normal coin. Most people would agree that the likelihood of the coin landing heads is about 50%, because this is a likelihood of one event happening (heads) out of two possible and equally likely outcomes: heads and tails. If we measure likelihood on a scale of 0 to 1, with 0 representing the likelihood of an event that can never occur and 1 the likelihood of an event that is certain to occur, the likelihood of our coin landing heads would be 0.5.

Similarly, consider the toss of a standard six-sided die.⁶ There are six possible and equally likely outcomes: one, two, three, four, five and six. The likelihood of any such outcome, e.g. "three", is therefore 1 out of 6, i.e. $\frac{1}{6}$. The likelihood of the outcome to be a number less than four is 3 out of 6, or $\frac{3}{6} = \frac{1}{2}$, as exactly three of the possible six outcomes of the toss (namely one, two and three) would satisfy this requirement. Likewise, the likelihood of an even number is also $\frac{1}{2}$. Lastly, the likelihood of the outcome to be a number between one and six is $\frac{6}{6} = 1$ and the likelihood of a number between 7 and 13 is, of course, $\frac{0}{6} = 0$.

Such likelihoods are measured by the notion of **probability**, the precise meaning of which we will formally define shortly. Before that, we need some preparation.

6.10.2. Sample spaces

Now let us abstract some important concepts from these two informal examples. In each case we considered an **experiment** or **trial** and carefully listed all the possible outcomes thereof. The set of all such outcomes is called **sample space**. Let us denote the sample space with S . We will only consider **finite sample spaces**.

Next, we considered some properties of, or restrictions on, the outcomes of our experiment, e.g. the property of the outcome of the die-toss being an even number. This corresponds to a subset of the sample space, namely the subset consisting of all outcomes that are even numbers. One can think of this as the event of the outcome being even, where **event** is simply any set of outcomes, i.e. any subset of the sample space S . Events containing exactly one outcome are called **simple events**.

Mutually exclusive events are events that have no outcomes in common, i.e. events $A, B \subseteq S$ such that $A \cap B = \emptyset$. If $A \subseteq S$ is an event, then $\bar{A} = S - A$ is called the **complementary event** of A .

Example 6.10.1

Let us return to the case of casting a die. The sample space is the set $S = \{1, 2, 3, 4, 5, 6\}$. Examples of events are $A = \{1, 3, 5\}$, $B = \{2, 4, 6\}$, $C = \{1, 2, 3, 5, 6\}$, $D = \{4\}$ and $E = \{5\}$. A and B are mutually exclusive, and so are C and D . The events D and E are simple, while the events C and D are complementary.

⁶ "Die" is the singular of "dice".

Example 6.10.2

Consider a lottery draw that consists in a random selection of 6 different numbers between 1 and 49. Thus, the sample space, S , consists of all possible draws of six numbers out of 49. Hence there are

$$C(49, 6) = \frac{49!}{43!6!} = \frac{49 \times 48 \times 47 \times 46 \times 45 \times 44}{720} = \frac{10\,068\,347\,520}{720} = 13\,983\,816$$

outcomes in S . By definition, every subset of S determines an event, but in this case some events are of much greater interest to players than others. Suppose John has a lottery ticket with numbers (3, 7, 19, 27, 39, 42). Let J_1 be the event consisting of all draws in which John has at least one number matching the draw. How many outcomes are there in J_1 ? In order to count them, we consider the complementary event of J_1 , that is the event $\overline{J_1}$ in which John has no matching numbers:

$$|\overline{J_1}| = C(49 - 6, 6) = C(43, 6) = \frac{43!}{37!6!} = \frac{4\,389\,446\,880}{720} = 6\,096\,454.$$

Thus we have $|J_1| = |S| - |\overline{J_1}| = 13\,983\,816 - 6\,096\,454 = 7\,887\,362$.

Let $J_{\geq 5}$ be the event in which John has at least five numbers right. Clearly $J_{\geq 5} = J^5 \cup J^6$, where J^5 and J^6 are the events in which John has exactly five numbers right and all six numbers right, respectively. Now $|J^5| = C(6, 5) \times C(43, 1) = 6 \times 43 = 258$ and $|J^6| = 1$ so $|J_{\geq 5}| = 258 + 1 = 259$.

Example 6.10.3

Consider a trial in which a coin is tossed and a die is cast. The sample space looks like this:

(Heads, 1)	(Heads, 2)	(Heads, 3)	(Heads, 4)	(Heads, 5)	(Heads, 6)
(Tails, 1)	(Tails, 2)	(Tails, 3)	(Tails, 4)	(Tails, 5)	(Tails, 6)

Each outcome represents a combination of a possible outcome of the tossed coin and the cast die. Some events of interest may be:

HEADS The coin comes up heads. This consists of all sample points in the first row.

SIX The die lands on a 6. This consists of all sample points in the last column.

ODD The die lands on an even number. This consists of all outcomes in the first, third and fifth columns.

TAILS \cap EVEN The coin comes up tails and the die lands on an even number. This consists of all outcomes in the bottom row in the second, fourth and sixth columns.

6.10.3. Discrete probability

Our reason for introducing sample spaces and events is to define probabilities on them. In the simplest case, when all events are equally likely, we have the following definition that goes back to the works of Laplace (see the end of Section 6.10).

Definition 6.10.4 Consider an experiment with a finite number of possible different outcomes, which are all equally likely. Then the **classical probability** of an event is defined to be the ratio of the number of possible favourable outcomes to the total number of possible outcomes of the experiment, where a favourable outcome is any outcome that belongs to the event.

Example 6.10.5

Returning to Example 6.10.1, it should be clear that the classical probability of the event A , denoted by $P(A)$, is equal to $\frac{3}{6} = \frac{1}{2}$. Similarly, $P(B) = \frac{1}{2}$, $P(C) = \frac{5}{6}$ and $P(D) = P(E) = \frac{1}{6}$. Lastly, $P(S) = 1$.

Notice that, in Example 6.10.5, $P(S) = 1$ and that, if E_1, E_2, \dots, E_n are pairwise mutually exclusive events, $P(E_1 \cup E_2 \cup \dots \cup E_n) = P(E_1) + P(E_2) + \dots + P(E_n)$. In fact these are exactly the properties that are used to define the general notion of a **probability measure**.

Definition 6.10.6 Let S be a finite sample space and P a function from the set $\mathcal{P}(S)$ of all events over S to the real interval $[0, 1]$. P is called a **probability measure** if

$$(P1) \quad P(S) = 1.$$

(P2) For any family $\{E_i \mid 1 \leq i \leq n\}$ of pairwise disjoint events it holds that

$$P\left(\bigcup_{i=1}^n E_i\right) = \sum_{i=1}^n P(E_i).$$

The pair (S, P) is called a **finite probability space**.

(P1) requires that the probability of S is 1, as we would expect, since intuitively S is certain to occur. (P2) is known as the principle of **additivity**.

Example 6.10.7

In the lottery Example 6.10.2, the lottery draw is supposed to be totally random. Hence, each selection of 6 numbers out of 49 should be equally likely. We saw that the sample space S contains 13 983 816 outcomes and, consequently, that there are 13 983 816 simple events. Each simple event then has probability $1/13\,983\,816$. This means that the probability that John wins the jackpot is 1 in 13 983 816. The probability that he has at least one number right is

$$P(J_1) = \frac{7\,887\,362}{13\,983\,816} \approx 0.564.$$

The probability that he has at least five numbers right is

$$P(J_{\geq 5}) = \frac{259}{13\,983\,816} \approx 0.0000185.$$

Example 6.10.8

Recall the combined experiment in Example 6.10.3, in which a coin is tossed and a die cast. Assuming that the coin and the die are fair, i.e. each outcome is equally likely, we have $P(E) = 1/12$ for each simple event. The probabilities of the events considered are then:

- $P(\text{HEADS}) = 1/2$.
- $P(\text{SIX}) = 2/12 = 1/6$.
- $P(\text{ODD}) = 6/12 = 1/2$.
- $P(\text{TAILS} \cap \text{EVEN}) = 3/12 = 1/4$.

Also,

$$\begin{aligned} P(\text{TAILS} \cup \text{EVEN}) &= \frac{|\text{TAILS} \cup \text{EVEN}|}{12} \\ &= \frac{|\text{TAILS}| + |\text{EVEN}| - |\text{TAILS} \cap \text{EVEN}|}{12} \\ &= \frac{6 + 6 - 3}{12} = \frac{3}{4} \end{aligned}$$

and

$$\begin{aligned} P(\text{TAILS} \setminus \text{EVEN}) &= \frac{|\text{TAILS} \setminus \text{EVEN}|}{12} \\ &= \frac{|\text{TAILS}| - |\text{TAILS} \cap \text{EVEN}|}{12} \\ &= \frac{6 - 3}{12} = \frac{1}{4}. \end{aligned}$$

6.10.4. Properties of probability measures

We will now draw some consequences from Definition 6.10.6, which will elucidate the nature of probability spaces and facilitate the computation of probabilities. As a first consequence of Definition 6.10.6, we have the following proposition, which lists some useful properties of probability measures.

Proposition 6.10.9 *Let (S, P) be a finite probability space. Then, for all events A and B and the empty events \emptyset we have:*

$$P(\emptyset) = 0$$

$$P(\overline{A}) = 1 - P(A)$$

$$P(A \setminus B) = P(A) - P(A \cap B)$$

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

If $A \subseteq B$, then $P(A) \leq P(B)$

Proof: For the first equation, first note that $S \cap \emptyset = \emptyset$. Therefore, by axiom **(P2)**, $P(S) + P(\emptyset) = P(S \cup \emptyset)$. Further, $S \cup \emptyset = S$ and $P(S) = 1$ by axiom **(P1)** and hence $1 + P(\emptyset) = 1$, so $P(\emptyset) = 0$.

For the second equation, note that $1 = P(S) = P(A \cup \overline{A}) = P(A) + P(\overline{A})$, where the first equality holds by **(P1)** and the last equality follows from **(P2)** and the fact that $A \cap \overline{A} = \emptyset$. Thus $1 = P(A) + P(\overline{A})$ and hence $P(\overline{A}) = 1 - P(A)$.

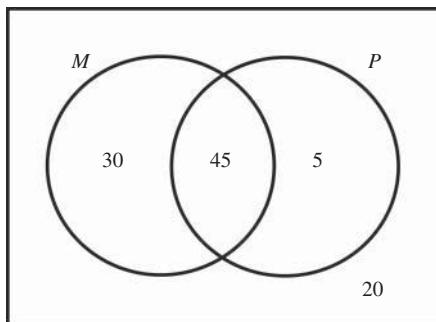
The proof of the third equation is left as Exercise 12.

For the fourth equation, we begin by noting that $A \cup B = (A \setminus B) \cup (B \setminus A) \cup (A \cap B)$ and that the sets $(A \setminus B)$, $(B \setminus A)$ and $(A \cap B)$ are pairwise disjoint. Hence, we may apply the axiom of additivity and obtain $P(A \cup B) = P(A \setminus B) + P(B \setminus A) + P(A \cap B) = P(A) - P(A \cap B) + P(B) - P(B \cap A) + P(A \cap B) = P(A) + P(B) - P(A \cap B)$, where the second equality follows from the third equation.

The proof of the fifth equation is left as Exercise 13.

Example 6.10.10

In Example 6.3.3 we considered a sample of 100 university students of which 75 take mathematics, 50 take physics and 45 take both. From this information, using the inclusion – exclusion principle (Section 6.3), we were able to extrapolate to the following Venn diagram:



Suppose we want to know how likely it is that a randomly selected student from among the 100 takes mathematics. The natural way to state this question in terms of the probability theory would be to take our sample space S to be the set of all 100 students. As we have no reason to think that any student is more likely to be selected than another, we can use classical probability, and hence $P(\text{Math}) = \frac{75}{100} = 0.75$.

The following proposition allows us to calculate the probability of an event by splitting it up according to a partition of the probability space.

Proposition 6.10.11 Let (S, P) be a finite probability space and suppose that $E_1, E_2, \dots, E_n \subseteq S$ form a partition of S , i.e. E_1, E_2, \dots, E_n are pairwise disjoint events such that $E_1 \cup E_2 \cup \dots \cup E_n = S$. Then for any event $S \subseteq S$ it holds that

$$P(A) = P(A \cap E_1) + P(A \cap E_2) + \dots + P(A \cap E_n).$$

Proof: The proof is left as Exercise 14.

6.10.5. Conditional probability and independent events

We now turn to questions of the type: “What is the probability of event A , given that event B has occurred”. For example, suppose that two players, Alice and Bob, are playing a card game. Alice is dealt a card from a standard 52 card deck. The probability that she has been dealt a heart is, of course, $\frac{13}{52} = \frac{1}{4}$. Respectively, the probability that her card is the Queen of hearts is $\frac{1}{52}$. However, suppose that Bob got a glimpse of the card she was dealt. He did not see enough to know which card it was, all he can tell was that it was a red card. What probability should he now assign to the event that she got a heart? Two of the suits, i.e. 13 + 13 of the cards, are red, so the probability is $\frac{13}{13+13} = \frac{1}{2}$. Respectively, the probability that Alice’s card is the Queen of hearts, given that it is red, is $\frac{1}{26}$.

What happened? We restricted the sample space to the red cards and calculated the ratio of the number of suitable outcomes (hearts) which are in the restricted sample space to the total number of outcomes in this restricted sample space.

Similarly, in Example 6.10.10 we might ask the question: how likely is it that a student who takes physics also takes mathematics? One way to approach this is to restrict the sample space to those 50 students who take physics. We are given that 45 of them take mathematics. Hence, it would be reasonable to assign the probability $\frac{45}{50} = 0.9$ to the event that a (randomly chosen) student who takes physics, also takes mathematics. The next definition formalizes these intuitions.

Definition 6.10.12 The conditional probability of an event A relative to an event B is denoted by $P(A|B)$, and is given by

$$P(A|B) = \frac{P(A \cap B)}{P(B)},$$

provided $P(B) > 0$. In the case that $P(B) = 0$, the conditional probability $P(A|B)$ is undefined.

Example 6.10.13

Students gain admission to the course Probability Theory B if, and only if, they have passed Probability Theory A. The probability that a student passes Probability Theory A is $\frac{2}{3}$, while the probability that (s)he passes both A and B is $\frac{1}{2}$. What is the probability that a student will pass Probability Theory B, given the fact that (s)he has passed Probability Theory A?

To model this situation we can construct a probability space (S, P) with $S = \{00, 10, 11\}$ containing the three outcomes “pass neither”, “pass only A” and “pass both”. (Note that “pass only B” is impossible, so we do not list it in S .) Let A and B denote the events “pass A” and “pass B”, respectively. From the given information we have $P(A) = P(\{10, 11\}) = \frac{2}{3}$ and $P(B) = P(\{11\}) = \frac{1}{2}$. Therefore the probability that a student passes B, given that (s)he has passed A, is

$$P(B|A) = \frac{P(B \cap A)}{P(A)} = \frac{P(\{11\})}{P(A)} = \frac{\frac{1}{2}}{\frac{2}{3}} = \frac{3}{4} = 0.75.$$

The probability of passing B, given that A has not been passed, is

$$P(B|\bar{A}) = \frac{P(B \cap \bar{A})}{P(\bar{A})} = \frac{P(\emptyset)}{1 - P(A)} = \frac{0}{1 - \frac{2}{3}} = 0,$$

as one would expect.

As an immediate consequence of Definition 6.10.12 we have that

$$P(A \cap B) = P(B) \cdot P(A|B),$$

provided that $P(B) > 0$. This generalizes to any finite number of events A_1, A_2, \dots, A_n , as follows.

Proposition 6.10.14 *Let (S, P) be a finite probability space. Then, for all events A_1, A_2, \dots, A_n such that $P(A_1 \cap A_2 \cap \dots \cap A_n) > 0$,*

$$\begin{aligned} & P(A_1 \cap A_2 \cap \dots \cap A_n) \\ &= P(A_1)P(A_2|A_1)P(A_3|A_1 \cap A_2) \cdots P(A_n|A_1 \cap A_2 \cap \dots \cap A_{n-1}). \end{aligned}$$

Proof: By the last equation in Proposition 6.10.9 we have $P(A_1) \geq P(A_1 \cap A_2) \geq \dots \geq P(A_1 \cap A_2 \cap \dots \cap A_{n-1}) > 0$, and hence the conditional probabilities $P(A_2|A_1)$, $P(A_3|A_1 \cap A_2)$, ... $P(A_n|A_1 \cap A_2 \cap \dots \cap A_{n-1})$ are defined. Now

$$\begin{aligned} & P(A_1)P(A_2|A_1)P(A_3|A_1 \cap A_2) \cdots P(A_n|A_1 \cap A_2 \cap \dots \cap A_{n-1}) \\ &= \frac{P(A_1)}{P(S)} \frac{P(A_1 \cap A_2)}{P(A_1)} \frac{P(A_1 \cap A_2 \cap A_3)}{P(A_1 \cap A_2)} \cdots \frac{P(A_1 \cap A_2 \cap \dots \cap A_n)}{P(A_1 \cap A_2 \cap \dots \cap A_{n-1})} \\ &= \frac{P(A_1 \cap A_2 \cap \dots \cap A_n)}{P(S)} \\ &= P(A_1 \cap A_2 \cap \dots \cap A_n). \end{aligned}$$

Example 6.10.15

Suppose a box contains a red ball and a blue ball. We consider a procedure of repeatedly drawing a ball from the box and replacing it back. Moreover, whenever a red ball is drawn it is replaced and another red ball is added to the box. Let us determine the probability that for the first 20 rounds only red balls are drawn. In other words, we are looking for $P(R_1 \cap R_2 \cap \dots \cap R_{20})$, where R_i is the event that in round i a red ball is drawn. Then

$$\begin{aligned} P(R_1 \cap R_2 \cap \dots \cap R_{20}) \\ = P(R_1)P(R_2|R_1)P(R_3|R_1 \cap R_2)\dots P(R_{20}|R_1 \cap \dots \cap R_{19}). \end{aligned}$$

Now $P(R_2|R_1) = \frac{2}{3}$, since after R_1 has occurred, there are three ball in the box, two of which are red. Similarly, $P(R_3|R_1 \cap R_2) = \frac{3}{4}$ and in general $R_i = i/(i+1)$. So the probability we are looking for is

$$\frac{1}{2} \cdot \frac{2}{3} \cdot \frac{3}{4} \cdots \frac{20}{21} = \frac{20!}{21!} = \frac{1}{21}.$$

All general results on probabilities hold for conditional probabilities, as the following “conditional” version of Theorem 6.10.9 illustrates.

Proposition 6.10.16 *Let (S, P) be a finite probability space; then, for all events A, B and C ,*

$$\begin{aligned} P(\bar{A}|C) &= 1 - P(A|C) \\ P(A \setminus B|C) &= P(A|C) - P(A \cap B|C) \\ P(A \cup B|C) &= P(A|C) + P(B|C) - P(A \cap B|C). \end{aligned}$$

Proof: The proof is left as Exercise 19.

Independent events

Intuitively, two events are independent if the occurrence of the one make the occurrence of the other nor more likely, nor less likely. For example, consider the experiment of drawing a single card from a standard deck, and the events H and F of the card being a heart and a face card, respectively. Intuitively, since the face cards are evenly distributed among the suits, these events should be independent. Indeed notice that

$$P(H|F) = \frac{P(H \cap F)}{P(F)} = \frac{\frac{3}{52}}{\frac{12}{52}} = \frac{1}{4} = \frac{13}{52} = P(H)$$

and

$$P(F|H) = \frac{P(F \cap H)}{P(H)} = \frac{\frac{3}{52}}{\frac{13}{52}} = \frac{3}{13} = \frac{12}{52} = P(F).$$

This is not something particular to this example: saying that $P(H|F) = P(H)$ and $(F|H) = P(F)$ is the same as saying that $P(H) = \frac{P(H \cap F)}{P(F)}$ and $P(F) = \frac{P(F \cap H)}{P(H)}$, which is equivalent to saying that $P(H \cap F) = P(H) \cdot P(F)$. We will turn this observation into a formal definition of independence.

Definition 6.10.17 Two events A and B are called **independent** if $P(A \cap B) = P(A) \cdot P(B)$.

Example 6.10.18

A coin is tossed and a die is cast. What is the probability that the outcome is heads and a six?

Solution

Let A be the event that the coin lands heads and let B be the event that the die lands on six. We are interested in $P(A \cap B)$, but since the outcome of the coin toss is in no way influenced by the outcome of the cast of the die and vice versa, these events are independent, and hence $P(A \cap B) = P(A) \cdot P(B) = \frac{1}{2} \cdot \frac{1}{6} = \frac{1}{12}$.

We can generalize the notion of independence to more than two events.

Definition 6.10.19 Events A_1, \dots, A_n are **independent** if for all $2 \leq k \leq n$ and all families of k sets A_{i_1}, \dots, A_{i_k} from among A_1, \dots, A_n it holds that

$$P(A_{i_1} \cap \dots \cap A_{i_k}) = P(A_{i_1}) \cdot \dots \cdot P(A_{i_k}).$$

If this holds only for $k = 2$, we call the events **pairwise independent**.

Example 6.10.20

Suppose the 26 letters of the alphabet are written on cards which are placed in a box. Then cards are drawn, one at a time, with replacement. What is the probability that the first 26 draws produces the sequence of the 26 alphabet letters in order?

Solution

Let A_i be the event that the i th letter of the alphabet is drawn in the i th draw. Since the draws are with replacement, the outcomes of the subsequent draws do not affect each other, so the events A_i are independent. Hence

$$P(A_1 \cap A_2 \cap \dots \cap A_{26}) = P(A_1) \cdot P(A_2) \cdot \dots \cdot P(A_{26}) = \left(\frac{1}{26}\right)^{26} \approx 1.6244 \times 10^{-37}.$$

This is a very small probability and, indeed, one would be very surprised to see the 26 letters of the alphabet appearing in order. Notice, however, that there is nothing special about this particular sequence: any other sequence of letters has exactly the same probability of being drawn!

Suppose now that the drawing proceeded without replacement. Now the events A_i are no longer independent, so we have to use Proposition 6.10.14:

$$\begin{aligned}
 & P(A_1 \cap A_2 \cap \cdots \cap A_{26}) \\
 &= P(A_1)P(A_2|A_1)P(A_3|A_1 \cap A_2) \cdots P(A_{26}|A_1 \cap \cdots \cap A_{25}) \\
 &= \frac{1}{26} \cdot \frac{1}{25} \cdot \frac{1}{24} \cdot \cdots \cdot \frac{1}{1} \\
 &= \frac{1}{26!} \approx 2.4796 \times 10^{-27}.
 \end{aligned}$$

6.10.6. Exercises

- ❶ A lottery ticket consists of six numbers between 1 and 49. A holder of a ticket wins the jackpot if the numbers match the six randomly drawn numbers. Any ticket with at least three matching numbers from the draw also qualifies for some prize money.
 - (a) How many different lottery tickets are there?
 - (b) How many different lottery tickets do you have to buy in order to have at least a chance of one in a million of winning the jackpot?
 - (c) How many different lottery tickets are there in which at least three numbers match numbers in the draw?
 - (d) How many different lottery tickets do you have to buy in order to have at least a chance of one in a million of winning some prize money?
 - (e) How many different lottery tickets are there in which no number matches any of the six in the draw?
- ❷ A patient is tested for deficiencies in vitamins A, B and C. Let E be the event “the patient is deficient in at least two of the vitamins” and let F be the event “the patient has no deficiencies”. Describe the following events in words.
 - (a) $E \cap F$
 - (b) $E \cup F$
 - (c) $\overline{E} \cap \overline{F}$
- ❸ What is the probability that an integer between 1 and 10 000 has exactly one 5 and one 4 among its digits? What is the sample space?
- ❹ What is the probability that in a random n -digit binary sequence there are no pairs of consecutive digits that are the same?
- ❺ What is the probability that a 5-card poker hand has the following?
 - (a) A Royal Flush, i.e. an Ace, King, Queen, Jack and Ten of the same suit.
 - (b) A Flush, i.e. any 5 cards of the same suit.

- (c) 4 aces.
 - (d) 4 of a kind, i.e. 4 cards of the same value, e.g. 4 Eights or 4 Queens.
 - (e) A Straight, i.e. a set of 5 consecutive values.
 - (f) A Straight that is not a Flush.
 - (g) The same number of hearts and diamonds.
 - (h) More spades than hearts.
- ⑥ What is the probability that 2 people out of a meeting of 30 people have the same birthday?
- ⑦ What is the probability that an arrangement of the letters in the word ALTITUDE has:
- (a) no consecutive vowels?
 - (b) at least 2 consecutive vowels?
 - (c) at least 3 consecutive vowels?
- ⑧ What is the probability that in an arrangement of 7 letters A, 6 letters B and 5 letters C:
- (a) the first A precedes the first B?
 - (b) the first A precedes the first B and the first B precedes the first C?
 - (c) all A's precede all B's?
- ⑨ 50 butterflies are captured from a previously unexplored rainforest. 20 of these butterflies have stripes, 31 have spots and 14 have dots. 16 have stripes and spots, 9 have spots and dots and 5 have all three features. 12 of the 50 butterflies have none of these features.
- (a) What is the probability that a butterfly randomly selected from the sample has stripes and dots?
 - (b) What is the probability that a butterfly randomly selected from the sample has at least two of these three features?
- ⑩ Suppose 12 distinct objects are distributed into 12 different boxes. What is the probability that
- (a) no box is empty?
 - (b) exactly one box is empty?
 - (c) at least one box is empty?
 - (d) two boxes are empty?
- ⑪ What is the probability that, when 26 shirts are distributed into 5 washing machines, that at least one washing machine contains 6 shirts or more?
- ⑫ Prove the third equation from Proposition 6.10.9.
- ⑬ Prove the last equation from Proposition 6.10.9.
- ⑭ Prove Proposition 6.10.11 and then generalize it to a partition consisting of countably many sets.

- ⑯ Two archers are shooting at a target. The probability that, with any given shot, the first archer hits the target is 0.6, while the probability that the second archer hits it is 0.75. A total of 5 shots are fired, 3 by the first archer and 2 by the second. What is the probability that
- all 5 shots hit the target?
 - no shots hit the target?
 - at least one shot hits the target?
- ⑰ The 20 volumes of an encyclopedia are arranged randomly on a shelf. What is the probability that they will be in numerical order?
- ⑱ An urn contains 12 identical black balls and 10 identical white balls. Balls are drawn two at a time, for 10 rounds. What is the probability that in each of the 10 rounds a black and a white ball will be drawn?
- ⑲ I'm trying to guess someone's age. If I know that she is between 30 and 39 years old, what is the probability that
- I will get it right on the third guess?
 - I will get it right in three or fewer guesses?
- ⑳ Prove the items in all three equations from Proposition 6.10.16.
- ㉑ A student passes Discrete Mathematics if he passes Probability Theory or if he passes Propositional Logic and First-Order Logic. The probability that he will pass Probability Theory is 0.6, while the probabilities that he will pass Propositional and First-Order Logic are 0.5 and 0.4, respectively. Passing these three topics represents independent events. What is the probability that the student will pass Discrete Mathematics?
- ㉒ In Example 6.10.15 no probability space or probability measure was specified. Under the assumption that exactly 30 balls are drawn in the experiment, specify a suitable probability space and define a probability measure on it.



Pierre-Simon de Laplace (1749–1827)

Pierre-Simon de Laplace was a French mathematician and astronomer who made fundamental contributions to mathematical astronomy and statistics.

Pierre-Simon de Laplace's parents were peasants living in Beaumont-en-Auge in France. Very little is known about Laplace's childhood, mostly because he tried to suppress all details of his humble beginnings in later life. In Beaumont-en-Auge the young Laplace's talent in theological disputations and mathematics gained him the support of some wealthy residents, which enabled him to study mathematics at the local military academy. After teaching for a while at this same academy, Laplace travelled to Paris where he introduced himself to D'Alembert (1717–1783). After initially receiving him rather coldly, D'Alembert became so impressed

with Laplace that he arranged for him to be appointed as Professor of Mathematics at the Military School of Paris.

The life's work that Laplace set himself was to work out the detailed application of Newton's laws of motion and gravitation to the solar system. This was a staggeringly complicated problem. The results were published as the *Mécanique Céleste*, Laplace's masterwork, which appeared in parts over a period of twenty-six years. The overall aim was to prove that the solar system is stable, i.e. that the planets will not eventually drift out of their orbits into space or crash into each other or into the sun. All of his work was subservient to this aim, including his ground breaking and fundamental work on probability and statistics, published as *Théorie Analytique des Probabilités* (1812).

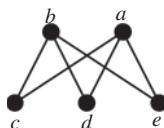
In 1785 one of the candidates that Laplace had to examine at the Military Academy was the young Napoleon Bonaparte (1769–1821). This was a fateful encounter: after the revolution Napoleon granted Laplace not only distinctions and honours for his work in mathematics, but also influential government positions, and made him a Count of the French Empire. Laplace spent his last days in comfortable retirement at his country estate at Arcueil outside Paris.

7

Graph Theory

7.1.	Introduction to graphs and digraphs	357
7.2.	Incidence and adjacency matrices	370
7.3.	Weighted graphs and path algorithms	377
7.4.	Trees	385
7.5.	Eulerian graphs and Hamiltonian graphs	395
7.6.	Planar graphs	404
7.7.	Graph colourings	411

Graph theory is the study of structures called graphs.¹ A **graph** is essentially a (usually finite) set of objects, called **vertices**, equipped with a symmetric and irreflexive binary **edge** relation between these vertices. Graphs are typically visualized in a way similar to the graphical representation of binary relations studied in Section 2.3.2, with dots representing vertices and lines representing edges, for example:



We do not put directions to the edges because the edge relation is assumed symmetric. When the assumption of symmetry is dropped, the resulting structures are called **directed graphs** or **digraphs**.

¹ These are not to be confused with function plotting graphs, like, for example the graph of a parabola.

Graph theory is closely related to combinatorics. It is a beautiful mathematical field: graphs are very pleasant objects to work with, being easy to visualize and to apply geometric intuitions to. The problems studied in graph theory are often relatively easy to formulate and understand, although these problems can sometimes turn out to be extremely difficult to solve. Some of the questions that we will be asking and answering in this chapter include: “*when can we find a path along the edges in a given graph which uses all the edges?*”, “*when can a graph be drawn in the plane without any of its edges crossing?*”, “*what is the least number of colours needed to colour the vertices of a given graph so that no two vertices linked by an edge have the same colour?*” and “*what is the relationship between the numbers of edges and vertices in a graph which does not contain any cycles?*”.

Graph theory has many and varied applications across many different fields. Indeed, graphs and digraphs can be used to model any situation in which pairs objects stand in a relation to each other, including computers linked in a network, web pages linking to each other, people being friends on social networking sites, radio transmitters having overlapping ranges and many more. In this chapter we will be applying the theory of graphs to solve problems like scheduling clash-free exam timetables, planning cheap and efficient routes through road or airline networks, and minimizing the number of layers needed in electronic circuit boards.

7.1. Introduction to graphs and digraphs

In this section we introduce the basic terminology and concepts relating to graphs and digraphs.

7.1.1. Graphs

Basic concepts

A **graph** G consists of a nonempty set of objects called **vertices** (singular: **vertex**) together with a (possibly empty) set of unordered pairs of *distinct* vertices called **edges**. In our presentation the set of vertices (and, hence, also the set of edges) will be assumed finite unless otherwise specified, but much of the theory presented here also applies to infinite graphs. The set of vertices (**vertex set**) of G and the set of edges (**edge set**) of G are denoted by $V(G)$ and $E(G)$, respectively. A graph G with vertex set V and edge set E will be denoted by $G(V, E)$. If $G(V, E)$ is a graph, then:

- We say the edge $e = \{u, v\}$ **joins** vertices u and v . Hereafter we will denote the edge $\{u, v\}$ simply as uv .
- If $e = uv$ is an edge of G , u and v are called **adjacent vertices**, while u and e (as well as v and e) are said to be **incident**.
- If e_1 and e_2 are distinct edges of G incident with a common vertex they are called **adjacent edges**.
- An edge uv will usually be denoted merely by uv .
- The number of vertices is called the **order** of the graph G , denoted by $n(G)$, or simply by n if the graph under consideration is clear from the context.

- The number of edges is called the **size** of the graph G and denoted by $m(G)$, or simply by m if the graph under consideration is clear from the context.
- An (n, m) **graph** has order n and size m .

We usually visualize a graph as follows: the vertices are represented by circles or dots and the edges as lines or arcs connecting these. It follows from the definition of a graph that there can be at most one edge joining two vertices and that an edge cannot join a vertex to itself.

We illustrate these concepts with an example.

Example 7.1.1

Consider the graph $G = (V, E)$ with $V = \{v_1, v_2, v_3, v_4\}$ and $E = \{v_1v_2, v_2v_3, v_3v_4, v_4v_1, v_1v_3\}$. For simplicity of notation we set $v_1v_2 = e_1, v_2v_3 = e_2, v_3v_4 = e_3, v_4v_1 = e_4$ and $v_1v_3 = e_5$. This graph is sketched in Figure 7.1. Note that $n(G) = 4$ and $m(G) = 5$, and hence G is a $(4, 5)$ graph; e_1 joins v_1 and v_2 , and hence v_1 and v_2 are adjacent and e_1 is incident with both; e_1 and e_5 are adjacent edges.

The **degree of a vertex** v in a graph G is the number of edges incident with the vertex in that graph, denoted $\deg_G v$ or simply $\deg v$ if G is clear from the context.

- A vertex is called **even** or **odd** depending on whether its degree is even or odd.
- A vertex with degree 0 is called **isolated**.
- A vertex with degree 1 is called an **end-vertex**.
- The **minimum degree** of a graph G is the minimum among the degrees of the vertices of G , denoted $\delta(G)$. The **maximum degree** of G , denoted by $\Delta(G)$, is defined likewise.
- A graph with order 1 can have no edges. Therefore there is only one graph of order 1, namely the graph consisting of a single isolated vertex. This graph is called the **trivial graph**. All other graphs are **nontrivial**.

Example 7.1.2

We consider the graph G in Figure 7.2. We have $\deg v_1 = 3 = \Delta(G)$, $\deg v_2 = \deg v_4 = \deg v_5 = 1$ and $\deg v_3 = 0 = \delta(G)$. Thus, v_2, v_4 and v_5 are end-vertices while v_3 is an isolated vertex.

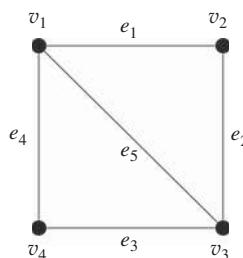


Figure 7.1: Example 7.1.1

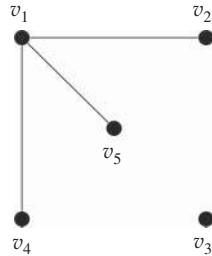


Figure 7.2: Example 7.1.2

These definitions give us enough of a conceptual framework to formulate and prove our first simple theorem about graphs.

Theorem 7.1.3 *Let G be an (n, m) graph with $V(G) = \{v_1, v_2, \dots, v_n\}$. Then*

$$\sum_{i=0}^n \deg v_i = 2m$$

Proof: Each edge is incident with precisely two vertices. Hence, when summing the degrees of the vertices each edge is counted precisely twice.

Theorem 7.1.3 is often called *The First Theorem of Graph Theory*. It has the following corollary.

Corollary 7.1.4 *Every graph has an even number of odd vertices.*

Proof: Let $G(V, E)$ be a (n, m) graph and let W be the set of odd vertices of G and U the set of even vertices. By Theorem 7.1.3

$$2m = \sum_{v \in V} \deg v = \sum_{v \in W} \deg v + \sum_{v \in U} \deg v.$$

Clearly $\sum_{v \in U} \deg v$, being a sum of even numbers, must be even. Then, as $\sum_{v \in U} \deg v$ and $\sum_{v \in W} \deg v$ add up to an even number, $\sum_{v \in W} \deg v$ must also be even. This implies that $|W|$ must be even, giving us an even number of odd vertices.

Subgraphs

A graph H is called a **subgraph** of a graph G if H is “contained in” G . Formally, H is a subgraph of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Note that we can take any subset of $E(G)$ as the vertex set of the subgraph, as long as each vertex incident with an edge in this subset is a vertex in $V(H)$.

H is called a **proper subgraph** of G if either of the containments $V(H) \subseteq V(G)$ or $E(H) \subseteq E(G)$ are proper. If H is a subgraph of G , then G is a **supergraph** of H . H being a subgraph of G is denoted by $H \subseteq G$.

If $I \subseteq V(G)$, then the **subgraph of G induced by I** , denoted $\langle I \rangle_G$, is the subgraph of G that has vertex set I and all the edges in G between vertices in I . In other words, the edges of $\langle I \rangle_G$

are exactly those edges of G between vertices in I . Similarly, if $X \subseteq E(G)$ then the subgraph of G **vertex induced** by X , denoted $\langle X \rangle_G$, is the graph with edge set X and all vertices of G incident with one or more edges from X .

Example 7.1.5

The graph shown in Figure 7.3 is known as the Petersen graph. The graphs shown in Figures 7.4 and 7.5 are subgraphs of the Petersen graph, but are not subgraphs of each other. Note that neither is an induced subgraph of the Petersen graph, but that both are edge induced subgraphs of the Petersen graph. You can check that the graph (called K_3) in Figure 7.6 is not a subgraph of the Petersen graph (so it cannot be a subgraph of any of these two graphs – why?).

Complete graphs and bipartite graphs

A graph is called **complete** if there is an edge between every pair of vertices, i.e. the graph has all the edges it can possibly have. The complete graph with n vertices is denoted by K_n . Note that K_n has $\binom{n}{2}$ edges. Figure 7.7 shows the complete graphs of order one to four.

A graph G is called **bipartite** if it is possible to express $V(G)$ as the union of two disjoint nonempty sets (known as **partite sets**), say V_1 and V_2 , such that every edge $e \in E(G)$ joins a vertex in V_1 to a vertex in V_2 ; in other words, there is no edge between two vertices in V_1 or between two vertices in V_2 .

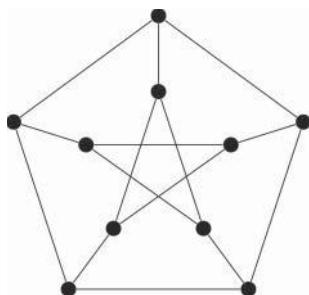


Figure 7.3: The Petersen graph

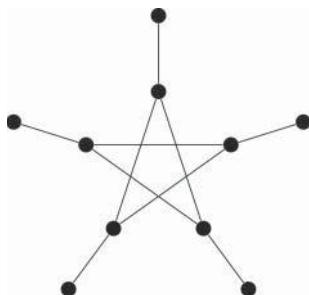


Figure 7.4: A subgraph of the Petersen graph

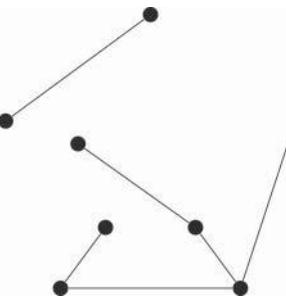


Figure 7.5: Another subgraph of the Petersen graph

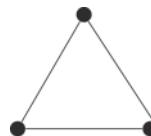


Figure 7.6: K_3 – the complete graph on three vertices

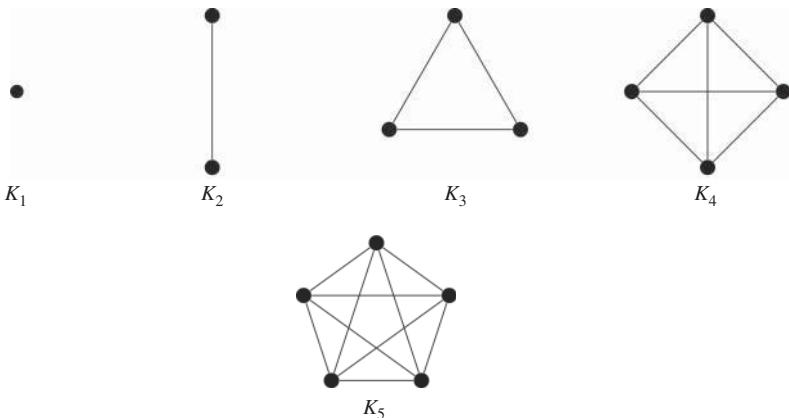


Figure 7.7: The complete graphs of order one to four.

Example 7.1.6

Consider the graph pictured in Figure 7.8.

This graph is bipartite. We have indicated the vertices in the two partite sets as respectively white and black circles.

A bipartite graph G is called **complete** if for all $u_1 \in V_1$ and $u_2 \in V_2$ it is the case that $u_1 u_2 \in E(G)$. If $|V_1| = r$ and $|V_2| = s$, then this graph is denoted by $K_{r,s}$ and has $r \times s$ edges.

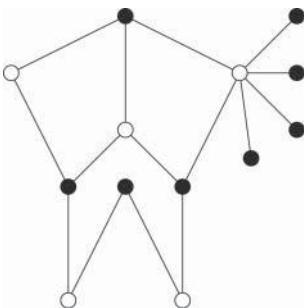
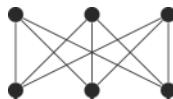
**Figure 7.8:** A bipartite graph**Figure 7.9:** The complete bipartite graph $K_{3,3}$ **Example 7.1.7**

Figure 7.9 shows the complete bipartite graph $K_{3,3}$. Note that K_2 is bipartite but that all complete graphs of order higher than two are not. Indeed, there is no way to split the vertex set of K_3 into two nonempty partite sets. Can you see why?

Paths and distance

In this subsection we introduce and study a number of notions that capture ideas of “moving around” in a graph from a vertex to a vertex along the edges. This is important for many applications. For instance, being able to move between two specific vertices a and b along the edges of the graph might be interpreted as “there is a sequence of hyper links leading from web page a to web page b ”, or “it is possible to get from city a to city b by using only flights of airline X ”, or “there is a sequence of events that will lead from state a to state b of the nuclear reactor”.

Let G be a graph, $u_0, u_1, \dots, u_k \in V(G)$ and $e_1, e_2, \dots, e_k \in E(G)$ such that $e_i = u_{i-1}u_i$ for all $1 \leq i \leq k$. Then the sequence

$$W : u_0, e_1, u_1, e_2, u_2, \dots, e_k, u_k$$

is called a **walk** of length k in G . The vertex u_0 is called the **initial vertex** of the walk and the vertex u_k the **terminal vertex**. If the edges e_1, e_2, \dots, e_k in the walk are all distinct, the walk is called a **trail** of length k in G , i.e. a trail is a walk in which no edge is repeated. Note, however, that vertices may still be repeated in a trail.

If the vertices u_0, u_1, \dots, u_k are all distinct the sequence $u_0, e_1, u_1, e_2, u_2, \dots, e_k, u_k$ is called a **path** of length k in G . Note that the length of a walk, trail or path is the number of edges it contains. When writing the sequence representing a walk or path we will usually not include

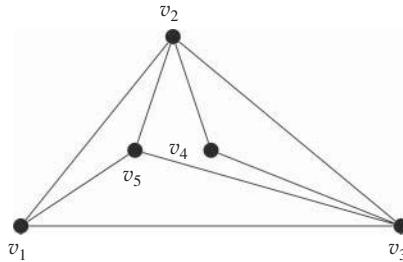


Figure 7.10: Example 7.1.8

the edges (as this adds no information) but only list the vertices. A path consisting of a single vertex is called a **trivial path**. The **distance** $d(u, v)$ between two vertices u and v in a graph is the length of the shortest path between u and v , if such a path exists.

A walk, trail or path from a vertex v to a vertex u is called a $u - v$ walk, trail or path, respectively. A $u - v$ walk is called **open** if $u \neq v$ and **closed** otherwise. A non-trivial closed trail is called a **circuit**. A circuit $v_1, v_2, \dots, v_n, v_1$ whose n vertices v_1, v_2, \dots, v_n are distinct is called a **cycle**. A graph that contains no cycles is called **acyclic**.

Example 7.1.8

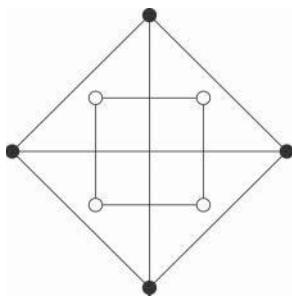
Consider the graph pictured in Figure 7.10.

- $W_1 : v_1, v_2, v_3, v_2, v_5, v_3, v_4$ is a walk (of length 6) between v_1 and v_4 but is not a trail, since the edge v_2v_3 is repeated. When an edge is repeated in a walk, one or both of the vertices incident with the edge also get repeated. In W_1 , when edge v_2v_3 is repeated vertex v_2 is also repeated. Therefore in general, a walk that fails to be a trail also fails to be a path.
- $W_2 : v_1, v_2, v_3, v_5, v_1, v_3$ is a trail of length 5. It is an open trail, since the initial vertex v_1 is not the same as the terminal vertex v_3 . Also, W_2 is not a path, as the vertices v_1 and v_3 are repeated.
- $W_3 : v_1, v_3, v_4$ is a path (of length 2) from v_1 to v_4 . In fact, as you can check, there is no shorter path from v_1 to v_4 , so $d(v_1, v_4) = 2$.
- $W_4 : v_1, v_5, v_2, v_3, v_4, v_2, v_1$ is a closed trail, making it a circuit per definition. It is not a cycle, however, since the vertex v_2 is repeated.
- v_1, v_5, v_2, v_1 is a cycle and hence the graph is not acyclic.

Two vertices are said to be **connected** if there is a path between them. A graph is called **connected** if there is a path between every pair of distinct vertices. A maximally connected subgraph H of a graph G is called a **connected component** of G . In other words, H is a connected component of G if $H \subseteq G$ and H is connected, and if H' is any connected subgraph of G such that $H \subseteq H'$, it must be the case that $H' = H$.

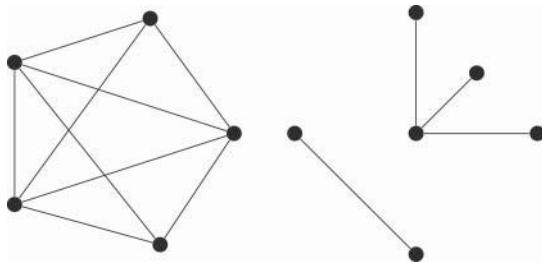
Example 7.1.9

Consider the following graph:



The white vertices are all connected and so are the black vertices, but no white vertex is connected to any black vertex. Therefore, the graph has two connected components, namely the subgraph induced by the white vertices and the one induced by the black vertices.

The following graph has three connected components:

**7.1.2. Digraphs**

The edges in (undirected) graphs have no associated direction, as they are defined as *unordered* pairs of vertices. It is often useful in applications to be able to associate a direction with an edge. For the modelling of situations where the edges have directions we use the concept of a **directed graph** (or **digraph**), which is the topic of this section.

A digraph D consists of a finite nonempty set of objects called **vertices** together with a (possibly empty) set of *ordered* pairs of vertices called **arcs** or **directed edges** between *different* vertices. A digraph D with vertex set V and arc set E is denoted $D(V, E)$. The vertex set of D will again be denoted by $V(D)$ and the arc set by $E(D)$. Note that it is only possible to have at most one arc from a vertex u to a vertex v .

Digraphs are graphically represented in much the same way as graphs, with dots representing the vertices, but with arrows representing arcs, as in Figure 7.11.

If $D(V, E)$ is a digraph, then:

- The arc $e = (u, v)$ **joins** vertices u and v ; u is the **initial vertex** of e and v is the **terminal vertex**.
- The arc $e = (u, v)$ is **incident from** its initial vertex u and **incident to** its terminal vertex v . The vertices u and v are also said to be **incident on** the arc e . Moreover, u is **adjacent to** v and v is **adjacent from** u .
- Two vertices u and v in a digraph are **nonadjacent** if u is neither adjacent to v nor adjacent from v .
- The **outdegree** of a vertex u , denoted $\text{out}(u)$, is the number of arcs incident from u (equivalently the number of edges of which u is the initial vertex). Similarly, the **indegree** of u , denoted $\text{in}(u)$, is the number of edges incident to u (equivalently the number of edges of which u is the terminal vertex).
- A vertex v such that $\text{in}(v) = 0$ is called a **source**; a vertex u such that $\text{out}(u) = 0$ is called a **sink**.
- The number of vertices in the vertex set V is called the **order** of the digraph D and denoted by $n(D)$, or simply by n if the digraph under consideration is clear from the context.
- The number of arcs in the arc set E is called the **size** of the digraph D and denoted by $m(D)$, or simply by m if the digraph under consideration is clear from the context.
- An (n, m) **digraph** has order n and size m .

Example 7.1.10

Consider the digraph D in Figure 7.11. Here we have

$$V(D) = \{a, b, c, d, e, f\}$$

and

$$E(D) = \{(a, b), (b, c), (c, d), (d, a), (a, c), (c, a), (e, f)\}.$$

Thus $n(D) = 6$ and $m(D) = 7$; (a, b) is incident from a to b ; b is adjacent to c and c is adjacent from b . Looking at the indegrees of the vertices we have $\text{in}(a) = \text{in}(c) = 2$, $\text{in}(b) = \text{in}(d) = \text{in}(f) = 1$ and $\text{in}(e) = 0$. As for the outdegrees: $\text{out}(a) = \text{out}(c) = 2$, $\text{out}(d) = \text{out}(b) = \text{out}(e) = 1$ and $\text{out}(f) = 0$. Vertex e is a source and f is a sink.

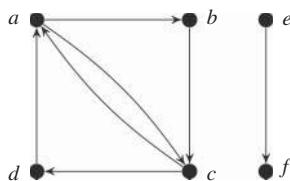


Figure 7.11: A digraph of order 6

Now we will state and prove the *First Theorem of Digraph Theory*.

Theorem 7.1.11 *If D is a digraph of order n and size m with $V(D) = \{v_1, v_2, \dots, v_n\}$ then*

$$\sum_{i=1}^n \text{in}(v_i) = \sum_{i=1}^n \text{out}(v_i) = m.$$

Proof: Straightforward, as every arc contributes with 1 to each of the two sums.

The notion of a subdigraph (or directed subgraph) is just what one would expect. A digraph D' is a subdigraph of a digraph D (denoted $D' \subseteq D$) if $V(D') \subseteq V(D)$ and $E(D') \subseteq E(D)$.

A **directed walk** in digraph D is a sequence of vertices

$$W : u_0, u_1, u_2, \dots, u_n$$

such that (u_{i-1}, u_i) is an arc of D for $i = 1, 2, \dots, n$. A walk in a digraph is just like a walk in a graph, except that the arcs now have to be followed in the right direction. The notions of **directed trail**, **directed path**, **directed circuit** and **directed cycle** are precise analogues of the corresponding notions for graphs. The **length** of a directed walk, trail, path, circuit or cycle is, as one would expect, the number of arcs. Often, when working with digraphs, we will omit the word “directed” when talking of directed paths, etc.

Example 7.1.12

The digraphs D' and D'' in Figures 7.12 and 7.13 are subgraphs of D in Figure 7.11, as you may check. Further, $W : a, b, b, c$ is a directed walk in D' but not a directed path and $W' : a, b, c$ is a directed path (contained in the directed walk W'). Note that D'' contains an a - a cycle while D' does not.

We can transform a digraph into a graph as follows: given a digraph D , replace each arc (u, v) or symmetric pair of arcs (u, v) and (v, u) by the edge uv . The graph so obtained is called the **underlying graph of D** .

We use the concept of an underlying graph to define that of a connected digraph. A digraph D is said to be **connected** if its underlying graph is connected. A digraph D is **strongly connected** if for every pair of vertices $a, b \in V(D)$ there is a directed path from a to b .

Our definitions of graph and digraph, as already mentioned, do not allow for the possibility of multiple edges or arcs between vertices. If the definition of a graph is altered to allow multiple edges between vertices the resulting concept is called a **multigraph**. If we modify

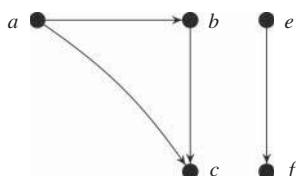


Figure 7.12: A subdigraph of D (Figure 7.11).

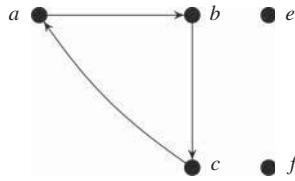
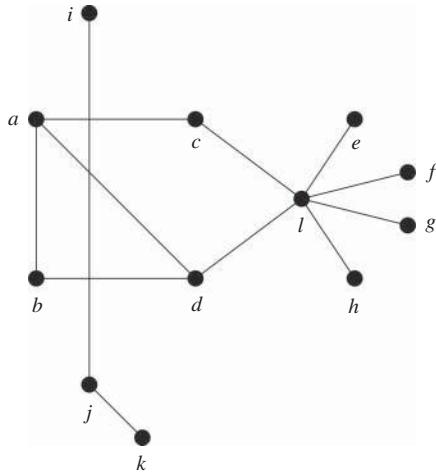


Figure 7.13: Another subdigraph of D

the definition of a digraph similarly we obtain what is known as a **multidigraph** or a **directed multigraph**. It is sometimes useful to label (or give names to) arcs: this also helps to distinguish multiple arcs between the same pair of vertices. Formally a labelled digraph consists of a set of vertices V , a set of labels L and a set of labelled arcs $E \subseteq V \times L \times V$.

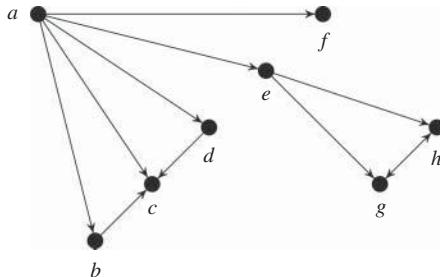
7.1.3. Exercises

① Consider the graph G pictured below:



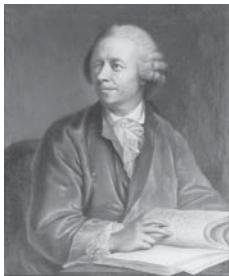
- Determine $n(G)$ and $m(G)$.
- For each vertex determine its degree, the vertices adjacent to it and the edges incident with it. Also state, for each vertex, whether it is even or odd and if it is an isolated vertex or an end vertex.
- Determine $\delta(G)$ and $\Delta(G)$.
- Determine the complete graph of maximum order that forms a subgraph of G .
- Is G bipartite? If so, give the two partite sets; if not, explain why not.
- Give a path of maximum length in G .
- Determine $d(a,f)$.
- Is G connected?
- How many components does G have?

- ② Give an example of a graph such that each vertex is adjacent to four vertices and each edge is adjacent to six edges.
- ③ Draw K_6 and $K_{4,3}$.
- ④ Is there a graph with:
- 6 vertices and 6 edges?
 - 6 vertices and 15 edges?
 - 6 vertices and 18 edges?
- ⑤ Does there exist a graph with five vertices, every vertex incident with at least one edge but with no two edges being adjacent? Motivate your answer.
- ⑥ Must a subgraph of a bipartite graph be bipartite? If yes, motivate, if no, give a counterexample.
- ⑦ Must a subgraph of a connected graph be connected? If yes, motivate, if no, give a counterexample.
- ⑧ Consider the set of vertices $\{v_1, v_2, \dots, v_n\}$. How many different complete bipartite graphs may be formed using this vertex set?
- ⑨ What is the largest possible number of vertices in a graph with 35 edges, all vertices having degree at least 3. (Hint: use Theorem 7.1.3.)
- ⑩ For the digraph pictured in the figure below, do the following:



- (a) For each vertex write down its indegree as well as its outdegree. Identify any sinks or sources.
- (b) Determine the order and size of the digraph.
- (c) Find a path of maximum length in the digraph.
- (d) Sketch the underlying graph.
- (e) Is the digraph connected?
- (f) Is it strongly connected?
- (g) Find four different subgraphs.
- (h) Find a strongly connected subgraph.
- ⑪ In a digraph with n vertices:

- (a) What is the maximum indegree of any vertex?
 - (b) What is the maximum outdegree of any vertex?
 - (c) What is the maximum number of arcs?
- ⑫ Does there exist a digraph with:
- (a) Order 5 and size 21? If yes, draw such a digraph, if no, why not?
 - (b) Three sources, two sinks and size two? If yes, draw such a digraph, if no, why not?
- ⑬ Prove Theorem 7.1.11.
- ⑭ Give a definition of a **directed cycle**.



Leonhard Euler (1707–1783)

Leonhard Euler was a Swiss mathematician and physicist who made important discoveries in many fields of mathematics, from infinitesimal calculus to graph theory, which he founded as a branch of mathematics.

Euler was born in Basel in Switzerland. His parents were Paul Euler, a pastor of the Reformed Church, and Marguerite Brucker, a pastor's daughter. Euler's father intended him to become a pastor like himself, and he accordingly studied theology and Hebrew at the University of Basel. Paul was friends with the mathematician Johann Bernoulli (1667–1748), who gave Euler one private mathematics lesson per week. Bernoulli was able to persuade Paul Euler that his son was a mathematical genius and that he should be allowed to pursue a career as a mathematician.

In 1726, after failing to obtain an academic position in Basel, Euler accepted a position at the Imperial Russian Academy of Sciences in St Petersburg. This position had been arranged with the help of Johann Bernoulli's son Daniel who was based at the Academy at that time. Euler quickly worked his way up through the ranks and was appointed head of the mathematics department when Daniel Bernoulli resigned in 1733. While in St Petersburg Euler married Katharina Gsell. They had thirteen children, of whom only five survived infancy. At the Academy Euler was enormously productive: apart from publishing great numbers of mathematical papers and a treatise that revolutionized mechanics, he also wrote elementary mathematical textbooks for Russian schools, supervised the geographical department of the government and reformed the Russian weights and measures.

The political situation in Russia was tense, and in 1741 Euler gladly accepted a position at the Berlin Academy of Sciences, offered to him by Emperor Frederick the Great of Prussia. Unfortunately Euler was not popular at Frederick's court, where he was unable to compete with the conversational brilliance and wit of philosophers like Voltaire with whom Frederick had surrounded himself. In 1766 he therefore returned to St Petersburg upon the invitation of Catherine the Great, who had succeeded to the Russian throne in 1762. He remained in St Petersburg for the rest of his life.

Euler was probably the most productive mathematician in history – it is estimated that during certain years he produced an average of one research paper per week. Even during the last years of his life, while he was completely blind, he continued to work with undiminished vigour. He made fundamental contributions to analysis, number theory, combinatorics, applied mathematics, physics and other fields.

7.2. Incidence and adjacency matrices

Associated with any graph or digraph there are two matrices, namely the **incidence matrix** and the **adjacency matrix**. These are useful for representing a graph in a form that can be easily stored in a computer. The matrix representation also lends itself more readily to algorithmic manipulation. We will consider one such algorithm on adjacency matrices, known as the Warshall algorithm. This algorithm computes the existence of paths in graphs.

Definition 7.2.1 Let $G = (V, E)$ be an (n, m) graph. Let A be an $n \times m$ matrix, the rows of which are labelled with the vertices of G and the columns of which are labelled with the edges of G . Moreover, the entry in the i th row and j th column of A , denoted A_{ij} , is 1 if the i th vertex is incident with the j th edge, and is 0 otherwise. The matrix A is called the **incidence matrix** of the graph G .

Example 7.2.2

Consider the graph in Figure 7.14. The incidence matrix for this graph is

$$\begin{array}{ccccccccc} \alpha & \beta & \gamma & \delta & \epsilon & \zeta & \eta & \theta \\ a & \left(\begin{array}{ccccccc} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right) \\ b & \\ c & \\ d & \\ e & \\ f & \end{array}.$$

Notice that in each row of an incidence matrix there are 1s precisely in the columns corresponding to the edges incident with the vertex with which that row is labelled. The degree of a vertex can then be calculated by summing the entries in its corresponding row. Also notice that in each column there are precisely two 1s, as each edge is incident with precisely two vertices.

Definition 7.2.3 Let $G = (V, E)$ be an (n, m) graph (digraph). Let A be an $n \times n$ matrix in which both the rows and columns are labelled with the vertices of G in the same order. Moreover, the entry in the i th row and j th column of A , denoted A_{ij} , is 1 if there is an edge (arc) from the vertex labelling row i to the vertex labelling column j , and 0 otherwise. The matrix A is called the **adjacency matrix** of the graph (digraph) G .

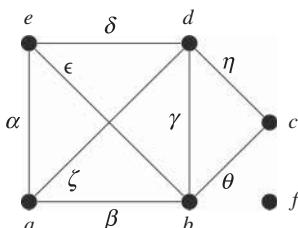


Figure 7.14: Incidence and adjacency matrices for this graph are given in Examples 7.2.2 and 7.2.4

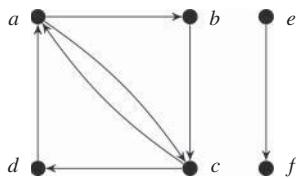
Example 7.2.4

The adjacency matrix for the graph in Figure 7.14 is

$$\begin{array}{cccccc} & a & b & c & d & e & f \\ a & \left(\begin{array}{cccccc} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right) \\ b & \\ c & \\ d & \\ e & \\ f & \end{array}.$$

Example 7.2.5

Here is an example of a digraph with its adjacency matrix:



$$\begin{array}{cccccc} & a & b & c & d & e & f \\ a & \left(\begin{array}{cccccc} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right) \\ b & \\ c & \\ d & \\ e & \\ f & \end{array}.$$

Notice that the adjacency matrix of a graph is symmetric about the main diagonal, which consists entirely of 0s. The main diagonal of the adjacency matrix of a digraph also consists entirely of 0s, but the matrix need not be symmetric. Summing all the entries in the adjacency matrix of a graph, one will get a number equal to twice the size of the graph, and doing so for the adjacency matrix of a digraph, one will get the size of the digraph.

As we indicated at the beginning of this section, matrix representations of graphs are amenable to algorithmic manipulation and are quite useful in the computation of graph parameters. To illustrate this we will now discuss an algorithm, known as the **Warshall algorithm**, which computes **reachability** in (di)graphs. We say that a vertex v of a (di)graph is reachable from a vertex u if there exists a (directed) walk from u to v . For example, in the

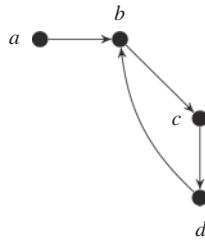


Figure 7.15: In Example 7.2.7 the Warshall algorithm is used to calculate reachability in this digraph

digraph pictured in Figure 7.15 every vertex is reachable from vertex a , while vertex a is only reachable from itself. Note that every vertex is reachable from itself through a walk of length 0.

Before presenting the Warshall algorithm we need the notion of **Boolean addition**. Boolean addition works on 0s and 1s and can be thought of as disjunction, where 0 represents \perp and 1 represents T . Thus, using Boolean addition we have $0 \oplus 0 = 0$ (analogy: $\perp \vee \perp \equiv \perp$), $0 \oplus 1 = 1$ (analogy: $\perp \vee T \equiv T \vee \perp \equiv T$) and $1 \oplus 1 = 1$ (analogy: $T \vee T \equiv T$). We will be adding rows of the adjacency matrix using Boolean addition.

The Warshall algorithm takes the $n \times n$ adjacency matrix A of a graph (digraph) G , and outputs an $n \times n$ matrix A' such that $A'_{ij} = 1$ if the vertex labelling column j is reachable from the vertex labelling row i , and $A'_{ij} = 0$ otherwise. In other words, the output is a matrix where the entries 1 represent the existence of (directed) walks between vertices. A' is obtained by successive transformations of A as follows.

Warshall algorithm

Given an adjacency matrix A , proceed as follows:

1. Change all entries on the diagonal of A to 1s. (Each vertex is reachable from itself, via a walk of length 0.)
2. Look at column 1 of the matrix obtained in (1). For every row in the matrix where there is a 1 in that column, add row 1 of the matrix to that row, using Boolean addition. (Explanation: 1 in column 1 of row i means the vertex labelling row i can reach the vertex labelling row 1 and, hence, also every vertex reachable from the vertex labelling row 1.)
3. Then look at column 2 of the matrix obtained in (2). Wherever there is a 1 in a row of that column, add row 2 of the matrix obtained in (1) to the row in which that 1 occurred, using Boolean addition. (Same explanation: 1 in column 2 of row i means the vertex labelling row i can reach the vertex labelling row 2, and hence, also every vertex reachable from the vertex labelling row 2.)
4. Continue this process until all columns have been examined.

Exercise 7.2.6 Prove that the Warshall algorithm is correct.

A more formal pseudo-code specification of the algorithm is given as Algorithm 1. For working by hand, the informal version of the algorithm proves more convenient. The following example demonstrates it in action.

Algorithm 1 Warshall algorithm

```

for  $i = 1$  to  $n$  do
     $A_{ii} := 1$ 
end for
for  $i = 1$  to  $n$  do
    for  $j = 1$  to  $n$  do
        if  $A_{ji} = 1$  then
            for  $k = 1$  to  $n$  do
                 $A_{jk} := A_{jk} \oplus A_{ik}$ 
            end for
        end if
    end for
end for

```

Example 7.2.7

Consider the digraph in Figure 7.15, and its adjacency matrix A :

$$\begin{array}{l}
 \begin{array}{cccc} & a & b & c & d \end{array} \\
 \begin{array}{c} a \\ b \\ c \\ d \end{array} \begin{array}{ccccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{array}
 \end{array}$$

First we change all entries on the diagonal to 1s, yielding:

$$\begin{array}{l}
 \begin{array}{cccc} & a & b & c & d \end{array} \\
 \begin{array}{c} a \\ b \\ c \\ d \end{array} \begin{array}{ccccc} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{array}
 \end{array}$$

We consider column 1. There is a 1 in the first row of this column, so we must add row 1 to row 1. This, of course, will not make any change because the addition is Boolean. Bearing this fact in mind can streamline our computations somewhat.

Now consider column 2. There is a 1 in row 1 of that column, so we add row 2 to row 1 using Boolean addition to obtain the matrix

$$\begin{array}{l}
 \begin{array}{cccc} & a & b & c & d \end{array} \\
 \begin{array}{c} a \\ b \\ c \\ d \end{array} \begin{array}{ccccc} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{array}
 \end{array}$$

Row 2 also has a one in column 2, but as before adding row 2 to itself yields the same matrix. There is another 1 in column 2, namely in the 4th row. Adding row 2 to row 4 we obtain

$$\begin{array}{l} \begin{matrix} & a & b & c & d \\ a & 1 & 1 & 1 & 0 \\ b & 0 & 1 & 1 & 0 \\ c & 0 & 0 & 1 & 1 \\ d & 0 & 1 & 1 & 1 \end{matrix} \\ \cdot \end{array}$$

Now consider column 3 of the matrix obtained so far. Every entry in column 3 is one; hence we have to add row 3 to every row of the matrix. Doing this we obtain the matrix

$$\begin{array}{l} \begin{matrix} & a & b & c & d \\ a & 1 & 1 & 1 & 1 \\ b & 0 & 1 & 1 & 1 \\ c & 0 & 0 & 1 & 1 \\ d & 0 & 1 & 1 & 1 \end{matrix} \\ \cdot \end{array}$$

Lastly consider column 4. Once again all the entries in this column are 1s, so we add row 4 to all the rows of the matrix (skipping row 4, as this will make no difference). We obtain

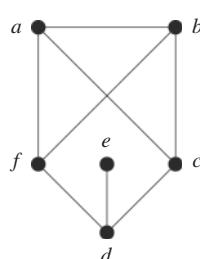
$$\begin{array}{l} \begin{matrix} & a & b & c & d \\ a & 1 & 1 & 1 & 1 \\ b & 0 & 1 & 1 & 1 \\ c & 0 & 1 & 1 & 1 \\ d & 0 & 1 & 1 & 1 \end{matrix} \\ \cdot \end{array}$$

Compare the matrix we have obtained with the graph in Figure 7.15. Notice that all the entries in column 1, except the first, are 0s, while all the other entries in the matrix are 1s. This corresponds to the fact that a is reachable only from itself, while every vertex other than a is reachable from every vertex in the digraph.

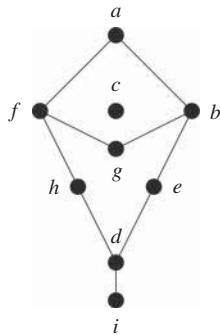
7.2.1. Exercises

- ① Determine the incidence and adjacency matrices of the following graphs:

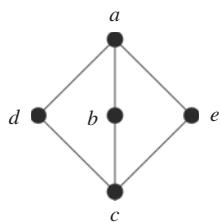
(a)



(b)



(c)



② Sketch the following graphs and give incidence and adjacency matrices for each.

(a) $G = (V, E)$ with $V = \{a, b, c, d, e, f\}$ with edges

$$e_1, e_2, e_3, \dots, e_6 = ab, ad, bc, cd, ae, ce.$$

(b) $G = (V, E)$ with $V = \{a, b, c, d, e, f, g\}$ with

$$\text{edges } e_1, e_2, e_3, \dots, e_{10} = ab, ad, af, bd, df, cd, ce, de, dg, eg.$$

(c) $G = (V, E)$ with $V = \{a, b, c, d, e, f\}$ with edges

$$e_1, e_2, e_3, \dots, e_9 = ab, ac, ae, bc, bd, cf, de, df, ef.$$

(d) $G = (V, E)$ with $V = \{a, b, c, d, e, f, g, i\}$ with edges

$$e_1, e_2, e_3, \dots, e_8 = ab, bc, bd, be, cd, de, df, gi.$$

③ Perform the Warshall algorithm on each of the adjacency matrices obtained in Exercise 2.

④ How would one determine whether the graph is connected from inspecting the matrix obtained from its adjacency matrix by the Warshall algorithm? For each graph in Exercise 2 determine whether it is connected by inspecting the adjacency matrix obtained for it in Exercise 3.

⑤ Sketch the following digraphs and give an adjacency matrix for each.

(a) $D = (V, E)$ with $V = \{a, b, c, d, e, f\}$ with arcs

$$e_1, e_2, e_3, \dots, e_9 = (a, d), (b, e), (c, a), (c, b), (c, e), (d, b), (d, c), (e, a), (e, d).$$

(b) $D = (V, E)$ with $V = \{a, b, c, d, e\}$ with arcs

$$e_1, e_2, e_3, \dots, e_7 = (a, c), (a, e), (b, a), (c, b), (c, d), (c, e), (d, a).$$

(c) $D = (V, E)$ with $V = \{a, b, c, d, e, f, g, h\}$ with arcs

$$e_1, e_2, e_3, \dots, e_6 = (a, b), (b, f), (f, e), (e, a), (c, d), (d, h).$$

(d) $D = (V, E)$ with $V = \{a, b, c, d, e, f, g, h\}$ with arcs

$$e_1, e_2, e_3, \dots, e_9 = (a, b), (a, d), (b, f), (f, e), (e, a), (c, d), (d, h), (h, g), (g, c).$$

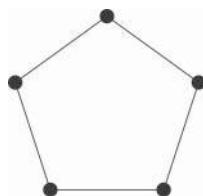
- ⑥ Perform the Warshall algorithm on each of the adjacency matrices obtained in Exercise 5.
- ⑦ How would one determine whether a digraph is strongly connected from inspecting the matrix obtained from its adjacency matrix by the Warshall algorithm? For each digraph in Exercise 5 determine whether it is strongly connected by inspecting the adjacency matrix obtained for it in Exercise 6.
- ⑧ How would you determine the order and size of a graph from its adjacency matrix?
- ⑨ How would you determine the order and size of a digraph from its adjacency matrix?
- ⑩ Prove that the Warshall algorithm is correct (Exercise 7.2.6).
- ⑪ When capturing an adjacency matrix, especially one for a large graph, it is easy to make mistakes. Describe some simple checks you could implement to pick up errors in an adjacency matrix. How would this differ for graphs and digraphs?

Ramsey theory

Frank P. Ramsey (1903–1930) was a British philosopher, mathematician and economist. In his very short life – he died at 26 – he made lasting contributions to these fields. He was a friend and translator of the philosopher Ludwig Wittgenstein and a student and collaborator of the economist John Maynard Keynes. He founded the field of combinatorics today known as *Ramsey theory*, the main gist of which is that in a system large enough, some order must emerge: complete chaos is impossible. Ramsey theory begins with questions like the following:

Is it true that if enough people are gathered in a room there will always be either three people that are friends or three people among which there are no friends? If so, what is the least number of people needed to guarantee this?

We can model this situation as a graph, with vertices representing people and edges representing the friendship relation. The question then becomes: will graphs over a certain order always contain either K_3 as a subgraph or three mutually nonadjacent vertices? If so, what is the smallest order for which this is guaranteed? Five vertices is not enough, as the following graph shows:



However, any graph with six vertices will contain K_3 or three mutually nonadjacent vertices. The proof is an easy exercise.

Denote by $R(n)$ the least integer such that a graph with order $R(n)$ must have K_n as a subgraph or n mutually nonadjacent vertices. Ramsey's theorem states that $R(n)$ exists for each n . As we saw above, $R(3) = 6$. It is known that $R(4) = 18$, but for $n > 4$ exact values of $R(n)$ are not known. Many years of hard work by mathematicians have produced some bounds, though. For example, it is known that $43 \leq R(5) \leq 49$, $102 \leq R(6) \leq 165$ and $17\,885 \leq R(9) \leq 9\,075\,135\,299$.



7.3. Weighted graphs and path algorithms

For many applications it is necessary to consider **weighted graphs** or **weighted digraphs**.² These are graphs (or digraphs) in which every edge (or arc) is assigned a number, called its weight. The weight of an edge can be variously interpreted as its cost, length, capacity, etc., depending on the intended application. For example, Figure 7.16 shows a typical example of a weighted graph. Here the vertices represent different cities in South Africa, the edges represent roads between the cities, while the weights of the edges represent their length in kilometers. In general the weight of an edge is intended as the “cost” (in money, time, distance, fuel, etc.) of traversing that edge. To name but one important application: weighted graphs can be used by Global Positioning System (GPS) devices as mathematical models of road networks.³

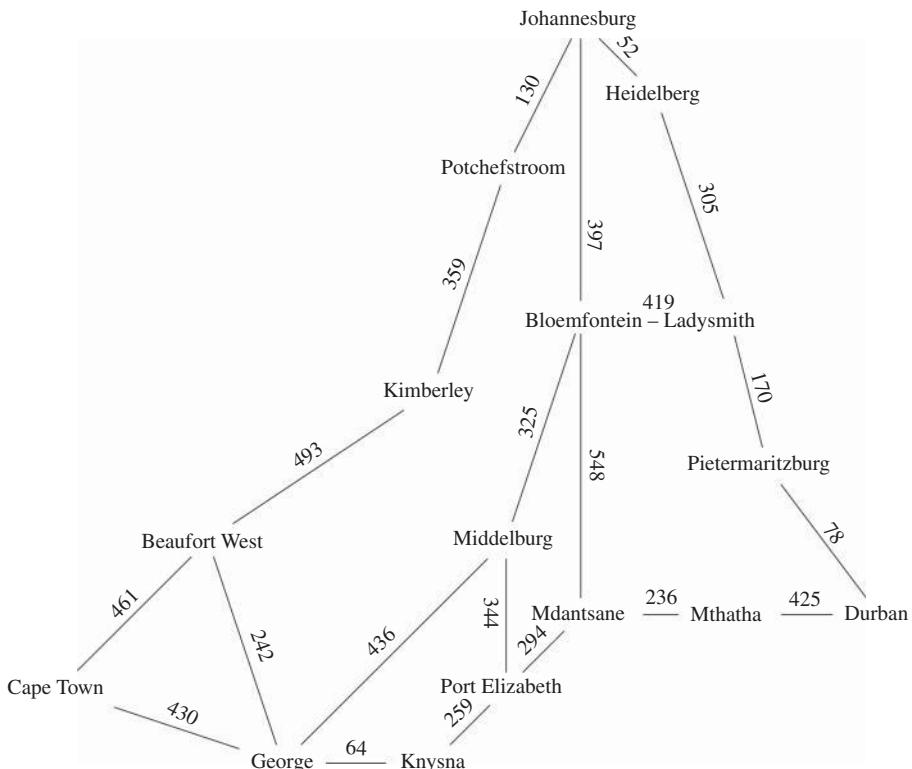


Figure 7.16: A weighted graph representing road links between some South African cities and the associated distances

² To keep the exposition to a reasonable length we will omit weighed digraphs and consider only weighted graphs.

³ See, for example, the article: G. Even-Tzur, Graph Theory Applications to GPS Networks, *GPS Solutions*, 5(1): 31–38, 2001.

Here is the formal definition of a weighted graph and some other associated terminology.

Definition 7.3.1 A **weighted graph** is a triple (V, E, w) such that (V, E) is a graph and $w : E \rightarrow R^*$ is a function assigning a non-negative real number $w(uv)$ to every edge $uv \in E$. We call $w(uv)$ the **weight** of uv .

The **weighted length** of a path v_0, v_1, \dots, v_n in a weighted graph is the sum of the weights of the edges on the path, i.e. $w(v_0v_1) + w(v_1v_2) + \dots + w(v_{n-1}v_n)$. The weighted distance between two vertices u and v , denoted $d(u, v)$, is the minimum of the weighted lengths of the paths between u and v if such paths exist.

For example, in the weighted graph depicted in Figure 7.16, the weighted length of the path “Johannesburg, Heidelberg, Laydysmith, Bloemfontein”, is $52 + 305 + 419 = 776$. However, the weighted distance between Johannesburg and Bloemfontein is 397.

Given vertices p and q in a weighted digraph, the key problem is to find the shortest path between p to q . We discuss two algorithms that solve this problem, namely Dijkstra’s algorithm and the Floyd–Warshall algorithm.

7.3.1. Dijkstra’s algorithm

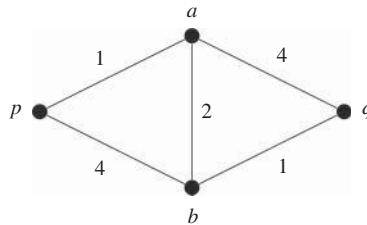
We begin with an informal specification of an algorithm for finding the shortest path between two given vertices in a weighted graph, due to the Dutch computer scientist Edsger Wybe Dijkstra (1930–2002). The algorithm takes as input a graph G and two specified vertices p and q . The output is the shortest path between p and q , if such a path exists. It maintains a set of labels of the form $(prev_v, dist_v)$, one for each vertex $v \in V(G)$. The value of $prev_v$ is the name of the vertex that precedes v in the shortest path from p to v , as calculated so far at the stage of the computation, and the value of $dist_v$ is the cost of that path. The algorithm also maintains a partition of the vertex set into two subsets Perm and Temp . The intention is that Perm contains those vertices whose labels are permanent and cannot be changed any more, whereas the labels of vertices in Temp are still subject to possible change.

Algorithm 2 Dijkstra’s algorithm

1. Initially set $(prev_p, dist_p) = (-, 0)$, and $(prev_v, dist_v) = (-, \infty)$ for all vertices v other than p . Set $\text{Perm} := \{p\}$ and $\text{Temp} = V - \{p\}$.
 2. Repeat the following two steps until q has been moved to Perm or an iteration is reached in which no vertex is moved to Perm .
 - (a) Take the last vertex that has been added to Perm , say it is c , and consider all its neighbours (i.e. vertices adjacent to c) in Temp . For each of these neighbours v calculate the weighted distance from p via c to v , i.e. $dist_c + w(cv)$, and if $dist_c + w(cv) < dist_v$ update $(prev_v, dist_v)$ to become equal to $(c, dist_c + w(cv))$.
 - (b) Let v be a vertex in Temp with minimal $dist$ and $dist_v < \infty$, if such exists, and transfer it to Perm , i.e. set $\text{Perm} := \text{Perm} \cup \{v\}$ and $\text{Temp} := \text{Temp} - \{v\}$.
-

Example 7.3.2

Let us apply Dijkstra's algorithm to the following weighted graph to calculate the shortest path from p to q :



We track the progress of the solution in the following table, which records the labels of each vertex after the step has finished, the last vertex transferred to Perm , and also the content of Perm .

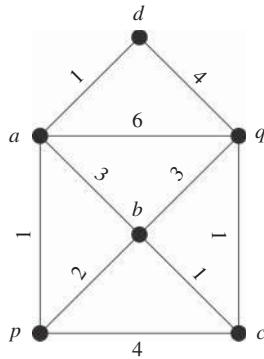
Step	p	a	b	q	Last transfer	Perm
Initial	$(-, 0)$	$(-, \infty)$	$(-, \infty)$	$(-, \infty)$	p	$\{p\}$
1	$(-, 0)$	$(p, 1)$	$(p, 4)$	$(-, \infty)$	a	$\{p, a\}$
2	$(-, 0)$	$(p, 1)$	$(a, 3)$	$(a, 5)$	b	$\{p, a, b\}$
3	$(-, 0)$	$(p, 1)$	$(a, 3)$	$(b, 4)$	q	$\{p, a, b, q\}$

When the algorithm terminates the shortest path from p to every vertex can be read off from the last row of the table. This is done by looking up the desired destination vertex and then working backwards by reading the prev coordinates. In this example, we begin by looking at the label of q , namely $(b, 4)$: this means that the weighted length of the shortest path from p to q is 4 and that q is preceded by b in this path. Now look at the label of b : it is $(a, 3)$, so b is preceded by a ; looking at the label of a , which is $(p, 1)$, we see that a is preceded by p . Therefore the shortest path from p to q is $pabq$.

Here is another example.

Example 7.3.3

We run Dijkstra's algorithm to the following weighted graph to calculate the shortest path from p to q :



Step	p	a	b	c	d	q	Last transf.	Perm
Init.	$(-, 0)$	$(-, \infty)$	p	$\{p\}$				
1	$(-, 0)$	$(p, 1)$	$(p, 2)$	$(p, 4)$	$(-, \infty)$	$(-, \infty)$	a	$\{p, a\}$
2	$(-, 0)$	$(p, 1)$	$(p, 2)$	$(p, 4)$	$(a, 2)$	$(a, 7)$	d	$\{p, a, d\}$
3	$(-, 0)$	$(p, 1)$	$(p, 2)$	$(p, 4)$	$(a, 2)$	$(d, 6)$	b	$\{p, a, d, b\}$
4	$(-, 0)$	$(p, 1)$	$(p, 2)$	$(b, 3)$	$(a, 2)$	$(b, 5)$	c	$\{p, a, d, b, c\}$
5	$(-, 0)$	$(p, 1)$	$(p, 2)$	$(b, 3)$	$(a, 2)$	$(c, 4)$	q	$\{p, a, d, b, c, q\}$

At termination the labels relevant for the shortest path from p to q are $(c, 4)$ (the label of q) and, working backwards from this label, $(b, 3)$ and $(p, 2)$. Thus, the shortest path from p to q is $pbcq$, which has the weighted length 4. Notice that at the end of step 2, both b and d were members of Temp with minimal $dist$, so either of them could have been chosen for transfer to Perm . We chose to transfer d , but the interested reader could easily compare the computation that would have resulted from transferring b at this point rather than d .

Theorem 7.3.4 *Dijkstra's algorithm is correct, i.e. given a weighted graph (V, E, w) and specified vertices p and q , the algorithm terminates and upon termination, for all $v \in \text{Perm}$, it holds that $dist_v = d(p, v)$ and prev_v is equal to a vertex preceding v that is some shortest path from p to v .*

Proof: We will show that the two properties claimed are in fact invariants of the algorithm. Before the first iteration of the loop $\text{Perm} := \{p\}$ and $(\text{prev}_p, dist_p) = (-, 0)$, and the conditions are clearly satisfied. Now suppose that the conditions hold up to some point in the execution of the algorithm and that now we enter the next iteration of the loop, in which a vertex v is picked to be transferred from Temp to Perm . Suppose, for the sake of definiteness, that $\text{prev}_v = u$. Note that $u \in \text{Perm}$, as the algorithm explores the graph only one step far outside the set Perm . We

must show that $dist_v$ and $prev_v$ satisfy the desired properties. It is clear that $d(p, v) \leq dist_v$. We need to show that in fact $dist_v = d(p, v)$. Suppose, to the contrary, that $d(p, v) < dist_v$, i.e. $d(p, v) < d(p, u) + w(uv)$. Thus, there exists a different shorter path from p to v . Let u' be the last vertex in Perm encountered on this path on the way from p to v and let $v' \in \text{Temp}$ be the successor of u' on this path. Then $dist_{v'} = d(p, u') + w(u'v') < d(p, u) + w(uv) = dist_v$. Now there are two cases. Firstly, if $v' = v$, then $prev_v = prev_{v'}$, i.e. $prev_v = u$ – a contradiction. Secondly, if $v' \neq v$, then it should have been v' added to Perm and not v , once again yielding a contradiction. We conclude that, indeed, $dist_v = d(p, v)$ and that u precedes v on the shortest path from p to v .

7.3.2. The Floyd–Warshall algorithm

The Floyd–Warshall algorithm is in essence a version of the Warshall algorithm discussed in Section 7.2. Unlike Dijkstra’s algorithm it does not compute shortest paths from a specified initial vertex, but computes the weighted distance between every two vertices in a weighted graph. Informally, the idea behind the Floyd–Warshall algorithm can be described as follows: suppose we are given a weighted graph $G = (V, E, w)$ with vertex set $V = \{v_1, v_2, \dots, v_n\}$. We maintain an $n \times n$ matrix of variables $dist(i, j)$, $1 \leq i, j \leq n$, where the value of $dist(i, j)$ is an approximation of $d(i, j)$. As the weighted distance from each vertex to itself is 0, we can set $dist(i, i) = 0$ for each $1 \leq i \leq n$. Moreover, if v_i and v_j are adjacent we can set $dist(i, j) = w(ij)$. We set all other entries in the matrix equal to ∞ . At this point the matrix of variables represents the costs of the paths consisting of 0 or 1 steps between the vertices. Now we incrementally try to improve on this by checking whether it is not possible to get cheaper paths by going via vertices in the set $\{v_1\}$, the set $\{v_1, v_2\}$, the set $\{v_1, v_2, v_3\}$, etc. The key idea is the following: suppose I know the cost of the shortest path between every two vertices v_i and v_j whose intermediate vertices (i.e. vertices other than the initial vertex v_i and terminal vertex v_j) consists only of vertices in $\{v_1, v_2, \dots, v_k\}$. If there were a shorter path with intermediate vertices contained in $\{v_1, v_2, \dots, v_k, v_{k+1}\}$ it would necessarily visit vertex v_{k+1} , and would hence consist of a path from v_i to v_{k+1} followed by a path from v_{k+1} to v_j with their intermediate vertices contained in $\{v_1, v_2, \dots, v_k\}$. To see whether such a path exists we need to check whether $dist(i, j) > dist(i, k+1) + dist(k+1, j)$, in which case we update $dist(i, j)$ to become equal to $dist(i, k+1) + dist(k+1, j)$.

Here is a pseudo code specification of this algorithm:

Algorithm 3 Floyd – Warshall algorithm

The algorithm takes a weighted graph (V, E, w) with vertex set $V = \{v_1, v_2, \dots, v_n\}$ as input. It maintains an $n \times n$ matrix of variables $dist(i, j)$, $1 \leq i, j \leq n$. At each point in the execution of the algorithm the value of $dist(i, j)$ is the approximation of $d(i, j)$ at that point. At the termination of the algorithm $dist(i, j) = d(i, j)$ for all $1 \leq i, j \leq n$.

Step 1. The weighted distance between each variable and itself is 0. Set the entries on the diagonal accordingly:

```
for i = 1 to n do
    dist(i, i) := 0
end for
```

Algorithm 3 Floyd – Warshall algorithm (*continued*)

Step 2. Set the weighted distance between any two adjacent nodes to the weight of the edge between them. All other entries are set to infinity:

```

for  $i = 1$  to  $n$  do
  for  $j = 1$  to  $n$  do
    if there is an edge between  $v_i$  and  $v_j$  then
      set  $dist(v_i, v_j) = w(v_i v_j)$ 
    else if there no edge between  $v_i$  and  $v_j$  and  $i \neq j$ 
      set  $dist(v_i, v_j) = \infty$ 
    end if
  end for
end for
```

Step 3. Now cycle through and look for shorter paths with intermediate vertices contained in $\{v_1, \dots, v_k\}$ with $k = 1, 2, \dots, n$:

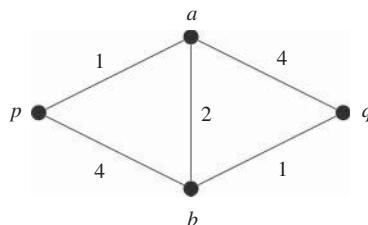
```

for  $k = 1$  to  $n$  do
  for  $i = 1$  to  $n$  do
    for  $j = 1$  do
       $dist(i, j) := \min\{dist(i, j), dist(i, k) + dist(k, j)\}$ 
    end for
  end for
end for
```

Let us look at an example.

Example 7.3.5

We run the Floyd–Warshall algorithm on the graph from Example 7.3.2:



Steps 1 and 2 (as specified in the pseudo code description) produce the matrix

	p	a	b	q
p	0	1	4	∞
a	1	0	2	4
b	4	2	0	1
q	∞	4	1	0

Now consider Step 3. The first iteration of the outer loop ($k = 1$) checks for shorter paths via vertex $v_1 = p$, but no such paths can be found. Indeed, $2 = \text{dist}(a, b) > \text{dist}(a, p) + \text{dist}(p, b) = 1 + 4 = 5$; $4 = \text{dist}(a, q) > \text{dist}(a, p) + \text{dist}(p, q) = 1 + \infty = \infty$; $1 = \text{dist}(b, q) > \text{dist}(b, p) + \text{dist}(p, q) = 4 + \infty = \infty$. Thus, the matrix remains unchanged.

In the second iteration ($k = 2$), checking for shorter paths via vertex $v_2 = a$, we find: $4 = \text{dist}(p, b) > \text{dist}(p, a) + \text{dist}(a, b) = 1 + 2 = 3$; $\infty = \text{dist}(p, q) > \text{dist}(p, a) + \text{dist}(a, q) = 1 + 4 = 5$; $4 = \text{dist}(b, p) > \text{dist}(b, a) + \text{dist}(a, p) = 2 + 1 = 3$; $\infty = \text{dist}(q, p) > \text{dist}(q, a) + \text{dist}(a, p) = 4 + 1 = 5$. This yields the following update matrix:

	p	a	b	q
p	0	1	3	5
a	1	0	2	4
b	3	2	0	1
q	5	4	1	0

In the second iteration ($k = 3$), checking for shorter paths via vertex $v_3 = b$, we find: $5 = \text{dist}(p, q) > \text{dist}(p, b) + \text{dist}(b, q) = 3 + 1 = 4$, $4 = \text{dist}(a, q) > \text{dist}(a, b) + \text{dist}(b, q) = 2 + 1 = 3$, and symmetrically for $\text{dist}(q, p)$ and $\text{dist}(q, a)$. Applying the corresponding updates gives the matrix

	p	a	b	q
p	0	1	3	4
a	1	0	2	3
b	3	2	0	1
q	4	3	1	0

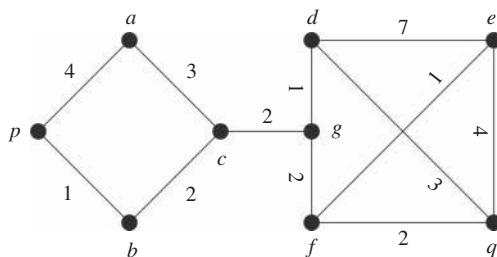
In the last iteration ($k = 4$) no shorter paths can be found via $v_4 = q$, so the matrix of variables remains as it was at the end of the previous iteration.

By comparing the matrix with the graph, it is easy to check that we have indeed found the weighted lengths of the shortest paths between every two vertices.

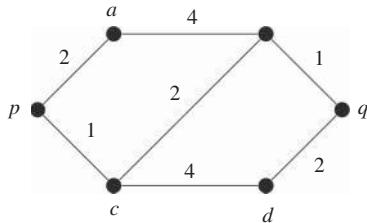
7.3.3. Exercises

- ① Apply Dijkstra's algorithm as well as the Floyd–Warshall algorithm to the following graphs:

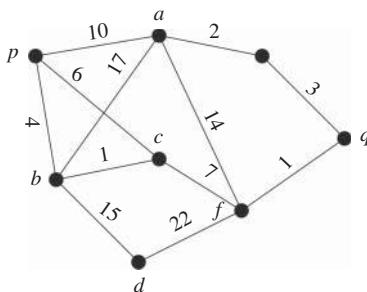
(a)



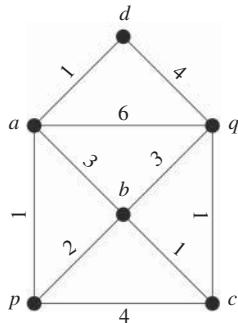
(b)



(c)



- ② Apply the Floyd–Warshall algorithm to the graph from Example 7.3.3.



- ③ Prove that the Floyd–Warshall algorithm is correct. The informal discussion of the algorithm above contains the key ideas needed for this proof.
 ④ How would Dijkstra’s algorithm have to be adapted to make it applicable to digraphs?
 ⑤ How would the Floyd–Warshall algorithm have to be adapted to make it applicable to digraphs?
 ⑥* (a) Describe how the Floyd–Warshall algorithm could be optimized by exploiting the symmetry in the matrix of variables it maintains.
 (b) Would a similar optimization apply to a version of the Floyd–Warshall algorithm for digraphs?
 ⑦* Analyse the runtime complexity of Dijskra’s algorithm.
 ⑧* Analyse the runtime complexity of Dijskra’s algorithm.



Edsger Wybe Dijkstra (1930–2002)

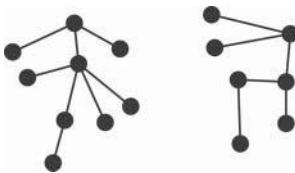
Dijkstra was a Dutch computer scientist. He was born in Rotterdam, the son of a chemist father and mathematician mother. After studying theoretical physics at the University of Leiden, he held a job as a programmer at the Mathematisch Centrum in Amsterdam. As a means of giving public demonstration of Centrum's newly built computer, Dijkstra proposed programming it to find the shortest route between Dutch cities, and it was for this purpose that he developed his now famous shortest path algorithm. He was a member of the team who developed the programming language ALGOL-60, starting in 1958. In 1959 he obtained a PhD from the University of Amsterdam for a Thesis entitled *Communication with an Automatic Computer*. In 1962 he was appointed as Professor of Mathematics at Eindhoven University of Technology. While in Eindhoven he developed the "THE" operating system, which was very influential for later operating systems. In 1984 he was appointed to the Schlumberger Centennial Chair in Computer Sciences at the University of Texas at Austin, where he remained until his retirement in 1999. Dijkstra wrote extensively on computer programming, which he regarded as an activity of the highest intellectual order. His opinions in this regard have been highly influential. He was also a pioneer in the field of distributed computing.

7.4. Trees

Graphs that do not contain cycles are very “well behaved” in many respects and are known as trees. Trees are ubiquitous in the mathematical sciences and in applications. They are used to organize information in hierarchical fashion and as such form one of the fundamental data types in computer science. In this section we will first study undirected trees and then move on to directed trees and their traversal.

7.4.1. Undirected trees

Definition 7.4.1 An acyclic graph is called a *forest*. A connected forest is a *tree*. In a tree, a vertex with degree 1 is called a *leaf* while a vertex with a higher degree is an *internal vertex*.

Example 7.4.2

The graph above is a forest, as it contains no cycles. Note that it has two connected components, which are trees. Indeed, a connected component of a forest, seen as a graph on its own, is always a tree, as it satisfies the definition by being connected and acyclic. The component on the left has 6 leaves and 3 internal vertices, while the component on the right has 4 leaves and 3 internal vertices.

We now explore some of the basic properties of tree in the following lemmas and theorems. This exploration will culminate in Theorem 7.4.10, which gives a very simple and precise relationship between the order and size of any tree.

Lemma 7.4.3 *Let G be a tree with at least two vertices. Then G has at least two leaves.*

Proof: Let P be a path of maximal length in G , say a path from a vertex u to a vertex v . Since G has at least two vertices and is connected, such a (nontrivial) path exists. Firstly, note that u cannot be adjacent to any vertex not on the path P , since that would mean that P could be extended to a longer path, contradicting the maximality of P . Similarly, v cannot be adjacent to any vertex not on P . Secondly, u cannot be adjacent to any vertex on P other than the one immediately following it on P , since this would imply the existence of a cycle in G . Similarly v cannot be adjacent to any vertex on P other than the one immediately preceding it. Putting this together, we conclude that u and v both have degree 1.

Theorem 7.4.4 *A graph G is a tree if and only if there exists a unique $u-v$ path in G for any two vertices $u, v \in V(G)$.*

Proof: Suppose that there is a unique $u-v$ path between every two vertices $u, v \in V(G)$. Then G is connected. Also, if G were to contain a cycle, then between every two distinct vertices, u and v , on the cycle there would be two distinct paths, contradicting the hypothesis. We therefore conclude that G must be a tree.

Conversely, suppose that G is a tree. Since G is connected there is at least one path between every two vertices $u, v \in V(G)$. If there were two distinct paths between some vertex u and another vertex v , there would exist a cycle in G consisting of vertices from these paths, contradicting the assumption that G is a tree.

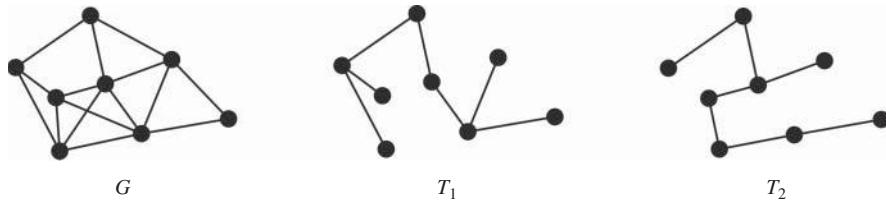
The next theorem provides a precise relationship between the size (number of edges) $m(T)$ and the order (number of vertices) $n(T)$ of a tree.

Theorem 7.4.5 For every tree T it is the case that $n(T) = m(T) + 1$.

Proof: Consider any tree T . We proceed by induction on the order $n(T)$. If $n(T) = 1$, then it must be the case that $m(T) = 0$, and hence we have $n(T) = 1 = 0 + 1 = m(T) + 1$, as desired. Now, assume that $n(T') = m(T') + 1$ holds for all trees T' with $n(T') \leq k$, and suppose that $n(T) = k + 1$. We know from Lemma 7.4.3 that T has at least two leaves. Let v be a leaf of T and $T - v$ the tree obtained from T by removing v and the edge incident with it. Then $m(T - v) = m(T) - 1$ and $n(T - v) = k$, and so $T - v$ falls under the inductive hypothesis. Applying the inductive hypothesis to $T - v$ we get $k = n(T - v) = m(T - v) + 1$, and hence $n(T) = k + 1 = m(T - v) + 1 + 1 = m(T) + 1$.

Definition 7.4.6 A tree T is called a *spanning tree* of a graph G if T is a subgraph of G containing every vertex of G .

Example 7.4.7



A graph G is depicted above together with two of its spanning trees T_1 and T_2 .

Theorem 7.4.8 Every connected graph has a spanning tree.

Proof: If G is acyclic it is itself a tree, and we are done. Suppose that G contains at least one cycle C . If uv is an edge of C then we may delete it and in the resulting graph G' there will still be a path from u to v .

Now let a and b be any vertices of G . Since G is connected there is an $a-b$ path in G . We claim that there is still an $a-b$ path in G' : if the edge uv occurs on the $a-b$ path in G , we may replace it by the $u-v$ path in G' , obtaining an $a-b$ path in G' . Hence G' is connected. If G' still contains a cycle we may repeat this procedure to delete another edge. We continue in this fashion until we obtain an acyclic connected subgraph T of G . As we never delete vertices this subgraph contains all the vertices of G and is the desired spanning tree.

Putting Theorems 7.4.5 and 7.4.8 together we obtain the following corollary.

Corollary 7.4.9 Every connected graph of order n has at least $n - 1$ edges.

Theorem 7.4.10 Let G be a connected (n, m) graph. Then G is a tree if, and only if, $n = m + 1$.

Proof: Let G be a connected (n, m) graph. If G is a tree then, by Theorem 7.4.5, $n = m + 1$. Conversely, suppose that $n = m + 1$. We need to show that G is a tree. By Corollary 7.4.9 G has

the least number of edges it can have as a connected graph. Hence G cannot contain a cycle, for if it did we would be able to remove a cycle edge and still keep the graph connected – a contradiction. Hence, G is a tree.



7.4.2. Computing spanning trees: Kruskal's algorithm

Theorem 7.4.8 tells us that every connected graph has a spanning tree. The proof also gives a method for constructing a spanning tree for any given connected graph, which works by repeatedly breaking cycles by removing edges. Another method, known as Kruskal's algorithm, works by rather choosing which edges to keep. In fact, Kruskal's algorithm also works on weighted graphs and computes a spanning tree of minimum weight of any given connected weighted graph. The weight of the spanning tree is the sum of the weights of its edges. Here is a pseudocode specification of the algorithm. Recall that if G is a graph and $X \subseteq E(G)$, then $\langle X \rangle_G$ denotes the subgraph of G edge-induced by X .

Algorithm 4 Kruskal's algorithm

Input. A connected weighted graph $G = (V, E, w)$

Set $S = E$ and $X = \emptyset$.

while $\langle X \rangle_G$ does not contain every vertex in V **do**

Let e be an edge in S of minimum weight

if $\langle X \cup \{e\} \rangle_G$ is acyclic **then**

Set $S := S - \{e\}$

$X := X \cup \{e\}$

else if $\langle X \cup \{e\} \rangle_G$ contains a cycle

Set $S := S - \{e\}$

end if

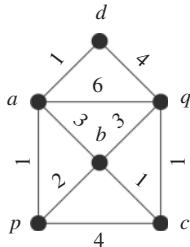
end while

return $\langle X \rangle_G$

For the proof of the correctness of the algorithm, see Exercise 13.

Example 7.4.11

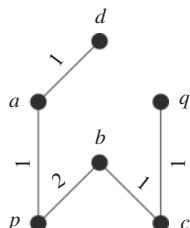
Let us run Kruskal's algorithm on the following weighted graph:



Note that the algorithm involves some nondeterminism: at each step an edge in S of minimal weight is chosen and if there is more than one such edge the choice between them is made randomly. As a result there may be more than one correct execution and more than one spanning tree of minimal weight.

Step	S	X	$V(\langle X \rangle_G) = V(G)?$
0	-	$\{ab, ad, ap, aq, bc, bp, bq, cp, cq, dq\}$	\emptyset NO
1	ad	$\{ab, ap, aq, bc, bp, bq, cp, cq, dq\}$	$\{ad\}$ NO
2	ap	$\{ab, aq, bc, bp, bq, cp, cq, dq\}$	$\{ad, ap\}$ NO
3	bc	$\{ab, aq, bp, bq, cp, cq, dq\}$	$\{ad, ap, bc\}$ NO
4	cq	$\{ab, aq, bp, bq, cp, dq\}$	$\{ad, ap, bc, cq\}$ NO
5	bp	$\{ab, aq, bq, cp, dq\}$	$\{ad, ap, bc, cq, bp\}$ YES

Here is the resulting spanning tree:



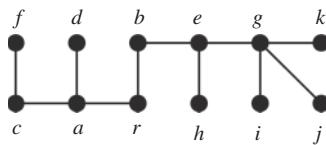
To run Kruskal's algorithm on unweighted graphs, one can simply assign weight 1 (or any other constant weight) to every edge in the graph and then run the algorithm.

7.4.3. Rooted trees

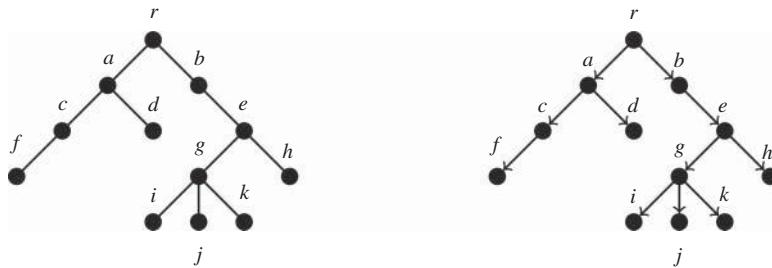
It is sometimes necessary to distinguish a vertex of a tree and call it the **root** of the tree. The root is usually drawn at the top: it is as if the tree has been picked up by the root and is now dangling from it. Once a root has been determined for a tree it is called a **rooted tree**. One can further transform a rooted tree T into a directed tree called a **rooted directed tree T' derived from T** . Directions are assigned to the edges in the direction of the unique path leading from the root to every other vertex in T , as illustrated in Example 7.4.12.

Example 7.4.12

Consider the following tree:



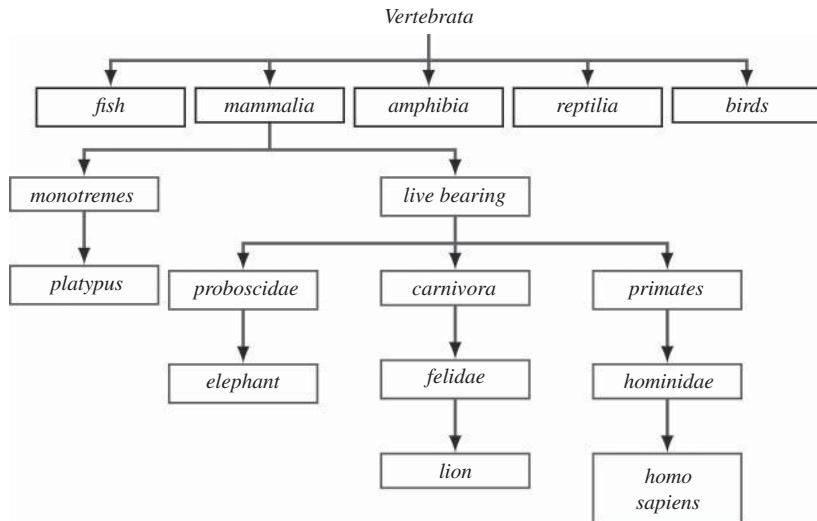
Let us distinguish r as the root of the tree. Redrawing the tree such that it is “dangling” from its root and orienting the edges into arcs yield the following rooted tree and derived rooted directed tree, respectively:



Example 7.4.13

We have encountered trees before in this book: the formula construction and parsing trees and the semantic tableaux of Chapters 3 and 4 are rooted trees.

Rooted trees are also the structures of choice for representing all kinds of hierarchical information. They are used, for example, in concept hierarchies like the taxonomic classification of animals, as in this subtree of the classification of the vertebrates:



Given a rooted tree T with root r , the **level** of a vertex v in the tree is the length of the path from r to v . Thus, the root is on level 0. The **height** of T is the length of the longest path from the root r to a leaf or, equivalently, the maximum level among the vertices.

In a rooted tree, a vertex v is called an **ancestor** of a vertex u if v precedes u on the path from the root to u . In this case u is called a **descendant** of v . If v is the ancestor of u that immediately precedes it on the path from the root to u , then v is the **parent** of u and u is the **child** of v . The children of the same parent vertex are **siblings**.

If the greatest number of children of any given vertex in a rooted tree T is m , then T is called an **m -ary tree**. In the case that $m = 2$ we speak of a **binary tree**. In a binary tree we may designate each child of a parent vertex as either the **left child** or the **right child**, but not as both.

Example 7.4.14

Consider the rooted tree given in Example 7.4.12. It is rooted at vertex r and has height 4. It is a 3-ary tree as no vertex has more than 3 children. Vertices g and h occur on level 3 and are siblings as they are both children of vertex e . The vertex e is an ancestor of vertices g, h, i, j and k , which are its descendants.

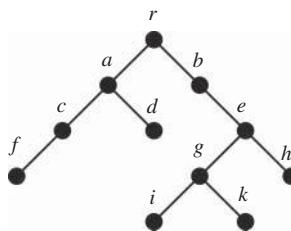


7.4.4. Traversing rooted trees

In many applications it is often necessary to visit every vertex in a rooted tree in some systematic way, e.g. if every member of a family must be read off a family tree or if every concept needs to be retrieved from a concept hierarchy. Systematically visiting each vertex in this way is known as **traversing** the tree. There are two general ways of traversing a rooted tree, known as **depth first** and **breadth first** traversal. We will outline these traversal protocols for binary trees and will then indicate how they can be generalized to arbitrary rooted trees.

Breadth first traversal

In breadth first traversal the vertices are visited level by level in some predefined order on the vertices on a given level, say from left to right, given the way the tree is drawn. For example, the breadth first traversal of the rooted tree



will visit the vertices in the order

$$r, a, b, c, d, e, f, g, h, i, j, k.$$

Depth first traversal

Depth first traversal explores completely a subtree before moving on to the next subtree. It starts at the root and visits all vertices (again, depth first) in the subtree rooted at the left child of the root before doing the same with the subtree rooted at the right child. The depth first traversal is done in **pre-order**, **in-order** or **post-order**, depending on whether the root of the (sub)tree is visited respectively before, in between or after the vertices in the subtrees rooted at the left child and right child are visited. For example, the pre-order depth first traversal of the tree above visits the vertices in the order

$$r, a, c, f, d, b, e, g, i, k, h.$$

The in-order depth first traversal of the same tree is

$$f, c, a, d, r, b, i, g, k, e, h.$$

The post-order depth first traversal of the same tree is

$$f, c, d, a, i, k, g, h, e, b, r.$$

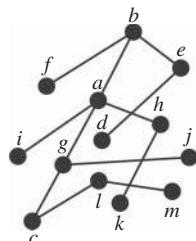


Exercises 2, 4, 13 and 14 below are intended for the computer science track only.

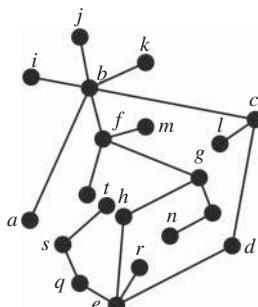
7.4.5. Exercises

- ❶ Consider the following graphs. For each graph:
 - (i) determine whether it is a tree;
 - (ii) if it is a tree, use vertex c as the root and draw the rooted tree and the derived rooted directed tree;
 - (iii) determine the height of the resulting rooted tree;
 - (iv) determine the level of vertex e in the resulting rooted tree.

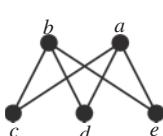
(a)



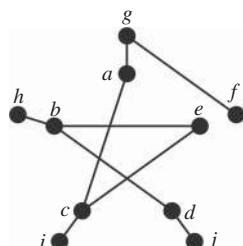
(b)



(c)



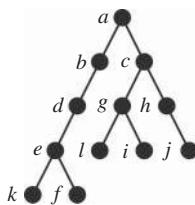
(d)



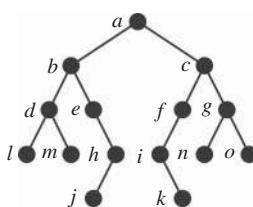
- ❷ Consider the following rooted trees. For each tree give the order in which the nodes are visited if the tree is traversed:

- (i) breadth first;
- (ii) depth first, pre-order;
- (iii) depth first, in-order;
- (iv) depth first, post-order.

(a)



(b)



- ③ For each graph in Exercise 1 that is not itself a tree, determine a spanning tree. Do this either by inspection or by applying Kruskal's algorithm.
- ④ Run Kruskal's algorithm on the weighted graphs in Exercise 1 in Section 7.3.3 to compute spanning trees of minimum weight.
- ⑤ Let T be a tree such that the sum of the degrees of the vertices in T is 22. How many vertices does T have?
- ⑥ Let T be a tree with 13 vertices. What is the sum of the degrees of all the vertices?
- ⑦ Draw a family tree, starting with yourself as the root and ending with your great grandparents as leaves.
- ⑧* Prove that every tree with at least two vertices is a bipartite graph.
- ⑨ Show that the assumption in Theorem 7.4.10 that G is connected is essential, i.e. show that the theorem does not hold for graphs that are not connected. (Hint: find a suitable counterexample.)
- ⑩ For which values of $i \in \mathbb{N}$ will the complete graph K_i be a tree? Prove your answer.
- ⑪ For which values of $i, j \in \mathbb{N}$ will the complete bipartite graph $K_{i,j}$ be a tree? Prove your answer.
- ⑫* Prove that if F is an (n, m) forest with k connected components then $m = n - k$. (Note that the formula given in Theorem 7.4.5 is a special case of this formula.)
- ⑬* Prove that Kruskal's algorithm returns a spanning tree. This proof can be broken down into two parts: firstly, prove that the graph returned by the algorithm is indeed a tree and, secondly, prove that it contains every vertex of the input graph. Proving that the returned tree is indeed one of minimal weight is more challenging and is deferred to Exercise 14
- ⑭** Prove that the spanning tree returned by Kruskal's algorithm is one of minimum weight among the spanning trees of the input graph.

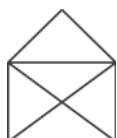


Paul Erdős (1913–1996)

Paul Erdős was one of the most prolific mathematicians of all time, comparable in his productivity only to Euler (see page Section 7.1). He had wide mathematical interests, mainly within discrete mathematics, where he made contributions to number theory, combinatorics, graph theory, probability theory and set theory. Erdős was born in Budapest in Hungary, the son of two mathematics teachers. He became fascinated with mathematics by the age of four, started publishing mathematical papers in his teens and was awarded a doctorate by the University Pázmány Péter in Budapest in 1934 at the age of 21. In the same year he moved to Manchester to take up a postdoctoral research position. The anti-Semitic climate in Hungary at the time made it very difficult for him to return there and he subsequently took up a temporary position at Princeton. This was the start of his life as a wanderer: he never held a permanent academic position. Not that there was a shortage of offers, but he preferred to travel from one mathematics meeting to the next, staying in the homes of colleagues and collaborators. It is said that he would often arrive unannounced at the door of a colleague, stay for a few weeks during which they would collaborate on some mathematics, and then depart again. All his belongings could fit into a suitcase. Erdős published over 1500 papers in his life (for comparison, 50 to 100 is considered a very good achievement for the average mathematician), with over 500 collaborators. This has led to the notion of the *Erdős number*: Erdős himself has Erdős number 0; anyone who has published a paper with him has Erdős number 1; those who do not have Erdős number 1 but have published a paper with someone with Erdős number 1 have number 2, etc. It has been estimated that 90% of the world's mathematicians have an Erdős number smaller than 8. Erdős died of a heart attack while attending a conference in Warsaw.

7.5. Eulerian graphs and Hamiltonian graphs

A well-known puzzle asks you to draw the following picture of a house without lifting your pen and without going over the same line more than once:



This is actually an instance of a graph-theoretic problem that has received a lot of serious mathematical study, namely the problem of determining whether a given graph has a trail (i.e. a walk without repeating edges) that uses every edge. This problem and the problem of determining whether a graph has a cycle visiting every vertex are the topics of the current section.

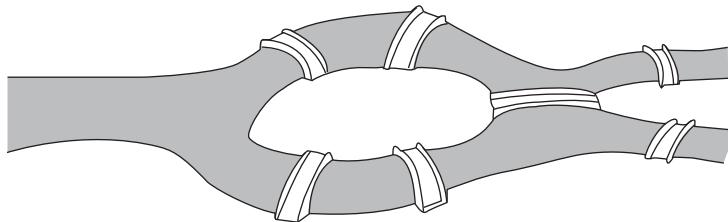
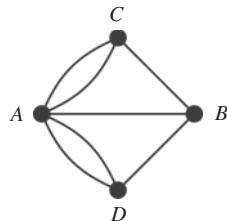


Figure 7.17: The bridges of Königsberg

7.5.1. Eulerian graphs and digraphs

In the 18th century the townsfolk of Königsberg amused themselves by trying to devise a route through the town that would cross every one of the seven bridges over the Pregel river precisely once. A map showing the river and the bridges is given in Figure 7.17. Try as they might, no one was able to come up with such a route and many started to suspect that it fact no such route existed. This story came to the ears of the Swiss mathematician Leonhard Euler, who subsequently proved that no such route can exist.

This episode is widely regarded as the beginning of graph theory. Indeed, we can model the Königsberg bridges as a **multigraph**. A multigraph is a graph in which we allow multiple edges between vertices. With vertices a and b representing the islands and vertices c and d representing the banks, the problem becomes equivalent to asking whether there exists a trail through the following multigraph, which contains every edge (exactly once):

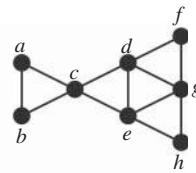
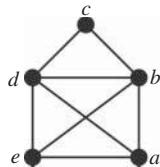


We will not treat multigraphs further, but will rather restrict ourselves to exploring this type of problem on graph. Here are the key definitions.

Definition 7.5.1 An *Eulerian trial* of a graph G is an open trial of G containing all the edges and vertices of G . An *Eulerian circuit* of G is a circuit containing all the edges and vertices of G . A graph that has an Eulerian circuit is called an *Eulerian graph*.

Note that an Eulerian graph must be nontrivial and connected.

Example 7.5.2



The graph on the left has an Eulerian trail, namely $e, b, d, c, b, a, d, e, a$, but no Eulerian circuit.

The graph on the right is Eulerian, as $a, b, c, e, h, g, f, d, g, e, d, c, a$ is an Eulerian circuit.

As it turns out, it is quite easy to determine whether a graph is Eulerian or not, thanks to the following theorem:

Theorem 7.5.3 *Let G be a nontrivial connected graph. Then G is Eulerian if and only if every vertex of G is even.*

Proof: Suppose that G has an Eulerian circuit C . Let v be any vertex of G . If v is not the initial vertex of C , then every time v is encountered on C it enters and leaves via two different edges. Therefore, each occurrence of v represents a contribution of 2 to its degree, and as all the edges incident with it occur in C , it must have an even degree. If, on the other hand, v is the initial vertex of C , the first and last occurrences represent a contribution of 1 each to its degree, while all other occurrences represent a contribution of 2. Hence, also in this case, v must have even degree.

To prove the converse we assume that every vertex of G is even and proceed by induction on the size m of G . The smallest m for which this is possible is 3. The only such graph is K_3 , which is obviously Eulerian. Now assume that $m \geq 4$ and that the claim holds for all graphs of size less than m . Let G be a connected graph of size m such that the degree of each of its vertices is even.

Let u be any vertex of G and let W be a $u-u$ circuit. To see that such a circuit W exists we argue as follows. Let W' be a trail starting at u of maximal length, say a $u-v$ trail. If $u \neq v$, i.e. W' is open, then an odd number of edges of G incident with u are present in W' . However, we know that u has an even degree and so W' can be extended to a trail W'' containing more edges of G . This is in contradiction with the maximality of W' . Hence the assumption that W' is open must be incorrect and W' is the desired $u-u$ circuit W .

If W is an Eulerian circuit for G we are done; otherwise there are edges in G not present in W . We obtain the subgraph G' from G by removing from G all the edges present in W . Since each vertex of W is incident with an even number of edges of W , every vertex in G' has an even degree. Hence every component of G' is a connected graph of which all the vertices have an even degree and with a size smaller than m , which is then Eulerian by hypothesis. Since G is connected, every component of G' has a vertex that also belongs to W . Hence an Eulerian

circuit for G may be constructed by inserting an Eulerian circuit of each component H of G' into W at a vertex of H belonging to W .

Theorem 7.5.4 *Let G be a nontrivial connected graph. Then G contains an Eulerian trail if and only if G has exactly two vertices of odd degree. Furthermore, the Eulerian trail begins at one of these odd vertices and ends at the other.*

Proof: See Exercise 11.

The next definition extends the notions of Eulerian trail, Eulerian circuit and Eulerian graph to digraphs.

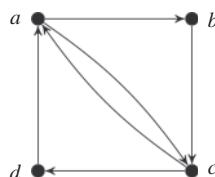
Definition 7.5.5 *Let D be a digraph. A **directed Eulerian trail** of D is a directed open trail containing all the vertices and arcs of D . A **directed Eulerian circuit** of D is a directed circuit containing all the vertices and arcs of D . A digraph that contains a directed Eulerian circuit is called an **Eulerian digraph**.*

The proof of the following theorem is left as Exercise 12.

Theorem 7.5.6 *Let D be a nontrivial connected digraph. Then D is Eulerian if and only if the indegree of every vertex is equal to its outdegree.*

Example 7.5.7

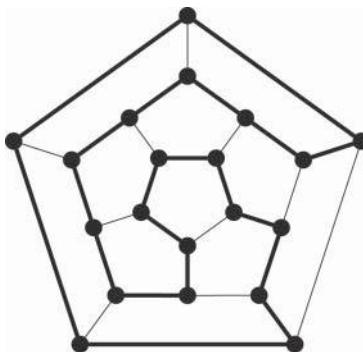
Consider the following digraph:



Since the indegree of every vertex is equal to its outdegree, this digraph should be Eulerian according to Theorem 7.5.6. Indeed, a, b, c, a, c, d, a is a directed Eulerian circuit.

7.5.2. Hamiltonian graphs and digraphs

So far we have considered trails and circuits that traversed every edge in a graph exactly once. A naturally related question is to ask for a path or cycle that visits every *vertex* of a graph exactly once. A graph that contains such a cycle is known as a **Hamiltonian graph**, after Sir William Rowan Hamilton, who was one of the first to study this type of problem. In fact, he invented a game consisting of a solid wooden dodecahedron with a peg at each corner and a piece of string. The object of the game was to wind the string so that it would form a cycle through the pegs passing along the edges of the dodecahedron. The dodecahedron can, like other Platonic solids, be represented as a graph. Here is such a representation, with a cycle visiting each vertex exactly once, indicated in bold:



The formal definition of a Hamiltonian graph is as follows.

Definition 7.5.8 Let G be a graph. A path containing every vertex of G is called a **Hamiltonian path** in G . A cycle in G containing every vertex of G is called a **Hamiltonian cycle** in G . A graph having a Hamiltonian cycle is called a **Hamiltonian graph**.

Thus, the graph of the dodecahedron is Hamiltonian.

Example 7.5.9

Some examples and nonexamples of Hamiltonian graphs:

- Every complete graph of order 3 or greater is Hamiltonian.
- The Petersen graph has a Hamiltonian path, but no Hamiltonian cycle.
- $K_{3,3}$ is Hamiltonian.
- $K_{3,2}$ has a Hamiltonian path, but no Hamiltonian cycle.
- $K_{1,3}$ has neither a Hamiltonian path nor a Hamiltonian cycle.

After the neat characterization of Eulerian graphs (Theorem 7.5.3) in the previous subsection, one might expect something similar for Hamiltonian graphs. Alas, no such simple necessary and sufficient condition for a graph being Hamiltonian is known. Several sufficient conditions are known, the most famous of which is perhaps Ore's theorem.

Theorem 7.5.10 Ore's theorem Let G be an (n, m) graph with $n \geq 3$. If

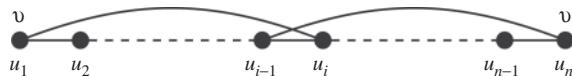
$$\deg u + \deg v \geq n,$$

for all distinct nonadjacent vertices u and v of G , then G is Hamiltonian.

Proof: The proof is by contradiction. Suppose that G is an (n, m) graph with $n \geq 3$ and that $\deg u + \deg v \geq n$ for all distinct nonadjacent vertices u and v of G , but that G is not Hamiltonian. Since any complete graph on $n \geq 3$ vertices is Hamiltonian, we could make G Hamiltonian by adding enough edges to it. Let G' be a maximal non-Hamiltonian graph obtained from

G by adding edges, i.e. adding any edge to G' would turn it into a Hamiltonian graph. Clearly, for any two nonadjacent vertices u and v of G' we have that $\deg_{G'} u + \deg_{G'} v \geq \deg_G u + \deg_G v \geq n$.

Let u and v be any two nonadjacent vertices of G' . From the discussion above $G' + uv$ (the graph obtained from G' by adding edge uv) would be Hamiltonian, i.e. it contains a Hamiltonian cycle $u = u_1, u_2, \dots, u_n = v, u$. Now, for every vertex u_i , $2 \leq i \leq n$, if u is adjacent to u_i then it cannot be the case that u_{i-1} is adjacent to v , or else $u_1, \dots, u_{i-1}, u_n, \dots, u_i, u_1$ would be a Hamiltonian cycle of G' :



This means that for every vertex in $\{u_1, \dots, u_n\}$ to which u is adjacent, there is a vertex to which v is not adjacent. Therefore we have that

$$\deg_{G'} v \leq (n - 1) - \deg_{G'} u,$$

i.e.

$$\deg_{G'} v + \deg_{G'} u \leq (n - 1),$$

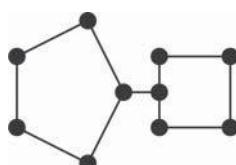
which is a contradiction.

The problem of finding a Hamiltonian cycle in a graph is closely related to the **travelling salesman problem**: a salesman needs to visit a number of cities and return to the city he started from. To save costs he would like to avoid visiting the same city more than once. The travelling salesman problem is known to be NP-complete, which means that it is very unlikely that an efficient sequential algorithm (working in the number of steps bounded by a polynomial on the order of the graph) can exist. It is also related to a variety of scheduling problems and has been very intensively studied by mathematicians and computer scientists.

7.5.3. Exercises

- ① For each of the following graphs, determine whether it is Eulerian (and if so, find an Eulerian circuit) and whether it contains an Eulerian trail (and if so, find such a trail).

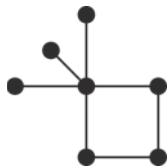
(a)



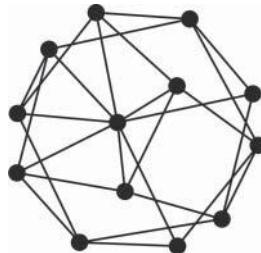
(b)



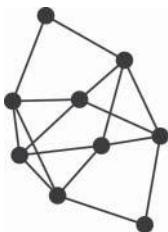
(c)



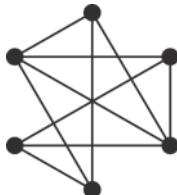
(d)



(e)

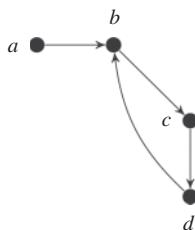


(f)

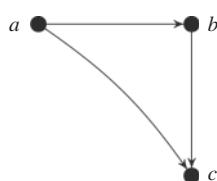


- ② For each digraph below, determine whether it is Eulerian (and if so, find an Eulerian directed circuit) and whether it contains a directed Eulerian trail (and if so, find such a trail).

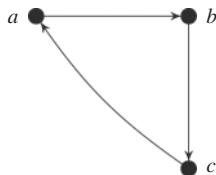
(a)



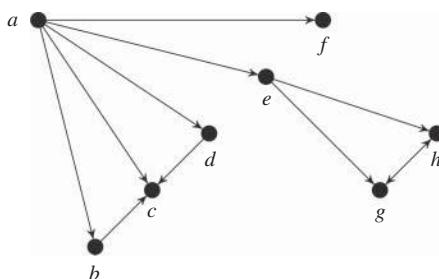
(b)



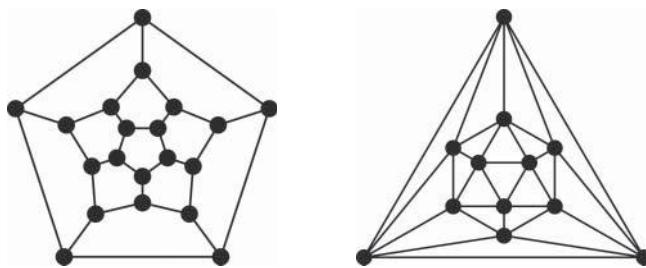
(c)



(d)

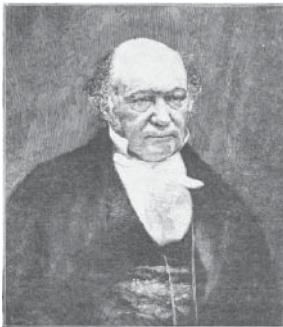


- ③ Below are graph representations of the five Platonic solids. The Platonic solids are the five geometric solids whose faces are identical regular polygons that meet at the same three-dimensional angles. They are known, respectively, as the tetrahedron, octahedron, cube (or hexahedron), dodecahedron and icosahedron, and their representations as graphs are given in the order below. For each of these, determine whether it is Eulerian (and if so, find an Eulerian circuit) and whether it contains an Eulerian trail (and if so, find such a trail).



- ④ For each graph in Exercise 1 determine whether it is Hamiltonian (in which case give a Hamiltonian cycle) and whether it has a Hamiltonian path (in which case give a Hamiltonian path).

- ⑤ Repeat Exercise 4 for the graphs of the Platonic solids given in Exercise 3.
- ⑥ Let G be a graph with two connected components, each of which has an Eulerian trail. Show that there is an Eulerian graph G' that can be obtained by adding two edges to G .
- ⑦ Show that the converse of Ore's theorem does not hold, by giving an example of a Hamiltonian graph that does not satisfy the condition of the theorem.
- ⑧ Characterize the Hamiltonian complete bipartite graphs, i.e. determine for exactly which values of j and k the complete bipartite graph $K_{j,k}$ will be a Hamiltonian graph. Prove that your answer is correct.
- ⑨* Prove that if a graph G is Eulerian, then there exists an Eulerian circuit of G starting and ending at v , for every vertex $v \in V(G)$.
- ⑩* What is the relationship between Eulerian and Hamiltonian graphs? Is every Hamiltonian graph Eulerian? Is every Eulerian graph Hamiltonian? Justify your answers.
- ⑪* Prove Theorem 7.5.4. The proof is similar to that of Theorem 7.5.3, but with the initial and terminal vertices of the trail being different and odd.
- ⑫* Prove Theorem 7.5.6.
- ⑬* State and prove an analogue of Theorem 7.5.4 for directed graphs.



Sir William Rowan Hamilton (1805–1865)

William Rowan Hamilton was an Irish physicist, astronomer and mathematician, who made important contributions to classical mechanics, optics and algebra.

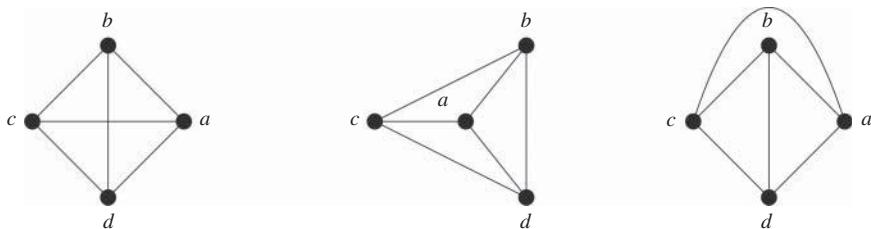
William Hamilton was born in Dublin, the youngest of three children. From a very young age it was clear that he possessed extraordinary intellectual gifts. At the age of three he was sent to live with and be educated by his uncle, the Reverend James Hamilton in Trim, about 50 kilometres from Dublin. Uncle James was an accomplished linguist, and by the age of thirteen Hamilton was fluent in thirteen languages. When the Persian ambassador visited Dublin, the fourteen year old Hamilton sent him a flowery letter of welcome, written in Persian.

When in 1813 the nine year old American calculating prodigy Zerah Colburn visited Dublin, a public contest was arranged between him and the local wonder boy, Hamilton, in the expectation that Hamilton would beat Colburn. In the event Colburn thoroughly trounced Hamilton, which led Hamilton to devote his attention henceforth mainly to mathematics.

In 1823 Hamilton enrolled at Trinity College, Dublin, where he studied classics and mathematics. In 1827, before he had even graduated, he was appointed as Professor of Astronomy at Trinity College, a position that he retained for the rest of his life. At the age of 23 he published the first part of his *A Theory of Systems of Rays* on optics. He went on to apply the methods he introduced there to reformulate classical dynamics in a way that greatly extended the range of problems that could be solved, and in so doing formulated what has become known as *Hamiltonian mechanics*. The later part of his career was devoted to the theory of quaternions, a number system extending the complex numbers.

7.6. Planar graphs

Consider the complete graph on four vertices, K_4 . Here it is drawn in three different ways:



On the left is K_4 as we are used to drawing it; in the middle is the same graph with the positioning of the vertices a and c adjusted so that no edges cross; on the right K_4 has been redrawn in yet another way to avoid edges crossing, this time by suitably bending the edge ac . This example motivates the question: can any graph be drawn on the plane so that the edges do not cross and, if not, which graphs can be drawn in this way?

These questions are interesting in their own right from a purely mathematical perspective, but the answers are also important from the point of view of applications. For example, consider the problem of laying out components on an electronic circuit board. The connections between components cannot be allowed to cross as this could cause a short circuit. If crossings cannot be avoided one or more extra layers need to be created in the circuit board to facilitate the crossings without short circuits, and this is more expensive. We can model this problem by representing the components as the vertices of a graph and drawing an edge between two vertices if and only if the corresponding components must be linked. Determining whether this graph can be drawn without edges crossing answers the question of whether a single layer is sufficient.

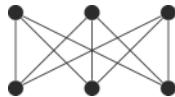
It is now time to introduce the formal definitions that will allow us to study this problem mathematically.

Definition 7.6.1 A graph G is called a **planar graph** if it can be drawn in the plane in such a way that none of its edges cross each other. A planar graph that has been drawn in the plane in such a way is called a **plane graph**.

Example 7.6.2

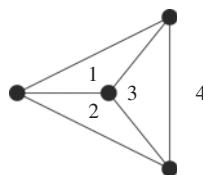
As we saw above, the graph K_4 is a planar graph.

You can try to draw the graph $K_{3,3}$ as a plane graph, but we bet you will fail.



At this stage we can prove that a graph is planar by showing how it can be drawn as a plane graph, but we have no way of proving that a graph is not planar. Later in this section, when we have developed the necessary theoretical tools, we will show that $K_{3,3}$ is indeed not planar.

A plane graph G divides the plane into a number of **regions** or **faces**; these are known as the **faces of G** . When drawn as a plane graph in the following way, K_4 has 4 faces: 3 **internal faces** as well as the rest of the plane representing the 1 **external face**:



Notice that all trees are planar graphs and that any tree drawn as a plane graph has only one face.

The following theorem shows that there is a precise relationship between the order, size and number of faces of a connected plane graph.

Theorem 7.6.3 *If G is a connected plane graph with n vertices, m edges and f faces, then*

$$n - m + f = 2.$$

Proof: We proceed by induction on the number of m edges of G . If $m = 0$, G consists of a single vertex, i.e. $n = 1$, $m = 0$ and $f = 1$ and hence the equality $n - m + f = 2$ holds. Now assume that $m \geq 1$ and that the theorem holds for all connected plane graphs with fewer than m edges. Suppose that G has m edges. If G is a tree we have $n = m + 1$ and $f = 1$ so we have $n - m + f = (m + 1) - m + 1 = 2$ and we are done. Suppose that G is not a tree, i.e. that G contains a cycle. Let e be an edge on this cycle. Then $G - e$, the graph obtained from G by removing edge e , is still connected and planar, and it has $m-1$ edges, n vertices and $f - 1$ faces. (The cycle on which e lies will necessarily enclose one or more faces, so e is on the border between two faces and removing e joins these two faces into one new face.) Hence, by the induction hypothesis we have $n - (m - 1) + (f - 1) = 2$, i.e. $n - m + f = 2$, and the proof is complete.

Notice that the above theorem says that, if G is a connected plane graph with n vertices, m edges and f faces, then $f = 2 - n + m$. In other words, the number of faces of G does not depend on the particular way in which it is drawn in the plane. Therefore, we are justified in speaking of the number of faces of a planar graph, as this number will be the same no matter how the graph is drawn as a plane graph.

Lemma 7.6.4 *Let G be a connected planar (n, m) graph with $n \geq 3$ vertices. Then*

$$3n - m \geq 6.$$

Proof: If G is acyclic we know that $m < n$. Also, as $n \geq 3$, we have $2n - 6 \geq 0$. Therefore, $m \leq n + 2n - 6 = 3n - 6$, i.e. $3n - m \geq 6$.

If G does contain a cycle we sum the number of edges on the boundaries of the faces. Since each face is bound by at least 3 edges the sum must be at least $3f$. As any edge can be on the boundary of at most two faces the sum can be at most $2m$. Therefore $3f \leq 2m$. However, $2 = n - m + f$ so $6 = 3n - 3m + 3f \leq 3n - 3m + 2m = 3n - m$, i.e. $3n - m \geq 6$.

The above theorem says that in a planar graph with at least 3 vertices we always have $m \leq 3n - 6$, i.e. it gives an upper bound on the number of edges in terms of the number of vertices. This makes intuitive sense, since the more edges a graph has in relation to the number of its vertices, the more likely it is that the edges will have to cross when the graph is drawn.

Theorem 7.6.5 *Every planar graph G contains a vertex of degree 5 or less.*

Proof: Let G be any planar (n, m) graph. If $n \leq 2$ then the maximum degree of the vertices in G is at most 1, and the theorem holds. Assume that $n \geq 3$. Then we may apply Lemma 7.6.4 and conclude that $3n - m \geq 6$, i.e. $m \leq 3n - 6$ or $2m \leq 6n - 12$. However, we know that the sum of the degrees of the vertices of G is equal to $2m$. Now if the degree of every vertex in G is at least 6, it must be the case that the sum of the degrees of the vertices is greater than or equal to $6n$, i.e. $6n \leq 2m$. However, that gives $6n \leq 2m \leq 6n - 12$. This cannot be, so there must be a vertex in G that has degree 5 or less.

We are now able to prove that $K_{3,3}$ and K_5 are not planar.

Theorem 7.6.6 *The graph K_5 is nonplanar.*

Proof: K_5 has $n = 5$ vertices and $m = 10$ edges. Therefore $3n - m = 15 - 10 = 5 < 6$, contradicting Lemma 7.6.4. Hence K_5 cannot be planar.

Theorem 7.6.7 *The graph $K_{3,3}$ is nonplanar.*

Proof: Suppose to the contrary that $K_{3,3}$ is planar, so we can draw it as a plane graph. Notice that the shortest length of any cycle in $K_{3,3}$ is 4, and hence no face is bounded by less than 4 edges. Let N denote the sum of the number of edges bounding a face, summed over all f faces. Then $N \geq 4f$. Each edge is counted precisely twice by the sum N as each edge is on the boundary of exactly 2 regions. Furthermore, there are 9 edges, so we must have $N \leq 9 \times 2 = 18$. By Theorem

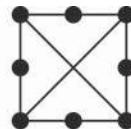
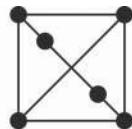
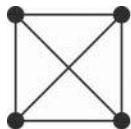
7.6.3 we have $f = 2 - 6 + 9 = 5$. Therefore we have $N \geq 4f = 20$ and $N \leq 18$ – a contradiction. Hence, $K_{3,3}$ cannot be planar.

It is an easy observation that any graph that contains a nonplanar graph as a subgraph cannot itself be planar (why?). Thus any graph that contains a K_5 or $K_{3,3}$ as a subgraph is nonplanar. By contraposition, every planar graph may not have K_5 or $K_{3,3}$ as subgraphs. As we will show further, it turns out that these two conditions are not only necessary but also essentially sufficient for a graph to be planar. This remarkable result is known as Kuratowski's theorem. In order to state that theorem precisely we need one more piece of terminology.

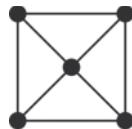
Definition 7.6.8 Let G be a graph. An **elementary subdivision** of G is a graph obtained from G by removing from it some edge $e = uv$ and adding to it a new vertex, say w , and 2 new edges, uw and wv . A **subdivision** of G is a graph obtained from G by a succession of finitely many (possibly 0) elementary subdivisions.

Example 7.6.9

The figure below illustrates three subdivisions of K_4 . The first is obtained by 0 elementary subdivisions, i.e. it is K_4 itself; the second by 2 elementary subdivisions; the third by 4 elementary subdivisions.



Notice that subdivisions do not allow us to eliminate edge crossings. Indeed, by definition an elementary subdivision splits a single edge into two new edges through the insertion of a new vertex and does not allow for the simultaneous splitting of two or more edges by inserting only one vertex. Thus, for example, the following is *not* a subdivision of K_4 :



In the light of the remarks above it is easy to see that a subdivision of a graph is planar if and only if the original graph is planar. The next theorem now follows from these observations and Theorems 7.6.6 and 7.6.7.

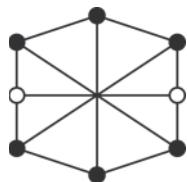
Proposition 7.6.10 If a graph G contains a subgraph that is a subdivision of K_5 or $K_{3,3}$, then G is nonplanar.

This statement is part of Kuratowski's theorem, which also claims the converse, but which is substantially more difficult to prove.

Theorem 7.6.11 (Kuratowski's theorem) *A graph G is planar if and only if it contains no subgraph that is a subdivision of K_5 or $K_{3,3}$.*

Example 7.6.12

The following graph is nonplanar as it contains a subdivision of $K_{3,3}$:

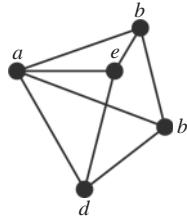


The vertices of the subgraph $K_{3,3}$ are coloured black.

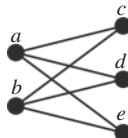
7.6.1. Exercises

- ① Determine whether each of the following graphs is planar or not. If it is planar, draw it as a plane graph. If it is nonplanar, apply Kuratowski's theorem to prove that it is not planar by identifying a subgraph that is a subdivision of $K_{3,3}$ or K_5 .

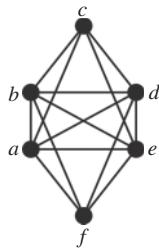
(a)



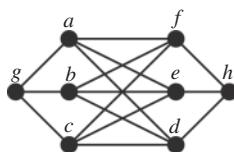
(b)



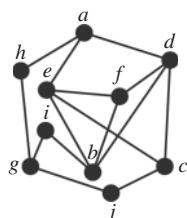
(c)



(d)



(e)



- ② Determine the greatest order a complete planar graph can have.
- ③ A certain connected planar graph has 6 vertices, each of degree 3. How many edges and faces does the graph have?
- ④ A certain connected planar graph has 6 vertices, 1 of degree 2, 4 of degree 3 and 1 of degree 4. How many edges and faces does the graph have?
- ⑤ Electronic components A to G need to be connected to each other as indicated in the following table, ✓ indicating the presence of a connection and ✗ indicating the absence of a connection:

	B	C	D	E	F	G
A	✓	✗	✗	✗	✗	✗
B		✓	✗	✗	✗	✗
C			✓	✓	✓	✓
D				✗	✓	✓
E					✓	✓
F						✓

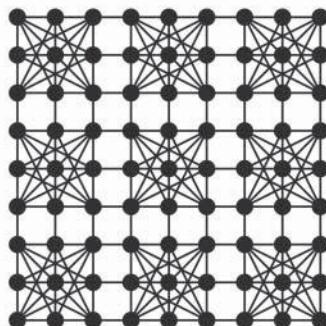
Model this information as a graph and use it to determine whether these components in this configuration can be embedded in a single layer on a circuit board.

- ⑥ (a) What is the maximum number of edges that a connected planar graph of order 6 can have?
 (b) Draw a connected plane graph of order 6 with this number of edges.
- ⑦ (a) What is the maximum number of edges that a connected planar graph of order k , $k \geq 3$, can have?
 (b) Prove that in a plane graph of order k with this number of edges each face is bounded by exactly 3 edges.
- ⑧ Draw two subdivisions of K_5 , one obtained by 1 elementary subdivision and the other obtained by 4 elementary subdivisions.
- ⑨ Show that the converse of Lemma 7.6.4 does not hold by giving a suitable graph as a counterexample.
- ⑩ Give an example of a planar graph in which every vertex has degree 5 or greater.
- ⑪ Show that the inequality $3n - m \geq 6$ given in Lemma 7.6.4 is, in a sense, the best possible, by giving an example of a connected planer graph with $n \geq 3$ in which $3n - m = 6$.
- ⑫ Let G be a planar (n, m) graph with f faces. Find the values of $n - m + f$ if G has
 (a) 2 components,
 (b) 3 components,
 (c) k components.
- ⑬* Let G be a graph and G' a subdivision of G . Prove that G is planar if and only if G' is planar.
 ⑭* Generalize Lemma 7.6.4 to planar graphs that are not necessarily connected, but in which each component has at least 3 vertices.

Graph theory and Sudoku

In the well-known Sudoku puzzle, players are presented with a 9×9 grid partially filled with numbers between 1 and 9. The object is to complete the grid by filling in further numbers in such a way that each of the numbers 1 to 9 appears in each column, each row and each of the 9 smaller 3×3 grids.

5	3		6	7				
6			8		9	3		
	9	8					4	
9				5				4
7			9		4			3
8				2	3			9
	1				4	6		
			4	1	7			5
				9		1	2	



This puzzle can be reformulated as a graph colouring problem by representing the grid as a graph with vertices corresponding to the cells of the grid and edges between vertices corresponding to cells in the same 3×3 grid or in the same column or row. The given numbers in the puzzle correspond to a partial colouring of the graph, and solving the puzzle corresponds to extending the partial colouring to a colouring of the graph using 9 colours. Such a graph of the Sudoku grid is given above. Note that some edges connecting vertices in the same row or in the same column have been omitted to reduce clutter. The missing edges can be easily inferred.

This graph theoretic modelling can be used to answer all sorts of interesting questions about Sudoku. It is known that it is possible to construct Sudoku puzzles with unique solutions by specifying only 17 numbers. Until recently it was unknown whether it is possible to do this using only 16 numbers. There are $C(81, 16) = 33\,594\,090\,947\,249\,085$ ways to choose 16 vertices to colour and approximately 2.241×10^{38} partial colourings of the Sudoku graph colouring 16 vertices. Accordingly, going through each possibility by brute force to check if it results in a unique colouring is not feasible: it would take an estimated 300 000 years on a current supercomputer. However, Gary McGuire of University College Dublin recast the problem using the graph theoretic notion of “hitting sets”, and in doing so, he was able to formulate a more efficient algorithm. He and his team at the Irish Centre for High-End Computing in Dublin ran this algorithm on supercomputers for the whole of 2011, and in 2012 they were able to announce that no colouring of 16 vertices of the Sudoku graph extends uniquely to a 9-colouring, i.e. no Sudoku puzzle with only 16 numbers given has a unique solution.

7.7. Graph colourings

In this section we consider assignments of colours to the vertices of graphs in such a way that no two adjacent vertices are assigned the same colour. This problem may sound artificial, but it has a number of real-life applications, e.g. to scheduling, image capturing and segmentation, clustering, networking, data mining, etc. Here we will formulate the famous four-colour theorem, which resisted numerous attempts to prove for many decades, and we prove the weaker five-colour theorem.

7.7.1. Colourings

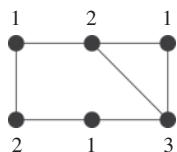
Graph colourings are used to solve a wide variety of problems, the classical example being the following. A radio station is assigned a frequency at which it broadcasts. If two radio stations with overlapping broadcast areas were to broadcast at the same frequency, or at frequencies that are not sufficiently different, interference would result. Since only a limited number of different frequencies are available, broadcast authorities typically have to solve the problem of having to assign the same frequency to different radio stations, but in such a way that no interference results. This is where graph theory comes into play: we model the problem by representing it as a graph where each radio station is represented by a vertex and two vertices are adjacent if and only if the radio stations they represent have overlapping broadcast areas. Suppose there are N different frequencies available. Solving the problem now becomes equivalent to finding a way to assign one of the numbers (or colours) $1, 2, 3, \dots, N$ to each of

the vertices in such a way that no two adjacent vertices are assigned the same number. Here are the formal definitions.

Definition 7.7.1 A *colouring* of a graph G is an assignment of colours to the vertices of G , one colour to each vertex, so that adjacent vertices are assigned different colours. A colouring using k colours is called a *k -colouring*. A graph is *k -colourable* if there exists an s -colouring of G with $s \leq k$. The minimum integer k for which G is k -colourable is called the *chromatic number* of G and is denoted by $\chi(G)$. If $\chi(G) = k$ then G is said to be *k -chromatic*.

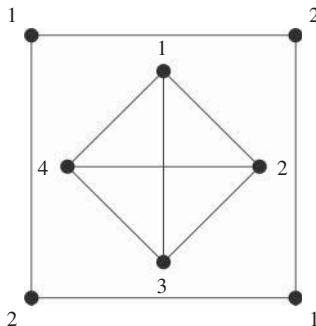
Example 7.7.2

Consider the following graph:



Indicated in the sketch is a 3-colouring of the graph using the three colours 1, 2 and 3. Thus the graph is 3-colourable. As you can check, it is impossible to colour the graph with 2 colours or less. Hence the minimum integer k for which the graph is k -colourable is 3, and its chromatic number is 3.

Now consider the graph:



This is 4-colourable, as we have indicated in the sketch. Indeed, the chromatic number is 4, as every two vertices on the outer square are adjacent to each other (they induce K_4 as a subgraph), so using fewer than four colours will result in two adjacent vertices being coloured the same colour.

In general, the chromatic number of the complete graph K_n is obviously k , since every two vertices are adjacent, so no colour can be used twice. It is also easy to see that if a graph G has

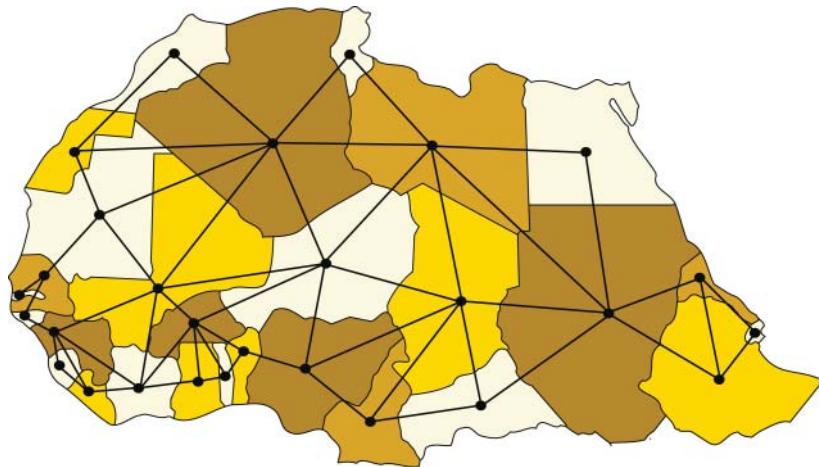


Figure 7.18: A map of Northern Africa represented as a graph and coloured using four colours

components G_1, G_2, \dots, G_k then $\chi(G) = \max\{\chi(G_1), \chi(G_2), \dots, \chi(G_k)\}$. This observation allows us to consider only the chromatic numbers of connected graphs.

7.7.2. The four- and five-colour theorems

For many years the most famous unsolved problem in graph theory was the **four-colour problem**, formulated at least as early as 1852. It seemed to be the case that any map (say a map of the countries of the world) could be coloured using only four colours, so that no two countries sharing a border were coloured using the same colour. What made the four-colour problem a *problem* was the fact that no one could come up with either a proof or a counterexample to this conjecture. The conjecture was at last proved to be true by Kenneth Appel and Wolfgang Haken in 1976.⁴ The proof required the checking of a large number of necessary case distinctions using a computer program. To check all the cases required 1200 hours of computer time.

Let us reformulate this statement into more familiar graph theoretic terms. We represent each country by a vertex and draw edges between any two countries that share a border. Then clearly the original map can be coloured in the desired way using four colours if and only if the graph so constructed is 4-colourable. Furthermore, any graph obtained in this way from a map will be a planar graph, and from every planar graph it is possible to draw a map. Hence there is a one-to-one correspondence between planar graphs and maps (in the plane). We illustrate this correspondence in Figure 7.18 with a map of the countries of Northern Africa with the appropriate graph superimposed.

Via this reformulation the four-colour theorem then becomes as follows.

Theorem 7.7.3 *Every planar graph is 4-colourable.*

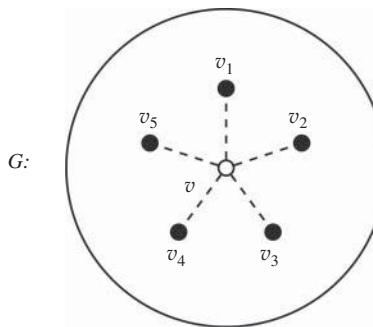
A weaker theorem that is considerably easier to prove, however, is the five-colour theorem.

Theorem 7.7.4 *Every planar graph is 5-colourable.*

⁴ K. Appel and W. Haken, Every Planar Map is Four Colorable, Part I. Discharging, *Illinois Journal of Mathematics*, 21: 429–490, 1977.

Proof: Let G be any connected plane graph. We proceed by induction on the number n of vertices. If $n \leq 5$ the graph is obviously 5-colourable. We therefore assume that all planar graphs with fewer than n vertices, $n > 5$, are 5-colourable, and suppose that G is a plane graph with n vertices. By Theorem 7.6.5 there is a vertex v in G such that $\deg v \leq 5$. We delete v from G to obtain the plane graph $G-v$. By the induction hypothesis $G-v$ is 5-colourable. Let a 5-colouring of $G-v$ with colours 1, 2, 3, 4 and 5 be given. If some colour is not used to colour the vertices adjacent to v in G then that colour may be used to colour v , producing a 5-colouring of G . If not, v has degree 5 and all 5 colours are used to colour the 5 vertices adjacent to v .

Assume, without loss of generality, that the vertices adjacent to v in G are v_1, v_2, v_3, v_4 and v_5 , and that v_i is coloured with colour i , $1 \leq i \leq 5$. We may also assume that v_1, v_2, v_3, v_4 and v_5 are arranged cyclically around v , in that order.



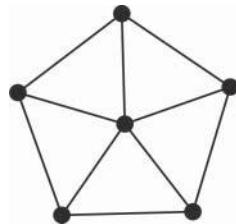
Now consider the vertices v_1 and v_3 and let H be the subgraph of $G-v$ induced by those vertices coloured 1 or 3. If v_1 and v_3 belong to different components of H , then we can interchange the colours (1 and 5) assigned to the vertices in the component of H containing v_1 and still have a 5-colouring of $G-v$ (why?). In the 5-colouring obtained in this way none of the vertices v_1, v_2, v_3, v_4 or v_5 is coloured 1. We may therefore colour v with colour 1, producing the desired 5-colouring of G .

Now suppose that v_1 and v_3 belong to the same component of H . Consequently there is a v_1-v_3 path P in $G-v$, the vertices of which are all coloured 1 and 3. This path P , together with the path v_1, v, v_3 produces a cycle C in G that encloses either v_2 or v_4 and v_5 . As G is connected it must contain a v_2-v_4 path. This path must cross the cycle C , but as G is a plane graph this must happen in a vertex, i.e. the v_2-v_4 path, and the cycle C must share a vertex. Hence there can be no v_2-v_4 path in $G-v$, the vertices of which are all coloured 2 and 4. Denote by F the subgraph of $G-v$ induced by all vertices coloured 2 or 4. Then v_2 and v_4 are in different components of F . Interchange the colours (2 and 4) of the vertices in the component containing v_2 . This results in a 5-colouring of $G-v$ in which none of the vertices adjacent to v is coloured 2. Hence colouring v with 2 results in the desired 5-colouring of G .

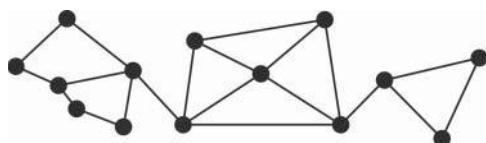
7.7.3. Exercises

- ① For each of the following graphs, determine its chromatic number. Give a colouring using the number of colours equal to the chromatic number and indicate why fewer colours will not do.

(a)



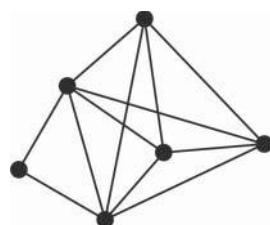
(b)



(c)



(d)



(e)



- ② What is the chromatic number of a graph consisting of a single cycle of even length and also of odd length?
- ③ What is the chromatic number of the complete bipartite graph $K_{3,3}$, of the complete bipartite graph $K_{4,6}$ and of the complete bipartite graph $K_{101,98}$?
- ④ Radio stations in cities A, B, C, D, E, F and G have to be assigned broadcasting frequencies. Two radio stations can only be assigned the same broadcasting frequencies if they are 500 km or more apart. The distances between the cities are given in the following table:

	A	B	C	D	E	F	G
A	123	773	596	372	373	501	
B		787	666	453	321	464	
C			372	500	519	393	
D				227	546	507	
E					419	443	
F						146	

Determine the least number of different broadcasting frequencies required. Hint: model the problem as a graph colouring problem by representing cities with nodes and making nodes adjacent if the corresponding cities are less than 500 km apart.

- ⑤ Many scheduling problems can naturally be modelled as graph colouring problems; for example: Ivy College must schedule exams in such a way that (a) there are no clashes, i.e. no student must write two or more exams in the same time slot, and (b) as few time slots as possible are used. There are students taking the following combinations of courses:

- Mathematics, Statistics, Applied Mathematics, Computer Science
- Chemistry, Physics, Mathematics, Statistics
- Zoology, Botany, Statistics, Chemistry
- Philosophy, English, Mathematics, Psychology

Model this problem as a graph colouring problem by representing the courses as vertices and making two vertices adjacent if and only if there is a student taking both of the corresponding subjects. The time slots are then the colours in a colouring of this graph. What is the minimum number of different time slots required?

- ⑥ The set $\{0, 1, 2, \dots, 11\}$ must be partitioned in such a way that no two elements in the same cluster of the partition are congruent modulo 3 (i.e. have the same remainder upon division by 3). Determine the least number of clusters such a partition can have by modelling the problem as a graph colouring problem.

- ⑦ Mrs Winterbottom has 9 cats. When going out she must put the cats in different rooms of her house so that no fights break out among the cats. She knows that: (a) Tiger fights with all the others except Socks; (b) Socks fights with Jack, Nefertiti and Pippin; (c) Jack fights with Tiger, Socks, Sylvie and Marx; (d) Nefertiti fights with Mopsy and Balthasar; (e) Balthasar fights with Tiger, Pippin and Marx.

Determine the number of different rooms Mrs Winterbottom requires for the cats, by modelling the situation as a graph colouring problem.

- ⑧ True or false: if G is planar then $\chi(G) = 4$. Motivate your answer.
 ⑨ True or false: if G contains no subdivision of K_5 or $K_{3,3}$ as a subgraph then G is 4-colourable. Motivate your answer.

- ⑩ Let T be a tree. What are the possible chromatic numbers of T ? What are the possible chromatic numbers of a forest?
 ⑪ (a) Formulate the converse of the four-colour theorem.
 (b) Is the converse of the four-colour theorem true? Motivate.

The four-colour problem

The four-colour problem had a long life before it eventually became the four-colour theorem. In 1852 Francis Guthrie (later Professor of Mathematics at the University of Cape Town) noticed that a map of the counties of England could be coloured using only four colours. He wondered if four colours would always suffice for any map. He, or his brother Frederick, proposed the problem to Augustus De Morgan (see the box at the end of Section 3.5 in Chapter 3) who liked it and suggested it to other mathematicians. Interest in the problem increased after Arthur Cayley presented it to the London Mathematical Society in 1878. The next year Alfred Bray Kempe (a British lawyer) gave a proof of the conjecture. His proof models the problem in terms of graphs and breaks it up into a number of necessary cases to be checked. Another proof was given by Peter Tait in 1880. It seemed that the four-colour problem had been settled in the affirmative.

However, in 1890 Percy John Heawood found that Kempe's proof missed one crucial case, but that the approach could still be used to prove that five colours are sufficient to colour any map. In the following year Tait's proof was also shown to be flawed, this time by Julius Petersen, after whom the Petersen graph is named. The four-colour problem was therefore again open, and would remain so for the next 86 years. In that time it attracted a lot of attention from professional mathematicians and good (and not so good) amateurs alike. In the words of Underwood Dudley:

The four-color conjecture was easy to state and easy to understand, no large amount of technical mathematics is needed to attack it, and errors in proposed proofs are hard to see, even for professionals; what an ideal combination to attract cranks!

The four-colour theorem was finally proved in 1976 by Kenneth Appel and Wolfgang Haken at the University of Illinois. They reduced the problem to a large number of cases, which were then checked by computer. This was the first mathematical proof that needed computer assistance. In 1997 N. Robertson, D.P. Sanders, P.D. Seymour and R. Thomas published a refinement of Appel on Haken's proof, which reduces the number of necessary cases, but which still relies on computer assistance. The search is still on for a short proof that does not require a computer.

Index

$R[X]$, 36
 $R^{-1}[X]$, 36
 $\Delta(G)$, 358
 $\delta(G)$, 358
 $\text{cod}(f)$, 28
 $\delta(G)$, 358
 $\text{dom}(f)$, 28
 $\mathbb{Z}, \mathbb{Z}^+, \mathbb{Q}, \mathbb{Q}^+, \mathbb{R}, \mathbb{R}^+$, 1
 $\text{rng}(f)$, 28
 $f[X]$, 28
 $R, |_B$, 37
 $f, |_X$, 30

absorption, 124
acyclic graph, 363
addition in \mathbb{Z} , 249
Addition principle, 280
Additive counting principle, 280
additive inverse in \mathbb{Z} , 251
adjacency in a digraph, 365
adjacency matrix, 370
adjacent from, 365
adjacent to, 365
algorithm, 235
antecedent, 91
arc, 364
Aristotle, 100
arithmetic modulo n , 250

arrangement, 278
nondistinct objects, 304
associativity, 124
assumptions, 102
attribute, 80, 81
auxiliary symbol, 139
axiom of choice, 66, 68

biconditional, 9, 90
bijection, 30
binomial theorem, 288
Boolean algebra, 25
Boolean connectives, 90
Braille, 312

canonical mapping, 53
Cantor, Georg, 35
capture, 166
cardinal number, 72
cardinality, 72
Catalan numbers, 319
Cauchy product, 330
chain, 65
characteristic equation, 322
characteristic roots, 323
Chinese remainder theorem, 257
chromatic number of a graph, 412
clausal form, 129, 198

- clause, 129
 - empty, 129
 - first-order, 198
 - unit, 129
- cluster, 52
- CNF, 127
- co-domain of a function, 6, 28
- combinatorial games, 336
- common divisor, 225
- common multiple, 232
- commutativity, 124
- complete mathematical induction, the principle of, 221
- complete residue system modulo n , 250
- composite number, 237
- compound propositions, 9
- conclusion, 104
- conditional, 9, 90
 - conditional probability, 348
- congruence, 243
 - congruence relations, 243
 - congruences, basic properties, 243
- conjunction, 9, 90
 - elementary, 127
- conjunctive normal form
 - first-order, 195
- connected component, 363
- consequent, 91
- constants, 137
- construction sequence of a formula, 92
 - construction tree, 92
- continuum hypothesis, 74
- contradiction, 121
- contradictory formula, 96
- countable set, 71
- countermodel of a formula, 171
- covering relation, 60
- CPHP, 301
- cryptography, 269
- data type, 81
- decryption, 269
- deductive consequence, 111
 - deductive system, 111
 - adequate, 112
 - complete, 111
 - sound, 111
 - definition
 - inductive, 92
 - recursive, 92
 - denumerable set, 71
 - derivation
 - resolution-based, 130
 - first-order, 204
 - derivative implications, 106
 - detachment rule, 104
 - digital signature, 269
 - digraph, 364
 - Dijkstra, Edsger Wybe, 385
 - Diophantine equations, 253
 - linear, 253
 - of two unknowns, 253
 - directed edge, 364
 - directed graph, 364
 - Dirichlet's principle, 299
 - Dirichlet, Johann Peter Gustav Lejeune, 304
 - disjunction, 9, 90
 - elementary, 127
 - disjunctive normal form
 - first-order, 195
 - distance in a graph, 363
 - distributions
 - different objects, 306
 - identical objects, 314
 - distributivity, 124
 - divide, 224
 - divisibility, 224
 - divisibility, properties of, 224
 - division of relations, 87
 - division with remainder, 224
 - divisor, 224
 - DNF, 127
 - domain
 - of a relation, 36
 - domain of a function, 6, 28
 - domain of discourse, 4, 136

- edge set, 357
- edge, 357
- edges
 - adjacent, 357
- Erdős, Paul, 395
- element, 2
- elementary conjunction
 - first-order, 195
- elementary disjunction
 - first-order, 195
- encryption, 269
- encryption function, key, 269
- equinumerous sets, 69
- equivalence class, 52, 248
- equivalence classes modulo n , 249
- equivalence relation, 46, 52, 248
- Euclidean algorithm, 230
- Euler's function, 261
- Euler's theorem, 262
- Euler, Leonhard, 369
- Eulerian circuit, 396
 - directed, 398
- Eulerian digraph, 398
- Eulerian graph, 396
- Eulerian trail
 - directed, 398
- Eulerian trial, 396
- evaluation game, 156
 - initial configuration, 156
- event, 343
- events
 - complementary, 343
 - mutually exclusive, 343
- existential quantifier, 139
- experiment, 343
- faces of a plane graph, 405
- falsifier, 156
- falsifying truth-assignment, 96
- Fermat primes, 241
- Fibonacci, 328
- Fibonacci numbers, 319, 325, 342
- Fibonacci spiral, 342
- finite probability space, 345
- finite set, 71
- first-order logic, 89, 135
- first-order instance, 143
- first-order language, 138
 - nonlogical symbols, 138
- first-order structure, 137
- forest, 385
- formation rules, 92
- formula
 - first-order, 141
 - atomic, 141
 - clean, 165
 - closed, 164
 - ground, 141
 - main connective, 142
 - open, 164
 - satisfiable, 172
 - sentence, 164
 - subformula, 143
 - true in a structure, 154
 - valid, 172
 - logically valid, 95
 - propositional, 91
 - satisfiable, 96
 - signed, 96
- formulae
 - Boolean, 92
 - first-order
 - logically equivalent, 173
 - logically equivalent, 121
 - signed
 - contradictory pair, 114
- function, 6, 28, 137
 - bijective, 30
 - co-domain, 28
 - co-domain of, 6
 - composition, 31
 - domain, 28
 - domain of, 6
 - extension, 30
 - graph of, 6, 47
 - image, 28
 - injective, 30
 - inverse, 31

- function, (*continued*)
 - pre-image, 28
 - range, 28
 - restriction, 30
 - surjective, 30
 - target set, 28
 - value, 28
- function:graph, 29
- fundamental theorem of arithmetic, 236, 238
- fuzzy set, 50
- generating function, 329
- generating functions:Cauchy product, 330
- Golden ratio, 342
- graph colouring, 412
- graph, 357
 - bipartite, 360
 - complete, 360
 - connected, 363
- graph of a function, 6, 47
- greatest common divisor, 225
- greatest lower bound, 63
- Hamilton, Sir William Rowan, 403
- Hamiltonian cycle, 399
- Hamiltonian graph, 399
- Hamiltonian path, 399
- Hasse diagram, 61
- idempotency, 124
- implication, 9, 90
 - contrapositive, 107
 - converse, 106
 - inverse, 107
- implication, fallacies, 106
- incidence, 357
- incidence in a digraph, 365
- incidence matrix, 370
- inclusion–exclusion principle, 293
- indegree, 365
- independent events, 351
- individual variable, 138
- inference rule, 104
- instance, 104
- logically sound, 104
- infimum, 63
- infinite set, 71
- initial vertex, 362, 365
- injection, 30
- integers modulo n , 249
- integral linear combination, 226
- internal vertex, 385
- interpretation, 152
 - standard, 152
- inverse
 - of a relation, 41
- k -combination with repetition, 313
- kernel equivalence, 56
- Laplace, Pierre-Simon de, 355
- law of excluded middle, 95
- law of noncontradiction, 95
- leaf, 385
- least common multiple, 232
- least upper bound, 63
- Leonardo of Pisa, 328
- lexicographic order, 66
- linear congruence, 254
- linear order, 46
- literal, 127
 - first-order, 195, 198
- logic
 - first-order, 89, 136
 - semantics, 136
 - syntax, 136
 - predicate, 89
 - propositional, 89
 - logical connectives, 9
 - logical consequence, 101, 176
 - logical equivalence, 121
 - logical law, 95
 - absorption, 123
 - associativity, 123
 - commutativity, 123
 - De Morgan's laws, 121

- distributivity, 123
- excluded middle, 121
- idempotency, 123
- lower bound, 63
- Lucas numbers, 319
- main connective, 92
- main subformula, 92
- map, 28
- mapping, 28
- mathematical induction, the principle of, 220
- maximal element, 64
- Mersenne primes, 241
- minimal element, 64
- model checking game, 156
- model of a formula, 171
- modular arithmetic, 250
- modulo
 - residue, 243
 - congruent, 243
 - remainder, 243
- modulo n , equivalence classes, 248
- modulo n , residue systems, 248
- Modus Ponens, 104
- multigraph, 396
- multinomial
 - coefficient, 309
 - theorem, 308, 309
- multiple, 224
- multiplication in \mathbb{Z} , 249
- multiplication principle, 275
 - commuting events, 276
 - generating functions, 330
- multiplicative inverse in \mathbb{Z} , 251
- natural join, 85
- necessary, 91
- negation, 9, 90
- negation normal form, 124
- Nim, 336
- nontrivial graph, 358
- normal form
 - conjunctive, 127
 - disjunctive, 127
- order
 - partial, 59
 - order isomorphism, 75
 - order of a digraph, 365
 - order of a graph, 357
 - order of an integer, 266
 - order preserving map, 75
 - ordered pair, 4
 - ordinal, 76, 77
 - limit, 77
 - successor, 77
 - ordinality of a set, 76, 77
 - outdegree, 365
- paramodulation, 207
- parsing tree, 93
- partial order, 46, 59
 - induced, 60
 - partially ordered set, 59
 - partite set, 360
- partition, 53, 248
 - cluster, 248
- Pascal's triangle, 292, 287
- Pascal, Blaise, 317
- path in a graph, 362
- Peirce, Charles, 88
- perfect k th power, 241
- permutation, 278
 - nondistinct objects, 304
 - with repetition/replacement, 315
- PHP, 299
- pigeonhole principle, 299
 - congested, 301
- planar graph, 404
- plane graph, 404
- poset, 59
- powerset, 4
- powerset algebra, 25
- pre-order, 59
- predicate, 137
- predicate logic, 89
- premise, 104

prenex conjunctive normal form, 195
 prenex disjunctive normal form, 195
 prenex form, 195
 matrix, 195
 prefix, 195
 primary residue system modulo n , 250
 prime number, 237
 prime numbers distribution, 239
 primitive propositions, 9
 principle of descending chains, 220
 principle of well-ordering, 220
 private key, 269
 probability
 conditional, 348
 probability measure, 345
 probability space, 345
 projection operation, 83
 proof
 by assumption of the contrary, 212
 by contradiction, 212
 by contraposition, 214
 proposition, 9
 propositional argument, 104
 propositional arithmetic, 10
 propositional connectives, 139
 propositional constants, 91
 propositional inference, 104
 propositional logic, 89
 propositional logical consequence, 101
 public key, 269
 public key cryptography, 269

quantifier, 13
 scope, 164
 quantifiers, 139
 first-order, 139
 quotient, 225
 quotient set, 53

r -Permutation, 278
 Ramsey, Frank Plumpton, 376
 range
 of a relation, 36
 range of a function, 28

reachability in graphs and digraphs, 371
 recurrence relation, 318
 characteristic equation, 322
 characteristic roots, 323
 constant coefficients, 322
 homogeneous, 322
 linear, 322
 reduced residue system modulo n , 250
 reflexive, 248
 regions of a plane graph, 405
 relation, 5, 35
 antisymmetric, 45
 binary, 5
 complementation, 39
 composition, 41
 direct image, 36
 domain, 36
 inverse, 41
 inverse image, 36
 irreflexive, 45
 range, 36
 reflexive, 45
 symmetric, 45
 transitive, 46
 relation schema, 81
 relational algebra, 84
 relational database model, 80
 relational database schema, 81
 relatively prime, 236
 relatively prime in pairs, 236
 remainder, 225
 renaming, 165
 residue systems, 250
 resolution
 propositional, 126
 resolution-based derivations, 130
 resolvent, 126
 restricted quantification, 162
 restriction of a relation, 37
 root of a tree, 390
 rooted tree, 390
 derived directed, 390
 RSA method, 270
 Russell's paradox, 27

- sample space, 343
- selection operation, 83
- semantic tableaux, 97, 113
 - formula-decomposition rules, 113
 - rules, 94
 - theorem, 115
- sentence
 - first-order, 164
- set, 2
 - complement, 4
 - notation, 3
 - universal, 4
- set-builder notation, 3
- sets
 - Cartesian product, 4
 - difference, 3
 - disjoint, 3
 - intersection, 3
 - operations on, 3
 - union, 3
- sink, 365
- size of a digraph, 365
- size of a graph, 358
- Skolem constant, 197
- Skolemization, 197
- source, 365
- subdivision of a graph, 407
 - elementary, 407
- subformulae, 92
- subgraph, 359
 - induced, 359
 - vertex-induced, 360
- subset, 2
- subset, proper, 2
- substitution
 - domain, 201
 - result, 166
- subterm, 140
- Sudoku, 410
- sufficient, 91
- supremum, 63
- surjection, 30
- syllogism, 101
- symmetric, 248
- tableau, 114
 - branch
 - closed, 114
 - open, 115
 - saturated, 115
 - closed, 115
 - open, 115
- tautology, 95
- term, 140
 - constant, 140
 - evaluation, 153
 - ground, 140
 - value, 153
- terminal vertex, 362, 365
- terms
 - nonunifiable, 202
 - unifiable, 202
- theorem, 111
- theorems of Euler and Fermat, 262
- total order, 46
- trial, 343
- trail in a graph, 362
- transitive, 248
- trapdoor function, 270
- travelling salesman problem, 400
- tree, 385
 - rooted, 390
- trivial divisors, 237
- trivial graph, 358
- truth tables, 10
- truth value, 9, 90
- tuple, 5
- n -tuple, 5
- underlying graph, 366
- unifier, 202
 - more general, 202
 - most general, 202
- universal closure, 172
- universal quantifier, 139
- upper bound, 63
- variable
 - Boolean, 91

- variable (*continued*)
 - bound, 164
 - free, 164
 - free for substitution,
 - 166
 - individual, 13
 - propositional, 91
- variable assignment, 153
 - variant, 155
- variation, 278
- Venn diagrams, 298
- Verifier, 156
- vertex, 357
 - degree, 358
 - end, 358
 - even/odd, 358
- isolated, 358
- vertex set, 357
- vertices
 - adjacent, 357
- walk in a graph, 362
- Warshall algorithm, 371
- weight of an edge, 378
- weighted graph, 377, 378
- weighted length of a path, 378
- well defined, 56
- well-founded poset, 65
- well order, 65
- Wilson's theorem, 266
- Zermelo–Fraenkel set theory, 58

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.

Antonio Gulli, Sujit Pal

Deep Learning with Keras

Implement neural networks with Keras on Theano and
TensorFlow



Packt

Deep Learning with Keras

Implement neural networks with Keras on Theano and TensorFlow

Antonio Gulli

Sujit Pal



BIRMINGHAM - MUMBAI

Deep Learning with Keras

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2017

Production reference: 1240417

Published by Packt Publishing Ltd.

Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78712-842-2

www.packtpub.com

Credits

Authors **Copy Editor**

Antonio Gulli Vikrant Phadkay
Sujit Pal

Reviewers **Project Coordinator**

Mike Dahlin Nidhi Joshi
Nick McClure
Corrado Zocollo

Commissioning Editor **Proofreader**

Amey Varangaonkar Safis Editing

Acquisition Editor **Indexer**

Divya Poojari Francy Puthiry

Content Development Editor **Graphics**

Cheryl Dsa Tania Dutta

Technical Editor **Production Coordinator**

Dinesh Pawar Arvindkumar Gupta

About the Authors

Antonio Gulli is a software executive and business leader with a passion for establishing and managing global technological talent, innovation, and execution. He is an expert in search engines, online services, machine learning, information retrieval, analytics, and cloud computing. So far, he has been lucky enough to gain professional experience in four different countries in Europe and managed people in six different countries in Europe and America. Antonio served as CEO, GM, CTO, VP, director, and site lead in multiple fields spanning from publishing (Elsevier) to consumer internet (Ask.com and Tiscali) and high-tech R&D (Microsoft and Google).

I would like to thank my coauthor, Sujit Pal, for being a such talented colleague, always willing to help with a humble spirit. I constantly appreciate his dedication to teamwork, which made this book a real thing.

I would like to thank Francois Chollet (and the many Keras contributors) for taking the time and effort to build an awesome deep learning toolkit that is easy to use without sacrificing too much power.

I would also like to thank our editors from Packt, Divya Poojari, Cheryl Dsa, and Dinesh Pawar, and our reviewers from Packt and Google, for their support and valuable suggestions. This book would not have been possible without you.

I would like to thank my manager, Brad, and my colleagues Mike and Corrado at Google for encouraging me to write this book, and for their constant help in reviewing the content. I would like to thank Same Fusy, Herbacichernia i Kawiarnia in Warsaw. I got the initial inspiration to write this book in front of a cup of tea chosen among hundreds of different offers. This place is magic and I strongly recommend visiting it if you are in search of a place to stimulate creativity (<http://www.samefusy.pl/>).

Then I would like to thank HRBP at Google for supporting my wish to donate all of this book's royalties in favor of a minority/diversity scholarship.

I would like to thank my friends Eric, Laura, Francesco, Ettore, and Antonella for supporting me when I was in need. Long-term friendship is a real thing, and you are true friends to me.

I would like to thank my son Lorenzo for encouraging me to join Google, my son Leonardo for his constant passion to discover new things, and my daughter Aurora for making me smile every day of my life. Finally thanks to my father Elio and my mother Maria for their love.

Sujit Pal is a technology research director at Elsevier Labs, working on building intelligent systems around research content and metadata. His primary interests are information retrieval, ontologies, natural language processing, machine learning, and distributed processing. He is currently working on image classification and similarity using deep learning models. Prior to this, he worked in the consumer healthcare industry, where he helped build ontology-backed semantic search, contextual advertising, and EMR data processing platforms. He writes about technology on his blog at *Salmon Run*.

I would like to thank my coauthor, Antonio Gulli, for asking me to join him in writing this book. This was an incredible opportunity and a great learning experience for me. Besides, had he not done so, I quite literally wouldn't have been here today.

I would like to thank Ron Daniel, the director of Elsevier Labs, and Bradley P Allen, chief architect at Elsevier, for introducing me to deep learning and making me a believer in its capabilities.

I would also like to thank Francois Chollet (and the many Keras contributors) for taking the time and effort to build an awesome deep learning toolkit that is easy to use without sacrificing too much power.

Thanks to our editors from Packt, Divya Poojari, Cheryl Dsa, and Dinesh Pawar, and our reviewers from Packt and Google, for their support and valuable suggestions. This book would not have been possible without you.

I would like to thank my colleagues and managers over the years, especially the ones who took their chances with me and helped me make discontinuous changes in my career.

Finally, I would like to thank my family for putting up with me these past few months as I juggled work, this book, and family, in that order. I hope you will agree that it was all worth it.

About the Reviewer

Nick McClure is currently a senior data scientist at PayScale Inc. in Seattle, Washington, USA. Prior to that, he worked at Zillow and Caesars Entertainment. He got his degrees in applied mathematics from the University of Montana and the College of Saint Benedict and Saint John's University. Nick has also authored *TensorFlow Machine Learning Cookbook* by Packt Publishing.

He has a passion for learning and advocating for analytics, machine learning, and artificial intelligence. Nick occasionally puts his thoughts and musing on his blog, fromdata.org, or through his Twitter account at @nfmclure.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1787128423>.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface	1
Chapter 1: Neural Networks Foundations	9
Perceptron	11
The first example of Keras code	11
Multilayer perceptron — the first example of a network	12
Problems in training the perceptron and a solution	13
Activation function — sigmoid	14
Activation function — ReLU	15
Activation functions	15
A real example — recognizing handwritten digits	16
One-hot encoding — OHE	17
Defining a simple neural net in Keras	17
Running a simple Keras net and establishing a baseline	21
Improving the simple net in Keras with hidden layers	22
Further improving the simple net in Keras with dropout	25
Testing different optimizers in Keras	28
Increasing the number of epochs	34
Controlling the optimizer learning rate	34
Increasing the number of internal hidden neurons	35
Increasing the size of batch computation	37
Summarizing the experiments run for recognizing handwritten charts	37
Adopting regularization for avoiding overfitting	38
Hyperparameters tuning	40
Predicting output	40
A practical overview of backpropagation	40
Towards a deep learning approach	42
Summary	43
Chapter 2: Keras Installation and API	44
Installing Keras	45
Step 1 — install some useful dependencies	45
Step 2 — install Theano	46
Step 3 — install TensorFlow	46
Step 4 — install Keras	47
Step 5 — testing Theano, TensorFlow, and Keras	47

Configuring Keras	48
Installing Keras on Docker	49
Installing Keras on Google Cloud ML	52
Installing Keras on Amazon AWS	55
Installing Keras on Microsoft Azure	57
Keras API	59
Getting started with Keras architecture	59
What is a tensor?	59
Composing models in Keras	60
Sequential composition	60
Functional composition	60
An overview of predefined neural network layers	60
Regular dense	61
Recurrent neural networks — simple, LSTM, and GRU	61
Convolutional and pooling layers	62
Regularization	62
Batch normalization	63
An overview of predefined activation functions	63
An overview of losses functions	64
An overview of metrics	65
An overview of optimizers	65
Some useful operations	65
Saving and loading the weights and the architecture of a model	65
Callbacks for customizing the training process	66
Checkpointing	67
Using TensorBoard and Keras	68
Using Quiver and Keras	69
Summary	70
Chapter 3: Deep Learning with ConvNets	71
Deep convolutional neural network — DCNN	72
Local receptive fields	72
Shared weights and bias	73
Pooling layers	74
Max-pooling	74
Average pooling	75
ConvNets summary	75
An example of DCNN — LeNet	76
LeNet code in Keras	76
Understanding the power of deep learning	83
Recognizing CIFAR-10 images with deep learning	84
Improving the CIFAR-10 performance with deeper a network	89

Improving the CIFAR-10 performance with data augmentation	91
Predicting with CIFAR-10	95
Very deep convolutional networks for large-scale image recognition	96
Recognizing cats with a VGG-16 net	97
Utilizing Keras built-in VGG-16 net module	98
Recycling pre-built deep learning models for extracting features	100
Very deep inception-v3 net used for transfer learning	101
Summary	104
Chapter 4: Generative Adversarial Networks and WaveNet	105
What is a GAN?	106
Some GAN applications	108
Deep convolutional generative adversarial networks	112
Keras adversarial GANs for forging MNIST	115
Keras adversarial GANs for forging CIFAR	122
WaveNet — a generative model for learning how to produce audio	130
Summary	139
Chapter 5: Word Embeddings	140
Distributed representations	141
word2vec	142
The skip-gram word2vec model	143
The CBOW word2vec model	147
Extracting word2vec embeddings from the model	149
Using third-party implementations of word2vec	152
Exploring GloVe	156
Using pre-trained embeddings	158
Learn embeddings from scratch	159
Fine-tuning learned embeddings from word2vec	164
Fine-tune learned embeddings from GloVe	168
Look up embeddings	169
Summary	173
Chapter 6: Recurrent Neural Network — RNN	175
SimpleRNN cells	176
SimpleRNN with Keras — generating text	178
RNN topologies	183
Vanishing and exploding gradients	184
Long short term memory — LSTM	187
LSTM with Keras — sentiment analysis	189
Gated recurrent unit — GRU	196

GRU with Keras — POS tagging	198
Bidirectional RNNs	205
Stateful RNNs	206
Stateful LSTM with Keras — predicting electricity consumption	206
Other RNN variants	213
Summary	214
Chapter 7: Additional Deep Learning Models	215
Keras functional API	217
Regression networks	219
Keras regression example — predicting benzene levels in the air	220
Unsupervised learning — autoencoders	224
Keras autoencoder example — sentence vectors	226
Composing deep networks	235
Keras example — memory network for question answering	236
Customizing Keras	243
Keras example — using the lambda layer	244
Keras example — building a custom normalization layer	245
Generative models	248
Keras example — deep dreaming	248
Keras example — style transfer	257
Summary	263
Chapter 8: AI Game Playing	264
Reinforcement learning	265
Maximizing future rewards	266
Q-learning	267
The deep Q-network as a Q-function	268
Balancing exploration with exploitation	270
Experience replay, or the value of experience	271
Example - Keras deep Q-network for catch	271
The road ahead	284
Summary	286
Chapter 9: Conclusion	287
Keras 2.0 — what is new	289
Installing Keras 2.0	289
API changes	290
Index	292

Preface

Hands-on deep learning with Keras is a concise yet thorough introduction to modern neural networks, artificial intelligence, and deep learning technologies designed especially for software engineers and data scientists.

Mission

The book presents more than 20 working deep neural networks coded in Python using Keras, a modular neural network library that runs on top of either Google's TensorFlow or Lisa Lab's Theano backends.

The reader is introduced step by step to supervised learning algorithms such as simple linear regression, classical multilayer perceptron, and more sophisticated deep convolutional networks and generative adversarial networks. In addition, the book covers unsupervised learning algorithms such as autoencoders and generative networks.

Recurrent networks and **long short-term memory (LSTM)** networks are also explained in detail. The book goes on to cover the Keras functional API and how to customize Keras in case the reader's use case is not covered by Keras's extensive functionality. It also looks at larger, more complex systems composed of the building blocks covered previously. The book concludes with an introduction to deep reinforcement learning and how it can be used to build game playing AIs.

Practical applications include code for the classification of news articles into predefined categories, syntactic analysis of texts, sentiment analysis, synthetic generation of texts, and parts of speech annotation. Image processing is also explored, with recognition of handwritten digit images, classification of images into different categories, and advanced object recognition with related image annotations. An example of identification of salient points for face detection will be also provided. Sound analysis comprises recognition of discrete speeches from multiple speakers. Reinforcement learning is used to build a deep Q-learning network capable of playing games autonomously.

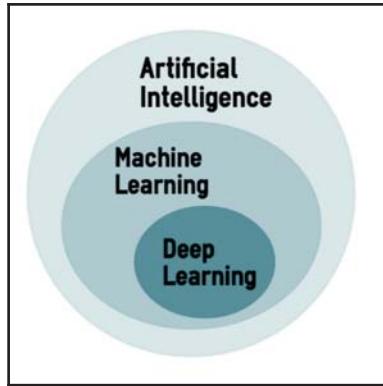
Experiments are the essence of the book. Each net is augmented by multiple variants that progressively improve the learning performance by changing the input parameters, the shape of the network, loss functions, and algorithms used for optimizations. Several comparisons between training on CPUs and GPUs are also provided.

How deep learning is different from machine learning and artificial intelligence

Artificial intelligence (AI) is a very large research field, where machines show *cognitive* capabilities such as learning behaviours, proactive interaction with the environment, inference and deduction, computer vision, speech recognition, problem solving, knowledge representation, perception, and many others (for more information, refer to this article: *Artificial Intelligence: A Modern Approach*, by S. Russell and P. Norvig, Prentice Hall, 2003). More colloquially, AI denotes any activity where machines mimic *intelligent* behaviors typically shown by humans. Artificial intelligence takes inspiration from elements of computer science, mathematics, and statistics.

Machine learning (ML) is a subbranch of AI that focuses on teaching computers how to learn without the need to be programmed for specific tasks (for more information refer to *Pattern Recognition and Machine Learning*, by C. M. Bishop, Springer, 2006). In fact, the key idea behind ML is that it is possible to create algorithms that learn from and make predictions on data. There are three different broad categories of ML. In supervised learning, the machine is presented with input data and desired output, and the goal is to learn from those training examples in such a way that meaningful predictions can be made for fresh unseen data. In unsupervised learning, the machine is presented with input data only and the machine has to find some meaningful structure by itself with no external supervision. In reinforcement learning, the machine acts as an agent interacting with the environment and learning what are the behaviours that generate rewards.

Deep learning (DL) is a particular subset of ML methodologies using **artificial neural networks (ANN)** slightly inspired by the structure of neurons located in the human brain (for more information, refer to the article *Learning Deep Architectures for AI*, by Y. Bengio, Found. Trends, vol. 2, 2009). Informally, the word *deep* refers to the presence of many layers in the artificial neural network, but this meaning has changed over time. While 4 years ago, 10 layers were already sufficient to consider a network as *deep*, today it is more common to consider a network as *deep* when it has hundreds of layers.



DL is a real tsunami (for more information, refer to *Computational Linguistics and Deep Learning* by C. D. Manning, "Computational Linguistics", vol. 41, 2015) for machine learning in that a relatively small number of clever methodologies have been very successfully applied to so many different domains (image, text, video, speech, and vision), significantly improving previous state-of-the-art results achieved over dozens of years. The success of DL is also due to the availability of more training data (such as ImageNet for images) and the relatively low-cost availability of GPUs for very efficient numerical computation. Google, Microsoft, Amazon, Apple, Facebook, and many others use those deep learning techniques every day for analyzing massive amounts of data. However, this kind of expertise is not limited any more to the domain of pure academic research and to large industrial companies. It has become an integral part of modern software production and therefore something that the reader should definitely master. The book does not require any particular mathematical background. However, it assumes that the reader is already a Python programmer.

What this book covers

Chapter 1, *Neural Networks Foundations*, teaches the basics of neural networks.

Chapter 2, *Keras Installation and API*, shows how to install Keras on AWS, Microsoft Azure, Google Cloud, and your own machine. In addition to that, we provide an overview of the Keras APIs.

Chapter 3, *Deep Learning with ConvNets*, introduces the concept of convolutional networks. It is a fundamental innovation in deep learning that has been used with success in multiple domains, from text to video to speech, going well beyond the initial image processing domain where it was originally conceived.

Chapter 4, *Generative Adversarial Networks and WaveNet*, introduces generative adversarial networks used to reproduce synthetic data that looks like data generated by humans. And we will present WaveNet, a deep neural network used for reproducing human voice and musical instruments with high quality.

Chapter 5, *Word Embeddings*, discusses word embeddings, a set of deep learning methodologies for detecting relationships between words and grouping together similar words.

Chapter 6, *Recurrent Neural Networks – RNN*, covers recurrent neural networks, a class of network optimized for handling sequence data such as text.

Chapter 7, *Additional Deep Learning Models*, gives a brief look into the Keras functional API, regression networks, autoencoders, and so on.

Chapter 8, *AI Game Playing*, teaches you deep reinforcement learning and how it can be used to build deep learning networks with Keras that learn how to play arcade games based on reward feedback.

Appendix, *Conclusion*, is a crisp refresher of the topics covered in this book and walks the users through what is new in Keras 2.0.

What you need for this book

To be able to smoothly follow through the chapters, you will need the following pieces of software:

- TensorFlow 1.0.0 or higher
- Keras 2.0.2 or higher
- Matplotlib 1.5.3 or higher
- Scikit-learn 0.18.1 or higher
- NumPy 1.12.1 or higher

The hardware specifications are as follows:

- Either 32-bit or 64-bit architecture
- 2+ GHz CPU
- 4 GB RAM
- At least 10 GB of hard disk space available

Who this book is for

If you are a data scientist with experience in machine learning or an AI programmer with some exposure to neural networks, you will find this book a useful entry point to deep learning with Keras. Knowledge of Python is required for this book.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "In addition, we load the true labels into `Y_train` and `Y_test` respectively and perform a one-hot encoding on them."

A block of code is set as follows:

```
from keras.models import Sequential
model = Sequential()
model.add(Dense(12, input_dim=8, kernel_initializer='random_uniform'))
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
# 10 outputs
# final stage is softmax
model = Sequential()
model.add(Dense(NB_CLASSES, input_shape=(RESHAPED,)))
model.add(Activation('softmax'))
model.summary()
```

Any command-line input or output is written as follows:

```
pip install quiver_engine
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Our simple net started with an accuracy of **92.22%**, which means that about eight handwritten characters out of 100 are not correctly recognized."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Deep-Learning-with-Keras>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/DeepLearningwithKeras_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Neural Networks Foundations

Artificial neural networks (briefly, *nets*) represent a class of machine learning models, loosely inspired by studies about the central nervous systems of mammals. Each net is made up of several interconnected *neurons*, organized in *layers*, which exchange messages (they *fire*, in jargon) when certain conditions happen. Initial studies were started in the late 1950s with the introduction of the perceptron (for more information, refer to the article: *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*, by F. Rosenblatt, Psychological Review, vol. 65, pp. 386 - 408, 1958), a two-layer network used for simple operations, and further expanded in the late 1960s with the introduction of the *backpropagation algorithm*, used for efficient multilayer networks training (according to the articles: *Backpropagation through Time: What It Does and How to Do It*, by P. J. Werbos, Proceedings of the IEEE, vol. 78, pp. 1550 - 1560, 1990, and *A Fast Learning Algorithm for Deep Belief Nets*, by G. E. Hinton, S. Osindero, and Y. W. Teh, Neural Computing, vol. 18, pp. 1527 - 1554, 2006). Some studies argue that these techniques have roots dating further back than normally cited (for more information, refer to the article: *Deep Learning in Neural Networks: An Overview*, by J. Schmidhuber, vol. 61, pp. 85 - 117, 2015). Neural networks were a topic of intensive academic studies until the 1980s, when other simpler approaches became more relevant. However, there has been a resurrection of interest starting from the mid-2000s, thanks to both a breakthrough fast-learning algorithm proposed by G. Hinton (for more information, refer to the articles: *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*, *Neural Networks*, by S. Leven, vol. 9, 1996 and *Learning Representations by Backpropagating Errors*, by D. E. Rumelhart, G. E. Hinton, and R. J. Williams, vol. 323, 1986) and the introduction of GPUs, roughly in 2011, for massive numeric computation.

These improvements opened the route for modern *deep learning*, a class of neural networks characterized by a significant number of layers of neurons, which are able to learn rather sophisticated models based on progressive levels of abstraction. People called it *deep* with 3-5 layers a few years ago, and now it has gone up to 100-200.

This learning via progressive abstraction resembles vision models that have evolved over millions of years in the human brain. The human visual system is indeed organized into different layers. Our eyes are connected to an area of the brain called the **visual cortex V1**, which is located in the lower posterior part of our brain. This area is common to many mammals and has the role of discriminating basic properties and small changes in visual orientation, spatial frequencies, and colors. It has been estimated that V1 consists of about 140 million neurons, with 10 billion connections between them. V1 is then connected with other areas V2, V3, V4, V5, and V6, doing progressively more complex image processing and recognition of more sophisticated concepts, such as shapes, faces, animals, and many more. This organization in layers is the result of a huge number of attempts tuned over several 100 million years. It has been estimated that there are ~16 billion human cortical neurons, and about 10%-25% of the human cortex is devoted to vision (for more information, refer to the article: *The Human Brain in Numbers: A Linearly Scaled-up Primate Brain*, by S. Herculano-Houzel, vol. 3, 2009). Deep learning has taken some inspiration from this layer-based organization of the human visual system: early artificial neuron layers learn basic properties of images, while deeper layers learn more sophisticated concepts.

This book covers several major aspects of neural networks by providing working nets coded in Keras, a minimalist and efficient Python library for deep learning computations running on the top of either Google's TensorFlow (for more information, refer to <https://www.tensorflow.org/>) or University of Montreal's Theano (for more information, refer to <http://deeplearning.net/software/theano/>) backend. So, let's start.

In this chapter, we will cover the following topics:

- Perceptron
- Multilayer perceptron
- Activation functions
- Gradient descent
- Stochastic gradient descent
- Backpropagation

Perceptron

The perceptron is a simple algorithm which, given an input vector x of m values (x_1, x_2, \dots, x_n) often called input features or simply features, outputs either 1 (yes) or 0 (no).

Mathematically, we define a function:

$$f(x) = \begin{cases} 1 & wx + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Here, w is a vector of weights, wx is the dot product $\sum_{j=1}^m w_j x_j$, and b is a bias. If you remember elementary geometry, $wx + b$ defines a boundary hyperplane that changes position according to the values assigned to w and b . If x lies above the straight line, then the answer is positive, otherwise it is negative. Very simple algorithm! The perception cannot express a *maybe* answer. It can answer *yes* (1) or *no* (0) if we understand how to define w and b , that is the training process that will be discussed in the following paragraphs.

The first example of Keras code

The initial building block of Keras is a model, and the simplest model is called **sequential**. A sequential Keras model is a linear pipeline (a stack) of neural networks layers. This code fragment defines a single layer with 12 artificial neurons, and it expects 8 input variables (also known as features):

```
from keras.models import Sequential
model = Sequential()
model.add(Dense(12, input_dim=8, kernel_initializer='random_uniform'))
```

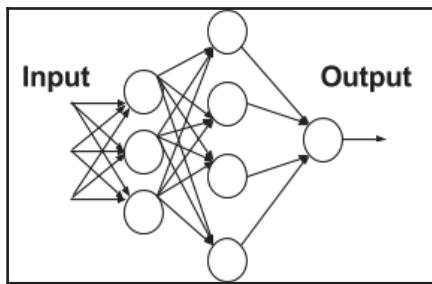
Each neuron can be initialized with specific weights. Keras provides a few choices, the most common of which are listed as follows:

- **random_uniform**: Weights are initialized to uniformly random small values in (-0.05, 0.05). In other words, any value within the given interval is equally likely to be drawn.
- **random_normal**: Weights are initialized according to a Gaussian, with a zero mean and small standard deviation of 0.05. For those of you who are not familiar with a Gaussian, think about a symmetric *bell curve* shape.
- **zero**: All weights are initialized to zero.

A full list is available at <https://keras.io/initializations/>.

Multilayer perceptron — the first example of a network

In this chapter, we define the first example of a network with multiple linear layers. Historically, perceptron was the name given to a model having one single linear layer, and as a consequence, if it has multiple layers, you would call it **multilayer perceptron (MLP)**. The following image represents a generic neural network with one input layer, one intermediate layer and one output layer.



In the preceding diagram, each node in the first layer receives an input and fires according to the predefined local decision boundaries. Then the output of the first layer is passed to the second layer, the results of which are passed to the final output layer consisting of one single neuron. It is interesting to note that this layered organization vaguely resembles the patterns of human vision we discussed earlier.

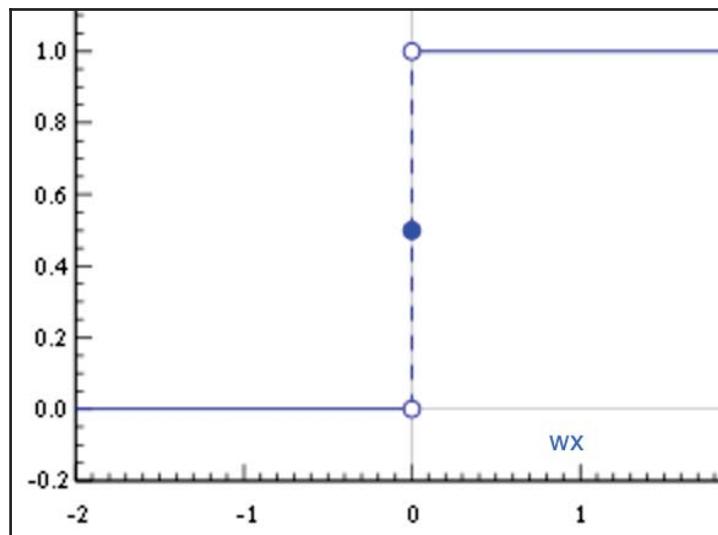


The *net* is dense, meaning that each neuron in a layer is connected to all neurons located in the previous layer and to all the neurons in the following layer.

Problems in training the perceptron and a solution

Let's consider a single neuron; what are the best choices for the weight w and the bias b ? Ideally, we would like to provide a set of training examples and let the computer adjust the weight and the bias in such a way that the errors produced in the output are minimized. In order to make this a bit more concrete, let's suppose we have a set of images of cats and another separate set of images not containing cats. For the sake of simplicity, assume that each neuron looks at a single input pixel value. While the computer processes these images, we would like our neuron to adjust its weights and bias so that we have fewer and fewer images wrongly recognized as non-cats. This approach seems very intuitive, but it requires that a small change in weights (and/or bias) causes only a small change in outputs.

If we have a big output jump, we cannot *progressively* learn (rather than trying things in all possible directions—a process known as exhaustive search—without knowing if we are improving). After all, kids learn little by little. Unfortunately, the perceptron does not show this little-by-little behavior. A perceptron is either 0 or 1 and that is a big jump and it will not help it to learn, as shown in the following graph:



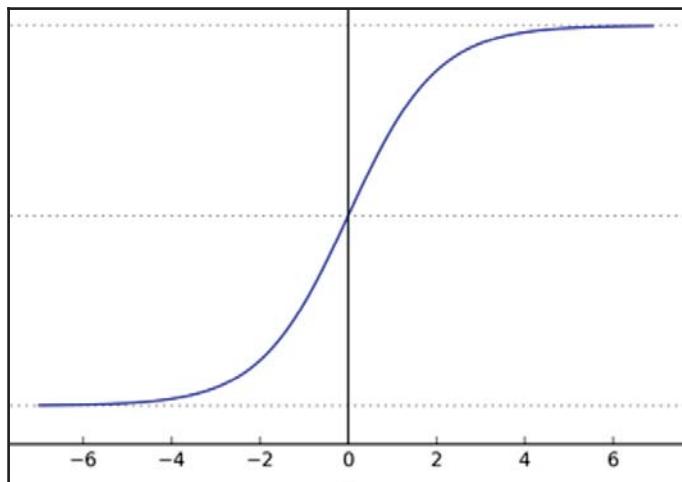
We need something different, smoother. We need a function that progressively changes from 0 to 1 with no discontinuity. Mathematically, this means that we need a continuous function that allows us to compute the derivative.

Activation function — sigmoid

The sigmoid function is defined as follows:

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

As represented in the following graph, it has small output changes in (0, 1) when the input varies in $(-\infty, \infty)$. Mathematically, the function is continuous. A typical sigmoid function is represented in the following graph:

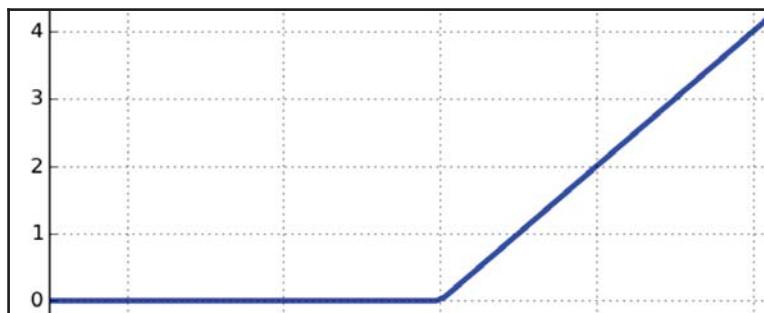


A neuron can use the sigmoid for computing the nonlinear function $\sigma(z = wx + b)$. Note that, if $z = wx + b$ is very large and positive, then $e^{-z} \rightarrow 0$, so $\sigma(z) \rightarrow 1$, while $z = wx + b$ if is very large and negative $e^{-z} \rightarrow \infty$ so $\sigma(z) \rightarrow 0$. In other words, a neuron with sigmoid activation has a behavior similar to the perceptron, but the changes are gradual and output values, such as 0.5539 or 0.123191, are perfectly legitimate. In this sense, a sigmoid neuron can answer *maybe*.

Activation function — ReLU

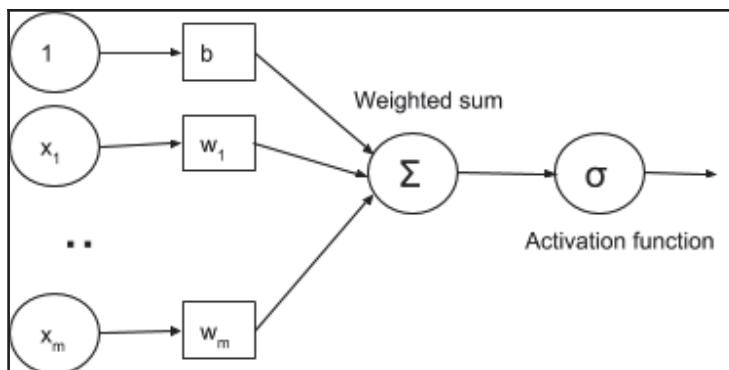
The sigmoid is not the only kind of smooth activation function used for neural networks. Recently, a very simple function called **rectified linear unit (ReLU)** became very popular because it generates very good experimental results. A ReLU is simply defined as

$f(x) = \max(0, x)$, and the nonlinear function is represented in the following graph. As you can see in the following graph, the function is zero for negative values, and it grows linearly for positive values:



Activation functions

Sigmoid and ReLU are generally called *activation functions* in neural network jargon. In the *Testing different optimizers in Keras* section, we will see that those gradual changes, typical of sigmoid and ReLU functions, are the basic building blocks to developing a learning algorithm which adapts little by little, by progressively reducing the mistakes made by our nets. An example of using the activation function σ with the (x_1, x_2, \dots, x_m) input vector, (w_1, w_2, \dots, w_m) weight vector, b bias, and Σ summation is given in the following diagram:



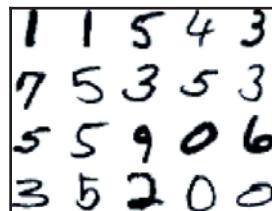
Keras supports a number of activation functions, and a full list is available at <https://keras.io/activations/>.

A real example — recognizing handwritten digits

In this section, we will build a network that can recognize handwritten numbers. For achieving this goal, we use MNIST (for more information, refer to <http://yann.lecun.com/exdb/mnist/>), a database of handwritten digits made up of a training set of 60,000 examples and a test set of 10,000 examples. The training examples are annotated by humans with the correct answer. For instance, if the handwritten digit is the number three, then three is simply the label associated with that example.

In machine learning, when a dataset with correct answers is available, we say that we can perform a form of *supervised learning*. In this case, we can use training examples for tuning up our net. Testing examples also have the correct answer associated with each digit. In this case, however, the idea is to pretend that the label is unknown, let the network do the prediction, and then later on, reconsider the label to evaluate how well our neural network has learned to recognize digits. So, not unsurprisingly, testing examples are just used to test our net.

Each MNIST image is in gray scale, and it consists of 28 x 28 pixels. A subset of these numbers is represented in the following diagram:



One-hot encoding — OHE

In many applications, it is convenient to transform categorical (non-numerical) features into numerical variables. For instance, the categorical feature digit with the value d in $[0-9]$ can be encoded into a binary vector with 10 positions, which always has 0 value, except the d -th position where a 1 is present. This type of representation is called **one-hot encoding (OHE)** and is very common in data mining when the learning algorithm is specialized for dealing with numerical functions.

Defining a simple neural net in Keras

Here, we use Keras to define a network that recognizes MNIST handwritten digits. We start with a very simple neural network and then progressively improve it.

Keras provides suitable libraries to load the dataset and split it into training sets `X_train`, used for fine-tuning our net, and tests set `X_test`, used for assessing the performance. Data is converted into `float32` for supporting GPU computation and normalized to $[0, 1]$. In addition, we load the true labels into `Y_train` and `Y_test` respectively and perform a one-hot encoding on them. Let's see the code:

```
from __future__ import print_function
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD
from keras.utils import np_utils
np.random.seed(1671) # for reproducibility

# network and training
NB_EPOCH = 200
BATCH_SIZE = 128
VERBOSE = 1
NB_CLASSES = 10 # number of outputs = number of digits
OPTIMIZER = SGD() # SGD optimizer, explained later in this chapter
N_HIDDEN = 128
VALIDATION_SPLIT=0.2 # how much TRAIN is reserved for VALIDATION

# data: shuffled and split between train and test sets
#
(X_train, y_train), (X_test, y_test) = mnist.load_data()
#X_train is 60000 rows of 28x28 values --> reshaped in 60000 x 784
RESHAPED = 784
#
```

```
X_train = X_train.reshape(60000, RESHAPE)
X_test = X_test.reshape(10000, RESHAPE)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# normalize
#
X_train /= 255
X_test /= 255
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
# convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, NB_CLASSES)
Y_test = np_utils.to_categorical(y_test, NB_CLASSES)
```

The input layer has a neuron associated with each pixel in the image for a total of $28 \times 28 = 784$ neurons, one for each pixel in the MNIST images.

Typically, the values associated with each pixel are normalized in the range $[0, 1]$ (which means that the intensity of each pixel is divided by 255, the maximum intensity value). The output is 10 classes, one for each digit.

The final layer is a single neuron with activation function softmax, which is a generalization of the sigmoid function. Softmax *squashes* a k-dimensional vector of arbitrary real values into a k-dimensional vector of real values in the range $(0, 1)$. In our case, it aggregates 10 answers provided by the previous layer with 10 neurons:

```
# 10 outputs
# final stage is softmax
model = Sequential()
model.add(Dense(NB_CLASSES, input_shape=(RESHAPE,)))
model.add(Activation('softmax'))
model.summary()
```

Once we define the model, we have to compile it so that it can be executed by the Keras backend (either Theano or TensorFlow). There are a few choices to be made during compilation:

- We need to select the *optimizer* that is the specific algorithm used to update weights while we train our model
- We need to select the *objective function* that is used by the optimizer to navigate the space of weights (frequently, objective functions are called *loss function*, and the process of optimization is defined as a process of loss *minimization*)
- We need to evaluate the trained model

Some common choices for the objective function (a complete list of Keras objective functions is at <https://keras.io/objectives/>) are as follows:

- **MSE**: This is the mean squared error between the predictions and the true values. Mathematically, if Υ is a vector of n predictions, and Y is the vector of n observed values, then they satisfy the following equation:

$$MSE = \frac{1}{n} \sum_{i=1}^n (\Upsilon - Y)^2$$



These objective functions average all the mistakes made for each prediction, and if the prediction is far from the true value, then this distance is made more evident by the squaring operation.

- **Binary cross-entropy**: This is the binary logarithmic loss. Suppose that our model predicts p while the target is t , then the binary cross-entropy is defined as follows:

$$-t \log(p) - (1 - t) \log(1 - p)$$



This objective function is suitable for binary labels prediction.

- **Categorical cross-entropy**: This is the multiclass logarithmic loss. If the target is t_{ij} and the prediction is p_{ij} , then the categorical cross-entropy is this:

$$L_i = -\sum_j t_{i,j} \log(p_{i,j})$$



This objective function is suitable for multiclass labels predictions. It is also the default choice in association with softmax activation.

Some common choices for metrics (a complete list of Keras metrics is at <https://keras.io/metrics/>) are as follows:

- **Accuracy:** This is the proportion of correct predictions with respect to the targets
- **Precision:** This denotes how many selected items are relevant for a multilabel classification
- **Recall:** This denotes how many selected items are relevant for a multilabel classification

Metrics are similar to objective functions, with the only difference that they are not used for training a model but only for evaluating a model. Compiling a model in Keras is easy:

```
model.compile(loss='categorical_crossentropy', optimizer=OPTIMIZER,
metrics=['accuracy'])
```

Once the model is compiled, it can be then trained with the `fit()` function, which specifies a few parameters:

- `epochs`: This is the number of times the model is exposed to the training set. At each iteration, the optimizer tries to adjust the weights so that the objective function is minimized.
- `batch_size`: This is the number of training instances observed before the optimizer performs a weight update.

Training a model in Keras is very simple. Suppose we want to iterate for `NB_EPOCH` steps:

```
history = model.fit(X_train, Y_train,
batch_size=BATCH_SIZE, epochs=NB_EPOCH,
verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
```



We reserved part of the training set for validation. The key idea is that we reserve a part of the training data for measuring the performance on the validation while training. This is a good practice to follow for any machine learning task, which we will adopt in all our examples.

Once the model is trained, we can evaluate it on the test set that contains new unseen examples. In this way, we can get the minimal value reached by the objective function and best value reached by the evaluation metric.

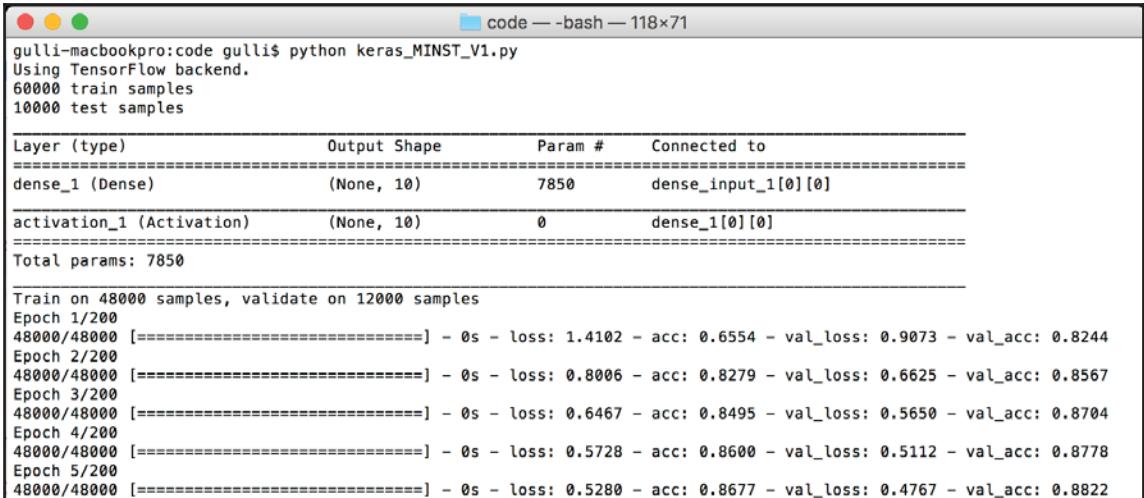
Note that the training set and the test set are, of course, rigorously separated. There is no point in evaluating a model on an example that has already been used for training. Learning is essentially a process intended to generalize unseen observations and not to memorize what is already known:

```
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

So, congratulations, you have just defined your first neural network in Keras. A few lines of code, and your computer is able to recognize handwritten numbers. Let's run the code and see what the performance is.

Running a simple Keras net and establishing a baseline

So let's see what will happen when we run the code in the following screenshot:



The screenshot shows a terminal window titled "code -- bash -- 118x71". The command run is "python keras_MINST_V1.py". The output shows the TensorFlow backend being used, and it prints the number of train and test samples (60000 and 10000 respectively). It then displays the network architecture with one dense layer of 10 units connected to the input, and an activation layer. The total number of parameters is 7850. The training loop follows, showing epochs from 1 to 5 over 48000 samples, with validation loss and accuracy values improving over time.

```
gulli-macbookpro:code gulli$ python keras_MINST_V1.py
Using TensorFlow backend.
60000 train samples
10000 test samples

Layer (type)          Output Shape         Param #  Connected to
=====
dense_1 (Dense)      (None, 10)           7850     dense_input_1[0][0]
activation_1 (Activation) (None, 10)           0        dense_1[0][0]
=====
Total params: 7850

Train on 48000 samples, validate on 12000 samples
Epoch 1/200
48000/48000 [=====] - 0s - loss: 1.4102 - acc: 0.6554 - val_loss: 0.9073 - val_acc: 0.8244
Epoch 2/200
48000/48000 [=====] - 0s - loss: 0.8006 - acc: 0.8279 - val_loss: 0.6625 - val_acc: 0.8567
Epoch 3/200
48000/48000 [=====] - 0s - loss: 0.6467 - acc: 0.8495 - val_loss: 0.5650 - val_acc: 0.8704
Epoch 4/200
48000/48000 [=====] - 0s - loss: 0.5728 - acc: 0.8600 - val_loss: 0.5112 - val_acc: 0.8778
Epoch 5/200
48000/48000 [=====] - 0s - loss: 0.5280 - acc: 0.8677 - val_loss: 0.4767 - val_acc: 0.8822
```

First, the net architecture is dumped, and we can see the different types of layers used, their output shape, how many parameters they need to optimize, and how they are connected. Then, the network is trained on 48,000 samples, and 12,000 are reserved for validation. Once the neural model is built, it is then tested on 10,000 samples. As you can see, Keras is internally using TensorFlow as a backend system for computation. For now, we don't go into the internals on how the training happens, but we can notice that the program runs for 200 iterations, and each time, the accuracy improves.

When the training ends, we test our model on the test set and achieve about 92.36% accuracy on training, 92.27% on validation, and 92.22% on the test.

This means that a bit less than one handwritten character out of ten is not correctly recognized. We can certainly do better than that. In the following screenshot, we can see the test accuracy:

```
Epoch 198/200
48000/48000 [=====] - 0s - loss: 0.2761 - acc: 0.9230 - val_loss: 0.2762 - val_acc: 0.9224
Epoch 199/200
48000/48000 [=====] - 0s - loss: 0.2760 - acc: 0.9231 - val_loss: 0.2762 - val_acc: 0.9223
Epoch 200/200
48000/48000 [=====] - 0s - loss: 0.2758 - acc: 0.9236 - val_loss: 0.2761 - val_acc: 0.9227
9888/10000 [=====>.] - ETA: 0s
Test score: 0.277792117235
Test accuracy: 0.9222
gulli-macbookpro:code gulli$
```

Improving the simple net in Keras with hidden layers

We have a baseline accuracy of 92.36% on training, 92.27% on validation, and 92.22% on the test. This is a good starting point, but we can certainly improve it. Let's see how.

A first improvement is to add additional layers to our network. So, after the input layer, we have a first dense layer with the `N_HIDDEN` neurons and an activation function `relu`. This additional layer is considered *hidden* because it is not directly connected to either the input or the output. After the first hidden layer, we have a second hidden layer, again with the `N_HIDDEN` neurons, followed by an output layer with 10 neurons, each of which will fire when the relative digit is recognized. The following code defines this new network:

```
from __future__ import print_function
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD
from keras.utils import np_utils
np.random.seed(1671) # for reproducibility
# network and training
NB_EPOCH = 20
BATCH_SIZE = 128
VERBOSE = 1
NB_CLASSES = 10 # number of outputs = number of digits
OPTIMIZER = SGD() # optimizer, explained later in this chapter
N_HIDDEN = 128
```

```
VALIDATION_SPLIT=0.2 # how much TRAIN is reserved for VALIDATION
# data: shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()
#X_train is 60000 rows of 28x28 values --> reshaped in 60000 x 784
RESHAPED = 784
#
X_train = X_train.reshape(60000, RESHAPED)
X_test = X_test.reshape(10000, RESHAPED)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# normalize
X_train /= 255
X_test /= 255
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
# convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, NB_CLASSES)
Y_test = np_utils.to_categorical(y_test, NB_CLASSES)
# M_HIDDEN hidden layers
# 10 outputs
# final stage is softmax
model = Sequential()
model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
model.add(Activation('relu'))
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dense(NB_CLASSES))
model.add(Activation('softmax'))
model.summary()
model.compile(loss='categorical_crossentropy',
optimizer=OPTIMIZER,
metrics=['accuracy'])
history = model.fit(X_train, Y_train,
batch_size=BATCH_SIZE, epochs=N_EPOCH,
verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

Let's run the code and see which result we get with this multilayer network. Not bad. By adding two hidden layers, we reached 94.50% on the training set, 94.63% on validation, and 94.41% on the test. This means that we gained an additional 2.2% accuracy on the test with respect to the previous network. However, we dramatically reduced the number of iterations from 200 to 20. That's good, but we want more.

If you want, you can play by yourself and see what happens if you add only one hidden layer instead of two, or if you add more than two layers. I leave this experiment as an exercise. The following screenshot shows the output of the preceding example:

```
code -- bash -- 118x66
gulli-macbookpro:code gulli$ python keras_MINST_V2.py
Using TensorFlow backend.
60000 train samples
10000 test samples
Layer (type)          Output Shape       Param #  Connected to
=====
dense_1 (Dense)      (None, 128)        100480    dense_input_1[0][0]
activation_1 (Activation) (None, 128)        0         dense_1[0][0]
dense_2 (Dense)      (None, 128)        16512     activation_1[0][0]
activation_2 (Activation) (None, 128)        0         dense_2[0][0]
dense_3 (Dense)      (None, 10)         1290      activation_2[0][0]
activation_3 (Activation) (None, 10)         0         dense_3[0][0]
=====
Total params: 118282
Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [=====] - 1s - loss: 1.5266 - acc: 0.6101 - val_loss: 0.7839 - val_acc: 0.8296
Epoch 2/20
48000/48000 [=====] - 1s - loss: 0.6108 - acc: 0.8464 - val_loss: 0.4603 - val_acc: 0.8796
Epoch 3/20
48000/48000 [=====] - 1s - loss: 0.4422 - acc: 0.8794 - val_loss: 0.3765 - val_acc: 0.8963
Epoch 4/20
48000/48000 [=====] - 1s - loss: 0.3796 - acc: 0.8946 - val_loss: 0.3374 - val_acc: 0.9065
Epoch 5/20
48000/48000 [=====] - 1s - loss: 0.3450 - acc: 0.9027 - val_loss: 0.3119 - val_acc: 0.9116
Epoch 6/20
48000/48000 [=====] - 1s - loss: 0.3214 - acc: 0.9090 - val_loss: 0.2940 - val_acc: 0.9165
Epoch 7/20
48000/48000 [=====] - 1s - loss: 0.3033 - acc: 0.9148 - val_loss: 0.2794 - val_acc: 0.9213
Epoch 8/20
48000/48000 [=====] - 1s - loss: 0.2885 - acc: 0.9181 - val_loss: 0.2668 - val_acc: 0.9251
Epoch 9/20
48000/48000 [=====] - 1s - loss: 0.2763 - acc: 0.9220 - val_loss: 0.2569 - val_acc: 0.9287
Epoch 10/20
48000/48000 [=====] - 1s - loss: 0.2654 - acc: 0.9245 - val_loss: 0.2491 - val_acc: 0.9304
Epoch 11/20
48000/48000 [=====] - 1s - loss: 0.2556 - acc: 0.9274 - val_loss: 0.2400 - val_acc: 0.9335
Epoch 12/20
48000/48000 [=====] - 1s - loss: 0.2464 - acc: 0.9299 - val_loss: 0.2329 - val_acc: 0.9355
Epoch 13/20
48000/48000 [=====] - 1s - loss: 0.2382 - acc: 0.9321 - val_loss: 0.2279 - val_acc: 0.9369
Epoch 14/20
48000/48000 [=====] - 1s - loss: 0.2309 - acc: 0.9342 - val_loss: 0.2208 - val_acc: 0.9388
Epoch 15/20
48000/48000 [=====] - 1s - loss: 0.2237 - acc: 0.9365 - val_loss: 0.2140 - val_acc: 0.9413
Epoch 16/20
48000/48000 [=====] - 1s - loss: 0.2172 - acc: 0.9380 - val_loss: 0.2085 - val_acc: 0.9423
Epoch 17/20
48000/48000 [=====] - 1s - loss: 0.2110 - acc: 0.9397 - val_loss: 0.2035 - val_acc: 0.9435
Epoch 18/20
48000/48000 [=====] - 1s - loss: 0.2051 - acc: 0.9415 - val_loss: 0.1993 - val_acc: 0.9445
Epoch 19/20
48000/48000 [=====] - 1s - loss: 0.1997 - acc: 0.9427 - val_loss: 0.1954 - val_acc: 0.9461
Epoch 20/20
48000/48000 [=====] - 1s - loss: 0.1947 - acc: 0.9450 - val_loss: 0.1914 - val_acc: 0.9463
9696/10000 [=====] - ETA: 0s
Test score: 0.191052276902
Test accuracy: 0.9441
gulli-macbookpro:code gulli$
```

Further improving the simple net in Keras with dropout

Now our baseline is 94.50% on the training set, 94.63% on validation, and 94.41% on the test. A second improvement is very simple. We decide to randomly drop with the dropout probability some of the values propagated inside our internal dense network of hidden layers. In machine learning, this is a well-known form of regularization. Surprisingly enough, this idea of randomly dropping a few values can improve our performance:

```
from __future__ import print_function
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import SGD
from keras.utils import np_utils
np.random.seed(1671) # for reproducibility
# network and training
NB_EPOCH = 250
BATCH_SIZE = 128
VERBOSE = 1
NB_CLASSES = 10 # number of outputs = number of digits
OPTIMIZER = SGD() # optimizer, explained later in this chapter
N_HIDDEN = 128
VALIDATION_SPLIT=0.2 # how much TRAIN is reserved for VALIDATION
DROPOUT = 0.3
# data: shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()
#X_train is 60000 rows of 28x28 values --> reshaped in 60000 x 784
RESHAPED = 784
#
X_train = X_train.reshape(60000, RESHAPED)
X_test = X_test.reshape(10000, RESHAPED)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
# normalize
X_train /= 255
X_test /= 255
# convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, NB_CLASSES)
Y_test = np_utils.to_categorical(y_test, NB_CLASSES)
# M_HIDDEN hidden layers 10 outputs
model = Sequential()
model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
model.add(Activation('relu'))
model.add(Dropout(DROPOUT))
```

```
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dropout(DROPOUT))
model.add(Dense(NB_CLASSES))
model.add(Activation('softmax'))
model.summary()
model.compile(loss='categorical_crossentropy',
optimizer=OPTIMIZER,
metrics=['accuracy'])
history = model.fit(X_train, Y_train,
batch_size=BATCH_SIZE, epochs=N_EPOCH,
verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
score = model.evaluate(X_test, Y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

Let's run the code for 20 iterations as previously done, and we will see that this net achieves an accuracy of 91.54% on the training, 94.48% on validation, and 94.25% on the test:

```

code -- bash -- 118x70
gulli-macbookpro:code gulli$ python keras_MINST_V3_1.py
Using TensorFlow backend.
60000 train samples
10000 test samples

Layer (type)          Output Shape       Param #  Connected to
===== ====== =====
dense_1 (Dense)      (None, 128)        100480    dense_input_1[0][0]
activation_1 (Activation) (None, 128)        0         dense_1[0][0]
dropout_1 (Dropout)   (None, 128)        0         activation_1[0][0]
dense_2 (Dense)      (None, 128)        16512     dropout_1[0][0]
activation_2 (Activation) (None, 128)        0         dense_2[0][0]
dropout_2 (Dropout)   (None, 128)        0         activation_2[0][0]
dense_3 (Dense)      (None, 10)         1290      dropout_2[0][0]
activation_3 (Activation) (None, 10)         0         dense_3[0][0]
=====
Total params: 118282

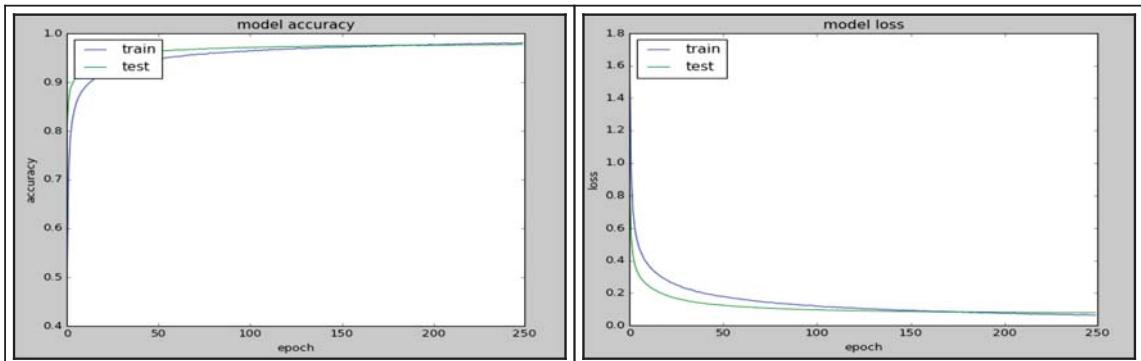
Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [=====] - 1s - loss: 1.7206 - acc: 0.4625 - val_loss: 0.9125 - val_acc: 0.8036
Epoch 2/20
48000/48000 [=====] - 1s - loss: 0.9254 - acc: 0.7149 - val_loss: 0.5374 - val_acc: 0.8621
Epoch 3/20
48000/48000 [=====] - 1s - loss: 0.6938 - acc: 0.7883 - val_loss: 0.4240 - val_acc: 0.8872
Epoch 4/20
48000/48000 [=====] - 1s - loss: 0.5917 - acc: 0.8205 - val_loss: 0.3724 - val_acc: 0.8958
Epoch 5/20
48000/48000 [=====] - 1s - loss: 0.5307 - acc: 0.8398 - val_loss: 0.3370 - val_acc: 0.9038
Epoch 6/20
48000/48000 [=====] - 1s - loss: 0.4868 - acc: 0.8546 - val_loss: 0.3126 - val_acc: 0.9084
Epoch 7/20
48000/48000 [=====] - 1s - loss: 0.4563 - acc: 0.8654 - val_loss: 0.2939 - val_acc: 0.9126
Epoch 8/20
48000/48000 [=====] - 1s - loss: 0.4322 - acc: 0.8726 - val_loss: 0.2789 - val_acc: 0.9173
Epoch 9/20
48000/48000 [=====] - 1s - loss: 0.4061 - acc: 0.8799 - val_loss: 0.2666 - val_acc: 0.9196
Epoch 10/20
48000/48000 [=====] - 1s - loss: 0.3908 - acc: 0.8848 - val_loss: 0.2556 - val_acc: 0.9236
Epoch 11/20
48000/48000 [=====] - 1s - loss: 0.3758 - acc: 0.8893 - val_loss: 0.2463 - val_acc: 0.9263
Epoch 12/20
48000/48000 [=====] - 1s - loss: 0.3592 - acc: 0.8938 - val_loss: 0.2372 - val_acc: 0.9297
Epoch 13/20
48000/48000 [=====] - 1s - loss: 0.3491 - acc: 0.8970 - val_loss: 0.2294 - val_acc: 0.9323
Epoch 14/20
48000/48000 [=====] - 1s - loss: 0.3361 - acc: 0.9009 - val_loss: 0.2224 - val_acc: 0.9344
Epoch 15/20
48000/48000 [=====] - 1s - loss: 0.3266 - acc: 0.9036 - val_loss: 0.2165 - val_acc: 0.9348
Epoch 16/20
48000/48000 [=====] - 1s - loss: 0.3182 - acc: 0.9064 - val_loss: 0.2102 - val_acc: 0.9371
Epoch 17/20
48000/48000 [=====] - 1s - loss: 0.3073 - acc: 0.9103 - val_loss: 0.2035 - val_acc: 0.9395
Epoch 18/20
48000/48000 [=====] - 1s - loss: 0.2998 - acc: 0.9109 - val_loss: 0.1987 - val_acc: 0.9418
Epoch 19/20
48000/48000 [=====] - 1s - loss: 0.2930 - acc: 0.9131 - val_loss: 0.1930 - val_acc: 0.9423
Epoch 20/20
48000/48000 [=====] - 1s - loss: 0.2855 - acc: 0.9154 - val_loss: 0.1893 - val_acc: 0.9448
9888/10000 [=====>.] - ETA: 0s
Test score: 0.191873697177
Test accuracy: 0.9425
gulli-macbookpro:code gulli$ 

```

Note that training accuracy should still be above the test accuracy, otherwise we are not training long enough. So let's try to increase significantly the number of epochs up to 250, and we get 98.1% accuracy on training, 97.73% on validation, and 97.7% on the test:

```
Epoch 248/250
48000/48000 [=====] - 1s - loss: 0.0630 - acc: 0.9804 - val_loss: 0.0785 - val_acc: 0.9769
Epoch 249/250
48000/48000 [=====] - 1s - loss: 0.0634 - acc: 0.9799 - val_loss: 0.0789 - val_acc: 0.9775
Epoch 250/250
48000/48000 [=====] - 1s - loss: 0.0616 - acc: 0.9810 - val_loss: 0.0787 - val_acc: 0.9773
9696/10000 [=====>.] - ETA: 0s
Test score: 0.0726828922328
Test accuracy: 0.9777
gulli-macbookpro:code gulli$
```

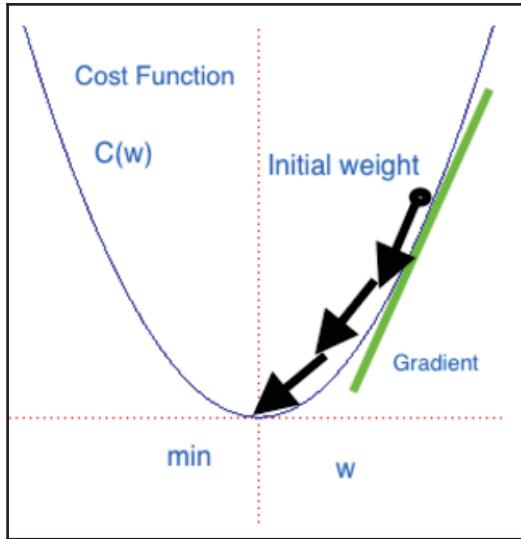
It is useful to observe how accuracy increases on training and test sets when the number of epochs increases. As you can see in the following graph, these two curves touch at about 250 epochs, and therefore, there is no need to train further after that point:



Note that it has been frequently observed that networks with random dropout in internal hidden layers can generalize better on unseen examples contained in test sets. Intuitively, one can think of this as each neuron becoming more capable because it knows it cannot depend on its neighbors. During testing, there is no dropout, so we are now using all our highly tuned neurons. In short, it is generally a good approach to test how a net performs when some dropout function is adopted.

Testing different optimizers in Keras

We have defined and used a network; it is useful to start giving an intuition about how networks are trained. Let's focus on one popular training technique known as **gradient descent (GD)**. Imagine a generic cost function $C(w)$ in one single variable w like in the following graph:



The gradient descent can be seen as a hiker who aims at climbing down a mountain into a valley. The mountain represents the function C , while the valley represents the minimum C_{\min} . The hiker has a starting point w_0 . The hiker moves little by little. At each step r , the gradient is the direction of maximum increase. Mathematically, this direction is the value of

the partial derivative $\frac{\partial c}{\partial w}$ evaluated at point w_r , reached at step r . Therefore by taking the opposite direction, $-\frac{\partial c}{\partial w}(w_r)$, the hiker can move towards the valley. At each step, the hiker can decide what the leg length is before the next step. This is the *learning rate* $\eta \geq 0$ in gradient descent jargon. Note that if η is too small, then the hiker will move slowly. However, if η is too high, then the hiker will possibly miss the valley.

Now you should remember that a sigmoid is a continuous function, and it is possible to compute the derivative. It can be proven that the sigmoid is shown as follows:

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

It has the following derivative:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

ReLU is not differentiable in 0. We can, however, extend the first derivative in 0 to a function over the whole domain by choosing it to be either 0 or 1. The point-wise derivative of ReLU $y = \max(0, x)$ is as follows:

$$\frac{dy}{dx} = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

Once we have the derivative, it is possible to optimize the nets with a gradient descent technique. Keras uses its backend (either TensorFlow or Theano) for computing the derivative on our behalf so we don't need to worry about implementing or computing it. We just choose the activation function, and Keras computes its derivative on our behalf.

A neural network is essentially a composition of multiple functions with thousands, and sometimes millions, of parameters. Each network layer computes a function whose error should be minimized in order to improve the accuracy observed during the learning phase. When we discuss backpropagation, we will discover that the minimization game is a bit more complex than our toy example. However, it is still based on the same intuition of descending a valley.

Keras implements a fast variant of gradient descent known as **stochastic gradient descent (SGD)** and two more advanced optimization techniques known as **RMSprop** and **Adam**. RMSprop and Adam include the concept of momentum (a velocity component) in addition to the acceleration component that SGD has. This allows faster convergence at the cost of more computation. A full list of Keras-supported optimizers is at <https://keras.io/optimizers/>. SGD was our default choice so far. So now let's try the other two. It is very simple, we just need to change few lines:

```
from keras.optimizers import RMSprop, Adam  
...  
OPTIMIZER = RMSprop() # optimizer,
```

That's it. Let's test it as shown in the following screenshot:

```

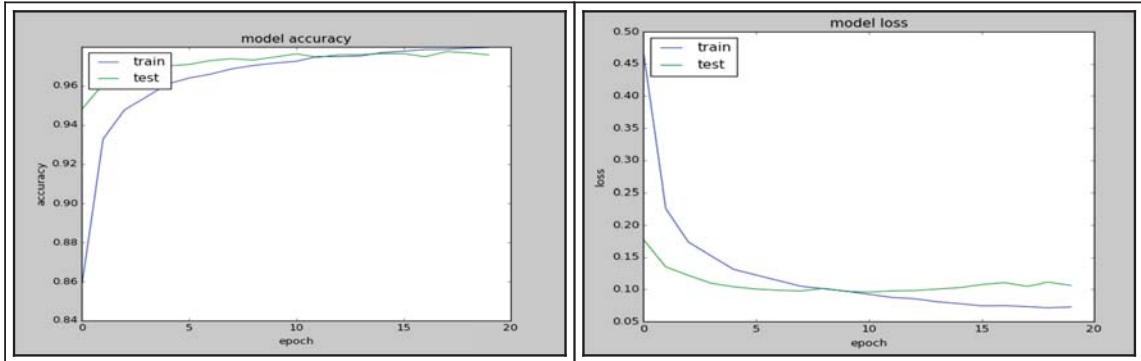
gulli-macbookpro:code gulli$ python keras_MINST_V4.py
Using TensorFlow backend.
60000 train samples
10000 test samples

Layer (type)      Output Shape       Param #
dense_1 (Dense)  (None, 128)        100480
activation_1 (Activation) (None, 128)       0
dropout_1 (Dropout) (None, 128)        0
dense_2 (Dense)  (None, 128)        16512
activation_2 (Activation) (None, 128)       0
dropout_2 (Dropout) (None, 128)        0
dense_3 (Dense)  (None, 10)         1290
activation_3 (Activation) (None, 10)        0
=====
Total params: 118282

Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [=====] - 2s - loss: 0.4714 - acc: 0.8571 - val_loss: 0.1780 - val_acc: 0.9478
Epoch 2/20
48000/48000 [=====] - 1s - loss: 0.2257 - acc: 0.9328 - val_loss: 0.1350 - val_acc: 0.9608
Epoch 3/20
48000/48000 [=====] - 1s - loss: 0.1737 - acc: 0.9477 - val_loss: 0.1217 - val_acc: 0.9643
Epoch 4/20
48000/48000 [=====] - 1s - loss: 0.1522 - acc: 0.9542 - val_loss: 0.1095 - val_acc: 0.9687
Epoch 5/20
48000/48000 [=====] - 1s - loss: 0.1312 - acc: 0.9609 - val_loss: 0.1039 - val_acc: 0.9703
Epoch 6/20
48000/48000 [=====] - 1s - loss: 0.1222 - acc: 0.9640 - val_loss: 0.1004 - val_acc: 0.9710
Epoch 7/20
48000/48000 [=====] - 1s - loss: 0.1134 - acc: 0.9660 - val_loss: 0.0985 - val_acc: 0.9730
Epoch 8/20
48000/48000 [=====] - 1s - loss: 0.1046 - acc: 0.9688 - val_loss: 0.0975 - val_acc: 0.9739
Epoch 9/20
48000/48000 [=====] - 1s - loss: 0.1009 - acc: 0.9705 - val_loss: 0.1014 - val_acc: 0.9732
Epoch 10/20
48000/48000 [=====] - 1s - loss: 0.0970 - acc: 0.9717 - val_loss: 0.0967 - val_acc: 0.9748
Epoch 11/20
48000/48000 [=====] - 1s - loss: 0.0922 - acc: 0.9726 - val_loss: 0.0956 - val_acc: 0.9764
Epoch 12/20
48000/48000 [=====] - 1s - loss: 0.0874 - acc: 0.9751 - val_loss: 0.0975 - val_acc: 0.9747
Epoch 13/20
48000/48000 [=====] - 1s - loss: 0.0853 - acc: 0.9750 - val_loss: 0.0980 - val_acc: 0.9760
Epoch 14/20
48000/48000 [=====] - 1s - loss: 0.0807 - acc: 0.9754 - val_loss: 0.1003 - val_acc: 0.9760
Epoch 15/20
48000/48000 [=====] - 1s - loss: 0.0777 - acc: 0.9771 - val_loss: 0.1025 - val_acc: 0.9766
Epoch 16/20
48000/48000 [=====] - 1s - loss: 0.0742 - acc: 0.9778 - val_loss: 0.1074 - val_acc: 0.9765
Epoch 17/20
48000/48000 [=====] - 1s - loss: 0.0746 - acc: 0.9786 - val_loss: 0.1104 - val_acc: 0.9750
Epoch 18/20
48000/48000 [=====] - 1s - loss: 0.0730 - acc: 0.9788 - val_loss: 0.1046 - val_acc: 0.9776
Epoch 19/20
48000/48000 [=====] - 1s - loss: 0.0711 - acc: 0.9793 - val_loss: 0.1112 - val_acc: 0.9769
Epoch 20/20
48000/48000 [=====] - 1s - loss: 0.0725 - acc: 0.9797 - val_loss: 0.1060 - val_acc: 0.9759
9888/10000 [=====] - ETA: 0s
Test score: 0.962571567255
Test accuracy: 0.9784
['acc', 'loss', 'val_acc', 'val_loss']

```

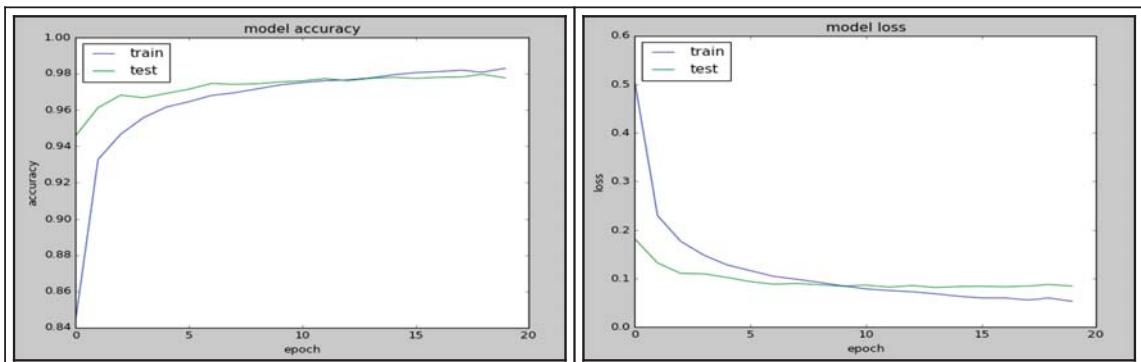
As you can see in the preceding screenshot, RMSprop is faster than SDG since we are able to achieve an accuracy of 97.97% on training, 97.59% on validation, and 97.84% on the test improving SDG with only 20 iterations. For the sake of completeness, let's see how the accuracy and loss change with the number of epochs, as shown in the following graphs:



OK, let's try the other optimizer, Adam(). It is pretty simple, as follows:

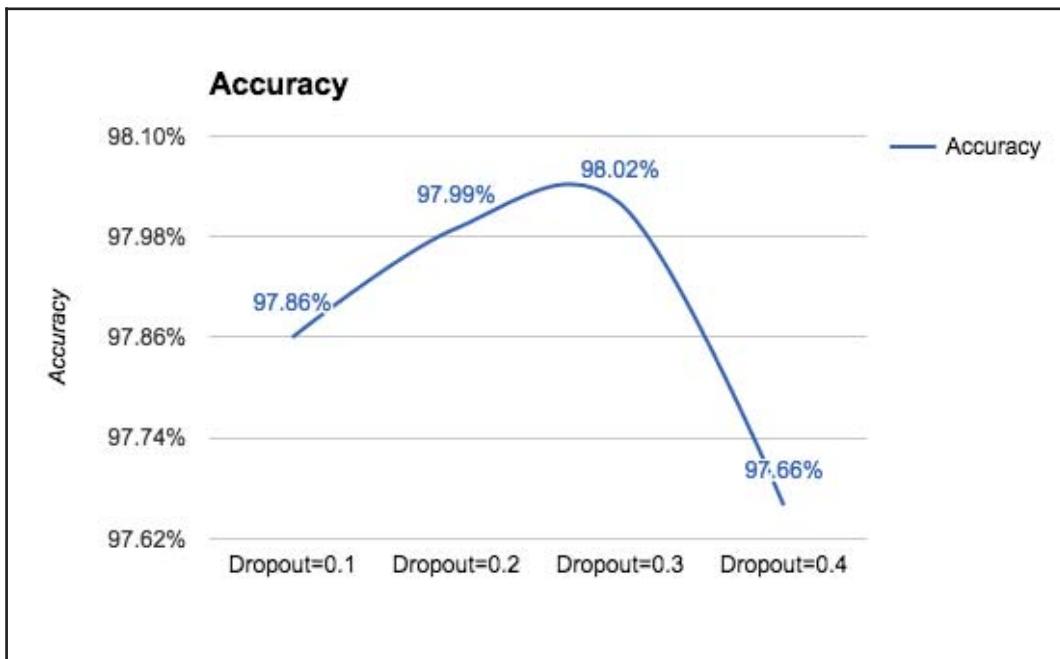
```
OPTIMIZER = Adam() # optimizer
```

As we have seen, Adam is slightly better. With Adam, we achieve 98.28% accuracy on training, 98.03% on validation, and 97.93% on the test with 20 iterations, as shown in the following graphs:



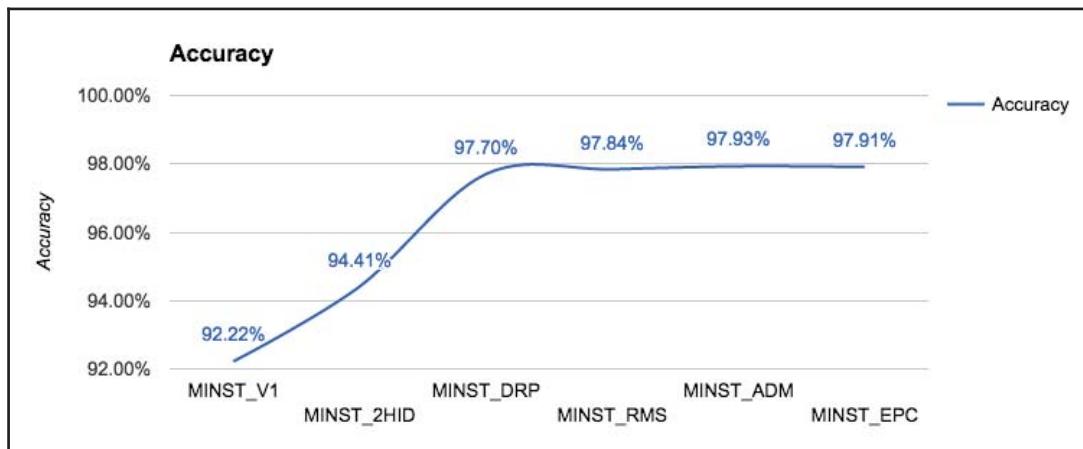
This is our fifth variant, and remember that our initial baseline was at 92.36%.

So far, we made progressive improvements; however, the gains are now more and more difficult. Note that we are optimizing with a dropout of 30%. For the sake of completeness, it could be useful to report the accuracy on the test only for other dropout values with Adam() chosen as optimizer, as shown in the following graph:



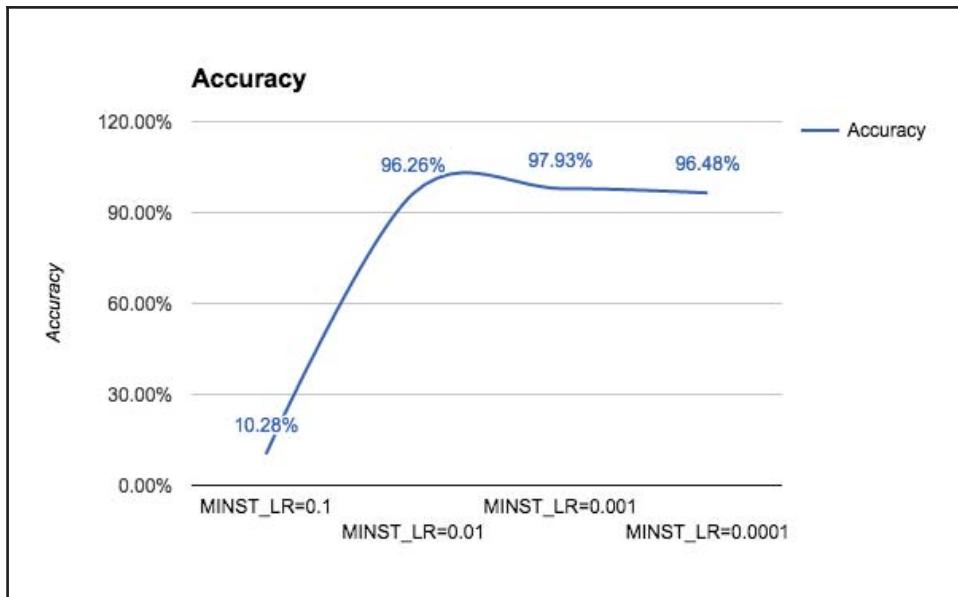
Increasing the number of epochs

Let's make another attempt and increase the number of epochs used for training from 20 to 200. Unfortunately, this choice increases our computation time by 10, but it gives us no gain. The experiment is unsuccessful, but we have learned that if we spend more time learning, we will not necessarily improve. Learning is more about adopting smart techniques and not necessarily about the time spent in computations. Let's keep track of our sixth variant in the following graph:



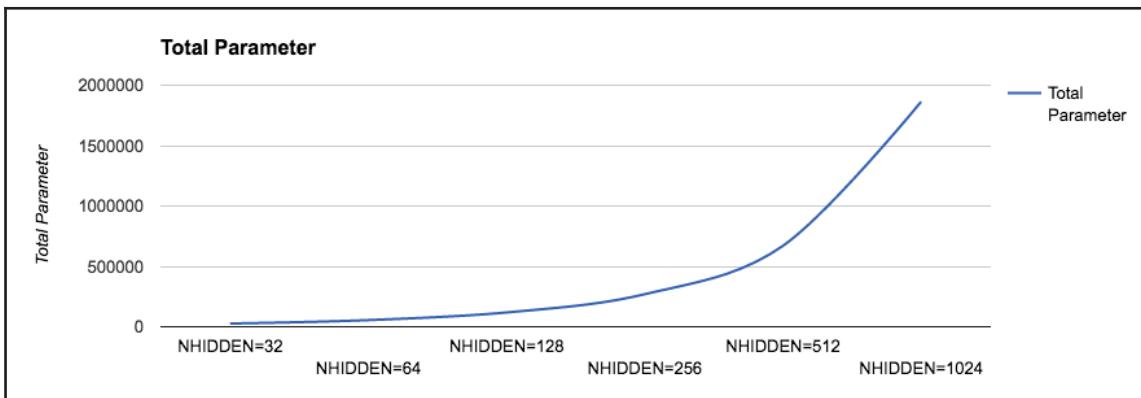
Controlling the optimizer learning rate

There is another attempt we can make, which is changing the learning parameter for our optimizer. As you can see in the following graph, the optimal value is somewhere close to 0.001, which is the default learning rate for the optimizer. Good! Adam works well out of the box:

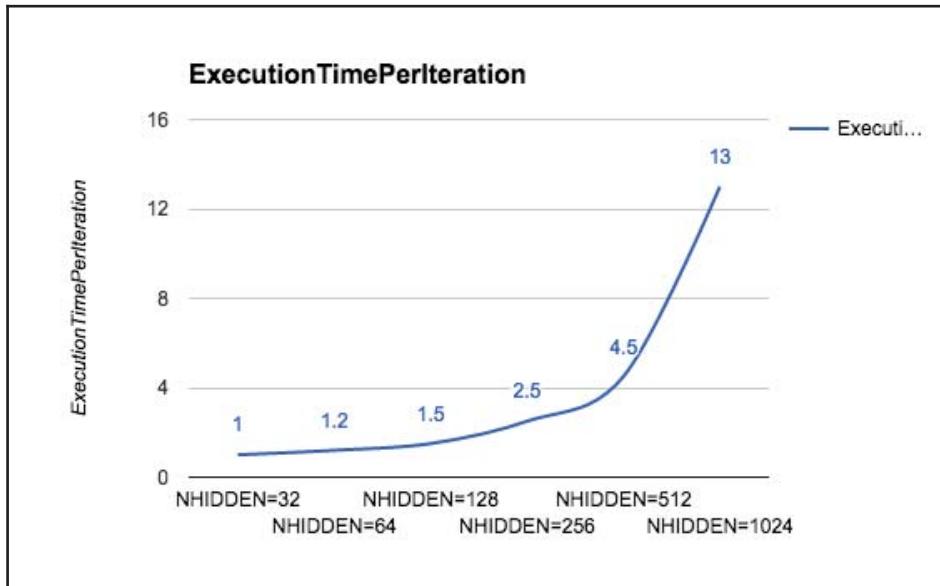


Increasing the number of internal hidden neurons

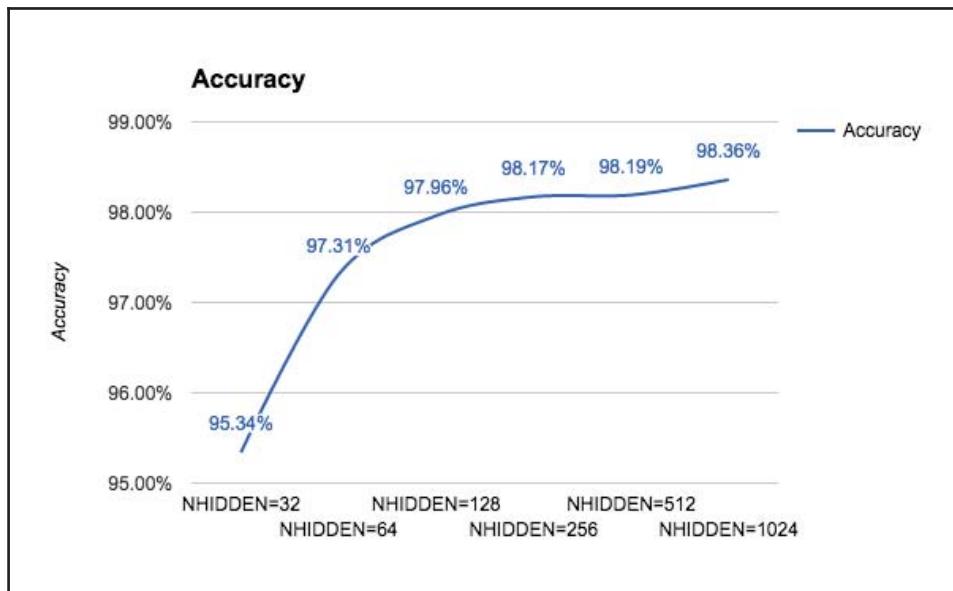
We can make yet another attempt, that is, changing the number of internal hidden neurons. We report the results of the experiments with an increasing number of hidden neurons. We can see in the following graph that by increasing the complexity of the model, the run time increases significantly because there are more and more parameters to optimize. However, the gains that we are getting by increasing the size of the network decrease more and more as the network grows:



In the following graph, we show the time needed for each iteration as the number of hidden neurons grow:

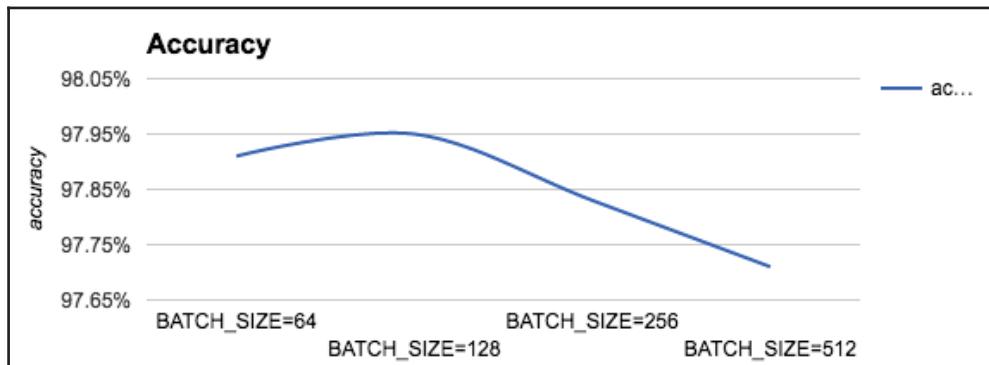


The following graph shows the accuracy as the number of hidden neurons grow:



Increasing the size of batch computation

Gradient descent tries to minimize the cost function on all the examples provided in the training sets and, at the same time, for all the features provided in the input. Stochastic gradient descent is a much less expensive variant, which considers only `BATCH_SIZE` examples. So, let's see what the behavior is by changing this parameter. As you can see, the optimal accuracy value is reached for `BATCH_SIZE=128`:



Summarizing the experiments run for recognizing handwritten charts

So, let's summarize: with five different variants, we were able to improve our performance from 92.36% to 97.93%. First, we defined a simple layer network in Keras. Then, we improved the performance by adding some hidden layers. After that, we improved the performance on the test set by adding a few random dropouts to our network and by experimenting with different types of optimizers. Current results are summarized in the following table:

Model/Accuracy	Training	Validation	Test
Simple	92.36%	92.37%	92.22%
Two hidden (128)	94.50%	94.63%	94.41%
Dropout (30%)	98.10%	97.73%	97.7% (200 epochs)
RMSprop	97.97%	97.59%	97.84% (20 epochs)
Adam	98.28%	98.03%	97.93% (20 epochs)

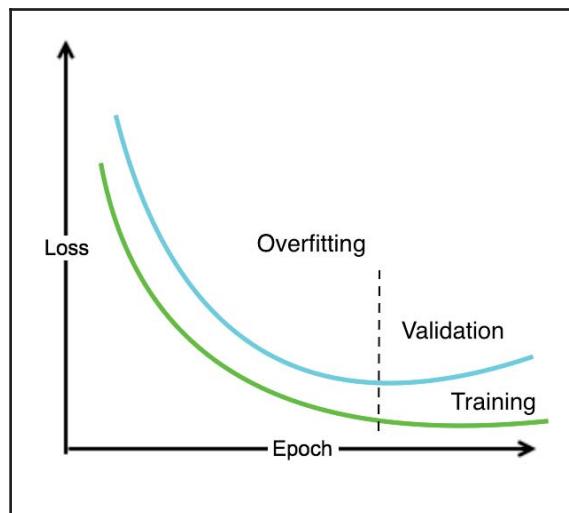
However, the next two experiments did not provide significant improvements. Increasing the number of internal neurons creates more complex models and requires more expensive computations, but it provides only marginal gains. We get the same experience if we increase the number of training epochs. A final experiment consisted in changing the `BATCH_SIZE` for our optimizer.

Adopting regularization for avoiding overfitting

Intuitively, a good machine learning model should achieve low error on training data. Mathematically, this is equivalent to minimizing the loss function on the training data given the machine learning model built. This is expressed by the following formula.:

$$\min : \{loss(Training\ Data|Model)\}$$

However, this might not be enough. A model can become excessively complex in order to capture all the relations inherently expressed by the training data. This increase of complexity might have two negative consequences. First, a complex model might require a significant amount of time to be executed. Second, a complex model can achieve very good performance on training data—because all the inherent relations in trained data are memorized, but not so good performance on validation data—as the model is not able to generalize on fresh unseen data. Again, learning is more about generalization than memorization. The following graph represents a typical loss function decreasing on both validation and training sets. However, a certain point the loss on validation starts to increase because of overfitting:



As a rule of thumb, if during the training we see that the loss increases on validation, after an initial decrease, then we have a problem of model complexity that overfits training. Indeed, overfitting is the word used in machine learning for concisely describing this phenomenon.

In order to solve the overfitting problem, we need a way to capture the complexity of a model, that is, how complex a model can be. What could be the solution? Well, a model is nothing more than a vector of weights. Therefore the complexity of a model can be conveniently represented as the number of nonzero weights. In other words, if we have two models, M_1 and M_2 , achieving pretty much the same performance in terms of loss function, then we should choose the simplest model that has the minimum number of nonzero weights. We can use a hyperparameter $\lambda \geq 0$ for controlling what the importance of having a simple model is, as in this formula:

$$\min : \{loss(Training\ Data|Model)\} + \lambda * complexity(Model)$$

There are three different types of regularizations used in machine learning:

- **L1 regularization** (also known as **lasso**): The complexity of the model is expressed as the sum of the absolute values of the weights
- **L2 regularization** (also known as **ridge**): The complexity of the model is expressed as the sum of the squares of the weights
- **Elastic net regularization**: The complexity of the model is captured by a combination of the two preceding techniques

Note that the same idea of regularization can be applied independently to the weights, to the model, and to the activation.

Therefore, playing with regularization can be a good way to increase the performance of a network, in particular when there is an evident situation of overfitting. This set of experiments is left as an exercise for the interested reader.

Note that Keras supports both l1, l2, and elastic net regularizations. Adding regularization is easy; for instance, here we have a l2 regularizer for kernel (the weight W):

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
kernel_regularizer=regularizers.l2(0.01)))
```

A full description of the available parameters is available at: <https://keras.io/regularizers/>.

Hyperparameters tuning

The preceding experiments gave a sense of what the opportunities for fine-tuning a net are. However, what is working for this example is not necessarily working for other examples. For a given net, there are indeed multiple parameters that can be optimized (such as the number of hidden neurons, `BATCH_SIZE`, number of epochs, and many more according to the complexity of the net itself).

Hyperparameter tuning is the process of finding the optimal combination of those parameters that minimize cost functions. The key idea is that if we have n parameters, then we can imagine that they define a space with n dimensions, and the goal is to find the point in this space which corresponds to an optimal value for the cost function. One way to achieve this goal is to create a grid in this space and systematically check for each grid vertex what the value assumed by the cost function is. In other words, the parameters are divided into buckets, and different combinations of values are checked via a brute force approach.

Predicting output

When a net is trained, it can be course be used for predictions. In Keras, this is very simple. We can use the following method:

```
# calculate predictions
predictions = model.predict(x)
```

For a given input, several types of output can be computed, including a method:

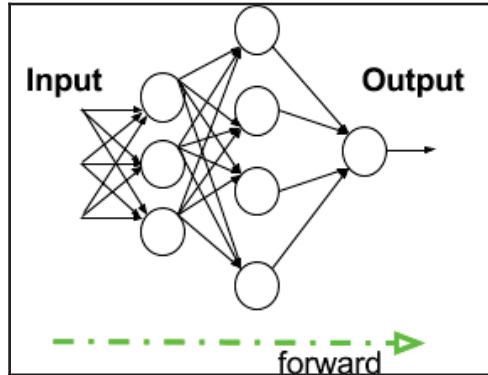
- `model.evaluate()`: This is used to compute the loss values
- `model.predict_classes()`: This is used to compute category outputs
- `model.predict_proba()`: This is used to compute class probabilities

A practical overview of backpropagation

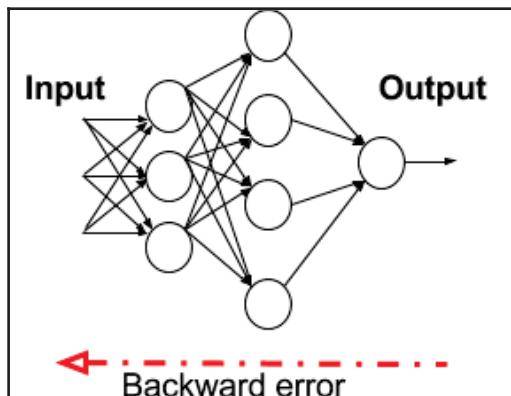
Multilayer perceptrons learn from training data through a process called backpropagation. The process can be described as a way of progressively correcting mistakes as soon as they are detected. Let's see how this works.

Remember that each neural network layer has an associated set of weights that determines the output values for a given set of inputs. In addition to that, remember that a neural network can have multiple hidden layers.

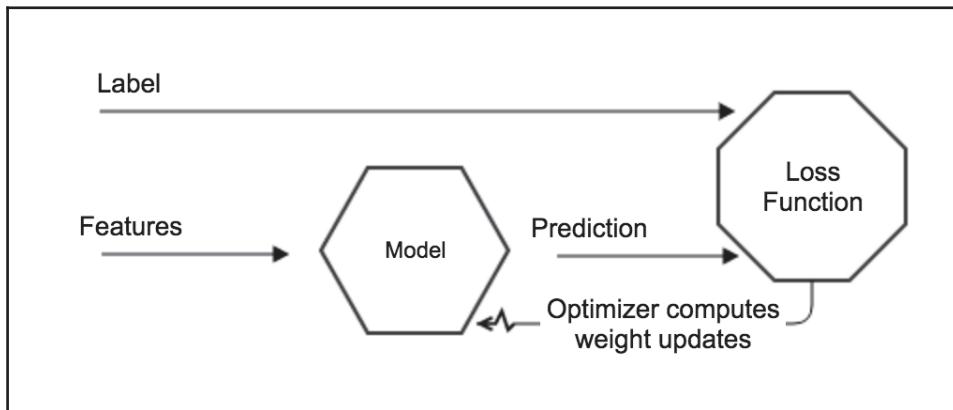
In the beginning, all the weights have some random assignment. Then the net is activated for each input in the training set: values are propagated *forward* from the input stage through the hidden stages to the output stage where a prediction is made (note that we have kept the following diagram simple by only representing a few values with green dotted lines, but in reality, all the values are propagated forward through the network):



Since we know the true observed value in the training set, it is possible to calculate the error made in prediction. The key intuition for backtracking is to propagate the error back and use an appropriate optimizer algorithm, such as a gradient descent, to adjust the neural network weights with the goal of reducing the error (again for the sake of simplicity, only a few error values are represented):



The process of forward propagation from input to output and backward propagation of errors is repeated several times until the error gets below a predefined threshold. The whole process is represented in the following diagram:



The features represent the input and the labels are here used to drive the learning process. The model is updated in such a way that the loss function is progressively minimized. In a neural network, what really matters is not the output of a single neuron but the collective weights adjusted in each layer. Therefore, the network progressively adjusts its internal weights in such a way that the prediction increases the number of labels correctly forecasted. Of course, using the right set features and having a quality labeled data is fundamental to minimizing the bias during the learning process.

Towards a deep learning approach

While playing with handwritten digit recognition, we came to the conclusion that the closer we get to the accuracy of 99%, the more difficult it is to improve. If we want to have more improvements, we definitely need a new idea. What are we missing? Think about it.

The fundamental intuition is that, so far, we lost all the information related to the local spatiality of the images. In particular, this piece of code transforms the bitmap, representing each written digit into a flat vector where the spatial locality is gone:

```
#X_train is 60000 rows of 28x28 values --> reshaped in 60000 x 784
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
```

However, this is not how our brain works. Remember that our vision is based on multiple cortex levels, each one recognizing more and more structured information, still preserving the locality. First we see single pixels, then from that, we recognize simple geometric forms and then more and more sophisticated elements such as objects, faces, human bodies, animals and so on.

In Chapter 3, *Deep Learning with ConvNets*, we will see that a particular type of deep learning network known as **convolutional neural network (CNN)** has been developed by taking into account both the idea of preserving the spatial locality in images (and, more generally, in any type of information) and the idea of learning via progressive levels of abstraction: with one layer, you can only learn simple patterns; with more than one layer, you can learn multiple patterns. Before discussing CNN, we need to discuss some aspects of Keras architecture and have a practical introduction to a few additional machine learning concepts. This will be the topic of the next chapters.

Summary

In this chapter, you learned the basics of neural networks, more specifically, what a perceptron is, what a multilayer perceptron is, how to define neural networks in Keras, how to progressively improve metrics once a good baseline is established, and how to fine-tune the hyperparameter's space. In addition to that, you now also have an intuitive idea of what some useful activation functions (sigmoid and ReLU) are, and how to train a network with backpropagation algorithms based on either gradient descent, on stochastic gradient descent, or on more sophisticated approaches, such as Adam and RMSprop.

In the next chapter, we will see how to install Keras on AWS, Microsoft Azure, Google Cloud, and on your own machine. In addition to that, we will provide an overview of Keras APIs.

2

Keras Installation and API

In the previous chapter, we discussed the basic principles of neural networks and provided a few examples of nets that are able to recognize MNIST handwritten numbers.

This chapter explains how to install Keras, Theano, and TensorFlow. Step by step, we will look at how to get the environment working and move from intuition to working nets in very little time. Then we will discuss how to install on a dockerized infrastructure based on containers, and in the cloud with Google GCP, Amazon AWS, and Microsoft Azure. In addition to that, we will present an overview of Keras APIs, and some commonly useful operations such as loading and saving neural networks' architectures and weights, early stopping, history saving, checkpointing, and interactions with TensorBoard and Quiver. Let us start.

By the end of this chapter, we will have covered the following topics:

- Installing and configuring Keras
- Keras architecture

Installing Keras

In the sections to follow, we will show how to install Keras on multiple platforms.

Step 1 — install some useful dependencies

First, we install the `numpy` package, which provides support for large, multidimensional arrays and matrices as well as high-level mathematical functions. Then we install `scipy`, a library used for scientific computation. After that, it might be appropriate to install `scikit-learn`, a package considered the Python Swiss army knife for machine learning. In this case, we will use it for data exploration. Optionally, it could be useful to install `pillow`, a library useful for image processing, and `h5py`, a library useful for data serialization used by Keras for model saving. A single command line is enough for installing what is needed. Alternatively, one can install Anaconda Python, which will automatically install `numpy`, `scipy`, `scikit-learn`, `h5py`, `pillow`, and a lot of other libraries that are needed for scientific computing (for more information, refer to: *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, by S. Ioffe and C. Szegedy, arXiv.org/abs/1502.03167, 2015). You can find the packages available in Anaconda Python at <https://docs.continuum.io/anaconda/pkg-docs>. The following screenshot shows how to install the packages for our work:



```
code — -bash — 103x20
gulli-macbookpro:code gulli$ pip install numpy scipy scikit-learn pillow h5py
Collecting numpy
  Using cached numpy-1.11.2-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Collecting scipy
  Using cached scipy-0.18.1-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Collecting scikit-learn
  Using cached scikit_learn-0.18.1-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Collecting pillow
  Using cached Pillow-3.4.2-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Collecting h5py
  Using cached h5py-2.6.0-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Requirement already satisfied: six in /Users/gulli/miniconda2/lib/python2.7/site-packages (from h5py)
Installing collected packages: numpy, scipy, scikit-learn, pillow, h5py
Successfully installed h5py-2.6.0 numpy-1.11.2 pillow-3.4.2 scikit-learn-0.18.1 scipy-0.18.1
gulli-macbookpro:code gulli$
```

Step 2 — install Theano

We can use pip to install Theano, as shown in the following screenshot:



```
gulli-macbookpro:google-cloud-sdk gulli$ pip install Theano
Collecting Theano
  Requirement already satisfied: numpy>=1.7.1 in /Users/gulli/miniconda2/lib/python2.7/site-packages (from Theano)
  Requirement already satisfied: scipy>=0.11 in /Users/gulli/miniconda2/lib/python2.7/site-packages (from Theano)
  Requirement already satisfied: six>=1.9.0 in /Users/gulli/miniconda2/lib/python2.7/site-packages (from Theano)
  Installing collected packages: Theano
    Successfully installed Theano-0.8.2
gulli-macbookpro:google-cloud-sdk gulli$
```

Step 3 — install TensorFlow

Now we can install TensorFlow using the instructions found on the TensorFlow website at https://www.tensorflow.org/versions/r0.11/get_started/os_setup.html#pip-installation. Again, we simply use pip for installing the correct package, as shown in the following screenshot. For instance, if we need to use GPUs, it is important to pick the appropriate package:



```
gulli-macbookpro:code gulli$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/mac/cpu/tensorflow-0.11.0-py2-none-any.whl
gulli-macbookpro:code gulli$ sudo pip install --upgrade $TF_BINARY_URL --ignore-installed
Collecting tensorflow==0.11.0 from https://storage.googleapis.com/tensorflow/mac/cpu/tensorflow-0.11.0-py2-none-any.whl
  Using cached https://storage.googleapis.com/tensorflow/mac/cpu/tensorflow-0.11.0-py2-none-any.whl
Collecting mock>=2.0.0 (from tensorflow==0.11.0)
  Using cached mock-2.0.0-py2.py3-none-any.whl
Collecting protobuf==3.0.0 (from tensorflow==0.11.0)
  Using cached protobuf-3.0.0-py2.py3-none-any.whl
Collecting numpy>=1.11.0 (from tensorflow==0.11.0)
  Using cached numpy-1.11.2-cp27-cp27m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.macosx_10_10_intel.macosx_10_10_x86_64.whl
Collecting wheel (from tensorflow==0.11.0)
  Using cached wheel-0.29.0-py2.py3-none-any.whl
Collecting six>=1.10.0 (from tensorflow==0.11.0)
  Using cached six-1.10.0-py2.py3-none-any.whl
Collecting funcsigs>=1; python_version < "3.3" (from mock>=2.0.0->tensorflow==0.11.0)
  Using cached funcsigs-1.0.2-py2.py3-none-any.whl
Collecting pbr>=0.11 (from mock>=2.0.0->tensorflow==0.11.0)
  Using cached pbr-1.10.0-py2.py3-none-any.whl
Collecting setuptools (from protobuf==3.0.0->tensorflow==0.11.0)
  Using cached setuptools-28.8.0-py2.py3-none-any.whl
Installing collected packages: six, funcsigs, pbr, mock, setuptools, protobuf, numpy, wheel, tensorflow
  Successfully installed funcsigs-1.0.2 mock-2.0.0 numpy-1.11.2 pbr-1.10.0 protobuf-3.0.0 setuptools-28.8.0 six-1.10.0 tensorflow-0.11.0 wheel-0.29.0
gulli-macbookpro:code gulli$
```

Step 4 — install Keras

Now we can simply install Keras and start testing the installed environment. Pretty simple; let's use pip again, as shown in this screenshot:

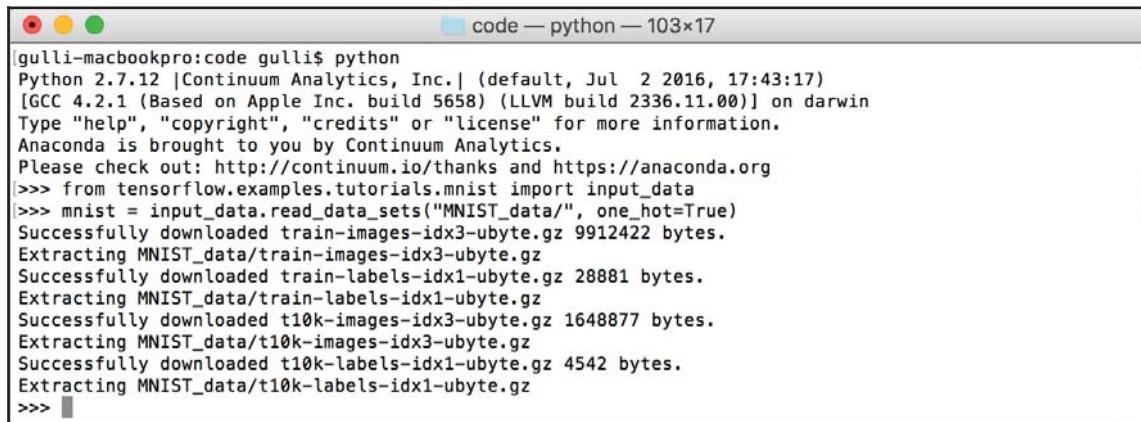
```
gulli-macbookpro:code gulli$ pip install keras
Collecting keras
  Requirement already satisfied: theano in /Users/gulli/miniconda2/lib/python2.7/site-packages (from keras)
Requirement already satisfied: pyyaml in /Users/gulli/miniconda2/lib/python2.7/site-packages (from keras)
Requirement already satisfied: six in /Users/gulli/miniconda2/lib/python2.7/site-packages (from keras)
Requirement already satisfied: numpy>=1.9.1 in /Users/gulli/miniconda2/lib/python2.7/site-packages (from theano->keras)
Requirement already satisfied: scipy>=0.14 in /Users/gulli/miniconda2/lib/python2.7/site-packages (from theano->keras)
Installing collected packages: keras
Successfully installed keras-1.1.1
gulli-macbookpro:code gulli$
```

Step 5 — testing Theano, TensorFlow, and Keras

Now let's test the environment. First let's look at how to define the sigmoid function in Theano. As you see, it is very simple; we just write the mathematical formula and compute the function element-wise on a matrix. Just run the Python Shell and write the code as shown in the following screenshot to get the result:

```
>>> import theano
>>> import theano.tensor as T
>>> x = T.dmatrix('x')
>>> s = 1 / (1 + T.exp(-x))
>>> logistic = theano.function([x], s)
>>> logistic([[0, 1], [-1, -2]])
array([[ 0.5          ,  0.73105858],
       [ 0.26894142,  0.11920292]])
>>>
```

So, Theano works. Let's test TensorFlow by simply importing the MNIST dataset as shown in the following screenshot. We have already seen, in *Chapter 1, Neural Networks Foundations*, a few working examples of the Keras network:



The screenshot shows a terminal window on a Macbook Pro. The title bar says "code — python — 103x17". The terminal content is as follows:

```
gulli-macbookpro:code gulli$ python
Python 2.7.12 |Continuum Analytics, Inc.| (default, Jul 2 2016, 17:43:17)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.11.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>> from tensorflow.examples.tutorials.mnist import input_data
>>> mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting MNIST_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
>>> █
```

Configuring Keras

Keras has a very minimalist configuration file. Let's load it with a `vi` session. The parameters are very simple:

Parameters	Values
image_dim_ordering	Can be either <code>tf</code> for the TensorFlow image ordering or <code>th</code> for Theano image ordering
epsilon	The <code>epsilon</code> value used during computation
floatx	Can be either <code>float32</code> or <code>float64</code>
backend	Can be either <code>tensorflow</code> or <code>theano</code>

The `image_dim_ordering` of `th` value gives you a somewhat non-intuitive dimension ordering for images (depth, width, and height), instead of (width, height, and depth), for `tf`. The following are the default parameters in my machine:



```
code — vi ~/keras/keras.json — 103x8
[{"image_dim_ordering": "th",
 "epsilon": 1e-07,
 "floatx": "float32",
 "backend": "tensorflow"}]
~/keras/keras.json" [noeol] 6L, 113C
```



If you install a GPU-enabled TensorFlow version, then Keras will automatically use your configured GPU when TensorFlow is selected as the backend.

Installing Keras on Docker

One of the easiest ways to get started with TensorFlow and Keras is running in a Docker container. A convenient solution is to use a predefined Docker image for deep learning created by the community that contains all the popular DL frameworks (TensorFlow, Theano, Torch, Caffe, and so on). Refer to the GitHub repository at <https://github.com/saiprashanths/dl-docker> for the code files. Assuming that you already have Docker up and running (for more information, refer to <https://www.docker.com/products/overview>), installing it is pretty simple and is shown as follows:

```
gulli-macbookpro:dl-docker gulli$ git clone https://github.com/saiprashanths/dl-docker
.git
Cloning into 'dl-docker'...
remote: Counting objects: 89, done.
remote: Total 89 (delta 0), reused 0 (delta 0), pack-reused 89
Unpacking objects: 100% (89/89), done.
gulli-macbookpro:dl-docker gulli$
```

The following screenshot, says something like, after getting the image from Git, we build the Docker image:

```
[gulli-macbookpro:dl-docker gulli$ cd dl-docker/
[gulli-macbookpro:dl-docker gulli$ docker build -t floydhub/dl-docker:cpu -f Dockerfile]
.cpu .
Sending build context to Docker daemon 284.2 kB
Step 1 : FROM ubuntu:14.04
--> 3f755ca42730
Step 2 : MAINTAINER Sai Soundararaj <saip@outlook.com>
--> Using cache
--> af02b42bde1c
Step 3 : ARG THEANO_VERSION=rel-0.8.2
--> Using cache
--> c8d03ba70cff
Step 4 : ARG TENSORFLOW_VERSION=0.8.0
--> Using cache
--> de0ed51e5732
Step 5 : ARG TENSORFLOW_ARCH=cpu
--> Using cache
--> 270d4bfbccaa
Step 6 : ARG KERAS_VERSION=1.0.3
--> Using cache
--> 61219a95474f
Step 7 : ARG LASAGNE_VERSION=v0.1
--> Using cache
--> 585e125f1e76
Step 8 : ARG TORCH_VERSION=latest
--> Using cache
--> fa5c4246c2ec
Step 9 : ARG CAFFE_VERSION=master
--> Using cache
--> 989ad8491f04
Step 10 : RUN apt-get update && apt-get install -y          bc           build-
```

In this screenshot, we see how to run it:

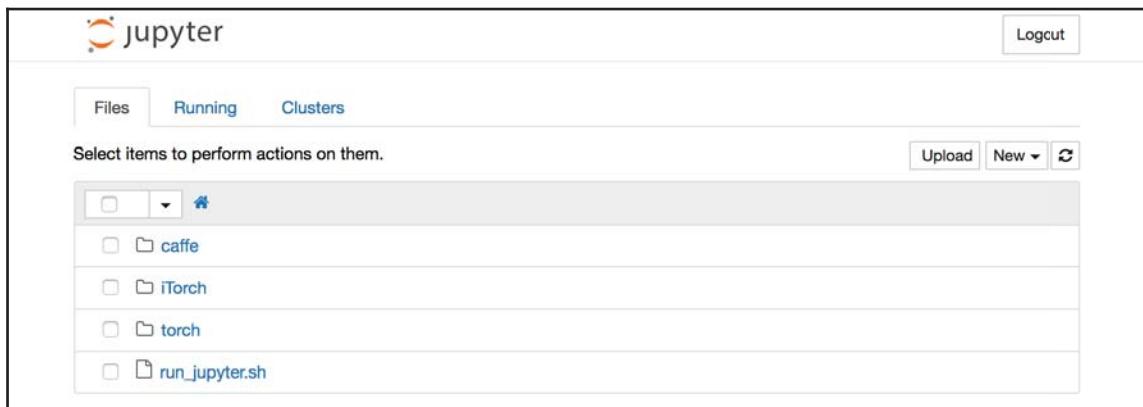
```
[gulli-macbookpro:dl-docker gulli$ docker run -it -p 8888:8888 -p 6006:6006 floydhub/dl]
-docker:cpu bash
[root@780e0d54bfc0:~# ls
caffe iTorch run_jupyter.sh torch
root@780e0d54bfc0:~# ]
```

From within the container, it is possible to activate support for Jupyter Notebooks (for more information, refer to <http://jupyter.org/>):

```
root@780e0d54bfc0:~# sh run_jupyter.sh
[I 10:51:17.489 NotebookApp] Copying /root/.ipython/kernels -> /root/.local/share/jupyter/kernels
[I 10:51:17.498 NotebookApp] Writing notebook server cookie secret to /root/.local/share/jupyter/runtime/notebook_cookie_secret
[W 10:51:17.520 NotebookApp] WARNING: The notebook server is listening on all IP addresses and not using encryption. This is not recommended.
[I 10:51:17.536 NotebookApp] Serving notebooks from local directory: /root
[I 10:51:17.536 NotebookApp] 0 active kernels
[I 10:51:17.537 NotebookApp] The Jupyter Notebook is running at: http://[all ip addresses on your system]:8888/?token=503b59dc969d43f588638e3bd153dd1525837ff46d7b1eb9
[I 10:51:17.537 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 10:51:17.539 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
    http://localhost:8888/?token=503b59dc969d43f588638e3bd153dd1525837ff46d7b1eb9
[I 10:51:32.547 NotebookApp] 302 GET / (172.17.0.1) 0.60ms
[I 10:51:32.553 NotebookApp] 302 GET /tree? (172.17.0.1) 0.86ms
[I 10:51:40.207 NotebookApp] 302 GET /?token=503b59dc969d43f588638e3bd153dd1525837ff46d7b1eb9 (172.17.0.1) 0.36ms
```

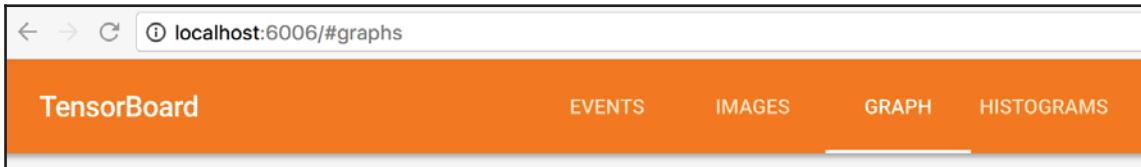
Access it directly from the host machine on port:



It is also possible to access TensorBoard (for more information, refer to https://www.tensorflow.org/how_tos/summaries_and_tensorboard/) with the help of the command in the screenshot that follows, which is discussed in the next section:

```
root@7b599d0dcaeb:~# tensorboard --logdir .
```

After running the preceding command, you will be redirected to the following page:



Installing Keras on Google Cloud ML

Installing Keras on Google Cloud is very simple. First, we can install Google Cloud (for the downloadable file, refer to <https://cloud.google.com/sdk/>), a command-line interface for Google Cloud Platform; then we can use CloudML, a managed service that enables us to easily build machine learning models with TensorFlow. Before using Keras, let's use Google Cloud with TensorFlow to train an MNIST example available on GitHub. The code is local and training happens in the cloud:

```
gulli-macbookpro:google-cloud-sdk gulli$ git clone https://github.com/GoogleCloudPlatform/cloudml-samples/
Cloning into 'cloudml-samples'...
remote: Counting objects: 118, done.
remote: Total 118 (delta 0), reused 0 (delta 0), pack-reused 118
Receiving objects: 100% (118/118), 84.40 KiB | 0 bytes/s, done.
Resolving deltas: 100% (49/49), done.
gulli-macbookpro:google-cloud-sdk gulli$
```

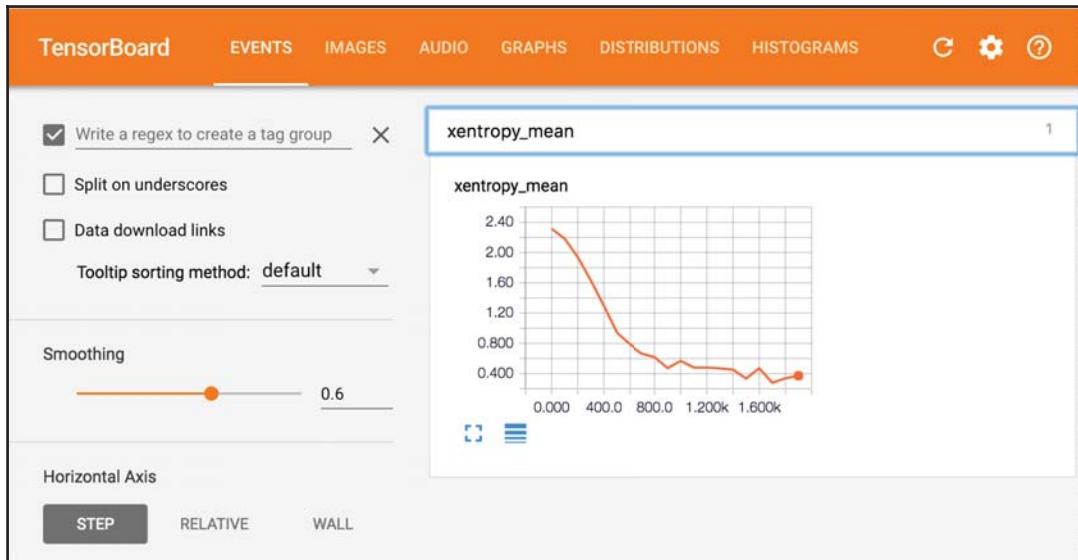
In the following screenshot, you can see how to run a training session:

```
gulli-macbookpro:codeBook gulli$ cd cloudml-samples/mnist/trainable/
gulli-macbookpro:trainable gulli$ ls
trainer
gulli-macbookpro:trainable gulli$ gcloud beta ml local train --package-path=trainer --module-name=trainer.task
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /var/folders/dx/s5b40ll92sz_sls6btf35mjr00cn0l/T/tmpcARKfj/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /var/folders/dx/s5b40ll92sz_sls6btf35mjr00cn0l/T/tmpcARKfj/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /var/folders/dx/s5b40ll92sz_sls6btf35mjr00cn0l/T/tmpcARKfj/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /var/folders/dx/s5b40ll92sz_sls6btf35mjr00cn0l/T/tmpcARKfj/t10k-labels-idx1-ubyte.gz
Step 0: loss = 2.32 (0.018 sec)
Step 100: loss = 2.19 (0.002 sec)
Step 200: loss = 1.94 (0.002 sec)
Step 300: loss = 1.64 (0.002 sec)
Step 400: loss = 1.30 (0.002 sec)
Step 500: loss = 0.95 (0.002 sec)
Step 600: loss = 0.80 (0.002 sec)
Step 700: loss = 0.67 (0.002 sec)
Step 800: loss = 0.62 (0.002 sec)
Step 900: loss = 0.48 (0.002 sec)
Training Data Eval:
  Num examples: 55000  Num correct: 47295  Precision @ 1: 0.8599
Validation Data Eval:
  Num examples: 5000  Num correct: 4347  Precision @ 1: 0.8694
Test Data Eval:
  Num examples: 10000  Num correct: 8649  Precision @ 1: 0.8649
Step 1000: loss = 0.58 (0.018 sec)
Step 1100: loss = 0.49 (0.115 sec)
Step 1200: loss = 0.49 (0.002 sec)
Step 1300: loss = 0.48 (0.002 sec)
Step 1400: loss = 0.46 (0.002 sec)
Step 1500: loss = 0.34 (0.002 sec)
Step 1600: loss = 0.49 (0.002 sec)
Step 1700: loss = 0.29 (0.002 sec)
Step 1800: loss = 0.35 (0.002 sec)
Step 1900: loss = 0.39 (0.002 sec)
Training Data Eval:
  Num examples: 55000  Num correct: 49243  Precision @ 1: 0.8953
Validation Data Eval:
  Num examples: 5000  Num correct: 4519  Precision @ 1: 0.9038
Test Data Eval:
  Num examples: 10000  Num correct: 9000  Precision @ 1: 0.9000
gulli-macbookpro:trainable gulli$
```

We can use TensorBoard to show how cross-entropy decreases across iterations:

```
gulli-macbookpro:trainable gulli$ tensorboard --logdir=data/ --port=8080
Starting TensorBoard 29 on port 8080
```

In the next screenshot, we see the graph of cross-entropy:



Now, if we want to use Keras on the top of TensorFlow, we simply download the Keras source from PyPI (for the downloadable file, refer to <https://pypi.python.org/pypi/Keras/1.2.0> or later versions) and then directly use Keras as a CloudML package solution, as in the following example:

```
gulli-macbookpro:trainable gulli$ gcloud beta ml local train --package-path=trainer --package-path=../../../../CloudML/fchollet-keras-1.2.0-0-g12d068f.tar.gz --module-name=trainer.task2
Using TensorFlow backend.
(0, 'input_1', (None, 224, 224, 3))
(1, 'block1_conv1', (None, 224, 224, 64))
(2, 'block1_conv2', (None, 224, 224, 64))
(3, 'block1_pool', (None, 112, 112, 64))
(4, 'block2_conv1', (None, 112, 112, 128))
(5, 'block2_conv2', (None, 112, 112, 128))
(6, 'block2_pool', (None, 56, 56, 128))
(7, 'block3_conv1', (None, 56, 56, 256))
(8, 'block3_conv2', (None, 56, 56, 256))
(9, 'block3_conv3', (None, 56, 56, 256))
(10, 'block3_pool', (None, 28, 28, 256))
(11, 'block4_conv1', (None, 28, 28, 512))
(12, 'block4_conv2', (None, 28, 28, 512))
(13, 'block4_conv3', (None, 28, 28, 512))
(14, 'block4_pool', (None, 14, 14, 512))
(15, 'block5_conv1', (None, 14, 14, 512))
(16, 'block5_conv2', (None, 14, 14, 512))
(17, 'block5_conv3', (None, 14, 14, 512))
(18, 'block5_pool', (None, 7, 7, 512))
(19, 'flatten', (None, 25088))
(20, 'fc1', (None, 4096))
(21, 'fc2', (None, 4096))
(22, 'predictions', (None, 1000))
gulli-macbookpro:trainable gulli$ ls
data    trainer
gulli-macbookpro:trainable gulli$
```

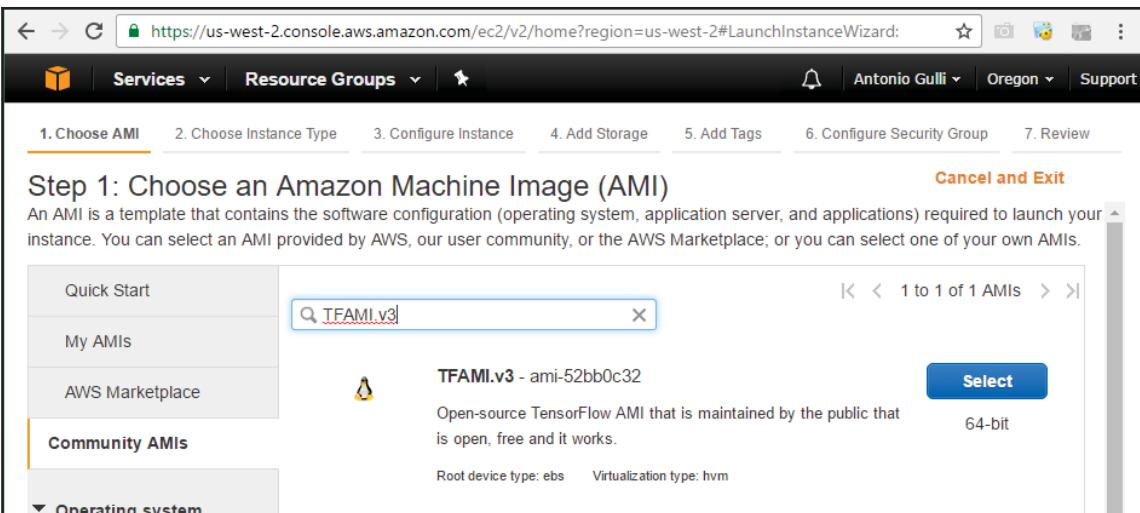
Here, `trainer.task2.py` is an example script:

```
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np

# pre-built and pre-trained deep learning VGG16 model
base_model = VGG16(weights='imagenet', include_top=True)
for i, layer in enumerate(base_model.layers):
    print (i, layer.name, layer.output_shape)
```

Installing Keras on Amazon AWS

Installing TensorFlow and Keras on Amazon is very simple. Indeed, it is possible to use a prebuilt AMI named `TFAMI.v3` that is open and free (for more information, refer to <https://github.com/ritchieng/tensorflow-aws-ami>), shown as follows:



The screenshot shows the AWS EC2 Launch Instance Wizard at Step 1: Choose an Amazon Machine Image (AMI). The search bar contains "TFAMI.v3". A single result is listed: "TFAMI.v3 - ami-52bb0c32". It is described as an "Open-source TensorFlow AMI that is maintained by the public that is open, free and it works." The "Select" button is highlighted in blue. The sidebar on the left shows categories like Quick Start, My AMIs, AWS Marketplace, and Community AMIs, with "Community AMIs" currently selected.

This AMI runs TensorFlow in less than five minutes and supports TensorFlow, Keras, OpenAI Gym, and all dependencies. As of January 2017, it supports the following:

- TensorFlow 0.12
- Keras 1.1.0
- TensorLayer 1.2.7
- CUDA 8.0
- CuDNN 5.1
- Python 2.7
- Ubuntu 16.04

In addition, TFAMI .v3 works on P2 computing instances (for more information, refer to <https://aws.amazon.com/ec2/instance-types/#p2>), as shown in the following screenshot:

The screenshot shows the AWS EC2 Launch Instance Wizard at Step 2. The URL in the browser is <https://us-west-2.console.aws.amazon.com/ec2/v2/home?region=us-west-2#LaunchInstanceWizard>. The page title is "Step 2: Choose an Instance Type". Below the title, a note states: "Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. [Learn more](#) about instance types and how they can meet your computing needs." A filter bar at the top allows filtering by "GPU compute" and "Current generation". A table below lists available instance types:

	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
<input type="checkbox"/>	GPU compute	p2.xlarge	4	61	EBS only	Yes	High
<input type="checkbox"/>	GPU compute	p2.8xlarge	32	488	EBS only	Yes	10 Gigabit
<input type="checkbox"/>	GPU compute	p2.16xlarge	64	732	EBS only	Yes	20 Gigabit

Some features of P2 instances are as follows:

- Intel Xeon E5-2686v4 (Broadwell) processors
- NVIDIA K80 GPUs, each with 2,496 parallel cores and 12 GB of GPU memory
- Supports peer-to-peer GPU communication
- Provides enhanced networking (for more information, refer to https://aws.amazon.com/ec2/faqs/#What_networking_capabilities_are_included_in_this_feature) with 20 Gbps of aggregate network bandwidth

The TFAMI.v3 also works on G2 computing instances (for more information, refer to <https://aws.amazon.com/ec2/instance-types/#g2>). Some features of G2 instances are as follows:

- Intel Xeon E5-2670 (Sandy Bridge) processors
- NVIDIA GPUs, each with 1,536 CUDA cores and 4 GB of video memory

Installing Keras on Microsoft Azure

One way to install Keras on Azure is to install the support for Docker and then get a containerized version of TensorFlow plus Keras. Online, it is also possible to find a detailed set of instructions on how to install Keras and TensorFlow with Docker, but this is essentially what we have seen already in a previous section (for more information, refer to https://blogs.msdn.microsoft.com/uk_faculty_connection/2016/09/26/tensorflow-on-docker-with-microsoft-azure/).

If you use Theano as the only backend, then Keras can run with just a click by loading a pre-built package available on Cortana Intelligence Gallery (for more information, refer to <https://gallery.cortanaintelligence.com/Experiment/Theano-Keras-1>).

The following sample shows how to import Theano and Keras into Azure ML directly as a ZIP file and use them in the Execute Python Script module. This example is due to Hai Ning (for more information, refer to <https://goo.gl/VLR25o>), and it essentially runs the Keras code within the `azureml_main()` method:

```
# The script MUST contain a function named azureml_main
# which is the entry point for this module.

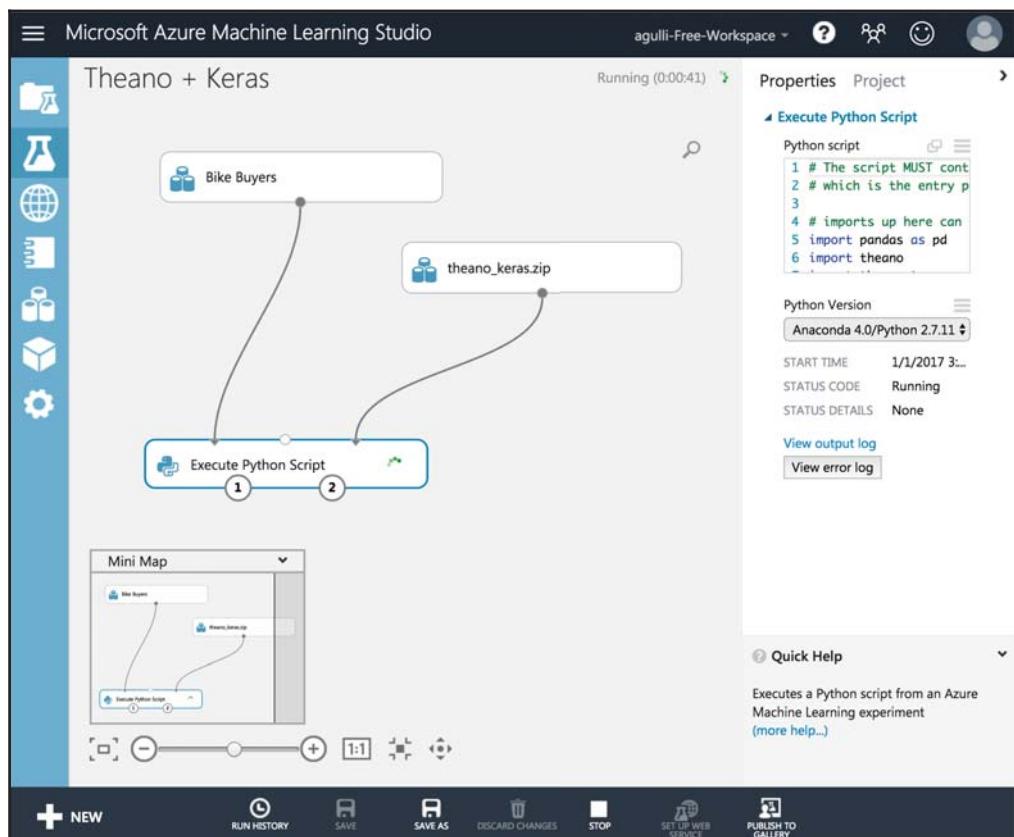
# imports up here can be used to
import pandas as pd
import theano
import theano.tensor as T
from theano import function
from keras.models import Sequential
from keras.layers import Dense, Activation
import numpy as np
# The entry point function can contain up to two input arguments:
#   Param<dataframe1>: a pandas.DataFrame
#   Param<dataframe2>: a pandas.DataFrame
def azureml_main(dataframe1 = None, dataframe2 = None):
    # Execution logic goes here
    # print('Input pandas.DataFrame #1:rn{0}'.format(dataframe1))

    # If a zip file is connected to the third input port is connected,
```

```
# it is unzipped under ".Script Bundle". This directory is added
# to sys.path. Therefore, if your zip file contains a Python file
# mymodule.py you can import it using:
# import mymodule
model = Sequential()
model.add(Dense(1, input_dim=784, activation="relu"))
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
metrics=['accuracy'])
data = np.random.random((1000,784))
labels = np.random.randint(2, size=(1000,1))
model.fit(data, labels, nb_epoch=10, batch_size=32)
model.evaluate(data, labels)

return datafram1,
```

In this screenshot, you see an example use of Microsoft Azure ML to run Theano and Keras:



Keras API

Keras has a modular, minimalist, and easy extendable architecture. Francois Chollet, the author of Keras, says:

The library was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

Keras defines high-level neural networks running on top of either TensorFlow (for more information, refer to <https://github.com/tensorflow/tensorflow>) or Theano (for more information, refer to <https://github.com/Theano/Theano>). In details:

- **Modularity:** A model is either a sequence or a graph of standalone modules that can be combined together like LEGO blocks for building neural networks. Namely, the library predefines a very large number of modules implementing different types of neural layers, cost functions, optimizers, initialization schemes, activation functions, and regularization schemes.
- **Minimalism:** The library is implemented in Python and each module is kept short and self-describing.
- **Easy extensibility:** The library can be extended with new functionalities, as we will describe in Chapter 7, *Additional Deep Learning Models*.

Getting started with Keras architecture

In this section, we review the most important Keras components used for defining neural networks. First, we define what a tensor is, then we discuss different ways of composing predefined modules, and we conclude with an overview of the ones most commonly used.

What is a tensor?

Keras uses either Theano or TensorFlow to perform very efficient computations on tensors. But what is a tensor anyway? A tensor is nothing but a multidimensional array or matrix. Both the backends are capable of efficient symbolic computations on tensors, which are the fundamental building blocks for creating neural networks.

Composing models in Keras

There are two ways of composing models in Keras. They are as follows:

- Sequential composition
- Functional composition

Let us take a look at each one in detail.

Sequential composition

The first one is the sequential composition, where different predefined models are stacked together in a linear pipeline of layers similar to a stack or a queue. In *Chapter 1, Neural Networks Foundations*, we saw a few examples of sequential pipelines. For instance:

```
model = Sequential()
model.add(Dense(N_HIDDEN, input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dropout(DROPOUT))
model.add(Dense(N_HIDDEN))
model.add(Activation('relu'))
model.add(Dropout(DROPOUT))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))
model.summary()
```

Functional composition

The second way of composing modules is via the functional API, where it is possible to define complex models, such as directed acyclic graphs, models with shared layers, or multi-output models. We will see such examples in *Chapter 7, Additional Deep Learning Models*.

An overview of predefined neural network layers

Keras has a number of prebuilt layers. Let us review the most commonly used ones and highlight in which chapter these layers are mostly used.

Regular dense

A dense model is a fully connected neural network layer. We have already seen examples of usage in Chapter 1, *Neural Networks Foundations*. Here is the prototype with a definition of the parameters:

```
keras.layers.core.Dense(units, activation=None, use_bias=True,  
kernel_initializer='glorot_uniform', bias_initializer='zeros',  
kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,  
kernel_constraint=None, bias_constraint=None)
```

Recurrent neural networks — simple, LSTM, and GRU

Recurrent neural networks are a class of neural networks that exploit the sequential nature of their input. Such inputs could be a text, a speech, time series, and anything else where the occurrence of an element in the sequence is dependent on the elements that appeared before it. We will discuss simple, LSTM, and GRU recurrent neural networks in Chapter 6, *Recurrent Neural Network — RNN*. Here you can see some prototypes with a definition of the parameters:

```
keras.layers.recurrent.Recurrent(return_sequences=False,  
go_backwards=False, stateful=False, unroll=False, implementation=0)  
  
keras.layers.recurrent.SimpleRNN(units, activation='tanh', use_bias=True,  
kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',  
bias_initializer='zeros', kernel_regularizer=None,  
recurrent_regularizer=None, bias_regularizer=None,  
activity_regularizer=None, kernel_constraint=None,  
recurrent_constraint=None, bias_constraint=None, dropout=0.0,  
recurrent_dropout=0.0)  
  
keras.layers.recurrent.GRU(units, activation='tanh',  
recurrent_activation='hard_sigmoid', use_bias=True,  
kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',  
bias_initializer='zeros', kernel_regularizer=None,  
recurrent_regularizer=None, bias_regularizer=None,  
activity_regularizer=None, kernel_constraint=None,  
recurrent_constraint=None, bias_constraint=None, dropout=0.0,  
recurrent_dropout=0.0)  
  
keras.layers.recurrent.LSTM(units, activation='tanh',  
recurrent_activation='hard_sigmoid', use_bias=True,  
kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',  
bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None,  
recurrent_regularizer=None, bias_regularizer=None,  
activity_regularizer=None, kernel_constraint=None,
```

```
recurrent_constraint=None, bias_constraint=None, dropout=0.0,
recurrent_dropout=0.0)
```

Convolutional and pooling layers

ConvNets are a class of neural networks using convolutional and pooling operations for progressively learning rather sophisticated models based on progressive levels of abstraction. This learning via progressive abstraction resembles vision models that have evolved over millions of years inside the human brain. People called it *deep* with 3-5 layers a few years ago, and now it has gone up to 100-200. We will discuss convolutional neural networks in Chapter 3, *Deep Learning with ConvNets*. Here are some prototypes with a definition of the parameters:

```
keras.layers.convolutional.Conv1D(filters, kernel_size, strides=1,
padding='valid', dilation_rate=1, activation=None, use_bias=True,
kernel_initializer='glorot_uniform', bias_initializer='zeros',
kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
kernel_constraint=None, bias_constraint=None)

keras.layers.convolutional.Conv2D(filters, kernel_size, strides=(1, 1),
padding='valid', data_format=None, dilation_rate=(1, 1), activation=None,
use_bias=True, kernel_initializer='glorot_uniform',
bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None,
activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

keras.layers.pooling.MaxPooling1D(pool_size=2, strides=None,
padding='valid')

keras.layers.pooling.MaxPooling2D(pool_size=(2, 2), strides=None,
padding='valid', data_format=None)
```

Regularization

Regularization is a way to prevent overfitting. We have already seen examples of usage in Chapter 1, *Neural Networks Foundations*. Multiple layers have parameters for regularization. The following is the list of regularization parameters commonly used for dense, and convolutional modules:

- **kernel_regularizer**: Regularizer function applied to the weight matrix
- **bias_regularizer**: Regularizer function applied to the bias vector
- **activity_regularizer**: Regularizer function applied to the output of the layer (its activation)

In addition is possible to use Dropout for regularization and that is frequently a very effective choice

```
keras.layers.core.Dropout(rate, noise_shape=None, seed=None)
```

Where:

- `rate`: It is a float between 0 and 1 which represents the fraction of the input units to drop
- `noise_shape`: It is a 1D integer tensor which represents the shape of the binary dropout mask that will be multiplied with the input
- `seed`: It is a integer which is used use as random seed

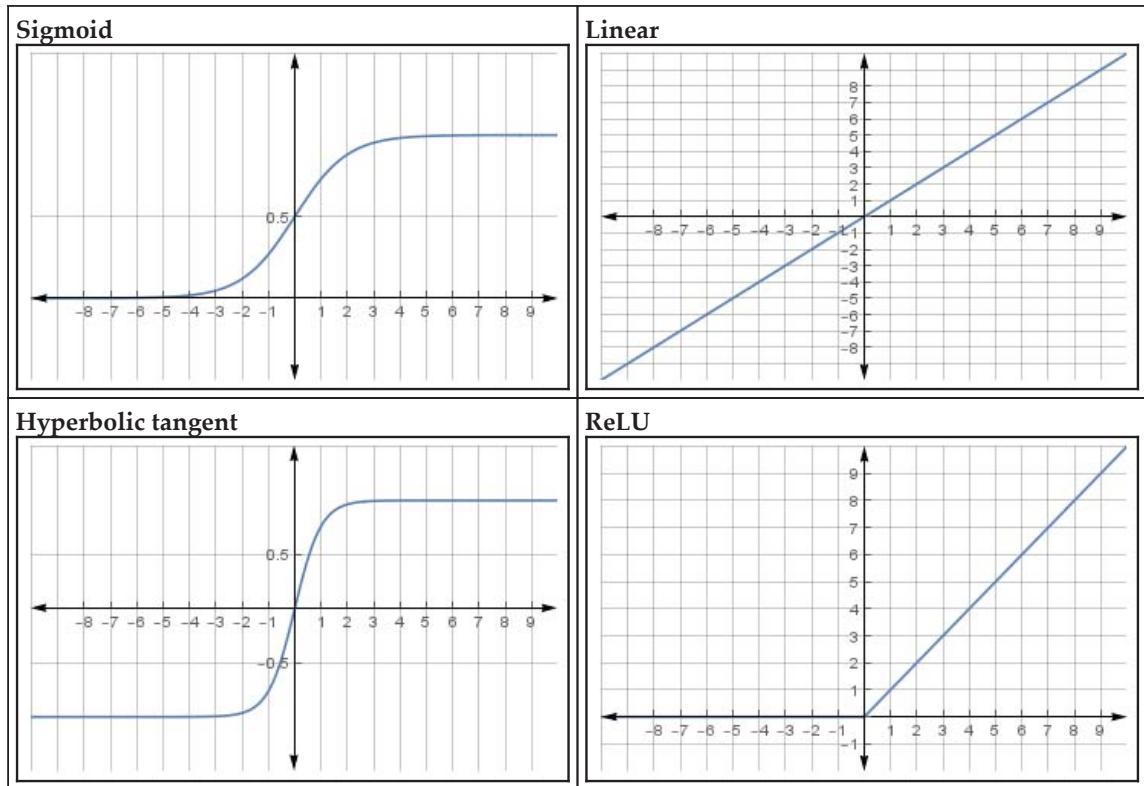
Batch normalization

Batch normalization (for more information, refer to <https://www.colwiz.com/cite-in-google-docs/cid=f20f9683aaf69ce>) is a way to accelerate learning and generally achieve better accuracy. We will look at examples of usage in Chapter 4, *Generative Adversarial Networks and WaveNet*, when we discuss GANs. Here is the prototype with a definition of the parameters:

```
keras.layers.normalization.BatchNormalization(axis=-1, momentum=0.99,
epsilon=0.001, center=True, scale=True, beta_initializer='zeros',
gamma_initializer='ones', moving_mean_initializer='zeros',
moving_variance_initializer='ones', beta_regularizer=None,
gamma_regularizer=None, beta_constraint=None, gamma_constraint=None)
```

An overview of predefined activation functions

Activation includes commonly used functions such as sigmoid, linear, hyperbolic tangent, and ReLU. We have seen a few examples of activation functions in Chapter 1, *Neural Networks Foundations*, and more examples will be presented in the next chapters. The following diagrams are examples of sigmoid, linear, hyperbolic tangent, and ReLU activation functions:



An overview of losses functions

Losses functions (or objective functions, or optimization score function; for more information, refer to <https://keras.io/losses/>) can be classified into four categories:

- Accuracy which is used for classification problems. There are multiple choices: `binary_accuracy` (mean accuracy rate across all predictions for binary classification problems), `categorical_accuracy` (mean accuracy rate across all predictions for multiclass classification problems), `sparse_categorical_accuracy` (useful for sparse targets), and `top_k_categorical_accuracy` (success when the target class is within the `top_k` predictions provided).

- Error loss, which measures the difference between the values predicted and the values actually observed. There are multiple choices: `mse` (mean square error between predicted and target values), `rmse` (root square error between predicted and target values), `mae` (mean absolute error between predicted and target values), `mape` (mean percentage error between predicted and target values), and `msle` (mean squared logarithmic error between predicted and target values).
- Hinge loss, which is generally used for training classifiers. There are two versions: `hinge` defined as $\max(1 - y_{true} * y_{pred}, 0)$ and `squared_hinge` defined as the squared value of the hinge loss.
- Class loss is used to calculate the cross-entropy for classification problems. There are multiple versions, including binary cross-entropy (for more information, refer to https://en.wikipedia.org/wiki/Cross_entropy), and categorical cross-entropy.

We have seen a few examples of objective functions in Chapter 1, *Neural Networks Foundations*, and more examples will be presented in the next chapters.

An overview of metrics

A metric function (for more information, refer to <https://keras.io/metrics/>) is similar to an objective function. The only difference is that the results from evaluating a metric are not used when training the model. We have seen a few examples of metrics in Chapter 1, *Neural Networks Foundations*, and more examples will be presented in the next chapters.

An overview of optimizers

Optimizers include SGD, RMSprop, and Adam. We have seen a few examples of optimizers in Chapter 1, *Neural Networks Foundations*, and more examples (Adagrad and Adadelta; for more information, refer to <https://keras.io/optimizers/>) will be presented in the next chapters.

Some useful operations

Here we report some utility operations that can be carried out with Keras APIs. The goal is to facilitate the creation of networks, the training process, and the saving of intermediate results.

Saving and loading the weights and the architecture of a model

Model architectures can be easily saved and loaded as follows:

```
# save as JSON json_string = model.to_json()
# save as YAML yaml_string = model.to_yaml()
# model reconstruction from JSON: from keras.models import model_from_json
model = model_from_json(json_string) # model reconstruction from YAML model
= model_from_yaml(yaml_string)
```

Model parameters (weights) can be easily saved and loaded as follows:

```
from keras.models import load_model model.save('my_model.h5')
# creates a HDF5 file 'my_model.h5' del model
# deletes the existing model
# returns a compiled model
# identical to the previous one model = load_model('my_model.h5')
```

Callbacks for customizing the training process

The training process can be stopped when a metric has stopped improving by using an appropriate callback:

```
keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0,
patience=0, verbose=0, mode='auto')
```

Loss history can be saved by defining a callback like the following:

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []
    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))
model = Sequential()
model.add(Dense(10, input_dim=784, init='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')
history = LossHistory()
model.fit(X_train, Y_train, batch_size=128, nb_epoch=20,
verbose=0, callbacks=[history])
print history.losses
```

Checkpointing

Checkpointing is a process that saves a snapshot of the application's state at regular intervals, so the application can be restarted from the last saved state in case of failure. This is useful during training of deep learning models, which can often be a time-consuming task. The state of a deep learning model at any point in time is the weights of the model at that time. Keras saves these weights in HDF5 format (for more information, refer to <https://www.hdfgroup.org/>) and provides checkpointing using its callback API.

Some scenarios where checkpointing can be useful include the following:

- If you want the ability to restart from your last checkpoint after your AWS Spot instance (for more information, refer to <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/how-spot-instances-work.html>) or Google preemptible virtual machine (for more information, refer to <https://cloud.google.com/compute/docs/instances/preemptible>) is unexpectedly terminated
- If you want to stop training, perhaps to test your model on test data, then continue training from the last checkpoint
- If you want to retain the best version (by some metric such as validation loss) as it trains over multiple epochs

The first and second scenarios can be handled by saving a checkpoint after each epoch, which is handled by the default usage of the `ModelCheckpoint` callback. The following code illustrates how to add checkpointing during training of your deep learning model in Keras:

```
from __future__ import division, print_function
from keras.callbacks import ModelCheckpoint
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout
from keras.utils import np_utils
import numpy as np
import os

BATCH_SIZE = 128
NUM_EPOCHS = 20
MODEL_DIR = "/tmp"

(Xtrain, ytrain), (Xtest, ytest) = mnist.load_data()
Xtrain = Xtrain.reshape(60000, 784).astype("float32") / 255
Xtest = Xtest.reshape(10000, 784).astype("float32") / 255
Ytrain = np_utils.to_categorical(ytrain, 10)
Ytest = np_utils.to_categorical(ytest, 10)
```

```
print(Xtrain.shape, Xtest.shape, Ytrain.shape, Ytest.shape)

model = Sequential()
model.add(Dense(512, input_shape=(784,), activation="relu"))
model.add(Dropout(0.2))
model.add(Dense(512, activation="relu"))
model.add(Dropout(0.2))
model.add(Dense(10, activation="softmax"))

model.compile(optimizer="rmsprop", loss="categorical_crossentropy",
              metrics=["accuracy"])

# save best model
checkpoint = ModelCheckpoint(
    filepath=os.path.join(MODEL_DIR, "model-{epoch:02d}.h5"))
model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE, nb_epoch=NUM_EPOCHS,
           validation_split=0.1, callbacks=[checkpoint])
```

The third scenario involves monitoring a metric, such as validation accuracy or loss, and only saving a checkpoint if the current metric is better than the previously saved checkpoint. Keras provides an additional parameter, `save_best_only`, which needs to be set to `true` when instantiating the `checkpoint` object in order to support this functionality.

Using TensorBoard and Keras

Keras provides a callback for saving your training and test metrics, as well as activation histograms for the different layers in your model:

```
keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0,
                            write_graph=True, write_images=False)
```

Saved data can then be visualized with TensorBoard launched at the command line:

```
tensorboard --logdir=/full_path_to_your_logs
```

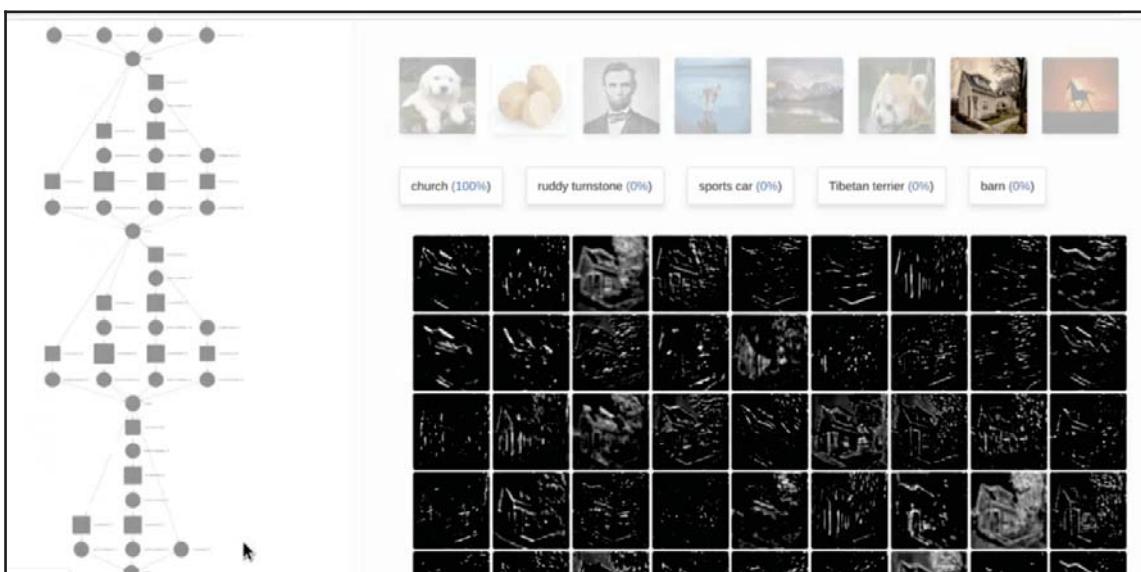
Using Quiver and Keras

In Chapter 3, *Deep Learning with ConvNets*, we will discuss ConvNets, which are an advanced deep learning technique for dealing with images. Here we give a preview of Quiver (for more information, refer to <https://github.com/jakebian/quiver>), a tool useful for visualizing ConvNets features in an interactive way. The installation is pretty simple, and after that Quiver can be used with one single line:

```
pip install quiver_engine

from quiver_engine import server      server.launch(model)
```

This will launch the visualization at `localhost:5000`. Quiver allows you to visually inspect a neural network, as in the following example:



Summary

In this chapter, we discussed how to install Theano, TensorFlow, and Keras on the following:

- Your local machine
- A dockerized infrastructure based on containers
- In the cloud with Google GCP, Amazon AWS, and Microsoft Azure

In addition to that, we looked at a few modules defining Keras APIs and some commonly useful operations such as loading and saving neural networks' architectures and weights, early stopping, history saving, checkpointing, interactions with TensorBoard, and interactions with Quiver.

In the next chapter, we will introduce the concept of convolutional networks a fundamental innovation in deep learning which has been used with success in multiple domains from text, to video, to speech going well beyond the initial image processing domain where they were originally conceived.

3

Deep Learning with ConvNets

In previous chapters, we discussed dense nets, in which each layer is fully connected to the adjacent layers. We applied those dense networks to classify the MNIST handwritten characters dataset. In that context, each pixel in the input image is assigned to a neuron for a total of 784 (28 x 28 pixels) input neurons. However, this strategy does not leverage the spatial structure and relations of each image. In particular, this piece of code transforms the bitmap representing each written digit into a flat vector, where the spatial locality is gone:

```
#X_train is 60000 rows of 28x28 values --> reshaped in 60000 x 784  
X_train = X_train.reshape(60000, 784)  
X_test = X_test.reshape(10000, 784)  
o
```

Convolutional neural networks (also called ConvNet) leverage spatial information and are therefore very well suited for classifying images. These nets use an ad hoc architecture inspired by biological data taken from physiological experiments done on the visual cortex. As discussed, our vision is based on multiple cortex levels, each one recognizing more and more structured information. First, we see single pixels; then from them, we recognize simple geometric forms. And then... more and more sophisticated elements such as objects, faces, human bodies, animals, and so on.

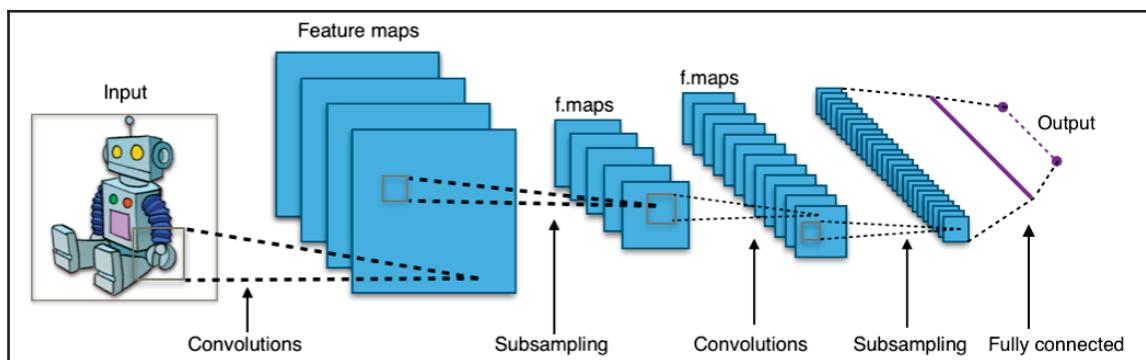
Convolutional neural networks are indeed fascinating. Over a short period of time, they become a *disruptive* technology, breaking all the state-of-the-art results in multiple domains, from text, to video, to speech going well beyond the initial image processing domain where they were originally conceived.

In this chapter, we will cover the following topics:

- Deep convolutional neural networks
- Image classification

Deep convolutional neural network — DCNN

A **deep convolutional neural network (DCNN)** consists of many neural network layers. Two different types of layers, convolutional and pooling, are typically alternated. The depth of each filter increases from left to right in the network. The last stage is typically made of one or more fully connected layers:



There are three key intuitions beyond ConvNets:

- Local receptive fields
- Shared weights
- Pooling

Let's review them.

Local receptive fields

If we want to preserve spatial information, then it is convenient to represent each image with a matrix of pixels. Then, a simple way to encode the local structure is to connect a submatrix of adjacent input neurons into one single hidden neuron belonging to the next layer. That single hidden neuron represents one local receptive field. Note that this operation is named convolution and it gives the name to this type of network.

Of course, we can encode more information by having overlapping submatrices. For instance, let's suppose that the size of each single submatrix is 5×5 and that those submatrices are used with MNIST images of 28×28 pixels. Then we will be able to generate 23×23 local receptive field neurons in the next hidden layer. In fact it is possible to slide the submatrices by only 23 positions before touching the borders of the images. In Keras, the size of each single submatrix is called **stride length**, and this is a hyperparameter that can be fine-tuned during the construction of our nets.

Let's define the feature map from one layer to another layer. Of course, we can have multiple feature maps that learn independently from each hidden layer. For instance, we can start with 28×28 input neurons for processing MINST images and then recall k feature maps of size 23×23 neurons each (again with a stride of 5×5) in the next hidden layer.

Shared weights and bias

Let's suppose that we want to move away from the pixel representation in a row by gaining the ability to detect the same feature independently from the location where it is placed in the input image. A simple intuition is to use the same set of weights and bias for all the neurons in the hidden layers. In this way, each layer will learn a set of position-independent latent features derived from the image.

Assuming that the input image has shape $(256, 256)$ on three channels with *tf* (TensorFlow) ordering, this is represented as $(256, 256, 3)$. Note that with *th* (Theano) mode, the channel's dimension (the depth) is at index 1; in *tf* (TensorFlow) mode, it is at index 3.

In Keras, if we want to add a convolutional layer with dimensionality of the output 32 and extension of each filter 3×3 , we will write:

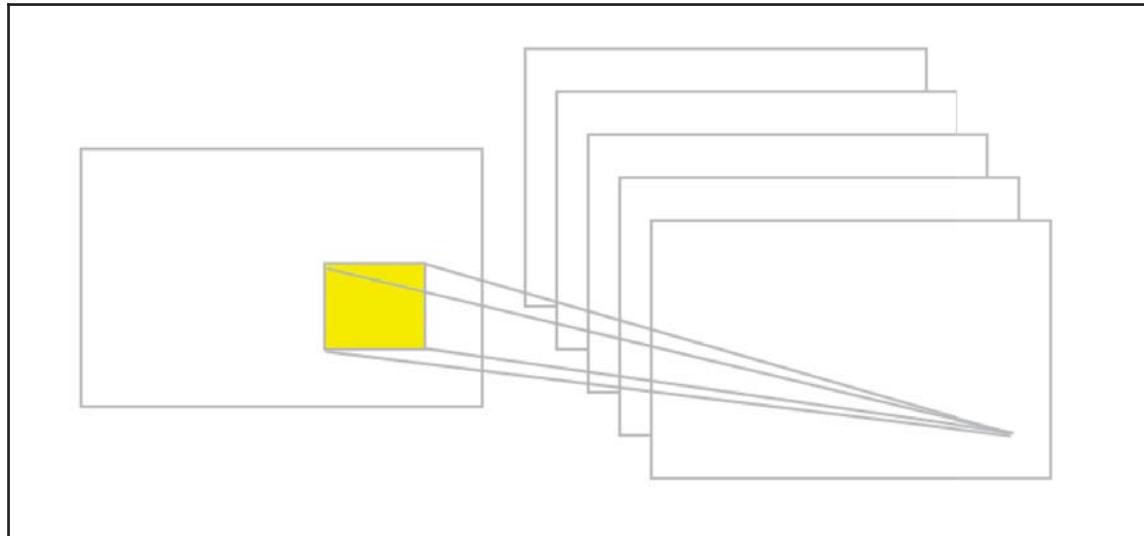
```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(256, 256, 3))
```

Alternatively, we will write:

```
model = Sequential()
model.add(Conv2D(32, kernel_size=3, input_shape=(256, 256, 3))
```

This means that we are applying a 3×3 convolution on a 256×256 image with three input channels (or input filters), resulting in 32 output channels (or output filters).

An example of convolution is provided in the following diagram:



Pooling layers

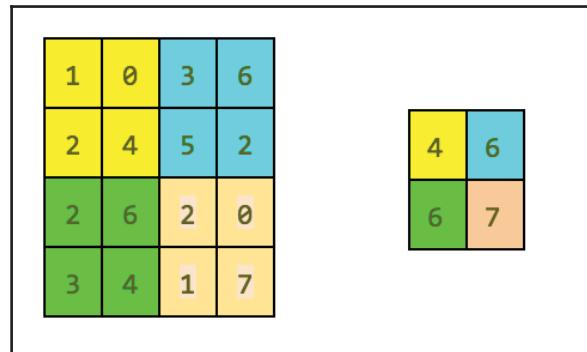
Let's suppose that we want to summarize the output of a feature map. Again, we can use the spatial contiguity of the output produced from a single feature map and aggregate the values of a submatrix into a single output value that synthetically describes the *meaning* associated with that physical region.

Max-pooling

One easy and common choice is *max-pooling*, which simply outputs the maximum activation as observed in the region. In Keras, if we want to define a max-pooling layer of size 2×2 , we will write:

```
model.add(MaxPooling2D(pool_size = (2, 2)))
```

An example of max-pooling is shown in the following diagram:



Average pooling

Another choice is average pooling, which simply aggregates a region into the average values of the activations observed in that region.

Note that Keras implements a large number of pooling layers and a complete list is available at: <https://keras.io/layers/pooling/>. In short, all pooling operations are nothing more than a summary operation on a given region.

ConvNets summary

So far, we have described the basic concepts of ConvNets. CNNs apply convolution and pooling operations in one dimension for audio and text data along the time dimension, in two dimensions for images along the (height x width) dimensions, and in three dimensions for videos along the (height x width x time) dimensions. For images, sliding the filter over input volume produces a map that gives the responses of the filter for each spatial position. In other words, a ConvNet has multiple filters stacked together which learn to recognize specific visual features independently of the location in the image. Those visual features are simple in the initial layers of the network, and then more and more sophisticated deeper in the network.

An example of DCNN — LeNet

Yann le Cun proposed (for more information refer to: *Convolutional Networks for Images, Speech, and Time-Series*, by Y. LeCun and Y. Bengio, brain theory neural networks, vol. 3361, 1995) a family of ConvNets named LeNet trained for recognizing MNIST handwritten characters with robustness to simple geometric transformations and to distortion. The key intuition here is to have low-layers alternating convolution operations with max-pooling operations. The convolution operations are based on carefully chosen local receptive fields with shared weights for multiple feature maps. Then, higher levels are fully connected layers based on a traditional MLP with hidden layers and softmax as the output layer.

LeNet code in Keras

To define LeNet code, we use a convolutional 2D module, which is:

```
keras.layers.convolutional.Conv2D(filters, kernel_size, padding='valid')
```

Here, `filters` is the number of convolution kernels to use (for example, the dimensionality of the output), `kernel_size` is an integer or tuple/list of two integers, specifying the width and height of the 2D convolution window (can be a single integer to specify the same value for all spatial dimensions), and `padding='same'` means that padding is used. There are two options: `padding='valid'` means that the convolution is only computed where the input and the filter fully overlap, and therefore the output is smaller than the input, while `padding='same'` means that we have an output that is the *same* size as the input, for which the area around the input is padded with zeros.

In addition, we use a `MaxPooling2D` module:

```
keras.layers.pooling.MaxPooling2D(pool_size=(2, 2), strides=(2, 2))
```

Here, `pool_size=(2, 2)` is a tuple of two integers representing the factors by which the image is vertically and horizontally downsampled. So `(2, 2)` will halve the image in each dimension, and `strides=(2, 2)` is the stride used for processing.

Now, let us review the code. First we import a number of modules:

```
from keras import backend as K
from keras.models import Sequential
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dense
from keras.datasets import mnist
from keras.utils import np_utils
from keras.optimizers import SGD, RMSprop, Adam
import numpy as np
import matplotlib.pyplot as plt
```

Then we define the LeNet network:

```
#define the ConvNet
class LeNet:
    @staticmethod
    def build(input_shape, classes):
        model = Sequential()
        # CONV => RELU => POOL
```

We have a first convolutional stage with ReLU activations followed by a max-pooling. Our net will learn 20 convolutional filters, each one of which has a size of 5×5 . The output dimension is the same one of the input shape, so it will be 28×28 . Note that since the Convolution2D is the first stage of our pipeline, we are also required to define its input_shape. The max-pooling operation implements a sliding window that slides over the layer and takes the maximum of each region with a step of two pixels vertically and horizontally:

```
model.add(Convolution2D(20, kernel_size=5, padding="same",
input_shape=input_shape))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
# CONV => RELU => POOL
```

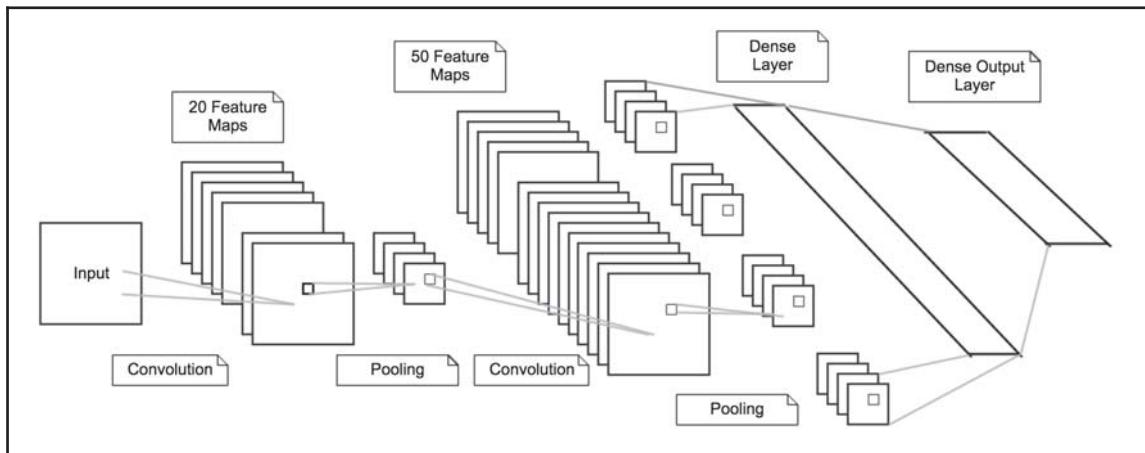
Then a second convolutional stage with ReLU activations follows, again by a max-pooling. In this case, we increase the number of convolutional filters learned to 50 from the previous 20. Increasing the number of filters in deeper layers is a common technique used in deep learning:

```
model.add(Conv2D(50, kernel_size=5, border_mode="same"))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
```

Then we have a pretty standard flattening and a dense network of 500 neurons, followed by a softmax classifier with 10 classes:

```
# Flatten => RELU layers
model.add(Flatten())
model.add(Dense(500))
model.add(Activation("relu"))
# a softmax classifier
model.add(Dense(classes))
model.add(Activation("softmax"))
return model
```

Congratulations, You have just defined the first deep learning network! Let's see how it looks visually:

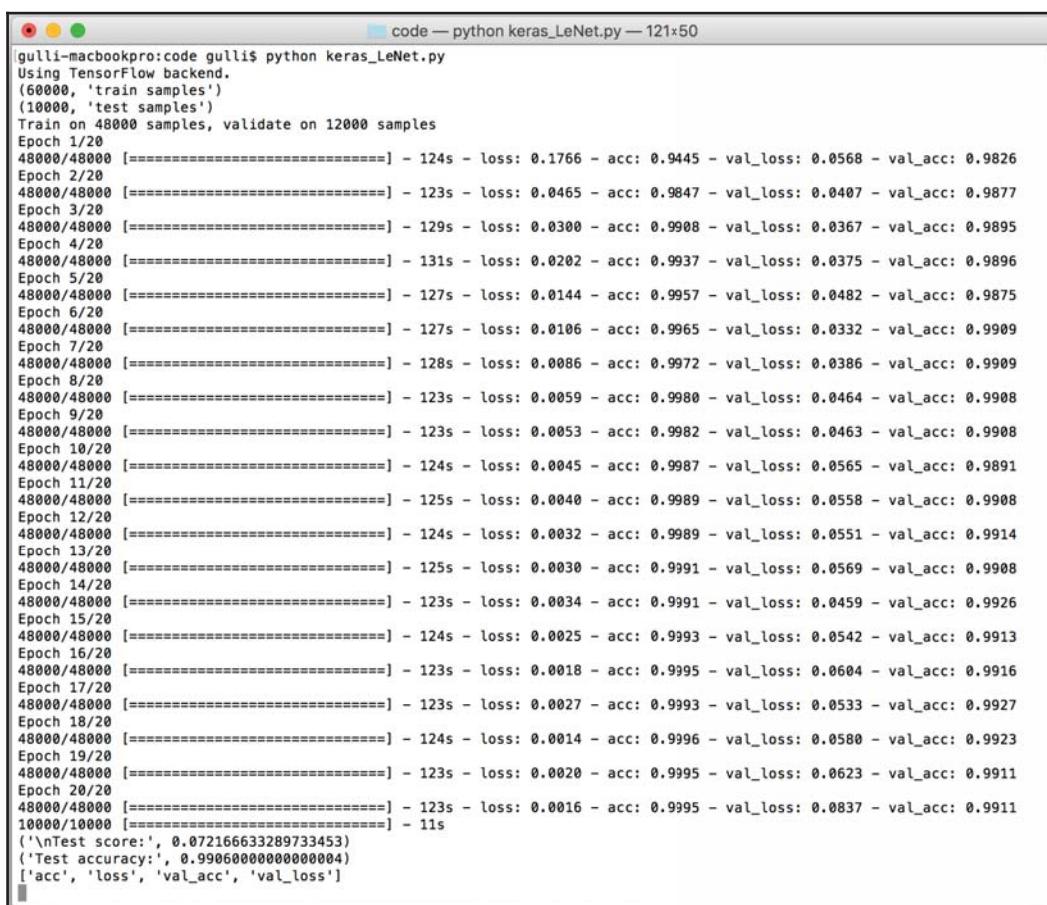


Now we need some additional code for training the network, but this is very similar to what we have already described in Chapter 1, *Neural Network Foundations*. This time, we also show the code for printing the loss:

```
# network and training
NB_EPOCH = 20
BATCH_SIZE = 128
VERBOSE = 1
OPTIMIZER = Adam()
VALIDATION_SPLIT=0.2
IMG_ROWS, IMG_COLS = 28, 28 # input image dimensions
NB_CLASSES = 10 # number of outputs = number of digits
INPUT_SHAPE = (1, IMG_ROWS, IMG_COLS)
# data: shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()
k.set_image_dim_ordering("th")
# consider them as float and normalize
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
# we need a 60K x [1 x 28 x 28] shape as input to the CONVNET
X_train = X_train[:, np.newaxis, :, :]
X_test = X_test[:, np.newaxis, :, :]
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
# convert class vectors to binary class matrices
y_train = np_utils.to_categorical(y_train, NB_CLASSES)
y_test = np_utils.to_categorical(y_test, NB_CLASSES)
# initialize the optimizer and model
model = LeNet.build(input_shape=INPUT_SHAPE, classes=NB_CLASSES)
model.compile(loss="categorical_crossentropy", optimizer=OPTIMIZER,
metrics=["accuracy"])
history = model.fit(X_train, y_train,
batch_size=BATCH_SIZE, epochs=NB_EPOCH,
verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
score = model.evaluate(X_test, y_test, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
```

```
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

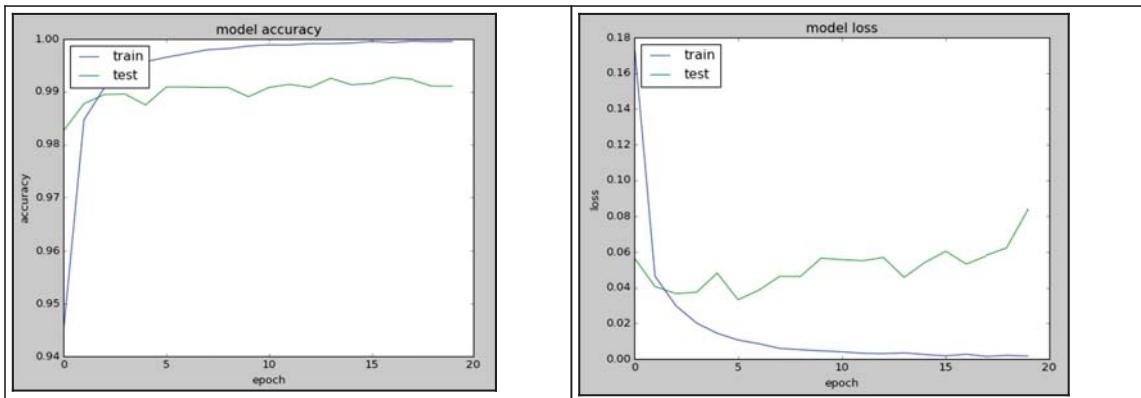
Now let's run the code. As you can see, the time had a significant increase and each iteration in our deep net now takes ~134 seconds against ~1-2 seconds for the net defined in Chapter 1, *Neural Network Foundations*. However, the accuracy has reached a new peak at 99.06%:



The terminal window shows the command `gulli$ python keras_LeNet.py` being run. The output displays the training progress over 20 epochs. Each epoch shows the training loss, accuracy, validation loss, and validation accuracy. The validation accuracy reaches 99.06% by epoch 20.

```
code — python keras_LeNet.py — 121x50
gulli-macbookpro:code gulli$ python keras_LeNet.py
Using TensorFlow backend.
(60000, 'train samples')
(10000, 'test samples')
Train on 48000 samples, validate on 12000 samples
Epoch 1/20
48000/48000 [=====] - 124s - loss: 0.1766 - acc: 0.9445 - val_loss: 0.0568 - val_acc: 0.9826
Epoch 2/20
48000/48000 [=====] - 123s - loss: 0.0465 - acc: 0.9847 - val_loss: 0.0407 - val_acc: 0.9877
Epoch 3/20
48000/48000 [=====] - 129s - loss: 0.0300 - acc: 0.9908 - val_loss: 0.0367 - val_acc: 0.9895
Epoch 4/20
48000/48000 [=====] - 131s - loss: 0.0202 - acc: 0.9937 - val_loss: 0.0375 - val_acc: 0.9896
Epoch 5/20
48000/48000 [=====] - 127s - loss: 0.0144 - acc: 0.9957 - val_loss: 0.0482 - val_acc: 0.9875
Epoch 6/20
48000/48000 [=====] - 127s - loss: 0.0106 - acc: 0.9965 - val_loss: 0.0332 - val_acc: 0.9909
Epoch 7/20
48000/48000 [=====] - 128s - loss: 0.0086 - acc: 0.9972 - val_loss: 0.0386 - val_acc: 0.9909
Epoch 8/20
48000/48000 [=====] - 123s - loss: 0.0059 - acc: 0.9980 - val_loss: 0.0464 - val_acc: 0.9908
Epoch 9/20
48000/48000 [=====] - 123s - loss: 0.0053 - acc: 0.9982 - val_loss: 0.0463 - val_acc: 0.9908
Epoch 10/20
48000/48000 [=====] - 124s - loss: 0.0045 - acc: 0.9987 - val_loss: 0.0565 - val_acc: 0.9891
Epoch 11/20
48000/48000 [=====] - 125s - loss: 0.0040 - acc: 0.9989 - val_loss: 0.0558 - val_acc: 0.9908
Epoch 12/20
48000/48000 [=====] - 124s - loss: 0.0032 - acc: 0.9989 - val_loss: 0.0551 - val_acc: 0.9914
Epoch 13/20
48000/48000 [=====] - 125s - loss: 0.0030 - acc: 0.9991 - val_loss: 0.0569 - val_acc: 0.9908
Epoch 14/20
48000/48000 [=====] - 123s - loss: 0.0034 - acc: 0.9991 - val_loss: 0.0459 - val_acc: 0.9926
Epoch 15/20
48000/48000 [=====] - 124s - loss: 0.0025 - acc: 0.9993 - val_loss: 0.0542 - val_acc: 0.9913
Epoch 16/20
48000/48000 [=====] - 123s - loss: 0.0018 - acc: 0.9995 - val_loss: 0.0604 - val_acc: 0.9916
Epoch 17/20
48000/48000 [=====] - 123s - loss: 0.0027 - acc: 0.9993 - val_loss: 0.0533 - val_acc: 0.9927
Epoch 18/20
48000/48000 [=====] - 124s - loss: 0.0014 - acc: 0.9996 - val_loss: 0.0580 - val_acc: 0.9923
Epoch 19/20
48000/48000 [=====] - 123s - loss: 0.0020 - acc: 0.9995 - val_loss: 0.0623 - val_acc: 0.9911
Epoch 20/20
48000/48000 [=====] - 123s - loss: 0.0016 - acc: 0.9995 - val_loss: 0.0837 - val_acc: 0.9911
10000/10000 [=====] - 11s
('Test score:', 0.072166633289733453)
('Test accuracy:', 0.9906000000000004)
['acc', 'loss', 'val_acc', 'val_loss']
```

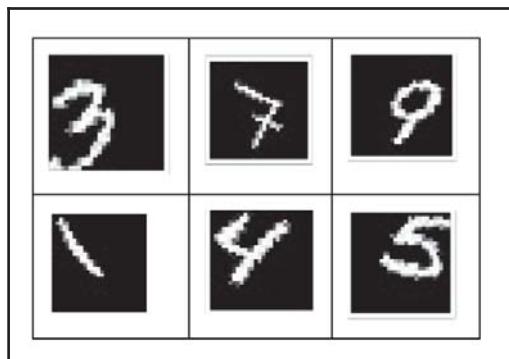
Let's plot the model accuracy and the model loss, and we understand that we can train in only 4 - 5 iterations to achieve a similar accuracy of 99.2%:



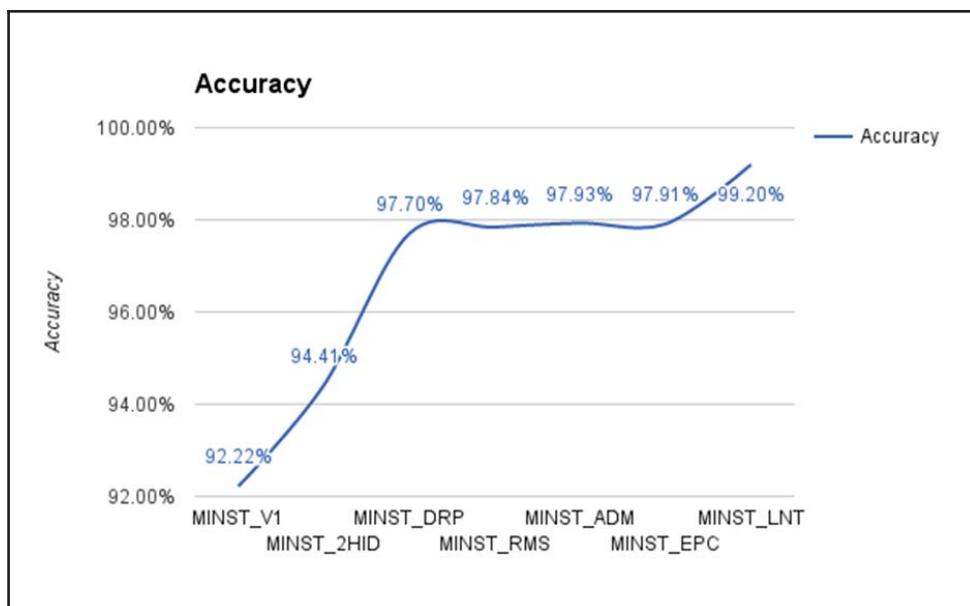
In the following screenshot, we show the final accuracy achieved by our model:

```
gulli-macbookpro:code gulli$ python keras_LeNet.py -- 121x18
Using TensorFlow backend.
(60000, 'train samples')
(10000, 'test samples')
Train on 48000 samples, validate on 12000 samples
Epoch 1/4
48000/48000 [=====] - 139s - loss: 0.1758 - acc: 0.9450 - val_loss: 0.0618 - val_acc: 0.9806
Epoch 2/4
48000/48000 [=====] - 136s - loss: 0.0461 - acc: 0.9849 - val_loss: 0.0408 - val_acc: 0.9878
Epoch 3/4
48000/48000 [=====] - 130s - loss: 0.0294 - acc: 0.9905 - val_loss: 0.0413 - val_acc: 0.9889
Epoch 4/4
48000/48000 [=====] - 129s - loss: 0.0199 - acc: 0.9936 - val_loss: 0.0373 - val_acc: 0.9900
10000/10000 [=====] - 12s
('Test score:', 0.027107118735135736)
('Test accuracy:', 0.9920999999999998)
['acc', 'loss', 'val_acc', 'val_loss']
```

Let's see some of the MNIST images just to understand how good the number 99.2% is! For instance, there are many ways in which humans write a 9, one of them appearing in the following diagram. The same holds for 3, 7, 4, and 5. The number 1 in this diagram is so difficult to recognize that probably even a human will have issues with it:



We can summarize all the progress made so far with our different models in the following graph. Our simple net started with an accuracy of **92.22%**, which means that about 8 handwritten characters out of 100 are not correctly recognized. Then, we gained 7% with the deep learning architecture by reaching an accuracy of **99.20%**, which means that about 1 handwritten character out of 100 is incorrectly recognized:



Understanding the power of deep learning

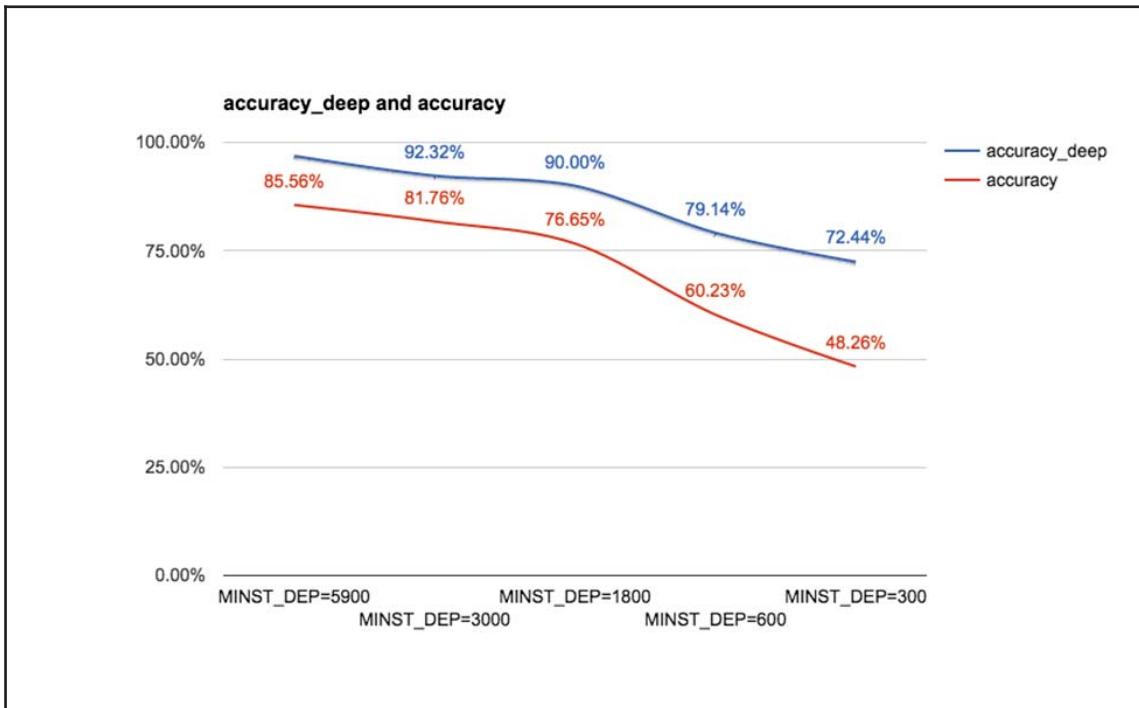
Another test that we can run to better understand the power of deep learning and ConvNet is to reduce the size of the training set and observe the consequent decay in performance. One way to do this is to split the training set of 50,000 examples into two different sets:

- The proper training set used for training our model will progressively reduce its size of (5,900, 3,000, 1,800, 600, and 300) examples
- The validation set used to estimate how well our model has been trained will consist of the remaining examples

Our test set is always fixed and it consists of 10,000 examples.

With this setup, we compare the just-defined deep learning ConvNet against the first example of neural network defined in [Chapter 1, Neural Network Foundations](#). As we can see in the following graph, our deep network always outperforms the simple network and the gap is more and more evident when the number of examples provided for training is progressively reduced. With 5,900 training examples the deep learning net had an accuracy of 96.68% against an accuracy of 85.56% of the simple net. More important, with only 300 training examples our deep learning net still has an accuracy of 72.44% while the simple net shows a significant decay at 48.26%. All the experiments are run for only four training iterations. This confirms the breakthrough progress achieved with deep learning. At first glance this could be surprising from a mathematical point of view because the deep network has many more unknowns (the weights), so one would think we need many more data points.

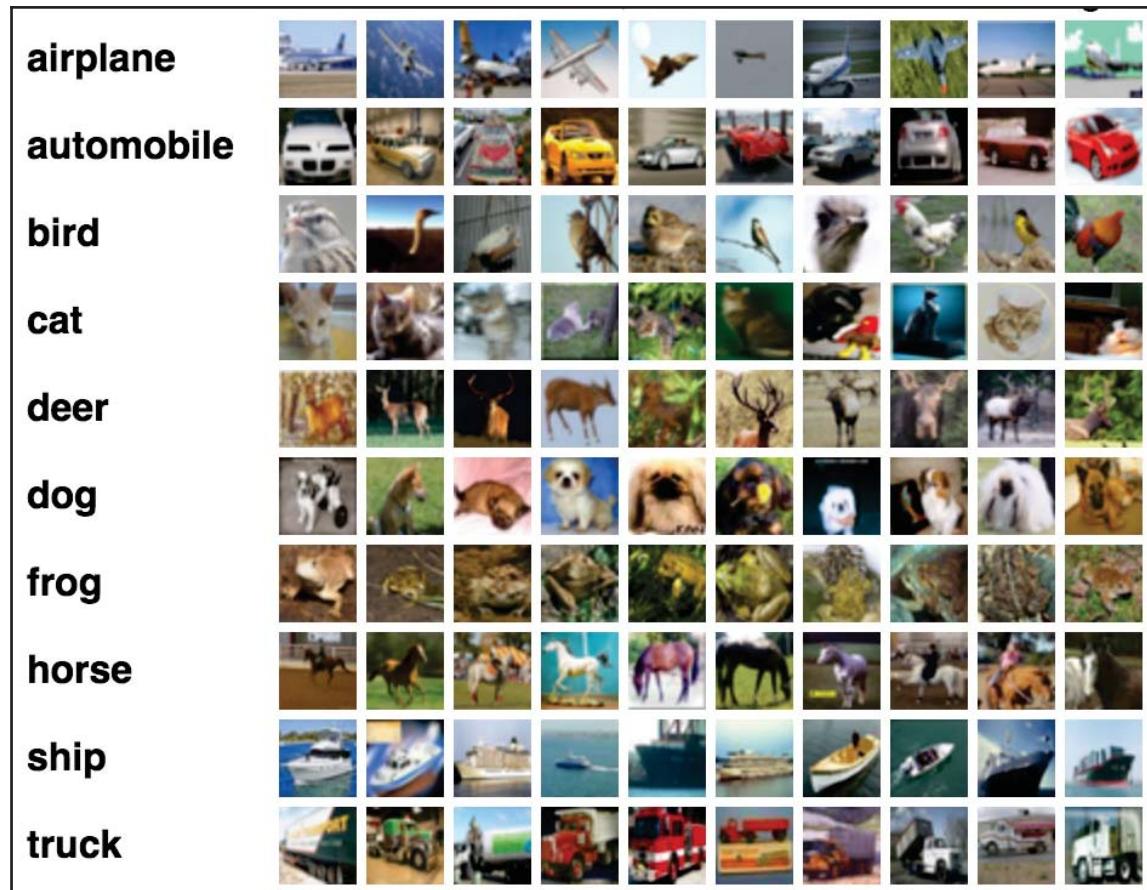
However, preserving the spatial information, adding convolution, pooling, and feature maps is innovation with ConvNets, and this was optimized on millions of years (since this organization has been inspired by the visual cortex):



A list of state-of-the-art results for MNIST is available at: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html. As of January, 2017, the best result has an error rate of 0.21%.

Recognizing CIFAR-10 images with deep learning

The CIFAR-10 dataset contains 60,000 color images of 32×32 pixels in 3 channels divided into 10 classes. Each class contains 6,000 images. The training set contains 50,000 images, while the test sets provides 10,000 images. This image taken from the CIFAR repository (<http://www.cs.toronto.edu/~kriz/cifar.html>) describes a few random examples from the 10 classes:



The goal is to recognize previously unseen images and assign them to one of the 10 classes. Let us define a suitable deep net.

First of all we import a number of useful modules, define a few constants, and load the dataset:

```

from keras.datasets import cifar10
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.optimizers import SGD, Adam, RMSprop
import matplotlib.pyplot as plt

# CIFAR_10 is a set of 60K images 32x32 pixels on 3 channels
IMG_CHANNELS = 3

```

```
IMG_ROWS = 32
IMG_COLS = 32

#constant
BATCH_SIZE = 128
NB_EPOCH = 20
NB_CLASSES = 10
VERBOSE = 1
VALIDATION_SPLIT = 0.2
OPTIM = RMSprop()

#load dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

Now let's do a one-hot encoding and normalize the images:

```
# convert to categorical
Y_train = np_utils.to_categorical(y_train, NB_CLASSES)
Y_test = np_utils.to_categorical(y_test, NB_CLASSES)

# float and normalization
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
```

Our net will learn 32 convolutional filters, each of which with a 3×3 size. The output dimension is the same one of the input shape, so it will be 32×32 and activation is ReLU, which is a simple way of introducing non-linearity. After that we have a max-pooling operation with pool size 2×2 and a dropout at 25%:

```
# network
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
                input_shape=(IMG_ROWS, IMG_COLS, IMG_CHANNELS)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

The next stage in the deep pipeline is a dense network with 512 units and ReLU activation followed by a dropout at 50% and by a softmax layer with 10 classes as output, one for each category:

```
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(NB_CLASSES))
model.add(Activation('softmax'))
model.summary()
```

After defining the network, we can train the model. In this case, we split the data and compute a validation set in addition to the training and testing sets. The training is used to build our models, the validation is used to select the best performing approach, while the test set is to check the performance of our best models on fresh unseen data:

```
# train
model.compile(loss='categorical_crossentropy', optimizer=OPTIM,
metrics=['accuracy'])
model.fit(X_train, Y_train, batch_size=BATCH_SIZE,
epochs=N_EPOCH, validation_split=VALIDATION_SPLIT,
verbose=VERBOSE)
score = model.evaluate(X_test, Y_test,
batch_size=BATCH_SIZE, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

In this case we save the architecture of our deep network:

```
#save model
model_json = model.to_json()
open('cifar10_architecture.json', 'w').write(model_json)
And the weights learned by our deep network on the training set
model.save_weights('cifar10_weights.h5', overwrite=True)
```

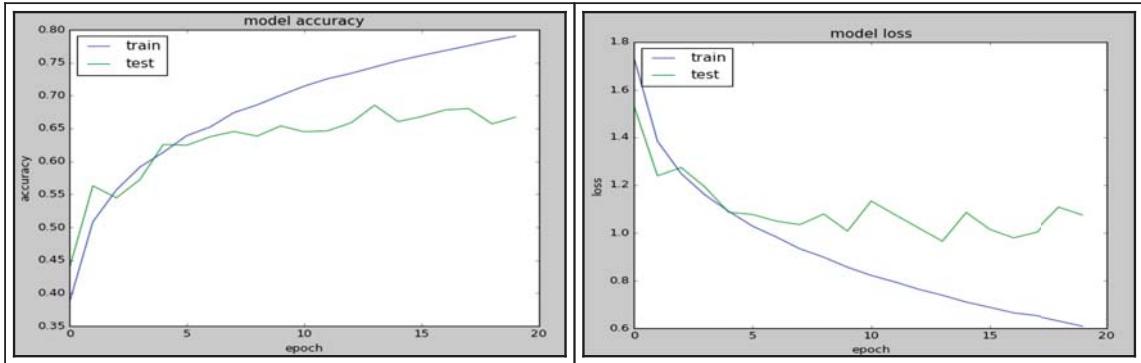
Let us run the code. Our network reaches a test accuracy of 66.4% with 20 iterations. We also print the accuracy and loss plot, and dump the network with `model.summary()`:

```
code — python keras_CIFAR10_simple.py — 121x77
gulli-macbookpro:code gulli$ python keras_CIFAR10_simple.py
Using TensorFlow backend.
('X_train shape:', (50000, 3, 32, 32))
(50000, 'train samples')
(10000, 'test samples')

Layer (type)          Output Shape         Param #     Connected to
=====
convolution2d_1 (Convolution2D)  (None, 32, 32, 32)  896        convolution2d_input_1[0][0]
activation_1 (Activation)       (None, 32, 32, 32)  0          convolution2d_1[0][0]
maxpooling2d_1 (MaxPooling2D)   (None, 32, 16, 16)   0          activation_1[0][0]
dropout_1 (Dropout)            (None, 32, 16, 16)   0          maxpooling2d_1[0][0]
flatten_1 (Flatten)           (None, 8192)        0          dropout_1[0][0]
dense_1 (Dense)              (None, 512)         4194816    flatten_1[0][0]
activation_2 (Activation)     (None, 512)         0          dense_1[0][0]
dropout_2 (Dropout)           (None, 512)         0          activation_2[0][0]
dense_2 (Dense)              (None, 10)          5130       dropout_2[0][0]
activation_3 (Activation)     (None, 10)          0          dense_2[0][0]
=====
Total params: 4200842

Train on 40000 samples, validate on 10000 samples
Epoch 1/20
40000/40000 [=====] - 114s - loss: 1.7380 - acc: 0.3855 - val_loss: 1.5353 - val_acc: 0.4376
Epoch 2/20
40000/40000 [=====] - 114s - loss: 1.3847 - acc: 0.5081 - val_loss: 1.2392 - val_acc: 0.5629
Epoch 3/20
40000/40000 [=====] - 116s - loss: 1.2481 - acc: 0.5566 - val_loss: 1.2737 - val_acc: 0.5446
Epoch 4/20
40000/40000 [=====] - 114s - loss: 1.1590 - acc: 0.5913 - val_loss: 1.1919 - val_acc: 0.5722
Epoch 5/20
40000/40000 [=====] - 116s - loss: 1.0904 - acc: 0.6138 - val_loss: 1.0860 - val_acc: 0.6257
Epoch 6/20
40000/40000 [=====] - 115s - loss: 1.0282 - acc: 0.6391 - val_loss: 1.0771 - val_acc: 0.6245
Epoch 7/20
40000/40000 [=====] - 115s - loss: 0.9828 - acc: 0.6523 - val_loss: 1.0491 - val_acc: 0.6375
Epoch 8/20
40000/40000 [=====] - 114s - loss: 0.9328 - acc: 0.6739 - val_loss: 1.0344 - val_acc: 0.6453
Epoch 9/20
40000/40000 [=====] - 114s - loss: 0.8978 - acc: 0.6858 - val_loss: 1.0789 - val_acc: 0.6384
Epoch 10/20
40000/40000 [=====] - 115s - loss: 0.8556 - acc: 0.7004 - val_loss: 1.0072 - val_acc: 0.6538
Epoch 11/20
40000/40000 [=====] - 114s - loss: 0.8215 - acc: 0.7142 - val_loss: 1.1334 - val_acc: 0.6450
Epoch 12/20
40000/40000 [=====] - 115s - loss: 0.7938 - acc: 0.7256 - val_loss: 1.0761 - val_acc: 0.6464
Epoch 13/20
40000/40000 [=====] - 118s - loss: 0.7631 - acc: 0.7337 - val_loss: 1.0204 - val_acc: 0.6587
Epoch 14/20
40000/40000 [=====] - 121s - loss: 0.7381 - acc: 0.7433 - val_loss: 0.9647 - val_acc: 0.6853
Epoch 15/20
40000/40000 [=====] - 114s - loss: 0.7094 - acc: 0.7529 - val_loss: 1.0852 - val_acc: 0.6604
Epoch 16/20
40000/40000 [=====] - 114s - loss: 0.6872 - acc: 0.7608 - val_loss: 1.0144 - val_acc: 0.6680
Epoch 17/20
40000/40000 [=====] - 115s - loss: 0.6642 - acc: 0.7682 - val_loss: 0.9787 - val_acc: 0.6781
Epoch 18/20
40000/40000 [=====] - 114s - loss: 0.6524 - acc: 0.7758 - val_loss: 1.0035 - val_acc: 0.6803
Epoch 19/20
40000/40000 [=====] - 114s - loss: 0.6302 - acc: 0.7834 - val_loss: 1.1080 - val_acc: 0.6571
Epoch 20/20
40000/40000 [=====] - 113s - loss: 0.6081 - acc: 0.7902 - val_loss: 1.0744 - val_acc: 0.6672
Testing...
10000/10000 [=====] - 13s
('Test score:', 1.0762448620796203)
('Test accuracy:', 0.6649000000000005)
['acc', 'loss', 'val_acc', 'val_loss']
```

In the following graph, we report the accuracy and the lost achieved by our net on both train and test datasets:



Improving the CIFAR-10 performance with deeper a network

One way to improve the performance is to define a deeper network with multiple convolutional operations. In this example, we have a sequence of modules:

conv+conv+maxpool+dropout+conv+conv+maxpool

Followed by a standard *dense+dropout+dense*. All the activation functions are ReLU.

Let us see the code for the new network:

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
input_shape=(IMG_ROWS, IMG_COLS, IMG_CHANNELS)))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512))
```

```
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(NB_CLASSES))
model.add(Activation('softmax'))
```

Congratulations! You have defined a deeper network. Let us run the code! First we dump the network, then we run for 40 iterations reaching an accuracy of 76.9%:

The screenshot shows a terminal window titled "code — python keras_CIFAR10_V2.py — 121x77". The window contains the following text:

```
~/Keras/codeBook/code — python keras_CIFAR10_V2.py — 121x77
~|/Keras/codeBook/code — python keras_CIFAR10_V1.py
gulli-macbookpro:code gulli$ python keras_CIFAR10_V1.py
Using TensorFlow backend.
('X_train shape:', (50000, 3, 32, 32)
(50000, 'train samples')
(10000, 'test samples')

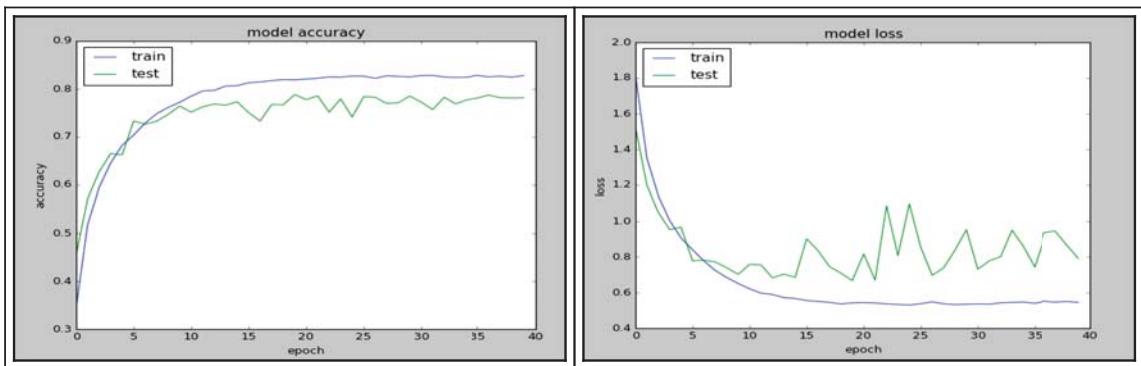
Layer (type)          Output Shape         Param #     Connected to
=====
convolution2d_1 (Convolution2D)  (None, 32, 32, 32)   896        convolution2d_input_1[0][0]
activation_1 (Activation)       (None, 32, 32, 32)   0          convolution2d_1[0][0]
convolution2d_2 (Convolution2D)  (None, 32, 32, 32)   9248       activation_1[0][0]
activation_2 (Activation)       (None, 32, 32, 32)   0          convolution2d_2[0][0]
maxpooling2d_1 (MaxPooling2D)   (None, 32, 16, 16)   0          activation_2[0][0]
dropout_1 (Dropout)            (None, 32, 16, 16)   0          maxpooling2d_1[0][0]
convolution2d_3 (Convolution2D)  (None, 64, 16, 16)   18496      dropout_1[0][0]
activation_3 (Activation)       (None, 64, 16, 16)   0          convolution2d_3[0][0]
convolution2d_4 (Convolution2D)  (None, 64, 14, 14)   36928       activation_3[0][0]
activation_4 (Activation)       (None, 64, 14, 14)   0          convolution2d_4[0][0]
maxpooling2d_2 (MaxPooling2D)   (None, 64, 7, 7)     0          activation_4[0][0]
dropout_2 (Dropout)            (None, 64, 7, 7)     0          maxpooling2d_2[0][0]
flatten_1 (Flatten)           (None, 3136)         0          dropout_2[0][0]
dense_1 (Dense)               (None, 512)          1606144    flatten_1[0][0]
activation_5 (Activation)       (None, 512)          0          dense_1[0][0]
dropout_3 (Dropout)            (None, 512)          0          activation_5[0][0]
dense_2 (Dense)               (None, 10)           5130       dropout_3[0][0]
activation_6 (Activation)       (None, 10)           0          dense_2[0][0]
=====
Total params: 1676842

Train on 40000 samples, validate on 10000 samples
Epoch 1/40
40000/40000 [=====] - 430s - loss: 1.8179 - acc: 0.3443 - val_loss: 1.5250 - val_acc: 0.4551
Epoch 2/40
40000/40000 [=====] - 382s - loss: 1.3506 - acc: 0.5182 - val_loss: 1.1998 - val_acc: 0.5714
```

In the following screenshot, we will see the accuracy reached after 40 iterations:

```
Epoch 39/40
40000/40000 [=====] - 348s - loss: 0.5497 - acc: 0.8246 - val_loss: 0.8669 - val_acc: 0.7811
Epoch 40/40
40000/40000 [=====] - 346s - loss: 0.5447 - acc: 0.8280 - val_loss: 0.7910 - val_acc: 0.7816
Testing...
10000/10000 [=====] - 41s
('Test score:', 0.79934534568786619)
('Test accuracy:', 0.7692999999999998)
['acc', 'loss', 'val_acc', 'val_loss']
```

So we have an improvement of 10.5% with respect to the previous simpler deeper network. For the sake of completeness, let us also report the accuracy and loss during training, shown as follows:



Improving the CIFAR-10 performance with data augmentation

Another way to improve the performance is to generate more images for our training. The key intuition is that we can take the standard CIFAR training set and augment this set with multiple types of transformations including rotation, rescaling, horizontal/vertical flip, zooming, channel shift, and many more. Let us see the code:

```
from keras.preprocessing.image import ImageDataGenerator
from keras.datasets import cifar10
import numpy as np
NUM_TO_AUGMENT=5

#load dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

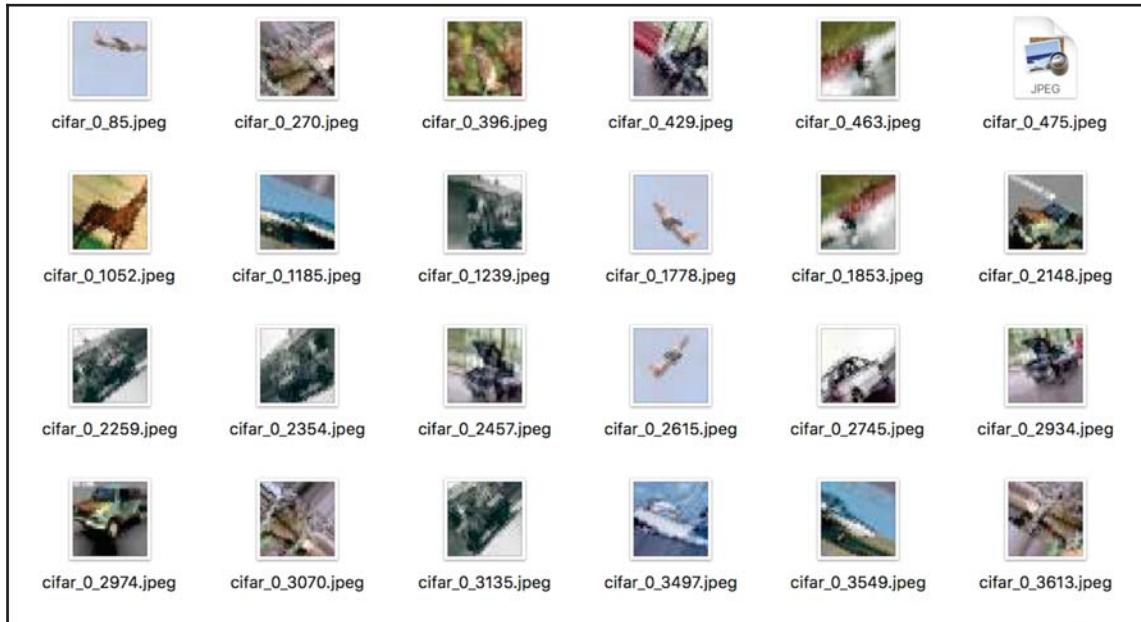
# augmenting
print("Augmenting training set images...")
```

```
datagen = ImageDataGenerator(  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    fill_mode='nearest')
```

The `rotation_range` is a value in degrees (0 - 180) for randomly rotating pictures. `width_shift` and `height_shift` are ranges for randomly translating pictures vertically or horizontally. `zoom_range` is for randomly zooming pictures. `horizontal_flip` is for randomly flipping half of the images horizontally. `fill_mode` is the strategy used for filling in new pixels that can appear after a rotation or a shift:

```
xtas, ytas = [], []  
for i in range(X_train.shape[0]):  
    num_aug = 0  
    x = X_train[i] # (3, 32, 32)  
    x = x.reshape((1,) + x.shape) # (1, 3, 32, 32)  
    for x_aug in datagen.flow(x, batch_size=1,  
        save_to_dir='preview', save_prefix='cifar', save_format='jpeg'):  
        if num_aug >= NUM_TO_AUGMENT:  
            break  
        xtas.append(x_aug[0])  
        num_aug += 1
```

After augmentation, we will have generated many more training images starting from the standard CIFAR-10 set:



Now we can apply this intuition directly for training. Using the same ConvNet defined previously we simply generate more augmented images and then we train. For efficiency, the generator runs in parallel to the model. This allows an image augmentation on the CPU and in parallel to training on the GPU. Here is the code:

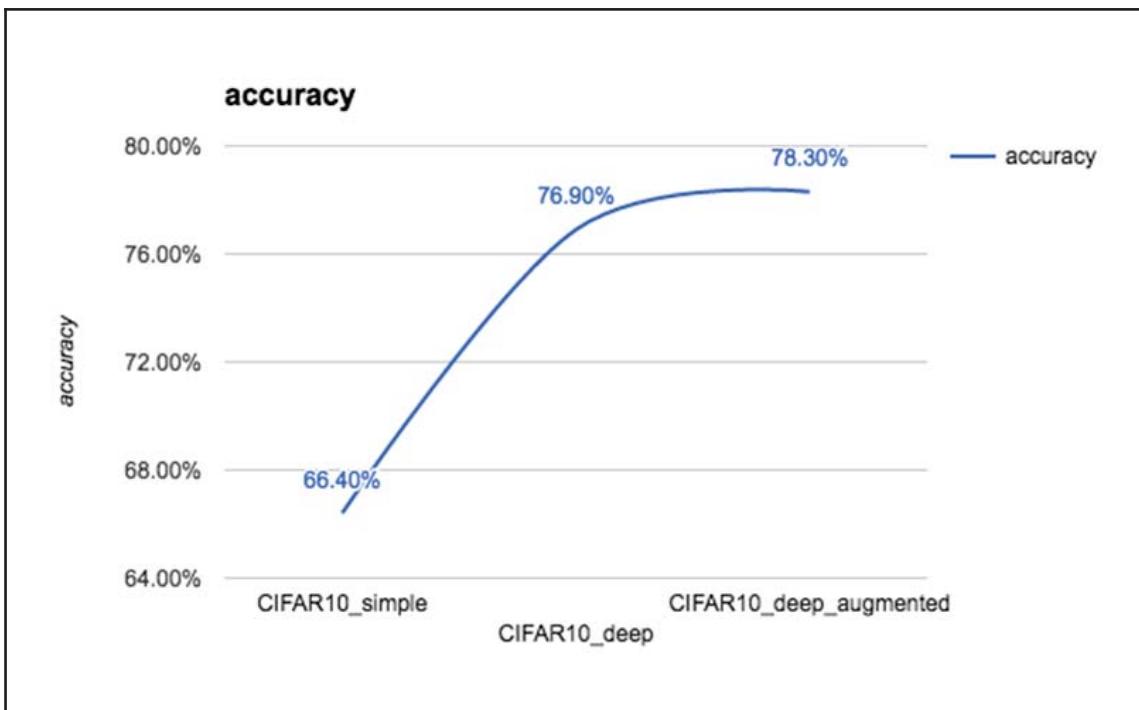
```
#fit the dataset
datagen.fit(X_train)

# train
history = model.fit_generator(datagen.flow(X_train, Y_train,
batch_size=BATCH_SIZE), samples_per_epoch=X_train.shape[0],
epochs=N_EPOCH, verbose=VERBOSE)
score = model.evaluate(X_test, Y_test,
batch_size=BATCH_SIZE, verbose=VERBOSE)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

Each iteration is now more expensive because we have more training data. So let us run for 50 iterations only and see that we reach an accuracy of 78.3%:

```
Epoch 46/50
50000/50000 [=====] - 405s - loss: 0.8288 - acc: 0.7297
Epoch 47/50
50000/50000 [=====] - 424s - loss: 0.8349 - acc: 0.7303
Epoch 48/50
50000/50000 [=====] - 408s - loss: 0.8319 - acc: 0.7295
Epoch 49/50
50000/50000 [=====] - 403s - loss: 0.8386 - acc: 0.7281
Epoch 50/50
50000/50000 [=====] - 398s - loss: 0.8394 - acc: 0.7267
Testing...
10000/10000 [=====] - 42s
('Test score:', 0.73110332846641546)
('Test accuracy:', 0.7836999999999995)
['acc', 'loss']
```

The results obtained during our experiments are summarized in the following graph:



A list of state-of-the-art results for CIFAR-10 is available at: http://rodrigob.github.io/re_we_there_yet/build/classification_datasets_results.html. As of January, 2017, the best result has an accuracy of 96.53%.

Predicting with CIFAR-10

Now let us suppose that we want to use the deep learning model we just trained for CIFAR-10 for a bulk evaluation of images. Since we saved the model and the weights, we do not need to train every time:

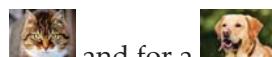
```
import numpy as np
import scipy.misc
from keras.models import model_from_json
from keras.optimizers import SGD

#load model
model_architecture = 'cifar10_architecture.json'
model_weights = 'cifar10_weights.h5'
model = model_from_json(open(model_architecture).read())
model.load_weights(model_weights)

#load images
img_names = ['cat-standing.jpg', 'dog.jpg']
imgs = [np.transpose(scipy.misc.imresize(scipy.misc.imread(img_name), (32, 32)), (1, 0, 2)).astype('float32')
for img_name in img_names]
imgs = np.array(imgs) / 255

# train
optim = SGD()
model.compile(loss='categorical_crossentropy', optimizer=optim,
metrics=['accuracy'])

# predict
predictions = model.predict_classes(imgs)
print(predictions)
```



Now let us get the prediction for a and for a .

We get categories 3 (cat) and 5 (dog) as output, as expected:

```
gulli-macbookpro:code gulli$ python keras_EvaluateCIFAR10.py
Using TensorFlow backend.
2/2 [=====] - 0s
[3 5]
gulli-macbookpro:code gulli$
```

Very deep convolutional networks for large-scale image recognition

In 2014, an interesting contribution for image recognition was presented (for more information refer to: *Very Deep Convolutional Networks for Large-Scale Image Recognition*, by K. Simonyan and A. Zisserman, 2014). The paper shows that, *a significant improvement on the prior-art configurations can be achieved by pushing the depth to 16-19 weight layers*. One model in the paper denoted as D or VGG-16 has 16 deep layers. An implementation in Java Caffe (<http://caffe.berkeleyvision.org/>) has been used for training the model on the ImageNet ILSVRC-2012 (<http://image-net.org/challenges/LSVRC/2012/>) dataset, which includes images of 1,000 classes and is split into three sets: training (1.3 million images), validation (50,000 images), and testing (100,000 images). Each image is (224 x 224) on three channels. The model achieves 7.5% top 5 error on ILSVRC-2012-val and 7.4% top 5 error on ILSVRC-2012-test.

According to the ImageNet site:

The goal of this competition is to estimate the content of photographs for the purpose of retrieval and automatic annotation using a subset of the large hand-labeled ImageNet dataset (10 million labeled images depicting 10,000 + object categories) as training. Test images will be presented with no initial annotation—no segmentation or labels—and algorithms will have to produce labelings specifying what objects are present in the images.

The weights learned by the model implemented in Caffe have been directly converted in Keras (for more information refer to: <https://gist.github.com/baraldilorenzo/07d7802847aaad0a35d3>) and can be used for preloading into the Keras model, which is implemented next as described in the paper:

```
from keras.models import Sequential
from keras.layers.core import Flatten, Dense, Dropout
from keras.layers.convolutional import Conv2D, MaxPooling2D, ZeroPadding2D
from keras.optimizers import SGD
import cv2, numpy as np

# define a VGG16 network
def VGG_16(weights_path=None):
    model = Sequential()
    model.add(ZeroPadding2D((1,1),input_shape=(3,224,224)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2,2), strides=(2,2)))
    model.add(ZeroPadding2D((1,1)))
```

```
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))
model.add(ZeroPadding2D((1,1)))
model.add(Conv2D(256, (3, 3), activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Conv2D(512, (3, 3), activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Flatten())
#top layer of the VGG net
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1000, activation='softmax'))
if weights_path:
    model.load_weights(weights_path)
return model
```

Recognizing cats with a VGG-16 net



Now let us test the image of a :

```
im = cv2.resize(cv2.imread('cat.jpg'), (224, 224)).astype(np.float32)
im = im.transpose((2,0,1))
im = np.expand_dims(im, axis=0)

# Test pretrained model
model = VGG_16('/Users/gulli/Keras/codeBook/code/data/vgg16_weights.h5')
```

```
optimizer = SGD()
model.compile(optimizer=optimizer, loss='categorical_crossentropy')
out = model.predict(im)
print np.argmax(out)
```

When the code is executed, the class 285 is returned, which corresponds (for more information refer to: <https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a>) to Egyptian cat:



```
code -- bash -- 108x5
~/Keras/codeBook/code -- bash ...
...
~/Keras/codeBook/code -- bash + ...

set properly.
gulli-macbookpro:code gulli$ python keras_VGG16.py
Using TensorFlow backend.
285
gulli-macbookpro:code gulli$
```

Utilizing Keras built-in VGG-16 net module

Keras applications are pre-built and pre-trained deep learning models. Weights are downloaded automatically when instantiating a model and stored at `~/.keras/models/`. Using built-in code is very easy:

```
from keras.models import Model
from keras.preprocessing import image
from keras.optimizers import SGD
from keras.applications.vgg16 import VGG16
import matplotlib.pyplot as plt
import numpy as np
import cv2

# prebuild model with pre-trained weights on imagenet
model = VGG16(weights='imagenet', include_top=True)
sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(optimizer=sgd, loss='categorical_crossentropy')

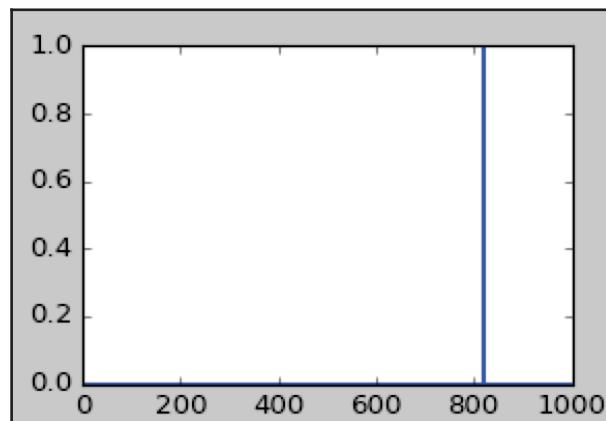
# resize into VGG16 trained images' format
im = cv2.resize(cv2.imread('steam-locomotive.jpg'), (224, 224))
im = np.expand_dims(im, axis=0)

# predict
out = model.predict(im)
plt.plot(out.ravel())
plt.show()
print np.argmax(out)
#this should print 820 for steaming train
```

Now, let us consider a train:



It's like the ones my grandfather drove. If we run the code, we get result 820, which is the image net code for *steaming train*. Equally important is the fact that all the other classes have very weak support, as shown in the following graph:



To conclude this section, note that VGG-16 is only one of the modules that are pre-built in Keras. A full list of pre-trained Keras models is available at: <https://keras.io/applications/>.

Recycling pre-built deep learning models for extracting features

One very simple idea is to use VGG-16 and, more generally, DCNN, for feature extraction. This code implements the idea by extracting features from a specific layer:

```
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np

# pre-built and pre-trained deep learning VGG16 model
base_model = VGG16(weights='imagenet', include_top=True)
for i, layer in enumerate(base_model.layers):
    print (i, layer.name, layer.output_shape)

# extract features from block4_pool block
model =
Model(input=base_model.input,
output=base_model.get_layer('block4_pool').output)
img_path = 'cat.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

# get the features from this block
features = model.predict(x)
```

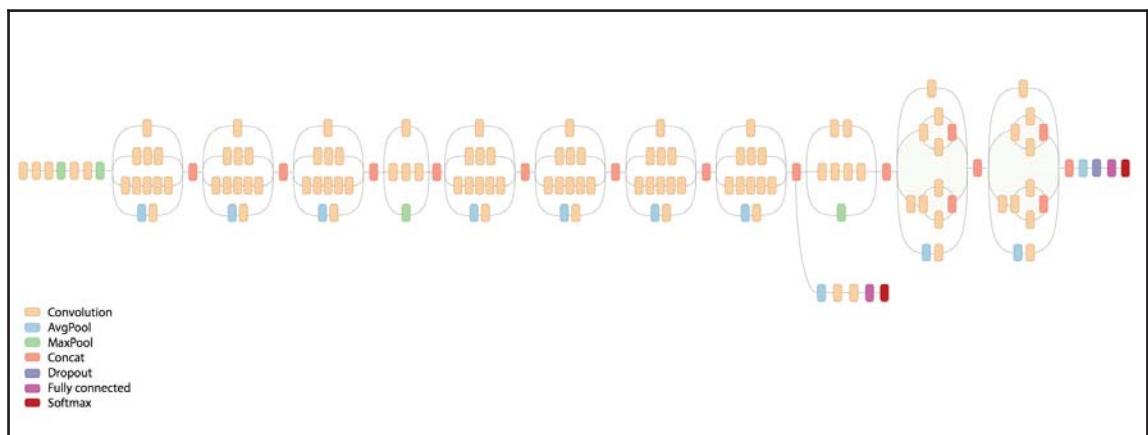
Now you might wonder why we want to extract the features from an intermediate layer in a DCNN. The key intuition is that, as the network learns to classify images into categories, each layer learns to identify the features that are necessary to do the final classification. Lower layers identify lower order features such as color and edges, and higher layers compose these lower order feature into higher order features such as shapes or objects. Hence the intermediate layer has the capability to extract important features from an image, and these features are more likely to help in different kinds of classification. This has multiple advantages. First, we can rely on publicly available large-scale training and transfer this learning to novel domains. Second, we can save time for expensive large training. Third, we can provide reasonable solutions even when we don't have a large number of training examples for our domain. We also get a good starting network shape for the task at hand, instead of guessing it.

Very deep inception-v3 net used for transfer learning

Transfer learning is a very powerful deep learning technique which has more applications in different domains. The intuition is very simple and can be explained with an analogy. Suppose you want to learn a new language, say Spanish; then it could be useful to start from what you already know in a different language, say English.

Following this line of thinking, computer vision researchers now commonly use pre-trained CNNs to generate representations for novel tasks, where the dataset may not be large enough to train an entire CNN from scratch. Another common tactic is to take the pre-trained ImageNet network and then to fine-tune the entire network to the novel task.

Inception-v3 net is a very deep ConvNet developed by Google. Keras implements the full network described in the following diagram and it comes pre-trained on ImageNet. The default input size for this model is 299 x 299 on three channels:



This skeleton example is inspired by a scheme available at: <https://keras.io/applications/>. We suppose to have a training dataset D in a domain, different from ImageNet. D has 1,024 features in input and 200 categories in output. Let us see a code fragment:

```
from keras.applications.inception_v3 import InceptionV3
from keras.preprocessing import image
from keras.models import Model
from keras.layers import Dense, GlobalAveragePooling2D
from keras import backend as K

# create the base pre-trained model
base_model = InceptionV3(weights='imagenet', include_top=False)
```

We use a trained inception-v3; we do not include the top model because we want to fine-tune on D . The top level is a dense layer with 1,024 inputs and where the last output level is a softmax dense layer with 200 classes of output. $x = \text{GlobalAveragePooling2D}()(x)$ is used to convert the input to the correct shape for the dense layer to handle. In fact, `base_model.output` tensor has the shape (*samples, channels, rows, cols*) for `dim_ordering="th"` or (*samples, rows, cols, channels*) for `dim_ordering="tf"` but `dense` needs them as (*samples, channels*) and `GlobalAveragePooling2D` averages across (*rows, cols*). So if you look at the last four layers (where `include_top=True`), you see these shapes:

```
# layer.name, layer.input_shape, layer.output_shape
('mixed10', [(None, 8, 8, 320), (None, 8, 8, 768), (None, 8, 8, 768),
(None, 8, 8, 192)], (None, 8, 8, 2048))
('avg_pool', (None, 8, 8, 2048), (None, 1, 1, 2048))
('flatten', (None, 1, 1, 2048), (None, 2048))
('predictions', (None, 2048), (None, 1000))
```

When you do `include_top=False`, you are removing the last three layers and exposing the `mixed10` layer, so the `GlobalAveragePooling2D` layer converts the (*None, 8, 8, 2048*) to (*None, 2048*), where each element in the (*None, 2048*) tensor is the average value for each corresponding (8, 8) subtensor in the (*None, 8, 8, 2048*) tensor:

```
# add a global spatial average pooling layer
x = base_model.output
x = GlobalAveragePooling2D()(x) # let's add a fully-connected layer as first
layer
x = Dense(1024, activation='relu')(x) # and a logistic layer with 200
classes as last layer
predictions = Dense(200, activation='softmax')(x) # model to train
model = Model(input=base_model.input, output=predictions)
```

All the convolutional levels are pre-trained, so we freeze them during the training of the full model:

```
# that is, freeze all convolutional InceptionV3 layers
for layer in base_model.layers: layer.trainable = False
```

The model is then compiled and trained for a few epochs so that the top layers are trained:

```
# compile the model (should be done *after* setting layers to non-
trainable)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# train the model on the new data for a few epochs model.fit_generator(...)
```

Then we freeze the top layers in inception and fine-tune some inception layer. In this example, we decide to freeze the first 172 layers (an hyperparameter to tune):

```
# we chose to train the top 2 inception blocks, that is, we will freeze
# the first 172 layers and unfreeze the rest:
for layer in
model.layers[:172]: layer.trainable = False
for layer in
model.layers[172:]: layer.trainable = True
```

The model is then recompiled for fine-tune optimization. We need to recompile the model for these modifications to take effect:

```
# we use SGD with a low learning rate
from keras.optimizers
import SGD
model.compile(optimizer=SGD(lr=0.0001, momentum=0.9),
loss='categorical_crossentropy')

# we train our model again (this time fine-tuning the top 2 inception
blocks)
# alongside the top Dense layers
model.fit_generator(...)
```

Now we have a new deep network that reuses the standard Inception-v3 network, but it is trained on a new domain D via transfer learning. Of course, there are many parameters to fine-tune for achieving good accuracy. However, we are now reusing a very large pre-trained network as a starting point via transfer learning. In doing so, we can save the need to train on our machines by reusing what is already available in Keras.

Summary

In this chapter, we learned how to use Deep Learning ConvNets for recognizing MNIST handwritten characters with high accuracy. Then we used the CIFAR 10 dataset to build a deep learning classifier in 10 categories, and the ImageNet datasets to build an accurate classifier in 1,000 categories. In addition, we investigated how to use large deep learning networks such as VGG16 and very deep networks such as InceptionV3. The chapter concluded with a discussion on transfer learning in order to adapt pre-built models trained on large datasets so that they can work well on a new domain.

In the next chapter, we will introduce generative adversarial networks used to reproduce synthetic data that looks like data generated by humans; and we will present WaveNet, a deep neural network used for reproducing human voice and musical instruments with high quality.

4

Generative Adversarial Networks and WaveNet

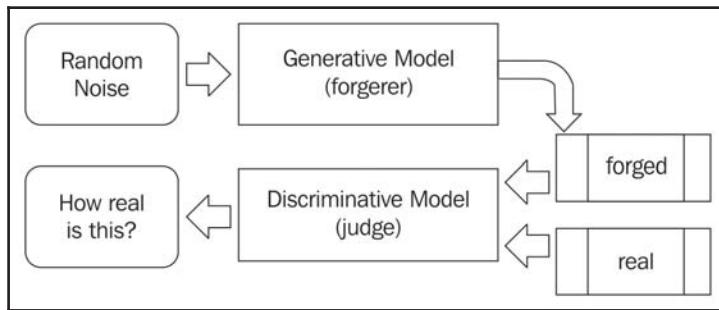
In this chapter, we will discuss **generative adversarial networks (GANs)** and WaveNets. GANs have been defined as *the most interesting idea in the last 10 years in ML* (<https://www.quora.com/What-are-some-recent-and-potentially-upcoming-breakthroughs-in-deep-learning>) by Yann LeCun, one of the fathers of deep learning. GANs are able to learn how to reproduce synthetic data that looks real. For instance, computers can learn how to paint and create realistic images. The idea was originally proposed by Ian Goodfellow (for more information refer to: *NIPS 2016 Tutorial: Generative Adversarial Networks*, by I. Goodfellow, 2016); he was worked with the University of Montreal, Google Brain, and recently OpenAI (<https://openai.com/>). WaveNet is a deep generative network proposed by Google DeepMind to teach computers how to reproduce human voices and musical instruments, both with impressive quality.

In this chapter, we will cover the following topics:

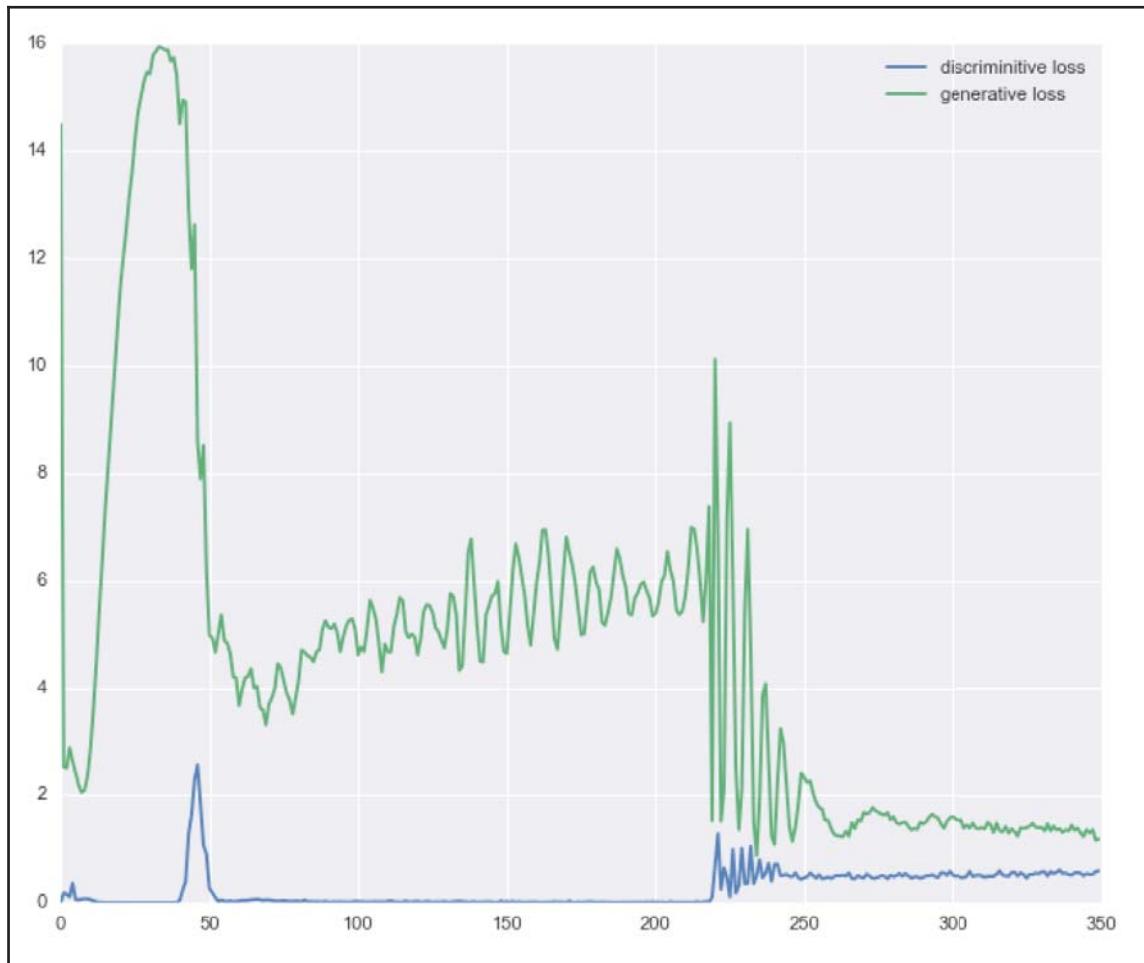
- What is GAN?
- Deep convolutional GAN
- Applications of GAN

What is a GAN?

The key intuition of GAN can be easily considered as analogous to *art forgery*, which is the process of creating works of art (<https://en.wikipedia.org/wiki/Art>) that are falsely credited to other, usually more famous, artists. GANs train two neural nets simultaneously, as shown in the next diagram. The generator $G(Z)$ makes the forgery, and the discriminator $D(Y)$ can judge how realistic the reproductions based on its observations of authentic pieces of arts and copies are. $D(Y)$ takes an input, Y , (for instance, an image) and expresses a vote to judge how real the input is--in general, a value close to zero denotes *real* and a value close to one denotes *forgery*. $G(Z)$ takes an input from a random noise, Z , and trains itself to fool D into thinking that whatever $G(Z)$ produces is real. So, the goal of training the discriminator $D(Y)$ is to maximize $D(Y)$ for every image from the true data distribution, and to minimize $D(Y)$ for every image not from the true data distribution. So, G and D play an opposite game; hence the name *adversarial training*. Note that we train G and D in an alternating manner, where each of their objectives is expressed as a loss function optimized via a gradient descent. The generative model learns how to forge more successfully, and the discriminative model learns how to recognize forgery more successfully. The discriminator network (usually a standard convolutional neural network) tries to classify whether an input image is real or generated. The important new idea is to backpropagate through both the discriminator and the generator to adjust the generator's parameters in such a way that the generator can learn how to fool the the discriminator for an increasing number of situations. At the end, the generator will learn how to produce forged images that are indistinguishable from real ones:



Of course, GANs require finding the equilibrium in a game with two players. For effective learning it is required that if a player successfully moves downhill in a round of updates, the same update must move the other player downhill too. Think about it! If the forger learns how to fool the judge on every occasion, then the forger himself has nothing more to learn. Sometimes the two players eventually reach an equilibrium, but this is not always guaranteed and the two players can continue playing for a long time. An example of learning from both sides has been provided in the following graph:



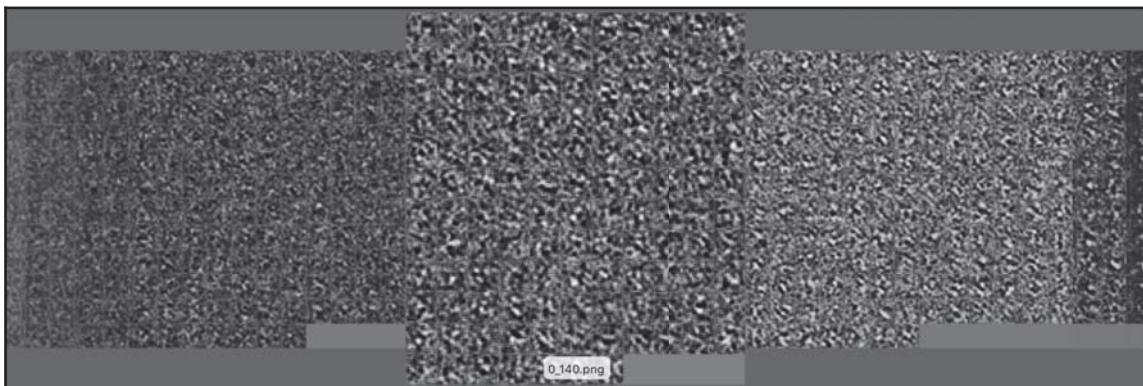
Some GAN applications

We have seen that the generator learns how to forge data. This means that it learns how to create new synthetic data, which is created by the network, that looks real and like it was created by humans. Before going into details of some GAN code, I'd like to share the results of a recent paper: *StackGAN: Text to Photo-Realistic Image Synthesis with Stacked Generative Adversarial Networks*, by Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaolei Huang, Xiaogang Wang, and Dimitris Metaxas (the code is available online at: <https://github.com/hanzhanggit/StackGAN>).

Here, a GAN has been used to synthesize forged images starting from a text description. The results are impressive. The first column is the real image in the test set, and the rest of the columns contain images generated from the same text description by **Stage-I** and **Stage-II** of StackGAN. More examples are available on YouTube (<https://www.youtube.com/watch?v=SuRyL5vhCIM&feature=youtu.be>):



Now let us see how a GAN can learn to *forge* the MNIST dataset. In this case, there is a combination of GAN and ConvNets (for more information refer to: *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, by A. Radford, L. Metz, and S. Chintala, arXiv: 1511.06434, 2015) used for the generator and the discriminator networks. At the beginning, the generator creates nothing understandable, but after a few iterations, synthetic forged numbers are progressively clearer and clearer. In the following image, the panels are ordered by increasing training epochs, and you can see the quality improving among panels:



The following image represents the forged handwritten numbers as the number of iterations increases:



The following image represents the forged handwritten numbers at the hand of computation. The results are virtually indistinguishable from the original:



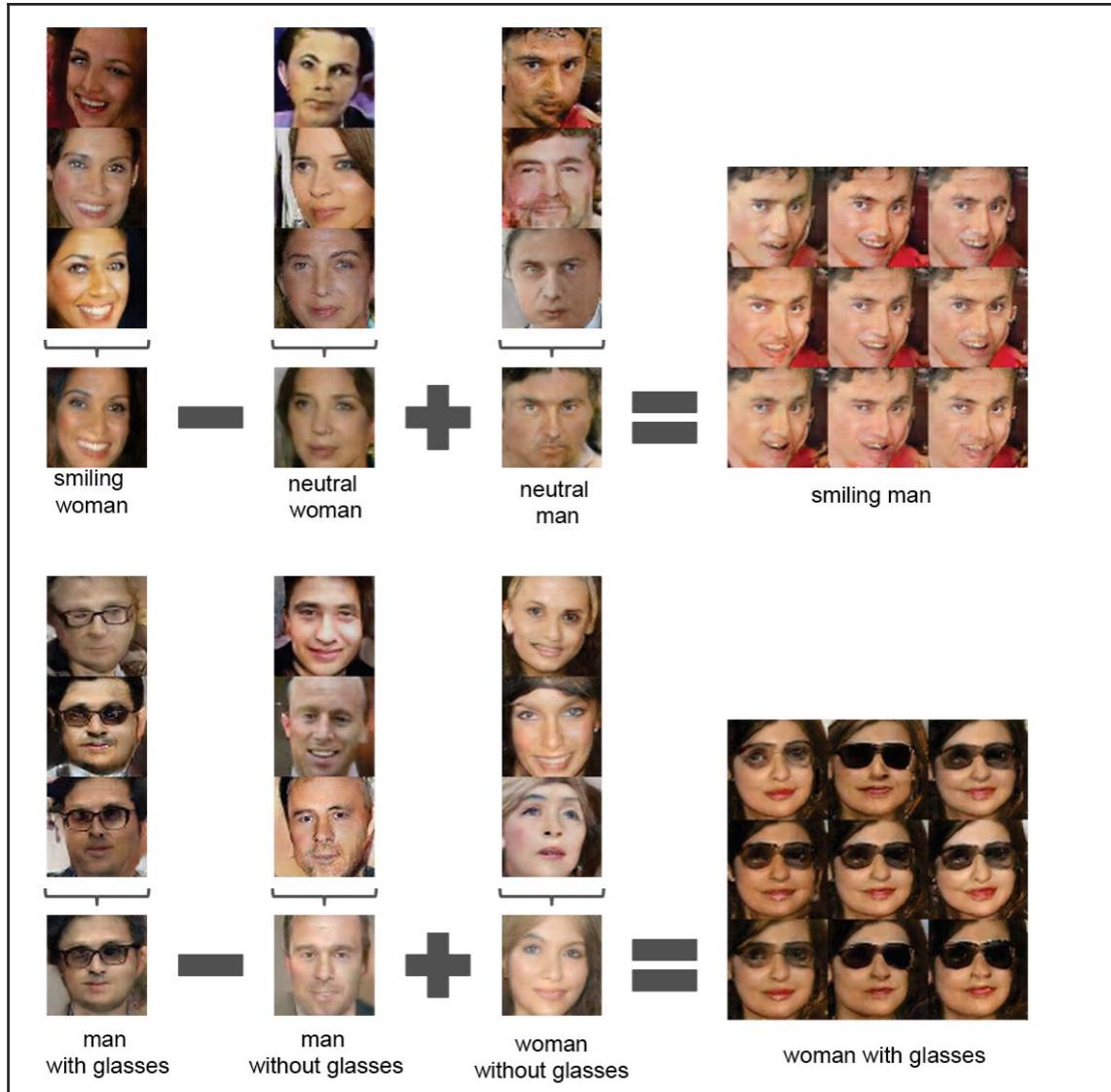
One of the coolest uses of GAN is arithmetic on faces in the generator's vector Z. In other words, if we stay in the space of synthetic forged images, it is possible to see things like this:

$$[\text{smiling woman}] - [\text{neutral woman}] + [\text{neutral man}] = [\text{smiling man}]$$

Or like this:

$$[\text{man with glasses}] - [\text{man without glasses}] + [\text{woman without glasses}] = [\text{woman with glasses}]$$

The next image is taken from the article, *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, by A. Radford, L. Metz, and S. Chintala, arXiv: 1511.06434, November, 2015:



Deep convolutional generative adversarial networks

The **deep convolutional generative adversarial networks (DCGAN)** are introduced in the paper: *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, by A. Radford, L. Metz, and S. Chintala, arXiv: 1511.06434, 2015. The generator uses a 100-dimensional, uniform distribution space, Z , which is then projected into a smaller space by a series of vis-a-vis convolution operations. An example is shown in the following figure:

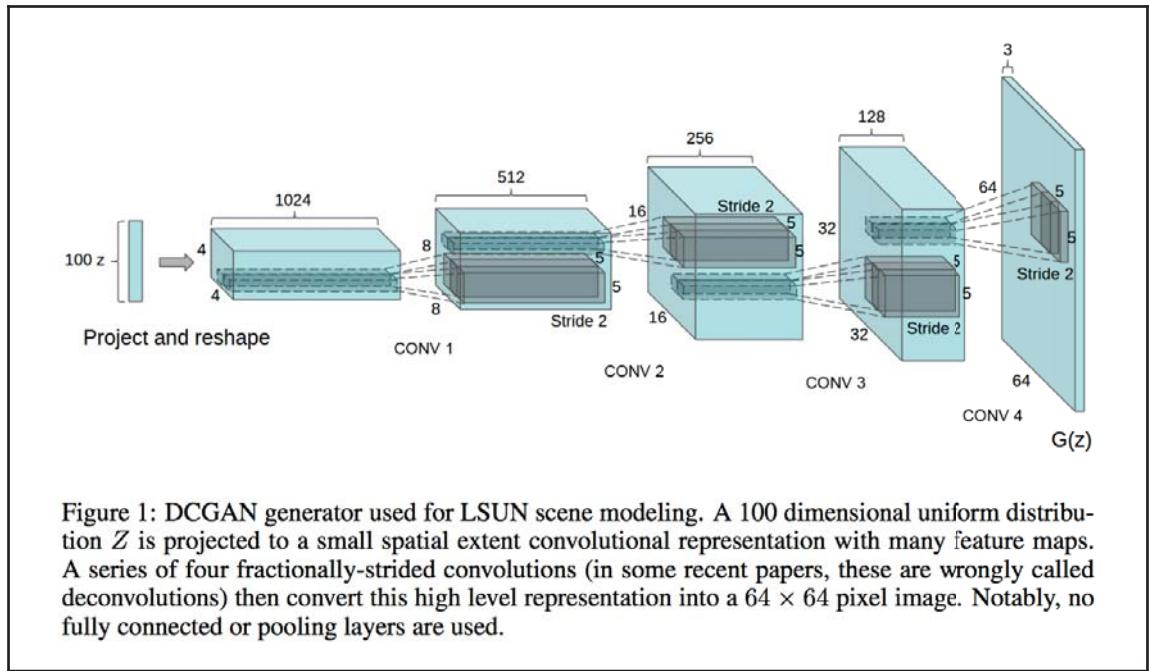


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a 64×64 pixel image. Notably, no fully connected or pooling layers are used.

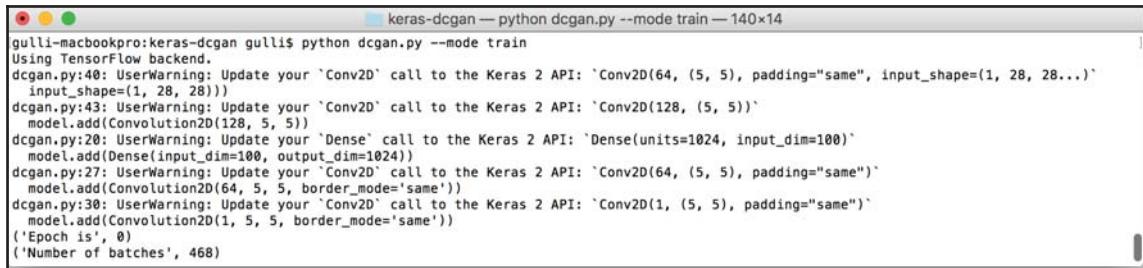
A DCGAN generator can be described by the following Keras code; it is also described by one implementation, available at: <https://github.com/jacobgil/keras-dcgan>:

```
def generator_model():
    model = Sequential()
    model.add(Dense(input_dim=100, output_dim=1024))
    model.add(Activation('tanh'))
    model.add(Dense(128*7*7))
    model.add(BatchNormalization())
    model.add(Activation('tanh'))
    model.add(Reshape((128, 7, 7), input_shape=(128*7*7,)))

```

```
model.add(UpSampling2D(size=(2, 2)))
model.add(Convolution2D(64, 5, 5, border_mode='same'))
model.add(Activation('tanh'))
model.add(UpSampling2D(size=(2, 2)))
model.add(Convolution2D(1, 5, 5, border_mode='same'))
model.add(Activation('tanh'))
return model
```

Note that the code runs with Keras 1.x syntax. However, it is possible to run it with Keras 2.0 thanks to the Keras legacy interfaces. In this case a few warnings are reported as shown in the following figure:



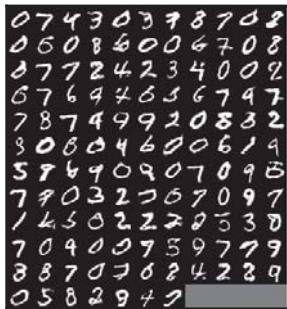
```
keras-dcgan — python dcgan.py --mode train — 140x14
gulli-macbookpro:keras-dcgan gulli$ python dcgan.py --mode train
Using TensorFlow backend.
dcgan.py:40: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(64, (5, 5), padding="same", input_shape=(1, 28, 28...))` →
dcgan.py:43: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(128, (5, 5))` →
    model.add(Convolution2D(128, 5, 5))
dcgan.py:20: UserWarning: Update your `Dense` call to the Keras 2 API: `Dense(units=1024, input_dim=100)` →
    model.add(Dense(input_dim=100, output_dim=1024))
dcgan.py:27: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(64, (5, 5), padding="same")` →
    model.add(Convolution2D(64, 5, 5, border_mode='same'))
dcgan.py:30: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(1, (5, 5), padding="same")` →
    model.add(Convolution2D(1, 5, 5, border_mode='same'))
('Epoch is', 0)
('Number of batches', 468)
```

Now let's see the code. The first dense layer takes a vector of 100 dimensions as input and it produces 1,024 dimensions with the activation function `tanh` as the output. We assume that the input is sampled from a uniform distribution in $[-1, 1]$. The next dense layer produces data of $128 \times 7 \times 7$ in the output using batch normalization (for more information refer to *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, by S. Ioffe and C. Szegedy, arXiv: 1502.03167, 2014), a technique that can help stabilize learning by normalizing the input to each unit to zero mean and unit variance. Batch normalization has been empirically proven to accelerate the training in many situations, reduce the problems of poor initialization, and more generally produce more accurate results. There is also a `Reshape()` module that produces data of $127 \times 7 \times 7$ (127 channels, 7 width, and 7 height), `dim_ordering` to `tf`, and a `UpSampling()` module that produces a repetition of each one into a 2×2 square. After that, we have a convolutional layer producing 64 filters on 5×5 convolutional kernels with the activation `tanh`, followed by a new `UpSampling()` and a final convolution with one filter, and on 5×5 convolutional kernels with the activation `tanh`. Notice that this ConvNet has no pooling operations. The discriminator can be described with the following code:

```
def discriminator_model():
    model = Sequential()
    model.add(Convolution2D(64, 5, 5, border_mode='same',
                           input_shape=(1, 28, 28)))
    model.add(Activation('tanh'))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Convolution2D(128, 5, 5))
model.add(Activation('tanh'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(1024))
model.add(Activation('tanh'))
model.add(Dense(1))
model.add(Activation('sigmoid'))
return model
```

The code takes a standard MNIST image with the shape `(1, 28, 28)` and applies a convolution with 64 filters of size 5×5 with `tanh` as the activation function. This is followed by a max-pooling operation of size 2×2 and by a further convolution max-pooling operation. The last two stages are dense, with the final one being the prediction for forgery, which consists of only one neuron with a `sigmoid` activation function. For a chosen number of epochs, the generator and discriminator are in turn trained by using `binary_crossentropy` as loss function. At each epoch, the generator makes a number of predictions (for example, it creates forged MNIST images) and the discriminator tries to learn after mixing the prediction with real MNIST images. After 32 epochs, the generator learns to forge this set of handwritten numbers. No one has programmed the machine to write but it has learned how to write numbers that are indistinguishable from the ones written by humans. Note that training GANs could be very difficult because it is necessary to find the equilibrium between two players. If you are interested in this topic, I'd advise you to have a look at a series of tricks collected by practitioners (<https://github.com/soumith/ganhacks>):



Keras adversarial GANs for forging MNIST

Keras adversarial (<https://github.com/bstriner/keras-adversarial>) is an open source Python package for building GANs developed by Ben Striner (<https://github.com/bstriner> and <https://github.com/bstriner/keras-adversarial/blob/master/LICENSE.txt>). Since Keras just recently moved to 2.0, I suggest downloading latest Keras adversarial package:

```
git clone --depth=50 --branch=master  
https://github.com/bstriner/keras-adversarial.git
```

And install setup.py:

```
python setup.py install
```

Note that compatibility with Keras 2.0 is tracked in this issue <https://github.com/bstriner/keras-adversarial/issues/11>.

If the generator G and the discriminator D are based on the same model, M , then they can be combined into an adversarial model; it uses the same input, M , but separates targets and metrics for G and D . The library has the following API call:

```
adversarial_model = AdversarialModel(base_model=M,  
player_params=[generator.trainable_weights,  
discriminator.trainable_weights],  
player_names=["generator", "discriminator"])
```

If the generator G and the discriminator D are based on the two different models, then it is possible to use this API call:

```
adversarial_model = AdversarialModel(player_models=[gan_g, gan_d],  
player_params=[generator.trainable_weights,  
discriminator.trainable_weights],  
player_names=["generator", "discriminator"])
```

Let's see an example of a computation with MNIST:

```
import matplotlib as mpl  
# This line allows mpl to run with no DISPLAY defined  
mpl.use('Agg')
```

Let us see the open source code (https://github.com/bstriner/keras-adversarial/blob/master/examples/example_gan_convolutional.py). Note that the code uses the syntax of Keras 1.x, but it also runs on the top of Keras 2.x thanks to a convenient set of utility functions contained in `legacy.py`. The code for `legacy.py` is reported in [Appendix, Conclusion, and is available at](#) https://github.com/bstriner/keras-adversarial/blob/master/keras_adversarial/legacy.py.

First, the open source example imports a number of modules. We have seen all of them previously, with the exception of LeakyReLU, a special version of ReLU that allows a small gradient when the unit is not active. Experimentally, it has been shown that LeakyReLU can improve the performance of GANs (for more information refer to: *Empirical Evaluation of Rectified Activations in Convolutional Network*, by B. Xu, N. Wang, T. Chen, and M. Li, arXiv:1505.00853, 2014) in a number of situations:

```
from keras.layers import Dense, Reshape, Flatten, Dropout, LeakyReLU,
Input, Activation, BatchNormalization
from keras.models import Sequential, Model
from keras.layers.convolutional import Convolution2D, UpSampling2D
from keras.optimizers import Adam
from keras.regularizers import l1, l1l2
from keras.datasets import mnist

import pandas as pd
import numpy as np
```

Then, specific modules for GANs are imported:

```
from keras_adversarial import AdversarialModel, ImageGridCallback,
simple_gan, gan_targets
from keras_adversarial import AdversarialOptimizerSimultaneous,
normal_latent_sampling, AdversarialOptimizerAlternating
from image_utils import dim_ordering_fix, dim_ordering_input,
dim_ordering_reshape, dim_ordering_unfix
```

Adversarial models train for multiplayer games. Given a base model with n targets and k players, create a model with $n*k$ targets, where each player optimizes loss on that player's targets. In addition, `simple_gan` generates a GAN with the given `gan_targets`. Note that in the library, the labels for generator and discriminator are opposite; intuitively, this is a standard practice for GANs:

```
def gan_targets(n):
"""
Standard training targets [generator_fake, generator_real,
discriminator_fake,
discriminator_real] = [1, 0, 0, 1]
:param n: number of samples
```

```
:return: array of targets
"""
generator_fake = np.ones((n, 1))
generator_real = np.zeros((n, 1))
discriminator_fake = np.zeros((n, 1))
discriminator_real = np.ones((n, 1))
return [generator_fake, generator_real, discriminator_fake,
discriminator_real]
```

The example defines the generator in a similar way to what we have seen previously. However, in this case, we use the functional syntax—each module in our pipeline is simply passed as input to the following module. So, the first module is dense, initialized by using `glorot_normal`. This initialization uses Gaussian noise scaled by the sum of the inputs plus outputs from the node. The same kind of initialization is used for all of the other modules. The `mode=2` parameter in `BatchNormlization` function produces feature-wise normalization based on per-batch statistics. Experimentally, this produces better results:

```
def model_generator():
    nch = 256
    g_input = Input(shape=[100])
    H = Dense(nch * 14 * 14, init='glorot_normal')(g_input)
    H = BatchNormalization(mode=2)(H)
    H = Activation('relu')(H)
    H = dim_ordering_reshape(nch, 14)(H)
    H = UpSampling2D(size=(2, 2))(H)
    H = Convolution2D(int(nch / 2), 3, 3, border_mode='same',
                      init='glorot_uniform')(H)
    H = BatchNormalization(mode=2, axis=1)(H)
    H = Activation('relu')(H)
    H = Convolution2D(int(nch / 4), 3, 3, border_mode='same',
                      init='glorot_uniform')(H)
    H = BatchNormalization(mode=2, axis=1)(H)
    H = Activation('relu')(H)
    H = Convolution2D(1, 1, 1, border_mode='same',
                      init='glorot_uniform')(H)
    g_V = Activation('sigmoid')(H)
    return Model(g_input, g_V)
```

The discriminator is very similar to the one defined previously in this chapter. The only major difference is the adoption of LeakyReLU:

```
def model_discriminator(input_shape=(1, 28, 28), dropout_rate=0.5):
    d_input = dim_ordering_input(input_shape, name="input_x")
    nch = 512
    H = Convolution2D(int(nch / 2), 5, 5, subsample=(2, 2),
                      border_mode='same', activation='relu')(d_input)
    H = LeakyReLU(0.2)(H)
```

```
H = Dropout(dropout_rate)(H)
H = Convolution2D(nch, 5, 5, subsample=(2, 2),
                  border_mode='same', activation='relu')(H)
H = LeakyReLU(0.2)(H)
H = Dropout(dropout_rate)(H)
H = Flatten()(H)
H = Dense(int(nch / 2))(H)
H = LeakyReLU(0.2)(H)
H = Dropout(dropout_rate)(H)
d_V = Dense(1, activation='sigmoid')(H)
return Model(d_input, d_V)
```

Then, two simple functions for loading and normalizing MNIST data are defined:

```
def mnist_process(x):
    x = x.astype(np.float32) / 255.0
    return x

def mnist_data():
    (xtrain, ytrain), (xtest, ytest) = mnist.load_data()
    return mnist_process(xtrain), mnist_process(xtest)
```

As a next step, the GAN is defined as a combination of generator and discriminator in a joint GAN model. Note that the weights are initialized with `normal_latent_sampling`, which samples from a normal Gaussian distribution:

```
if __name__ == "__main__":
    # z in R^100
    latent_dim = 100
    # x in R^{28x28}
    input_shape = (1, 28, 28)
    # generator (z -> x)
    generator = model_generator()
    # discriminator (x -> y)
    discriminator = model_discriminator(input_shape=input_shape)
    # gan (x -> yfake, yreal), z generated on GPU
    gan = simple_gan(generator, discriminator,
                      normal_latent_sampling(latent_dim,)))
    # print summary of models
    generator.summary()
    discriminator.summary()
    gan.summary()
```

After this, the example creates our GAN and it compiles the model trained using the Adam optimizer, with `binary_crossentropy` used as a loss function:

```
# build adversarial model
model = AdversarialModel(base_model=gan,
    player_params=[generator.trainable_weights,
    discriminator.trainable_weights],
    player_names=["generator", "discriminator"])
model.adversarial_compile(adversarial_optimizer=AdversarialOptimizerSimulta-
neous(),
    player_optimizers=[Adam(1e-4, decay=1e-4), Adam(1e-3, decay=1e-4)],
    loss='binary_crossentropy')
```

The generator for creating new images that look like real ones is defined. Each epoch will generate a new forged image during training that looks like the original:

```
def generator_sampler():
    zsamples = np.random.normal(size=(10 * 10, latent_dim))
    gen = dim_ordering_unfix(generator.predict(zsamples))
    return gen.reshape((10, 10, 28, 28))

generator_cb = ImageGridCallback(
    "output/gan_convolutional/epoch-{:03d}.png", generator_sampler)
xtrain, xtest = mnist_data()
xtrain = dim_ordering_fix(xtrain.reshape((-1, 1, 28, 28)))
xtest = dim_ordering_fix(xtest.reshape((-1, 1, 28, 28)))
y = gan_targets(xtrain.shape[0])
ytest = gan_targets(xtest.shape[0])
history = model.fit(x=xtrain, y=y,
    validation_data=(xtest, ytest), callbacks=[generator_cb], nb_epoch=100,
    batch_size=32)
df = pd.DataFrame(history.history)
df.to_csv("output/gan_convolutional/history.csv")
generator.save("output/gan_convolutional/generator.h5")
discriminator.save("output/gan_convolutional/discriminator.h5")
```

Note that `dim_ordering_unfix` is utility function for supporting different image ordering defined in `image_utils.py`, as follows:

```
def dim_ordering_fix(x):
    if K.image_dim_ordering() == 'th':
        return x
    else:
        return np.transpose(x, (0, 2, 3, 1))
```

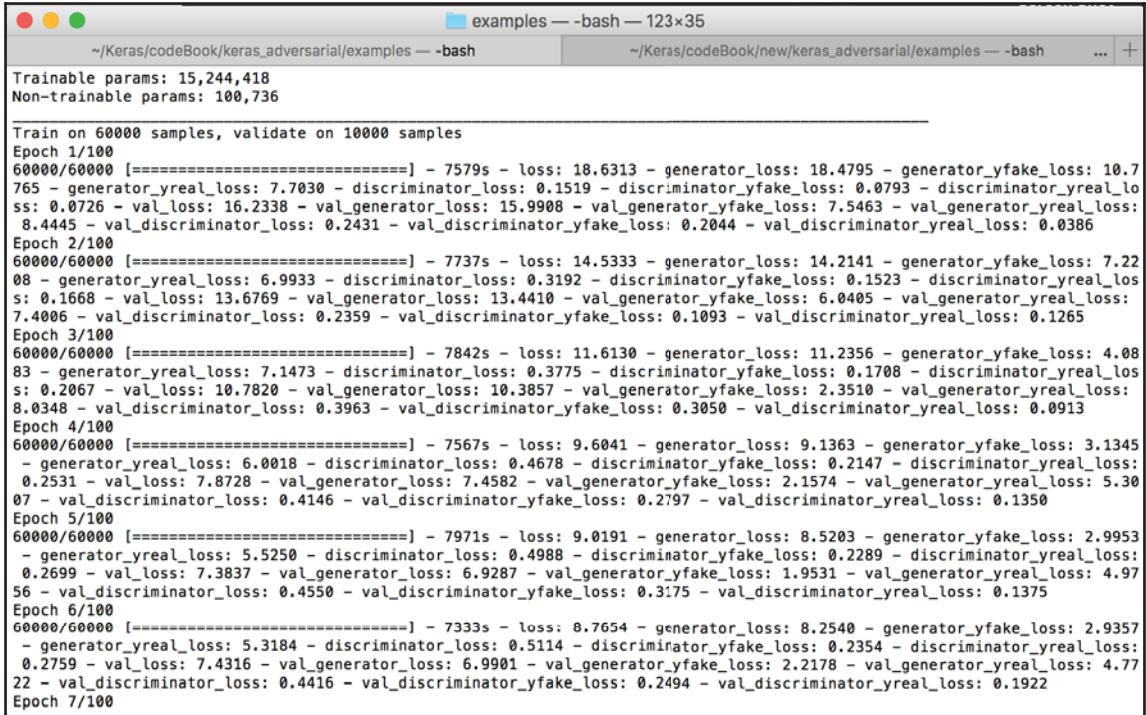
Now let's run the code and see the loss for the generator and discriminator. In the following screenshot, we see a dump of the networks for the discriminator and the generator:

```
examples — python example_gan_convolutional.py — 140x75
...ook/keras-dcgan — python dcgan.py --mode train ... ~/Keras/codeBook/keras-dcgan — bash .../examples — python example_gan_convolutional.py ...
gulli-macbookpro:examples gulli$ python example_gan_convolutional.py
Using TensorFlow backend.
~/Users/gulli/miniconda2/lib/python2.7/site-packages/keras_adversarial-0.0.3-py2.7.egg/keras_adversarial/legacy.py:48: UserWarning: Update your 'Conv2D' call to the Keras 2 API: 'Conv2D(256, (5, 5), padding='same', strides=(2, 2), activation='relu', kernel_initializer='glorot_uniform', kernel_regularizer=None)'
~/Users/gulli/miniconda2/lib/python2.7/site-packages/keras_adversarial-0.0.3-py2.7.egg/keras_adversarial/legacy.py:48: UserWarning: Update your 'Conv2D' call to the Keras 2 API: 'Conv2D(512, (5, 5), padding='same', strides=(2, 2), activation='relu', kernel_initializer='glorot_uniform', kernel_regularizer=None)'

Layer (type)          Output Shape         Param #
=====
input_1 (InputLayer) (None, 100)           0
dense_1 (Dense)      (None, 50176)        5067776
batch_normalization_1 (Batch (None, 50176) 200704
activation_1 (Activation) (None, 50176)    0
reshape_1 (Reshape)   (None, 14, 14, 256) 0
up_sampling2d_1 (UpSampling2 (None, 28, 28, 256) 0
conv2d_1 (Conv2D)    (None, 28, 28, 128) 295040
batch_normalization_2 (Batch (None, 28, 28, 128) 112
activation_2 (Activation) (None, 28, 28, 128) 0
conv2d_2 (Conv2D)    (None, 28, 28, 64)   73792
batch_normalization_3 (Batch (None, 28, 28, 64) 112
activation_3 (Activation) (None, 28, 28, 64) 0
conv2d_3 (Conv2D)    (None, 28, 28, 1)    65
activation_4 (Activation) (None, 28, 28, 1) 0
=====
Total params: 5,637,601.0
Trainable params: 5,537,137.0
Non-trainable params: 100,464.0

Layer (type)          Output Shape         Param #
=====
input_x (InputLayer) (None, 28, 28, 1) 0
conv2d_4 (Conv2D)    (None, 14, 14, 256) 6656
leaky_re_lu_1 (LeakyReLU) (None, 14, 14, 256) 0
dropout_1 (Dropout) (None, 14, 14, 256) 0
conv2d_5 (Conv2D)    (None, 7, 7, 512) 3277312
leaky_re_lu_2 (LeakyReLU) (None, 7, 7, 512) 0
dropout_2 (Dropout) (None, 7, 7, 512) 0
flatten_1 (Flatten) (None, 25088) 0
dense_2 (Dense)     (None, 256) 6422784
leaky_re_lu_3 (LeakyReLU) (None, 256) 0
dropout_3 (Dropout) (None, 256) 0
dense_3 (Dense)     (None, 1) 257
=====
Total params: 9,707,009.0
Trainable params: 9,707,009.0
Non-trainable params: 0.0
```

The following screenshot shows the number of sample used for training and for validation:



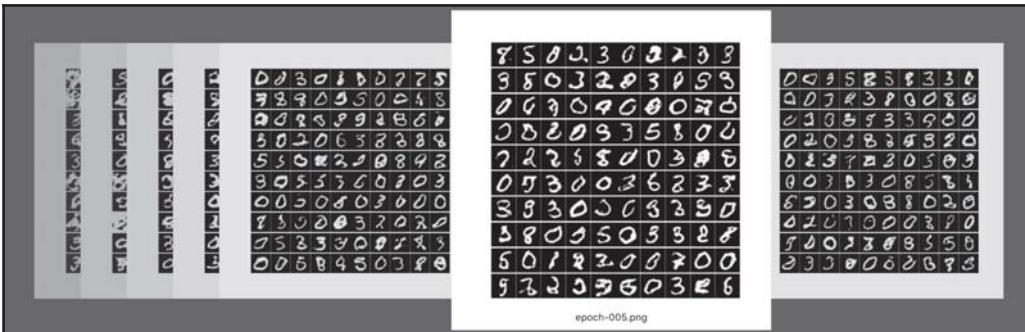
```

examples — bash — 123x35
~/Keras/codeBook/keras_adversarial/examples — bash ~/Keras/codeBook/new/keras_adversarial/examples — bash ...
Trainable params: 15,244,418
Non-trainable params: 100,736

Train on 60000 samples, validate on 10000 samples
Epoch 1/100
60000/60000 [=====] - 7579s - loss: 18.6313 - generator_loss: 18.4795 - generator_yfake_loss: 10.765 - generator_yreal_loss: 7.7030 - discriminator_loss: 0.1519 - discriminator_yfake_loss: 0.0793 - discriminator_yreal_loss: 0.0726 - val_loss: 16.2338 - val_generator_loss: 15.9908 - val_generator_yfake_loss: 7.5463 - val_generator_yreal_loss: 8.4445 - val_discriminator_loss: 0.2431 - val_discriminator_yfake_loss: 0.2044 - val_discriminator_yreal_loss: 0.0386
Epoch 2/100
60000/60000 [=====] - 7737s - loss: 14.5333 - generator_loss: 14.2141 - generator_yfake_loss: 7.2208 - generator_yreal_loss: 6.9933 - discriminator_loss: 0.3192 - discriminator_yfake_loss: 0.1523 - discriminator_yreal_loss: 0.1668 - val_loss: 13.6769 - val_generator_loss: 13.4410 - val_generator_yfake_loss: 6.0405 - val_generator_yreal_loss: 7.4006 - val_discriminator_loss: 0.2359 - val_discriminator_yfake_loss: 0.1093 - val_discriminator_yreal_loss: 0.1265
Epoch 3/100
60000/60000 [=====] - 7842s - loss: 11.6130 - generator_loss: 11.2356 - generator_yfake_loss: 4.0883 - generator_yreal_loss: 7.1473 - discriminator_loss: 0.3775 - discriminator_yfake_loss: 0.1708 - discriminator_yreal_loss: 0.2067 - val_loss: 10.7820 - val_generator_loss: 10.3857 - val_generator_yfake_loss: 2.3510 - val_generator_yreal_loss: 8.0348 - val_discriminator_loss: 0.3963 - val_discriminator_yfake_loss: 0.3050 - val_discriminator_yreal_loss: 0.0913
Epoch 4/100
60000/60000 [=====] - 7567s - loss: 9.6041 - generator_loss: 9.1363 - generator_yfake_loss: 3.1345 - generator_yreal_loss: 6.0018 - discriminator_loss: 0.4678 - discriminator_yfake_loss: 0.2147 - discriminator_yreal_loss: 0.2531 - val_loss: 7.8728 - val_generator_loss: 7.4582 - val_generator_yfake_loss: 2.1574 - val_generator_yreal_loss: 5.3007 - val_discriminator_loss: 0.4146 - val_discriminator_yfake_loss: 0.2797 - val_discriminator_yreal_loss: 0.1350
Epoch 5/100
60000/60000 [=====] - 7971s - loss: 9.0191 - generator_loss: 8.5203 - generator_yfake_loss: 2.9953 - generator_yreal_loss: 5.5250 - discriminator_loss: 0.4988 - discriminator_yfake_loss: 0.2289 - discriminator_yreal_loss: 0.2699 - val_loss: 7.3837 - val_generator_loss: 6.9287 - val_generator_yfake_loss: 1.9531 - val_generator_yreal_loss: 4.9756 - val_discriminator_loss: 0.4550 - val_discriminator_yfake_loss: 0.3175 - val_discriminator_yreal_loss: 0.1375
Epoch 6/100
60000/60000 [=====] - 7333s - loss: 8.7654 - generator_loss: 8.2540 - generator_yfake_loss: 2.9357 - generator_yreal_loss: 5.3184 - discriminator_loss: 0.5114 - discriminator_yfake_loss: 0.2354 - discriminator_yreal_loss: 0.2759 - val_loss: 7.4316 - val_generator_loss: 6.9901 - val_generator_yfake_loss: 2.2178 - val_generator_yreal_loss: 4.7722 - val_discriminator_loss: 0.4416 - val_discriminator_yfake_loss: 0.2494 - val_discriminator_yreal_loss: 0.1922
Epoch 7/100

```

After 5-6 iterations, we already have acceptable artificial images generated and the computer has learned how to reproduce handwritten characters, as shown in the following image:



Keras adversarial GANs for forging CIFAR

Now we can use a GAN approach to learn how to forge CIFAR-10 and create synthetic images that look real. Let's see the open source code (https://github.com/bstriner/keras-adversarial/blob/master/examples/example_gan_cifar10.py). Again, note that it uses the syntax of Keras 1.x, but it also runs on the top of Keras 2.x thanks to a convenient set of utility functions contained in `legacy.py` (https://github.com/bstriner/keras-adversarial/blob/master/keras_adversarial/legacy.py). First, the open source example imports a number of packages:

```
import matplotlib as mpl
# This line allows mpl to run with no DISPLAY defined
mpl.use('Agg')
import pandas as pd
import numpy as np
import os
from keras.layers import Dense, Reshape, Flatten, Dropout, LeakyReLU,
    Activation, BatchNormalization, SpatialDropout2D
from keras.layers.convolutional import Convolution2D, UpSampling2D,
    MaxPooling2D, AveragePooling2D
from keras.models import Sequential, Model
from keras.optimizers import Adam
from keras.callbacks import TensorBoard
from keras.regularizers import l1l2
from keras_adversarial import AdversarialModel, ImageGridCallback,
    simple_gan, gan_targets
from keras_adversarial import AdversarialOptimizerSimultaneous,
    normal_latent_sampling, fix_names
import keras.backend as K
from cifar10_utils import cifar10_data
from image_utils import dim_ordering_fix, dim_ordering_unfix,
    dim_ordering_shape
```

Next, it defines a generator that uses a combination of convolutions with 11 and 12 regularization, batch normalization, and upsampling. Note that `axis=1` says to normalize the dimension of the tensor first and `mode=0` says to adopt a feature-wise normalization. This particular net is the result of many fine-tuning experiments, but it is still essentially a sequence of convolution 2D and upsampling operations, which uses a `Dense` module at the beginning and a `sigmoid` at the end. In addition, each convolution uses a `LeakyReLU` activation function and `BatchNormalization`:

```
def model_generator():
    model = Sequential()
    nch = 256
    reg = lambda: l1l2(l1=1e-7, l2=1e-7)
```

```
h = 5
model.add(Dense(input_dim=100, output_dim=nch * 4 * 4,
W_regularizer=reg()))
model.add(BatchNormalization(mode=0))
model.add(Reshape(dim_ordering_shape((nch, 4, 4))))
model.add(Convolution2D(nch/2, h, h, border_mode='same',
W_regularizer=reg()))
model.add(BatchNormalization(mode=0, axis=1))
model.add(LeakyReLU(0.2))
model.add(UpSampling2D(size=(2, 2)))
model.add(Convolution2D(nch / 2, h, h, border_mode='same',
W_regularizer=reg()))
model.add(BatchNormalization(mode=0, axis=1))
model.add(LeakyReLU(0.2))
model.add(UpSampling2D(size=(2, 2)))
model.add(Convolution2D(nch / 4, h, h, border_mode='same',
W_regularizer=reg()))
model.add(BatchNormalization(mode=0, axis=1))
model.add(LeakyReLU(0.2))
model.add(UpSampling2D(size=(2, 2)))
model.add(Convolution2D(3, h, h, border_mode='same',
W_regularizer=reg()))
model.add(Activation('sigmoid'))
return model
```

Then, a discriminator is defined. Again, we have a sequence of convolution 2D operations, and in this case we adopt SpatialDropout2D, which drops entire 2D feature maps instead of individual elements. We also use MaxPooling2D and AveragePooling2D for similar reasons:

```
def model_discriminator():
    nch = 256
    h = 5
    reg = lambda: l1l2(l1=1e-7, l2=1e-7)
    c1 = Convolution2D(nch / 4, h, h, border_mode='same',
W_regularizer=reg(),
    input_shape=dim_ordering_shape((3, 32, 32)))
    c2 = Convolution2D(nch / 2, h, h, border_mode='same',
W_regularizer=reg())
    c3 = Convolution2D(nch, h, h, border_mode='same', W_regularizer=reg())
    c4 = Convolution2D(1, h, h, border_mode='same', W_regularizer=reg())
    def m(dropout):
        model = Sequential()
        model.add(c1)
        model.add(SpatialDropout2D(dropout))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(LeakyReLU(0.2))
        model.add(c2)
```

```
        model.add(SpatialDropout2D(dropout))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(LeakyReLU(0.2))
        model.add(c3)
        model.add(SpatialDropout2D(dropout))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(LeakyReLU(0.2))
        model.add(c4)
        model.add(AveragePooling2D(pool_size=(4, 4), border_mode='valid'))
        model.add(Flatten())
        model.add(Activation('sigmoid'))
    return model
return m
```

It is now possible to generate proper GANs. The following function takes multiple inputs, including a generator, a discriminator, the number of latent dimensions, and the GAN targets:

```
def example_gan(adversarial_optimizer, path, opt_g, opt_d, nb_epoch,
generator,
discriminator, latent_dim, targets=gan_targets,
loss='binary_crossentropy'):
    csvpath = os.path.join(path, "history.csv")
    if os.path.exists(csvpath):
        print("Already exists: {}".format(csvpath))
    return
```

Then two GANs are created, one with dropout and the other without dropout for the discriminator:

```
print("Training: {}".format(csvpath))
# gan (x -> yfake, yreal), z is gaussian generated on GPU
# can also experiment with uniform_latent_sampling
d_g = discriminator(0)
d_d = discriminator(0.5)
generator.summary()
d_d.summary()
gan_g = simple_gan(generator, d_g, None)
gan_d = simple_gan(generator, d_d, None)
x = gan_g.inputs[1]
z = normal_latent_sampling((latent_dim,))(x)
# eliminate z from inputs
gan_g = Model([x], fix_names(gan_g([z, x]), gan_g.output_names))
gan_d = Model([x], fix_names(gan_d([z, x]), gan_d.output_names))
```

The two GANs are now combined into an adversarial model with separate weights, and the model is then compiled:

```
# build adversarial model
model = AdversarialModel(player_models=[gan_g, gan_d],
    player_params=[generator.trainable_weights, d_d.trainable_weights],
    player_names=["generator", "discriminator"])
model.adversarial_compile(adversarial_optimizer=adversarial_optimizer,
    player_optimizers=[opt_g, opt_d], loss=loss)
```

Next, there is a simple callback to sample images and a print on the file where the method `ImageGridCallback` is defined:

```
# create callback to generate images
zsamples = np.random.normal(size=(10 * 10, latent_dim))
def generator_sampler():
    xpred = dim_ordering_unfix(generator.predict(zsamples)).transpose((0,
2, 3, 1))
    return xpred.reshape((10, 10) + xpred.shape[1:])
generator_cb =
    ImageGridCallback(os.path.join(path, "epoch-{:03d}.png"),
        generator_sampler, cmap=None)
```

Now, the CIFAR-10 data is loaded and the model is fit. If the backend is TensorFlow, then the loss information is saved into a TensorBoard to check how the loss decreases over time. The history is also conveniently saved into a CVS format, and the models' weights are also stored in an h5 format:

```
# train model
xtrain, xtest = cifar10_data()
y = targets(xtrain.shape[0])
ytest = targets(xtest.shape[0])
callbacks = [generator_cb]
if K.backend() == "tensorflow":
    callbacks.append(TensorBoard(log_dir=os.path.join(path, 'logs'),
        histogram_freq=0, write_graph=True, write_images=True))
history = model.fit(x=dim_ordering_fix(xtrain), y=y,
    validation_data=(dim_ordering_fix(xtest), ytest),
    callbacks=callbacks, nb_epoch=nb_epoch,
    batch_size=32)
# save history to CSV
df = pd.DataFrame(history.history)
df.to_csv(csvpath)
# save models
generator.save(os.path.join(path, "generator.h5"))
d_d.save(os.path.join(path, "discriminator.h5"))
```

Finally, the whole GANs can be run. The generator samples from a space with 100 latent dimensions, and we've used Adam as optimizer for both GANs:

```
def main():
    # z in R^100
    latent_dim = 100
    # x in R^{28x28}
    # generator (z -> x)
    generator = model_generator()
    # discriminator (x -> y)
    discriminator = model_discriminator()
    example_gan(AdversarialOptimizerSimultaneous(), "output/gan-cifar10",
                opt_g=Adam(1e-4, decay=1e-5),
                opt_d=Adam(1e-3, decay=1e-5),
                nb_epoch=100, generator=generator, discriminator=discriminator,
                latent_dim=latent_dim)
if __name__ == "__main__":
    main()
```

In order to have a complete view on the open source code, we need to include a few simple utility functions for storing the grid of images:

```
from matplotlib import pyplot as plt, gridspec
import os

def write_image_grid(filepath, imgs, figsize=None, cmap='gray'):
    directory = os.path.dirname(filepath)
    if not os.path.exists(directory):
        os.makedirs(directory)
    fig = create_image_grid(imgs, figsize, cmap=cmap)
    fig.savefig(filepath)
    plt.close(fig)

def create_image_grid(imgs, figsize=None, cmap='gray'):
    n = imgs.shape[0]
    m = imgs.shape[1]
    if figsize is None:
        figsize=(n,m)
    fig = plt.figure(figsize=figsize)
    gs1 = gridspec.GridSpec(n, m)
    gs1.update(wspace=0.025, hspace=0.025) # set the spacing between axes.
    for i in range(n):
        for j in range(m):
            ax = plt.subplot(gs1[i, j])
            img = imgs[i, j, :]
            ax.imshow(img, cmap=cmap)
            ax.axis('off')
    return fig
```

In addition, we need some utility methods for dealing with different image ordering (for example, Theano or TensorFlow):

```
import keras.backend as K
import numpy as np
from keras.layers import Input, Reshape

def dim_ordering_fix(x):
    if K.image_dim_ordering() == 'th':
        return x
    else:
        return np.transpose(x, (0, 2, 3, 1))

def dim_ordering_unfix(x):
    if K.image_dim_ordering() == 'th':
        return x
    else:
        return np.transpose(x, (0, 3, 1, 2))

def dim_ordering_shape(input_shape):
    if K.image_dim_ordering() == 'th':
        return input_shape
    else:
        return (input_shape[1], input_shape[2], input_shape[0])

def dim_ordering_input(input_shape, name):
    if K.image_dim_ordering() == 'th':
        return Input(input_shape, name=name)
    else:
        return Input((input_shape[1], input_shape[2], input_shape[0]),
                     name=name)

def dim_ordering_reshape(k, w, **kwargs):
    if K.image_dim_ordering() == 'th':
        return Reshape((k, w, w), **kwargs)
    else:
        return Reshape((w, w, k), **kwargs)

# One more utility function is used to fix names
def fix_names(outputs, names):
    if not isinstance(outputs, list):
        outputs = [outputs]
    if not isinstance(names, list):
        names = [names]
    return [Activation('linear', name=name)(output)
            for output, name in zip(outputs, names)]
```

The following screenshot, shows a dump of the defined networks:

```

examples — python example_gan_cifar10.py — 140x78
...ook/keras-dcgan — python dcgan.py --mode train ~Keras/codeBook/keras-dcgan — -bash ...rsarial/examples — python example_gan_cifar10.py +
gulli-macbookpro:examples gulli$ python example_gan_cifar10.py
Using TensorFlow backend.
Training: output/gan-cifar10/history.csv

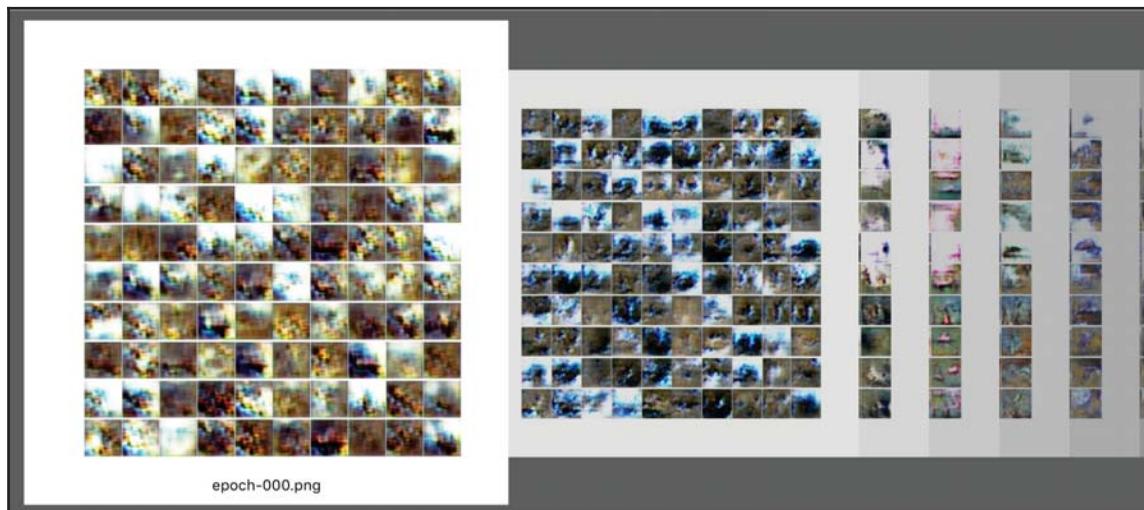
Layer (type)          Output Shape       Param #
=====================================================================
dense_1 (Dense)      (None, 4096)       413696
batch_normalization_1 (Batch Normalization) (None, 4096)   16384
reshape_1 (Reshape)   (None, 256, 4, 4)    0
conv2d_1 (Conv2D)     (None, 128, 4, 4)   819328
batch_normalization_2 (Batch Normalization) (None, 128, 4, 4) 512
leaky_re_lu_1 (LeakyReLU) (None, 128, 4, 4) 0
up_sampling2d_1 (UpSampling2) (None, 128, 8, 8) 0
conv2d_2 (Conv2D)     (None, 128, 8, 8)   409728
batch_normalization_3 (Batch Normalization) (None, 128, 8, 8) 512
leaky_re_lu_2 (LeakyReLU) (None, 128, 8, 8) 0
up_sampling2d_2 (UpSampling2) (None, 128, 16, 16) 0
conv2d_3 (Conv2D)     (None, 64, 16, 16)   204864
batch_normalization_4 (Batch Normalization) (None, 64, 16, 16) 256
leaky_re_lu_3 (LeakyReLU) (None, 64, 16, 16) 0
up_sampling2d_3 (UpSampling2) (None, 64, 32, 32) 0
conv2d_4 (Conv2D)     (None, 3, 32, 32)    4803
activation_1 (Activation) (None, 3, 32, 32) 0
=====
Total params: 1,870,083.0
Trainable params: 1,861,251.0
Non-trainable params: 8,832.0

Layer (type)          Output Shape       Param #
=====================================================================
conv2d_5 (Conv2D)     (None, 64, 32, 32) 4864
max_pooling2d_1 (MaxPooling2) (None, 64, 16, 16) 0
leaky_re_lu_4 (LeakyReLU) (None, 64, 16, 16) 0
conv2d_6 (Conv2D)     (None, 128, 16, 16) 204928
max_pooling2d_2 (MaxPooling2) (None, 128, 8, 8) 0
leaky_re_lu_5 (LeakyReLU) (None, 128, 8, 8) 0
conv2d_7 (Conv2D)     (None, 256, 8, 8)   819456
max_pooling2d_3 (MaxPooling2) (None, 256, 4, 4) 0
leaky_re_lu_6 (LeakyReLU) (None, 256, 4, 4) 0
conv2d_8 (Conv2D)     (None, 1, 4, 4)    6401
average_pooling2d_1 (Average) (None, 1, 1, 1) 0
flatten_1 (Flatten)   (None, 1)        0
activation_2 (Activation) (None, 1)        0
=====
Total params: 1,035,649.0
Trainable params: 1,035,649.0
Non-trainable params: 0.0

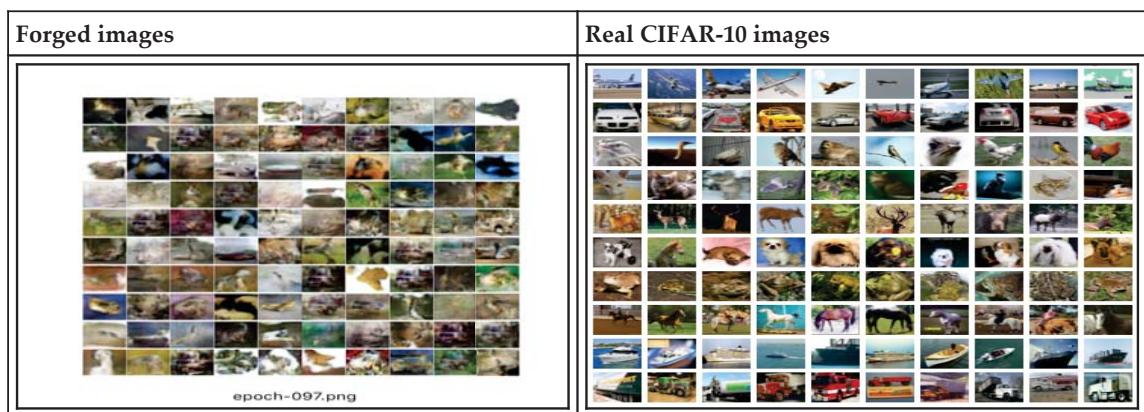
Train on 50000 samples, validate on 10000 samples

```

If we run the open source code, the very first iteration will generate unrealistic images. However, after 99 iterations, the network will learn to forge images that look like real CIFAR-10 images, as shown here:



In the following images, we see the real CIFAR-10 image on the right and the forged one on the left:



WaveNet — a generative model for learning how to produce audio

WaveNet is a deep generative model for producing raw audio waveforms. This breakthrough technology was introduced (<https://deepmind.com/blog/wavenet-generative-model-raw-audio/>) by Google DeepMind (<https://deepmind.com/>) for teaching users how to speak to computers. The results are truly impressive, and you can find online examples of synthetic voices where the computer learns how to talk with the voices of celebrities such as Matt Damon. So, you might wonder why learning to synthesize audio is so difficult. Well, each digital sound we hear is based on 16,000 samples per second (sometimes, 48,000 or more), and building a predictive model where we learn to reproduce a sample based on all the previous ones is a very difficult challenge. Nevertheless, there are experiments showing that WaveNet has improved current state-of-the-art **text-to-speech** (TTS) systems, reducing the difference with human voices by 50% for both US English and Mandarin Chinese. What is even cooler is that DeepMind proved that WaveNet can also be used to teach computers how to generate the sound of musical instruments such as piano music. Now it's time for some definitions. TTS systems are typically divided into two different classes:

- **Concatenative TTS:** This is where single speech voice fragments are first memorized and then recombined when the voice has to be reproduced. However, this approach does not scale because it is only possible to reproduce the memorized voice fragments, and it is not possible to reproduce new speakers or different types of audio without memorizing the fragments from the beginning.
- **Parametric TTS:** This is where a model is created for storing all the characteristic features of the audio to be synthesized. Before WaveNet, the audio generated with parametric TTS was less natural than concatenative TTS. WaveNet improved the state-of-the-art by modeling directly the production of audio sounds, instead of using intermediate signal processing algorithms that have been used in the past.

In principle, WaveNet can be seen as a stack of 1D convolutional layers (we have seen 2D convolution for images in Chapter 3, *Deep Learning with ConvNets*), with a constant stride of one and with no pooling layers. Note that the input and the output have by construction the same dimension, so ConvNet is well-suited to model sequential data such as audio.

However, it has been shown that in order to reach a large size for the receptive field (remember that the receptive field of a neuron in a layer is the cross section of the previous layer from which neurons provide inputs) in the output neuron it is necessary to either use a massive number of large filters or prohibitively increase the depth of the network. For this reason, pure ConvNets are not so effective in learning how to synthesize audio. The key intuition beyond WaveNet is the dilated causal convolutions (for more information refer to the article: *Multi-Scale Context Aggregation by Dilated Convolutions*, by Fisher Yu, Vladlen Koltun, 2016, available at: <https://www.semanticscholar.org/paper/Multi-Scale-Context-Aggregation-by-Dilated-Yu-Koltun/420c46d7cafcb841309f02ad04cf51cb1f190a48>) or sometime atrous convolution (*atrous* is the *bastardization* of the French expression *à trous*, meaning *with holes*, so an atrous convolution is a convolution with holes), which simply means that some input values are skipped when the filter of a convolutional layer is applied. As an example, in one dimension, a filter, w , of size 3 with dilatation 1 would compute the following sum:

$$w[0]x[0] + w[1]x[2] + w[3]x[4]$$

Thanks to this simple idea of introducing *holes*, it is possible to stack multiple dilated convolutional layers with exponentially increasing filters, and learn long range input dependencies without having an excessively deep network. A WaveNet is therefore a ConvNet where the convolutional layers have various dilation factors, allowing the receptive field to grow exponentially with depth and therefore efficiently cover thousands of audio time-steps. When we train, the input are sounds recorded from human speakers. The waveforms are quantized to a fixed integer range. A WaveNet defines an initial convolutional layer accessing only the current and previous input. Then, there is a stack of dilated ConvNet layers, still accessing only current and previous inputs. At the end, there is a series of dense layers that combine previous results, followed by a softmax activation function for categorical outputs. At each step, a value is predicted from the network and fed back into the input. At the same time, a new prediction for the next step is computed. The loss function is the cross-entropy between the output for the current step and the input at the next step. One Keras implementation developed by Bas Veeling is available at: <https://github.com/basveeling/wavenet> and can be easily installed via git:

```
pip install virtualenv
mkdir ~/virtualenvs && cd ~/virtualenvs
virtualenv wavenet
source wavenet/bin/activate
cd ~
```

```
git clone https://github.com/basveeling/wavenet.git
cd wavenet
pip install -r requirements.txt
```

Note that this code is compatible with Keras 1.x and please check the issue at <https://github.com/basveeling/wavenet/issues/29>, to understand what is the progress for porting it on the top of Keras 2.x. Training is very simple but requires a significant amount of computational power (so make sure that you have good GPU support):

```
$ python wavenet.py with 'data_dir=your_data_dir_name'
```

Sampling the network after training is equally very easy:

```
python wavenet.py predict with 'models/[run_folder]/config.json'
predict_seconds=1'
```

You can find a large number of hyperparameters online, which can be used for fine-tuning our training process. The network is really deep, as explained by this dump of internal layers. Note that the input waveform are divided into (`fragment_length = 1152` and `nb_output_bins = 256`), which is the tensor propagating into WaveNet. WaveNet is organized in repeated blocks called residuals, each consisting of a multiplied merge of two dilated convolutional modules (one with sigmoid and the other with tanh activation), followed by a sum merged convolutional. Note that each dilated convolution has holes of growing exponential size ($2^{** i}$) from 1 to 512, as defined in this piece of text:

```
def residual_block(x):
    original_x = x
    tanh_out = CausalAtrousConvolution1D(nb_filters, 2, atrous_rate=2 ** i,
        border_mode='valid', causal=True, bias=use_bias,
        name='dilated_conv_%d_tanh_s%d' % (2 ** i, s), activation='tanh',
        W_regularizer=l2(res_l2))(x)
    sigm_out = CausalAtrousConvolution1D(nb_filters, 2, atrous_rate=2 ** i,
        border_mode='valid', causal=True, bias=use_bias,
        name='dilated_conv_%d_sigm_s%d' % (2 ** i, s),
        activation='sigmoid',
        W_regularizer=l2(res_l2))(x)
    x = layers.Merge(mode='mul',
        name='gated_activation_%d_s%d' % (i, s))([tanh_out, sigm_out])
    res_x = layers.Convolution1D(nb_filters, 1, border_mode='same',
        bias=use_bias,
        W_regularizer=l2(res_l2))(x)
    skip_x = layers.Convolution1D(nb_filters, 1, border_mode='same',
        bias=use_bias,
        W_regularizer=l2(res_l2))(x)
    res_x = layers.Merge(mode='sum') ([original_x, res_x])
    return res_x, skip_x
```

After the residual dilated block, there is a sequence of merged convolutional modules, followed by two convolutional modules, followed by a softmax activation function in nb_output_bins categories. The full network structure is here:

```
Layer (type) Output Shape Param # Connected to
=====
=====
input_part (InputLayer) (None, 1152, 256) 0

=====
initial_causal_conv (CausalAtrou (None, 1152, 256) 131328 input_part[0][0]

=====
dilated_conv_1_tanh_s0 (CausalAt (None, 1152, 256) 131072
initial_causal_conv[0][0]

=====
dilated_conv_1_sigm_s0 (CausalAt (None, 1152, 256) 131072
initial_causal_conv[0][0]

=====
gated_activation_0_s0 (Merge) (None, 1152, 256) 0
dilated_conv_1_tanh_s0[0][0]
dilated_conv_1_sigm_s0[0][0]

=====
convolution1d_1 (Convolution1D) (None, 1152, 256) 65536
gated_activation_0_s0[0][0]

=====
merge_1 (Merge) (None, 1152, 256) 0 initial_causal_conv[0][0]
convolution1d_1[0][0]

=====
dilated_conv_2_tanh_s0 (CausalAt (None, 1152, 256) 131072 merge_1[0][0]

=====
dilated_conv_2_sigm_s0 (CausalAt (None, 1152, 256) 131072 merge_1[0][0]

=====
gated_activation_1_s0 (Merge) (None, 1152, 256) 0
dilated_conv_2_tanh_s0[0][0]
dilated_conv_2_sigm_s0[0][0]

=====
convolution1d_3 (Convolution1D) (None, 1152, 256) 65536
gated_activation_1_s0[0][0]
```

```
merge_2 (Merge) (None, 1152, 256) 0 merge_1[0][0]
convolution1d_3[0][0]

dilated_conv_4_tanh_s0 (CausalAt (None, 1152, 256) 131072 merge_2[0][0]

dilated_conv_4_sigm_s0 (CausalAt (None, 1152, 256) 131072 merge_2[0][0]

gated_activation_2_s0 (Merge) (None, 1152, 256) 0
dilated_conv_4_tanh_s0[0][0]
dilated_conv_4_sigm_s0[0][0]

convolution1d_5 (Convolution1D) (None, 1152, 256) 65536
gated_activation_2_s0[0][0]

merge_3 (Merge) (None, 1152, 256) 0 merge_2[0][0]
convolution1d_5[0][0]

dilated_conv_8_tanh_s0 (CausalAt (None, 1152, 256) 131072 merge_3[0][0]

dilated_conv_8_sigm_s0 (CausalAt (None, 1152, 256) 131072 merge_3[0][0]

gated_activation_3_s0 (Merge) (None, 1152, 256) 0
dilated_conv_8_tanh_s0[0][0]
dilated_conv_8_sigm_s0[0][0]

convolution1d_7 (Convolution1D) (None, 1152, 256) 65536
gated_activation_3_s0[0][0]

merge_4 (Merge) (None, 1152, 256) 0 merge_3[0][0]
convolution1d_7[0][0]

dilated_conv_16_tanh_s0 (CausalA (None, 1152, 256) 131072 merge_4[0][0]

dilated_conv_16_sigm_s0 (CausalA (None, 1152, 256) 131072 merge_4[0][0]
```

```
gated_activation_4_s0 (Merge) (None, 1152, 256) 0
dilated_conv_16_tanh_s0[0][0]
dilated_conv_16_sigm_s0[0][0]
```

```
convolution1d_9 (Convolution1D) (None, 1152, 256) 65536
gated_activation_4_s0[0][0]
```

```
merge_5 (Merge) (None, 1152, 256) 0 merge_4[0][0]
convolution1d_9[0][0]
```

```
dilated_conv_32_tanh_s0 (CausalA (None, 1152, 256) 131072 merge_5[0][0]
```

```
dilated_conv_32_sigm_s0 (CausalA (None, 1152, 256) 131072 merge_5[0][0]
```

```
gated_activation_5_s0 (Merge) (None, 1152, 256) 0
dilated_conv_32_tanh_s0[0][0]
dilated_conv_32_sigm_s0[0][0]
```

```
convolution1d_11 (Convolution1D) (None, 1152, 256) 65536
gated_activation_5_s0[0][0]
```

```
merge_6 (Merge) (None, 1152, 256) 0 merge_5[0][0]
convolution1d_11[0][0]
```

```
dilated_conv_64_tanh_s0 (CausalA (None, 1152, 256) 131072 merge_6[0][0]
```

```
dilated_conv_64_sigm_s0 (CausalA (None, 1152, 256) 131072 merge_6[0][0]
```

```
gated_activation_6_s0 (Merge) (None, 1152, 256) 0
dilated_conv_64_tanh_s0[0][0]
dilated_conv_64_sigm_s0[0][0]
```

```
convolution1d_13 (Convolution1D) (None, 1152, 256) 65536
gated_activation_6_s0[0][0]
```

```
merge_7 (Merge) (None, 1152, 256) 0 merge_6[0][0]
convolution1d_13[0][0]
```

```
dilated_conv_128_tanh_s0 (Causal (None, 1152, 256) 131072 merge_7[0][0]
```

```
dilated_conv_128_sigm_s0 (Causal (None, 1152, 256) 131072 merge_7[0][0]
```

```
gated_activation_7_s0 (Merge) (None, 1152, 256) 0
dilated_conv_128_tanh_s0[0][0]
dilated_conv_128_sigm_s0[0][0]
```

```
convolution1d_15 (Convolution1D) (None, 1152, 256) 65536
gated_activation_7_s0[0][0]
```

```
merge_8 (Merge) (None, 1152, 256) 0 merge_7[0][0]
convolution1d_15[0][0]
```

```
dilated_conv_256_tanh_s0 (Causal (None, 1152, 256) 131072 merge_8[0][0]
```

```
dilated_conv_256_sigm_s0 (Causal (None, 1152, 256) 131072 merge_8[0][0]
```

```
gated_activation_8_s0 (Merge) (None, 1152, 256) 0
dilated_conv_256_tanh_s0[0][0]
dilated_conv_256_sigm_s0[0][0]
```

```
convolution1d_17 (Convolution1D) (None, 1152, 256) 65536
gated_activation_8_s0[0][0]
```

```
merge_9 (Merge) (None, 1152, 256) 0 merge_8[0][0]
convolution1d_17[0][0]
```

```
dilated_conv_512_tanh_s0 (Causal (None, 1152, 256) 131072 merge_9[0][0]
```

```
dilated_conv_512_sigm_s0 (Causal (None, 1152, 256) 131072 merge_9[0][0]
```

```
gated_activation_9_s0 (Merge) (None, 1152, 256) 0
dilated_conv_512_tanh_s0[0][0]
dilated_conv_512_sigm_s0[0][0]

convolution1d_2 (Convolution1D) (None, 1152, 256) 65536
gated_activation_0_s0[0][0]

convolution1d_4 (Convolution1D) (None, 1152, 256) 65536
gated_activation_1_s0[0][0]

convolution1d_6 (Convolution1D) (None, 1152, 256) 65536
gated_activation_2_s0[0][0]

convolution1d_8 (Convolution1D) (None, 1152, 256) 65536
gated_activation_3_s0[0][0]

convolution1d_10 (Convolution1D) (None, 1152, 256) 65536
gated_activation_4_s0[0][0]

convolution1d_12 (Convolution1D) (None, 1152, 256) 65536
gated_activation_5_s0[0][0]

convolution1d_14 (Convolution1D) (None, 1152, 256) 65536
gated_activation_6_s0[0][0]

convolution1d_16 (Convolution1D) (None, 1152, 256) 65536
gated_activation_7_s0[0][0]

convolution1d_18 (Convolution1D) (None, 1152, 256) 65536
gated_activation_8_s0[0][0]

convolution1d_20 (Convolution1D) (None, 1152, 256) 65536
gated_activation_9_s0[0][0]

merge_11 (Merge) (None, 1152, 256) 0 convolution1d_2[0][0]
convolution1d_4[0][0]
convolution1d_6[0][0]
```

```
convolution1d_8[0][0]
convolution1d_10[0][0]
convolution1d_12[0][0]
convolution1d_14[0][0]
convolution1d_16[0][0]
convolution1d_18[0][0]
convolution1d_20[0][0]

activation_1 (Activation) (None, 1152, 256) 0 merge_11[0][0]

convolution1d_21 (Convolution1D) (None, 1152, 256) 65792 activation_1[0][0]

activation_2 (Activation) (None, 1152, 256) 0 convolution1d_21[0][0]

convolution1d_22 (Convolution1D) (None, 1152, 256) 65792 activation_2[0][0]

output_softmax (Activation) (None, 1152, 256) 0 convolution1d_22[0][0]
=====
=====
Total params: 4,129,536
Trainable params: 4,129,536
Non-trainable params: 0
```

DeepMind tried to train with data sets including multiple speakers, and this significantly improved the capacity to learn a shared representation of languages and tones and thus receive results close to natural speech. You'll find an amazing collection of examples of synthesized voice online (<https://deepmind.com/blog/wavenet-generative-model-raw-audio/>), and it is interesting to note that the quality of audio improves when WaveNet is conditioned on additional text that is transformed into a sequence of linguistic and phonetic features in addition to audio waveforms. My favorite examples are the ones where the same sentence is pronounced by the net with different tones of voice. Of course, it is also fascinating to hear WaveNet create piano music by itself. Check it out online!

Summary

In this chapter, we discussed GANs. A GAN typically consists of two networks; one is trained to forge synthetic data that looks authentic, and the second is trained to discriminate authentic data against forged data. The two networks continuously compete, and in doing so, they keep improving each other. We reviewed an open source code, learning to forge MNIST and CIFAR-10 images that look authentic. In addition, we discussed WaveNet, a deep generative network proposed by Google DeepMind for teaching computers how to reproduce human voices and musical instruments with impressive quality. WaveNet directly generates raw audio with a parametric text-to-speech approach based on dilated convolutional networks. Dilated convolutional networks are a special kind of ConvNets where convolution filters have holes, allowing the receptive field to grow exponentially in depth and therefore efficiently cover thousands of audio time-steps. DeepMind showed how it is possible to use WaveNet to synthesize human voice and musical instruments, and improved previous state-of-the-art. In the next chapter, we will discuss word embeddings—a set of deep learning methodologies for detecting relations among words and grouping together similar words.

5

Word Embeddings

Wikipedia defines word embedding as the collective name for a set of language modeling and feature learning techniques in **natural language processing (NLP)** where words or phrases from the vocabulary are mapped to vectors of real numbers.

Word embeddings are a way to transform words in text to numerical vectors so that they can be analyzed by standard machine learning algorithms that require vectors as numerical input.

You have already learned about one type of word embedding called **one-hot encoding**, in Chapter 1, *Neural Networks Foundations*. One-hot encoding is the most basic embedding approach. To recap, one-hot encoding represents a word in the text by a vector of the size of the vocabulary, where only the entry corresponding to the word is a one and all the other entries are zero.

A major problem with one-hot encoding is that there is no way to represent the similarity between words. In any given corpus, you would expect words such as (*cat, dog*), (*knife, spoon*), and so on to have some similarity. Similarity between vectors is computed using the dot product, which is the sum of element-wise multiplication between vector elements. In the case of one-hot encoded vectors, the dot product between any two words in a corpus is always zero.

To overcome the limitations of one-hot encoding, the NLP community has borrowed techniques from **information retrieval (IR)** to vectorize text using the document as the context. Notable techniques are TF-IDF (<https://en.wikipedia.org/wiki/Tf%E2%80%93idf>), **latent semantic analysis (LSA)** (https://en.wikipedia.org/wiki/Latent_semantic_analysis), and topic modeling (https://en.wikipedia.org/wiki/Topic_model). However, these representations capture a slightly different document-centric idea of semantic similarity.

Development of word embedding techniques began in earnest in 2000. Word embedding differs from previous IR-based techniques in that they use words as their context, which leads to a more natural form of semantic similarity from a human understanding perspective. Today, word embedding is the technique of choice for vectorizing text for all kinds of NLP tasks, such as text classification, document clustering, part of speech tagging, named entity recognition, sentiment analysis, and so on.

In this chapter, we will learn about two specific forms of word embedding, GloVe and word2vec, collectively known as distributed representations of words. These embeddings have proven more effective and have been widely adopted in the deep learning and NLP communities.

We will also learn different ways in which you can generate your own embeddings in your Keras code, as well as how to use and fine-tune pre-trained word2vec and GloVe models.

In this chapter, we will cover the following topics:

- Building various distributional representations of words in context
- Building models for leveraging embeddings to perform NLP tasks such as sentence parsing and sentiment analysis

Distributed representations

Distributed representations attempt to capture the meaning of a word by considering its relations with other words in its context. The idea is captured in this quote from J. R. Firth (for more information refer to the article: *Document Embedding with Paragraph Vectors*, by Andrew M. Dai, Christopher Olah, and Quoc V. Le, arXiv:1507.07998, 2015), a linguist who first proposed this idea:

You shall know a word by the company it keeps.

Consider the following pair of sentences:

Paris is the capital of France.

Berlin is the capital of Germany.

Even assuming you have no knowledge of world geography (or English for that matter), you would still conclude without too much effort that the word pairs (*Paris, Berlin*) and (*France, Germany*) were related in some way, and that corresponding words in each pair were related in the same way to each other, that is:

Paris : France :: Berlin : Germany

Thus, the aim of distributed representations is to find a general transformation function ϕ to convert each word to its associated vector such that relations of the following form hold true:

$$\varphi(\text{"Paris"}) - \varphi(\text{"France"}) \approx \varphi(\text{"Berlin"}) - \varphi(\text{"Germany"})$$

In other words, distributed representation aims to convert words to vectors where the similarity between the vectors correlate with the semantic similarity between the words.

The most well-known word embeddings are word2vec and GloVe, which we cover in more detail in subsequent sections.

word2vec

The word2vec group of models was created in 2013 by a team of researchers at Google led by Tomas Mikolov. The models are unsupervised, taking as input a large corpus of text and producing a vector space of words. The dimensionality of the word2vec embedding space is usually lower than the dimensionality of the one-hot embedding space, which is the size of the vocabulary. The embedding space is also more dense compared to the sparse embedding of the one-hot embedding space.

The two architectures for word2vec are as follows:

- **Continuous Bag Of Words (CBOW)**
- **Skip-gram**

In the CBOW architecture, the model predicts the current word given a window of surrounding words. In addition, the order of the context words does not influence the prediction (that is, the bag of words assumption). In the case of skip-gram architecture, the model predicts the surrounding words given the center word. According to the authors, CBOW is faster but skip-gram does a better job at predicting infrequent words.

An interesting thing to note is that even though word2vec creates embeddings that are used in deep learning NLP models, both flavors of word2vec that we will discuss, which also happens to be the most successful and acknowledged recent models, are shallow neural networks.

The skip-gram word2vec model

The skip-gram model is trained to predict the surrounding words given the current word. To understand how the skip-gram word2vec model works, consider the following example sentence:

I love green eggs and ham.

Assuming a window size of three, this sentence can be broken down into the following sets of (context, word) pairs:

([*I*, *green*], *love*)
([*love*, *eggs*], *green*)
([*green*, *and*], *eggs*)
...

Since the skip-gram model predicts a context word given the center word, we can convert the preceding dataset to one of (input, output) pairs. That is, given an input word, we expect the skip-gram model to predict the output word:

(*love*, *I*), (*love*, *green*), (*green*, *love*), (*green*, *eggs*), (*eggs*, *green*), (*eggs*, *and*), ...

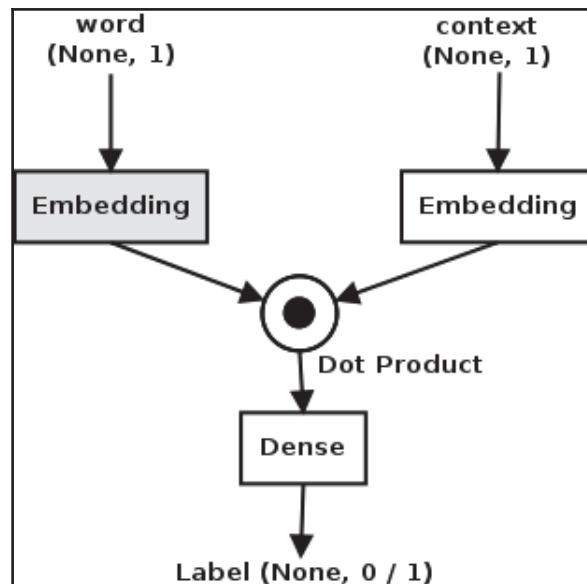
We can also generate additional negative samples by pairing each input word with some random word in the vocabulary. For example:

(*love*, *Sam*), (*love*, *zebra*), (*green*, *thing*), ...

Finally, we generate positive and negative examples for our classifier:

((*love*, *I*), 1), ((*love*, *green*), 1), ..., ((*love*, *Sam*), 0), ((*love*, *zebra*), 0), ...

We can now train a classifier that takes in a word vector and a context vector and learns to predict one or zero depending on whether it sees a positive or negative sample. The deliverables from this trained network are the weights of the word embedding layer (the gray box in the following figure):



The skip-gram model can be built in Keras as follows. Assume that the vocabulary size is set at 5000, the output embedding size is 300, and the window size is 1. A window size of one means that the context for a word is the words immediately to the left and right. We first take care of the imports and set our variables to their initial values:

```
from keras.layers import Merge
from keras.layers.core import Dense, Reshape
from keras.layers.embeddings import Embedding
from keras.models import Sequential

vocab_size = 5000
embed_size = 300
```

We then create a sequential model for the word. The input to this model is the word ID in the vocabulary. The embedding weights are initially set to small random values. During training, the model will update these weights using backpropagation. The next layer reshapes the input to the embedding size:

```
word_model = Sequential()
word_model.add(Embedding(vocab_size, embed_size,
                        embeddings_initializer="glorot_uniform",
                        input_length=1))
word_model.add(Reshape((embed_size, )))
```

The other model that we need is a sequential model for the context words. For each of our skip-gram pairs, we have a single context word corresponding to the target word, so this model is identical to the word model:

```
context_model = Sequential()
context_model.add(Embedding(vocab_size, embed_size,
                           embeddings_initializer="glorot_uniform",
                           input_length=1))
context_model.add(Reshape((embed_size, )))
```

The outputs of the two models are each a vector of size (`embed_size`). These outputs are merged into one using a dot product and fed into a dense layer, which has a single output wrapped in a sigmoid activation layer. You have seen the sigmoid activation function in Chapter 1, *Neural Network Foundations*. As you will recall, it modulates the output so numbers higher than 0.5 tend rapidly to 1 and flatten out, and numbers lower than 0.5 tend rapidly to 0 and also flatten out:

```
model = Sequential()
model.add(Merge([word_model, context_model], mode="dot"))
model.add(Dense(1, init="glorot_uniform", activation="sigmoid"))
model.compile(loss="mean_squared_error", optimizer="adam")
```

The loss function used is the `mean_squared_error`; the idea is to minimize the dot product for positive examples and maximize it for negative examples. If you recall, the dot product multiplies corresponding elements of two vectors and sums up the result—this causes similar vectors to have higher dot products than dissimilar vectors, since the former has more overlapping elements.

Keras provides a convenience function to extract skip-grams for a text that has been converted to a list of word indices. Here is an example of using this function to extract the first 10 of 56 skip-grams generated (both positive and negative).

We first declare the necessary imports and the text to be analyzed:

```
from keras.preprocessing.text import *
from keras.preprocessing.sequence import skipgrams

text = "I love green eggs and ham ."
```

The next step is to declare the `tokenizer` and run the text against it. This will produce a list of word tokens:

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts([text])
```

The `tokenizer` creates a dictionary mapping each unique word to an integer ID and makes it available in the `word_index` attribute. We extract this and create a two-way lookup table:

```
word2id = tokenizer.word_index
id2word = {v:k for k, v in word2id.items() }
```

Finally, we convert our input list of words to a list of IDs and pass it to the `skipgrams` function. We then print the first 10 of the 56 (pair, label) skip-gram tuples generated:

```
wids = [word2id[w] for w in text_to_word_sequence(text)]
pairs, labels = skipgrams(wids, len(word2id))
print(len(pairs), len(labels))
for i in range(10):
    print("({:s} ({:d}), {:s} ({:d})) -> {:d}".format(
        id2word[pairs[i][0]], pairs[i][0],
        id2word[pairs[i][1]], pairs[i][1],
        labels[i]))
```

The results from the code is shown below. Note that your results may be different since the skip-gram method randomly samples the results from the pool of possibilities for the positive examples. Additionally, the process of negative sampling, used for generating the negative examples, consists of randomly pairing up arbitrary tokens from the text. As the size of the input text increases, this is more likely to pick up unrelated word pairs. In our example, since our text is very short, there is a chance that it can end up generating positive examples as well.

```
(and (1), ham (3)) -> 0
(green (6), i (4)) -> 0
(love (2), i (4)) -> 1
(and (1), love (2)) -> 0
(love (2), eggs (5)) -> 0
(ham (3), ham (3)) -> 0
(green (6), and (1)) -> 1
(eggs (5), love (2)) -> 1
(i (4), ham (3)) -> 0
(and (1), green (6)) -> 1
```

The code for this example can be found in `skipgram_example.py` in the source code download for the chapter.

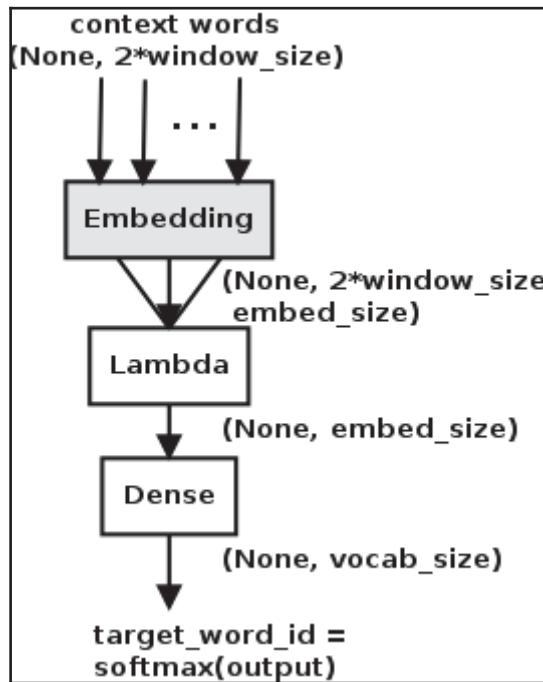
The CBOW word2vec model

Let us now look at the CBOW word2vec model. Recall that the CBOW model predicts the center word given the context words. Thus, in the first tuple in the following example, the CBOW model needs to predict the output word *love*, given the context words *I* and *green*:

$([I, \text{green}], \text{love}) ([\text{love}, \text{eggs}], \text{green}) ([\text{green}, \text{and}], \text{eggs}) \dots$

Like the skip-gram model, the CBOW model is also a classifier that takes the context words as input and predicts the target word. The architecture is somewhat more straightforward than the skip-gram model. The input to the model is the word IDs for the context words. These word IDs are fed into a common embedding layer that is initialized with small random weights. Each word ID is transformed into a vector of size (`embed_size`) by the embedding layer. Thus, each row of the input context is transformed into a matrix of size (`2*window_size, embed_size`) by this layer. This is then fed into a lambda layer, which computes an average of all the embeddings. This average is then fed to a dense layer, which creates a dense vector of size (`vocab_size`) for each row. The activation function on the dense layer is a softmax, which reports the maximum value on the output vector as a probability. The ID with the maximum probability corresponds to the target word.

The deliverable for the CBOW model is the weights from the embedding layer shown in gray in the following figure:



The corresponding Keras code for the model is shown as follows. Once again, assume a vocabulary size of 5000, an embedding size of 300, and a context window size of 1. Our first step is to set up all our imports and these values:

```
from keras.models import Sequential
from keras.layers.core import Dense, Lambda
from keras.layers.embeddings import Embedding
import keras.backend as K

vocab_size = 5000
embed_size = 300
window_size = 1
```

We then construct a sequential model, to which we add an embedding layer whose weights are initialized with small random values. Note that the `input_length` of this embedding layer is equal to the number of context words. So each context word is fed into this layer and will update the weights jointly during backpropagation. The output of this layer is a matrix of context word embeddings, which are averaged into a single vector (per row of input) by the lambda layer. Finally, the dense layer will convert each row into a dense vector of size (`vocab_size`). The target word is the one whose ID has the maximum value in the dense output vector:

```
model = Sequential()
model.add(Embedding(input_dim=vocab_size, output_dim=embed_size,
                     embeddings_initializer='glorot_uniform',
                     input_length=window_size*2))
model.add(Lambda(lambda x: K.mean(x, axis=1), output_shape=
                  (embed_size,)))
model.add(Dense(vocab_size, kernel_initializer='glorot_uniform',
                activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer="adam")
```

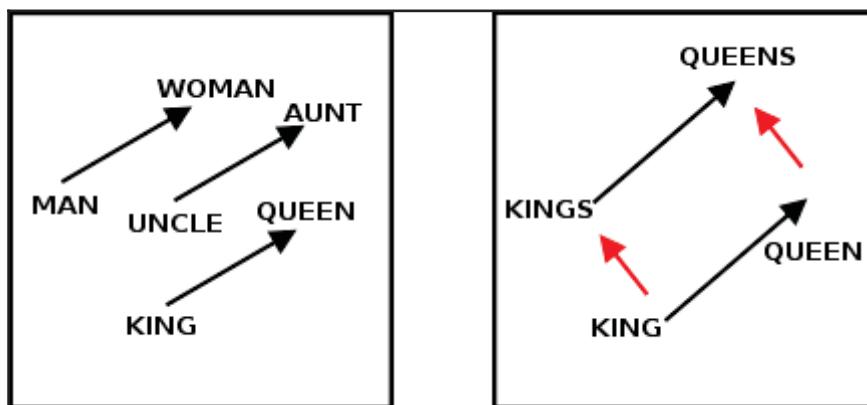
The loss function used here is `categorical_crossentropy`, which is a common choice for cases where there are two or more (in our case, `vocab_size`) categories.

The source code for the example can be found in the `keras_cbow.py` file in the source code download for the chapter.

Extracting word2vec embeddings from the model

As noted previously, even though both word2vec models can be reduced to a classification problem, we are not really interested in the classification problem itself. Rather, we are interested in the side effect of this classification process, that is, the weight matrix that transforms a word from the vocabulary to its dense, low-dimensional distributed representation.

There are many examples of how these distributed representations exhibit often surprising syntactic and semantic information. For example, as shown in the following figure from Tomas Mikolov's presentation at NIPS 2013 (for more information refer to the article: *Learning Representations of Text using Neural Networks*, by T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Q. Le, and T. Strohmann, NIPS 2013), vectors connecting words that have similar meanings but opposite genders are approximately parallel in the reduced 2D space, and we can often get very intuitive results by doing arithmetic with the word vectors. The presentation provides many other examples.



Intuitively, the training process imparts enough information to the internal encoding to predict an output word that occurs in the context of an input word. So points representing words shift in this space to be nearer to words with which it co-occurs. This causes similar words to clump together. Words that co-occur with these similar words also clump together in a similar way. As a result, vectors connecting points representing semantically related points tend to exhibit these regularities in the distributed representation.

Keras provides a way to extract weights from trained models. For the skip-gram example, the embedding weights can be extracted as follows:

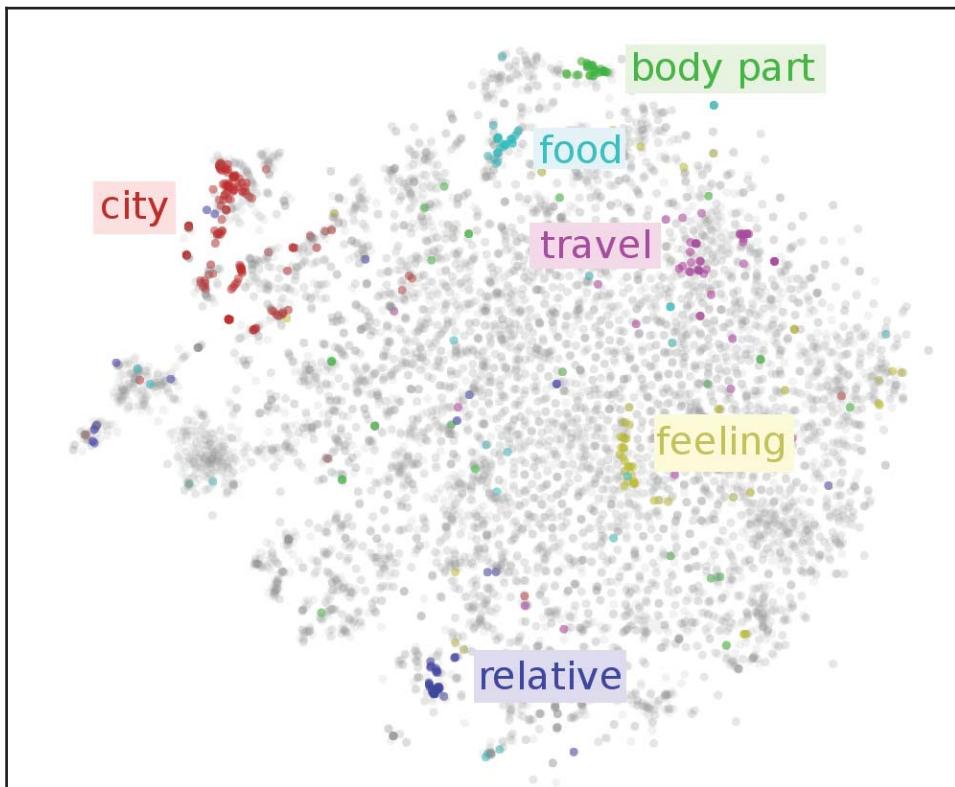
```
merge_layer = model.layers[0]
word_model = merge_layer.layers[0]
word_embed_layer = word_model.layers[0]
weights = word_embed_layer.get_weights()[0]
```

Similarly, the embedding weights for the CBOW example can be extracted using the following one-liner:

```
weights = model.layers[0].get_weights()[0]
```

In both cases, the shape of the weights matrix is `vocab_size` and `embed_size`. In order to compute the distributed representation for a word in the vocabulary, you will need to construct a one-hot vector by setting the position of the word index to one in a zero vector of size (`vocab_size`) and multiply it with the matrix to get the embedding vector of size (`embed_size`).

A visualization of word embeddings from work done by Christopher Olah (for more information refer to the article: *Document Embedding with Paragraph Vectors*, by Andrew M. Dai, Christopher Olah, and Quoc V. Le, arXiv:1507.07998, 2015) is shown as follows. This is a visualization of word embeddings reduced to two dimensions and visualized with T-SNE. The words forming entity types were chosen using WordNet synset clusters. As you can see, points corresponding to similar entity types tend to cluster together:



The source code for the example can be found in `keras_skipgram.py` in the source code download.

Using third-party implementations of word2vec

We have covered word2vec extensively over the past few sections. At this point, you understand how the skip-gram and CBOW models work and how to build your own implementation of these models using Keras. However, third-party implementations of word2vec are readily available, and unless your use case is very complex or different, it makes sense to just use one such implementation instead of rolling your own.

The gensim library provides an implementation of word2vec. Even though this is a book about Keras and not gensim, we include a discussion on this because Keras does not provide any support for word2vec, and integrating the gensim implementation into Keras code is very common practice.



Installation of gensim is fairly simple and described in detail on the gensim installation page (<https://radimrehurek.com/gensim/install.html>).

The following code shows how to build a word2vec model using gensim and train it with the text from the text8 corpus, available for download at: <http://mattmahoney.net/dc/text8.zip>. The text8 corpus is a file containing about 17 million words derived from Wikipedia text. Wikipedia text was cleaned to remove markup, punctuation, and non-ASCII text, and the first 100 million characters of this cleaned text became the text8 corpus. This corpus is commonly used as an example for word2vec because it is quick to train and produces good results. First we set up the imports as usual:

```
from gensim.models import KeyedVectors  
import logging  
import os
```

We then read in the words from the text8 corpus, and split up the words into sentences of 50 words each. The gensim library provides a built-in text8 handler that does something similar. Since we want to illustrate how to generate a model with any (preferably large) corpus that may or may not fit into memory, we will show you how to generate these sentences using a Python generator.

The `Text8Sentences` class will generate sentences of `maxlen` words each from the `text8` file. In this case, we do ingest the entire file into memory, but when traversing through directories of files, generators allows us to load parts of the data into memory at a time, process them, and yield them to the caller:

```
class Text8Sentences(object):
    def __init__(self, fname, maxlen):
        self.fname = fname
        self.maxlen = maxlen

    def __iter__(self):
        with open(os.path.join(DATA_DIR, "text8"), "rb") as ftext:
            text = ftext.read().split(" ")
            sentences, words = [], []
            for word in text:
                if len(words) >= self.maxlen:
                    yield words
                    words = []
                words.append(word)
            yield words
```

We then set up the caller code. The gensim word2vec uses Python logging to report on progress, so we first enable it. The next line declares an instance of the `Text8Sentences` class, and the line after that trains the model with the sentences from the dataset. We have chosen the size of the embedding vectors to be 300, and we only consider words that appear a minimum of 30 times in the corpus. The default window size is 5, so we will consider the words $w_{i-5}, w_{i-4}, w_{i-3}, w_{i-2}, w_{i-1}, w_{i+1}, w_{i+2}, w_{i+3}, w_{i+4}$, and w_{i+5} as the context for word w_i . By default, the word2vec model created is CBOW, but you can change that by setting `sg=1` in the parameters:

```
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',
level=logging.INFO)

DATA_DIR = ".../data/"
sentences = Text8Sentences(os.path.join(DATA_DIR, "text8"), 50)
model = word2vec.Word2Vec(sentences, size=300, min_count=30)
```

The word2vec implementation will make two passes over the data, first to generate a vocabulary and then to build the actual model. You can see its progress on the console as it runs:

```
2017-01-30 16:16:27,786 : INFO : PROGRESS: at 76.44% examples, 691859 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:28,801 : INFO : PROGRESS: at 77.74% examples, 693040 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:29,807 : INFO : PROGRESS: at 79.00% examples, 693746 words/s, in_qsize 2, out_qsize 0
2017-01-30 16:16:30,815 : INFO : PROGRESS: at 79.99% examples, 692107 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:31,819 : INFO : PROGRESS: at 80.03% examples, 682583 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:32,842 : INFO : PROGRESS: at 81.15% examples, 682090 words/s, in_qsize 1, out_qsize 0
2017-01-30 16:16:33,869 : INFO : PROGRESS: at 82.46% examples, 683117 words/s, in_qsize 0, out_qsize 1
2017-01-30 16:16:34,873 : INFO : PROGRESS: at 83.77% examples, 684403 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:35,882 : INFO : PROGRESS: at 85.02% examples, 685224 words/s, in_qsize 5, out_qsize 0
2017-01-30 16:16:36,884 : INFO : PROGRESS: at 86.36% examples, 686831 words/s, in_qsize 0, out_qsize 1
2017-01-30 16:16:37,925 : INFO : PROGRESS: at 87.51% examples, 686556 words/s, in_qsize 2, out_qsize 0
2017-01-30 16:16:38,925 : INFO : PROGRESS: at 88.57% examples, 685873 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:39,933 : INFO : PROGRESS: at 89.84% examples, 686756 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:40,936 : INFO : PROGRESS: at 91.17% examples, 688126 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:41,939 : INFO : PROGRESS: at 92.43% examples, 688894 words/s, in_qsize 0, out_qsize 1
2017-01-30 16:16:42,946 : INFO : PROGRESS: at 93.69% examples, 689612 words/s, in_qsize 1, out_qsize 0
2017-01-30 16:16:43,960 : INFO : PROGRESS: at 94.97% examples, 690484 words/s, in_qsize 1, out_qsize 0
2017-01-30 16:16:44,978 : INFO : PROGRESS: at 96.30% examples, 691348 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:45,982 : INFO : PROGRESS: at 97.58% examples, 692158 words/s, in_qsize 0, out_qsize 0
2017-01-30 16:16:46,980 : INFO : PROGRESS: at 98.83% examples, 692731 words/s, in_qsize 2, out_qsize 0
2017-01-30 16:16:48,092 : INFO : PROGRESS: at 99.92% examples, 691317 words/s, in_qsize 4, out_qsize 1
2017-01-30 16:16:48,124 : INFO : worker thread finished; awaiting finish of 2 more threads
2017-01-30 16:16:48,125 : INFO : worker thread finished; awaiting finish of 1 more threads
2017-01-30 16:16:48,128 : INFO : worker thread finished; awaiting finish of 0 more threads
2017-01-30 16:16:48,129 : INFO : training on 85026040 raw words (59645573 effective words) took 86.2s, 691572 effective words/s
2017-01-30 16:16:48,129 : INFO : precomputing L2-norms of word weight vectors
```

Once the model is created, we should normalize the resulting vectors. According to the documentation, this saves lots of memory. Once the model is trained, we can optionally save it to disk:

```
model.init_sims(replace=True)
model.save("word2vec_gensim.bin")
```

The saved model can be brought back into memory using the following call:

```
model = Word2Vec.load("word2vec_gensim.bin")
```

We can now query the model to find all the words it knows about:

```
>>> model.vocab.keys()[0:10]
['homomorphism',
 'woods',
 'spiders',
 'hanging',
```

```
'woody',
'localized',
'sprague',
'originality',
'alphabetic',
'hermann']
```

We can find the actual vector embedding for a given word:

```
>>> model["woman"]
array([-3.13099056e-01, -1.85702944e+00, 1.18816841e+00,
       -1.86561719e-01, -2.23673001e-01, 1.06527400e+00,
       &mldr;
      4.31755871e-01, -2.90115297e-01, 1.00955181e-01,
      -5.17173052e-01, 7.22485244e-01, -1.30940580e+00], dtype="float32")
```

We can also find words that are most similar to a certain word:

```
>>> model.most_similar("woman")
[('child', 0.7057571411132812),
 ('girl', 0.702182412147522),
 ('man', 0.6846336126327515),
 ('herself', 0.6292711496353149),
 ('lady', 0.6229539513587952),
 ('person', 0.6190367937088013),
 ('lover', 0.6062309741973877),
 ('baby', 0.5993420481681824),
 ('mother', 0.5954475402832031),
 ('daughter', 0.5871444940567017)]
```

We can provide hints for finding word similarity. For example, the following command returns the top 10 words that are like `woman` and `king` but unlike `man`:

```
>>> model.most_similar(positive=['woman', 'king'], negative=['man'],
topn=10)
[('queen', 0.6237582564353943),
 ('prince', 0.5638638734817505),
 ('elizabeth', 0.5557916164398193),
 ('princess', 0.5456407070159912),
 ('throne', 0.5439794063568115),
 ('daughter', 0.5364126563072205),
 ('empress', 0.5354889631271362),
 ('isabella', 0.5233952403068542),
 ('regent', 0.520746111869812),
 ('matilda', 0.5167444944381714)]
```

We can also find similarities between individual words. To give a feel of how the positions of the words in the embedding space correlates with their semantic meanings, let us look at the following word pairs:

```
>>> model.similarity("girl", "woman")
0.702182479574
>>> model.similarity("girl", "man")
0.574259909834
>>> model.similarity("girl", "car")
0.289332921793
>>> model.similarity("bus", "car")
0.483853497748
```

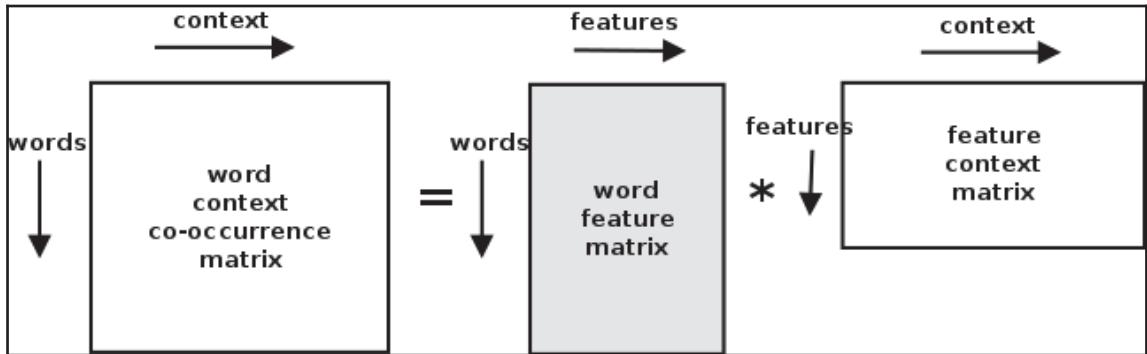
As you can see, girl and woman are more similar than girl and man, and car and bus are more similar than girl and car. This agrees very nicely with our human intuition about these words.

The source code for the example can be found in `word2vec_gensim.py` in the source code download.

Exploring GloVe

The global vectors for word representation, or GloVe, embeddings was created by Jeffrey Pennington, Richard Socher, and Christopher Manning (for more information refer to the article: *GloVe: Global Vectors for Word Representation*, by J. Pennington, R. Socher, and C. Manning, Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Pp. 1532–1543, 2013). The authors describe GloVe as an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space.

GloVe differs from word2vec in that word2vec is a predictive model while GloVe is a count-based model. The first step is to construct a large matrix of (word, context) pairs that co-occur in the training corpus. Each element of this matrix represents how often a word represented by the row co-occurs in the context (usually a sequence of words) represented by the column, as shown in the following figure:



The GloVe process converts the co-occurrence matrix into a pair of (word, feature) and (feature, context) matrices. This process is known as **matrix factorization** and is done using **stochastic gradient descent (SGD)**, an iterative numerical method. Rewriting in equation form:

$$\mathbf{R} = \mathbf{P} * \mathbf{Q} \approx \mathbf{R}'$$

Here, R is the original co-occurrence matrix. We first populate P and Q with random values and attempt to reconstruct a matrix R' by multiplying them. The difference between the reconstructed matrix R' and the original matrix R tells us how much we need to change the values of P and Q to move R' closer to R , to minimize the reconstruction error. This is repeated multiple times until the SGD converges and the reconstruction error is below a specified threshold. At that point, the (word, feature) matrix is the GloVe embedding. To speed up the process, SGD is often used in parallel mode, as outlined in the *HOGWILD!* paper.

One thing to note is that predictive neural network based models such as word2vec and count based models such as GloVe are very similar in intent. Both of them build a vector space where the position of a word is influenced by its neighboring words. Neural network models start with individual examples of word co-occurrences and count based models start with aggregate co-occurrence statistics between all words in the corpus. Several recent papers have demonstrated the correlation between these two types of model.

We will not cover generation of GloVe vectors in more detail in this book. Even though GloVe generally shows higher accuracy than word2vec and is faster to train if you use parallelization, Python tooling is not as mature as for word2vec. The only tool available to do this as of the time of writing is the GloVe-Python project (<https://github.com/maciejkula/glove-python>), which provides a toy implementation for GloVe on Python.

Using pre-trained embeddings

In general, you will train your own word2vec or GloVe model from scratch only if you have a very large amount of very specialized text. By far the most common use case for Embeddings is to use pre-trained embeddings in some way in your network. The three main ways in which you would use embeddings in your network are as follows:

- Learn embeddings from scratch
- Fine-tune learned embeddings from pre-trained GloVe/word2vec models
- Look up embeddings from pre-trained GloVe/word2vec models

In the first option, the embedding weights are initialized to small random values and trained using backpropagation. You saw this in the examples for skip-gram and CBOW models in Keras. This is the default mode when you use a Keras Embedding layer in your network.

In the second option, you build a weight matrix from a pre-trained model and initialize the weights of your embedding layer with this weight matrix. The network will update these weights using backpropagation, but the model will converge faster because of good starting weights.

The third option is to look up word embeddings from a pre-trained model, and transform your input to embedded vectors. You can then train any machine learning model (that is, not necessarily even a deep learning network) on the transformed data. If the pre-trained model is trained on a similar domain as the target domain, this usually works very well and is the least expensive option.

For general use with English language text, you can use Google's word2vec model trained over 10 billion words from the Google news dataset. The vocabulary size is about 3 million words and the dimensionality of the embedding is 300. The Google news model (about 1.5 GB) can be downloaded from here: <https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTT1SS21pQmM/edit?usp=sharing>.

Similarly, a pre-trained model trained on 6 billion tokens from English Wikipedia and the gigaword corpus can be downloaded from the GloVe site. The vocabulary size is about 400,000 words and the download provides vectors with dimensions 50, 100, 200, and 300. The model size is about 822 MB. Here is the direct download URL (<http://nlp.stanford.edu/data/glove.6B.zip>) for this model. Larger models based on the Common Crawl and Twitter are also available from the same location.

In the following sections, we will look at how to use these pre-trained models in the three ways listed.

Learn embeddings from scratch

In this example, we will train a one-dimensional **convolutional neural network (CNN)** to classify sentences as either positive or negative. You have already seen how to classify images using two-dimensional CNNs in Chapter 3, *Deep Learning with ConvNets*. Recall that CNNs exploit spatial structure in images by enforcing local connectivity between neurons of adjacent layers.

Words in sentences exhibit linear structure in the same way as images exhibit spatial structure. Traditional (non-deep learning) NLP approaches to language modeling involve creating word *n*-grams (<https://en.wikipedia.org/wiki/N-gram>) to exploit this linear structure inherent among words. One-dimensional CNNs do something similar, learning convolution filters that operate on sentences a few words at a time, and max pooling the results to create a vector that represents the most important ideas in the sentence.

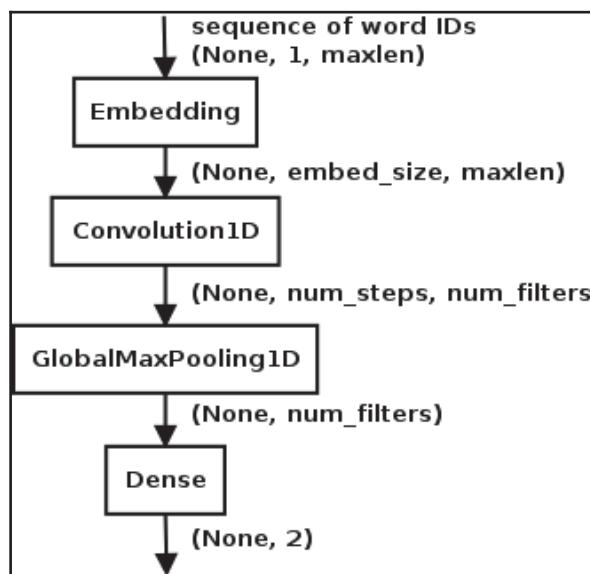
There is another class of neural network, called **recurrent neural network (RNN)**, which is specially designed to handle sequence data, including text, which is a sequence of words. The processing in RNNs is different from that in a CNN. We will learn about RNNs in a future chapter.

In our example network, the input text is converted to a sequence of word indices. Note that we have used the **natural language toolkit (NLTK)** to parse the text into sentences and words. We could also have used regular expressions to do this, but the statistical models supplied by NLTK are more powerful at parsing than regular expressions. If you are working with word embeddings, it is very likely that you are also working with NLP, in which case you would have NLTK installed already.



This link (<http://www.nltk.org/install.html>) has information to help you install NLTK on your machine. You will also need to install NLTK data, which is some trained corpora that comes standard with NLTK. Installation instructions for NLTK data are available here: <http://www.nltk.org/data.html>.

The sequence of word indices is fed into an array of embedding layers of a set size (in our case, the number of words in the longest sentence). The embedding layer is initialized by default to random values. The output of the embedding layer is connected to a 1D convolutional layer that convolves (in our example) word trigrams in 256 different ways (essentially, it applies different learned linear combinations of weights on the word embeddings). These features are then pooled into a single pooled word by a global max pooling layer. This vector (256) is then input to a dense layer, which outputs a vector (2). A softmax activation will return a pair of probabilities, one corresponding to positive sentiment and another corresponding to negative sentiment. The network is shown in the following figure:



Let us look at how to code this up using Keras. First we declare our imports. Right after the constants, you will notice that I set the `random.seed` value to 42. This is because we want consistent results between runs. Since the initializations of the weight matrices are random, differences in initialization can lead to differences in output, so this is a way to control that:

```
from keras.layers.core import Dense, Dropout, SpatialDropout1D
from keras.layers.convolutional import Conv1D
from keras.layers.embeddings import Embedding
from keras.layers.pooling import GlobalMaxPooling1D
from keras.models import Sequential
from keras.preprocessing.sequence import pad_sequences
from keras.utils import np_utils
from sklearn.model_selection import train_test_split
import collections
import matplotlib.pyplot as plt
import nltk
import numpy as np

np.random.seed(42)
```

We declare our constants. For all subsequent examples in this chapter, we will classify sentences from the UMICH SI650 sentiment classification competition on Kaggle. The dataset has around 7,000 sentences, and is labeled 1 for positive and 0 for negative. The `INPUT_FILE` defines the path to this file of sentences and labels. The format of the file is a sentiment label (0 or 1) followed by a tab, followed by a sentence.

The `VOCAB_SIZE` setting indicates that we will consider only the top 5,000 tokens in the text. The `EMBED_SIZE` setting is the size of the embedding that will be generated by the embedding layer in the network. `NUM_FILTERS` is the number of convolution filters we will train for our convolution layer, and `NUM_WORDS` is the size of each filter, that is, how many words we will convolve at a time. The `BATCH_SIZE` and `NUM_EPOCHS` is the number of records to feed the network each time and how many times we will run through the entire dataset during training:

```
INPUT_FILE = "../data/umich-sentiment-train.txt"
VOCAB_SIZE = 5000
EMBED_SIZE = 100
NUM_FILTERS = 256
NUM_WORDS = 3
BATCH_SIZE = 64
NUM_EPOCHS = 20
```

In the next block, we first read our input sentences and construct our vocabulary out of the most frequent words in the corpus. We then use this vocabulary to convert our input sentences into a list of word indices:

```
counter = collections.Counter()
fin = open(INPUT_FILE, "rb")
maxlen = 0
for line in fin:
    _, sent = line.strip().split("t")
    words = [x.lower() for x in nltk.word_tokenize(sent)]
    if len(words) > maxlen:
        maxlen = len(words)
    for word in words:
        counter[word] += 1
fin.close()

word2index = collections.defaultdict(int)
for wid, word in enumerate(counter.most_common(VOCAB_SIZE)):
    word2index[word[0]] = wid + 1
vocab_size = len(word2index) + 1
index2word = {v:k for k, v in word2index.items()}
```

We pad each of our sentences to predetermined length `maxlen` (in this case the number of words in the longest sentence in the training set). We also convert our labels to categorical format using a Keras utility function. The last two steps are a standard workflow for handling text input that we will see again and again:

```
xs, ys = [], []
fin = open(INPUT_FILE, "rb")
for line in fin:
    label, sent = line.strip().split("t")
    ys.append(int(label))
    words = [x.lower() for x in nltk.word_tokenize(sent)]
    wids = [word2index[word] for word in words]
    xs.append(wids)
fin.close()
X = pad_sequences(xs, maxlen=maxlen)
Y = np_utils.to_categorical(ys)
```

Finally, we split up our data into a 70/30 training and test set. The data is now in a form ready to be fed into the network:

```
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.3,
random_state=42)
```

We define the network that we described earlier in this section:

```
model = Sequential()
model.add(Embedding(vocab_size, EMBED_SIZE, input_length=maxlen))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(Conv1D(filters=NUM_FILTERS, kernel_size=NUM_WORDS,
activation="relu"))
model.add(GlobalMaxPooling1D())
model.add(Dense(2, activation="softmax"))
```

We then compile the model. Since our target is binary (positive or negative) we choose categorical_crossentropy as our loss function. For the optimizer, we choose adam. We then train the model using our training set, using a batch size of 64 and training for 20 epochs:

```
model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])
history = model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE,
epochs=NUM_EPOCHS,
validation_data=(Xtest, Ytest))
```

The output from the code looks as follows:

```
Epoch 9/20
4960/4960 [=====] - 3s - loss: 0.0337 - acc: 0.9855 - val_loss: 0.0263 - val_acc: 0.9882
Epoch 10/20
4960/4960 [=====] - 3s - loss: 0.0369 - acc: 0.9843 - val_loss: 0.0277 - val_acc: 0.9878
Epoch 11/20
4960/4960 [=====] - 3s - loss: 0.0331 - acc: 0.9881 - val_loss: 0.0303 - val_acc: 0.9878
Epoch 12/20
4960/4960 [=====] - 3s - loss: 0.0289 - acc: 0.9879 - val_loss: 0.0291 - val_acc: 0.9882
Epoch 13/20
4960/4960 [=====] - 3s - loss: 0.0261 - acc: 0.9901 - val_loss: 0.0305 - val_acc: 0.9878
Epoch 14/20
4960/4960 [=====] - 3s - loss: 0.0261 - acc: 0.9895 - val_loss: 0.0310 - val_acc: 0.9859
Epoch 15/20
4960/4960 [=====] - 3s - loss: 0.0355 - acc: 0.9857 - val_loss: 0.0307 - val_acc: 0.9873
Epoch 16/20
4960/4960 [=====] - 3s - loss: 0.0247 - acc: 0.9893 - val_loss: 0.0283 - val_acc: 0.9868
Epoch 17/20
4960/4960 [=====] - 3s - loss: 0.0249 - acc: 0.9891 - val_loss: 0.0329 - val_acc: 0.9854
Epoch 18/20
4960/4960 [=====] - 3s - loss: 0.0299 - acc: 0.9895 - val_loss: 0.0285 - val_acc: 0.9882
Epoch 19/20
4960/4960 [=====] - 3s - loss: 0.0282 - acc: 0.9887 - val_loss: 0.0287 - val_acc: 0.9882
Epoch 20/20
4960/4960 [=====] - 3s - loss: 0.0401 - acc: 0.9839 - val_loss: 0.0311 - val_acc: 0.9878

2126/2126 [=====] - 0s
Test score: 0.031, accuracy: 0.986
```

As you can see, the network gives us 98.6% accuracy on the test set.

The source code for this example can be found in `learn_embedding_from_scratch.py` in the source code download for the chapter.

Fine-tuning learned embeddings from word2vec

In this example, we will use the same network as the one we used to learn our embeddings from scratch. In terms of code, the only major difference is an extra block of code to load the word2vec model and build up the weight matrix for the embedding layer.

As always, we start with the imports and set up a random seed for repeatability. In addition to the imports we have seen previously, there is an additional one to import the word2vec model from gensim:

```
from gensim.models import KeyedVectors
from keras.layers.core import Dense, Dropout, SpatialDropout1D
from keras.layers.convolutional import Conv1D
from keras.layers.embeddings import Embedding
from keras.layers.pooling import GlobalMaxPooling1D
from keras.models import Sequential
from keras.preprocessing.sequence import pad_sequences
from keras.utils import np_utils
from sklearn.model_selection import train_test_split
import collections
import matplotlib.pyplot as plt
import nltk
import numpy as np

np.random.seed(42)
```

Next up is setting up the constants. The only difference here is that we reduced the `NUM_EPOCHS` setting from 20 to 10. Recall that initializing the matrix with values from a pre-trained model tends to set them to good values that converge faster:

```
INPUT_FILE = "../data/umich-sentiment-train.txt"
WORD2VEC_MODEL = "../data/GoogleNews-vectors-negative300.bin.gz"
VOCAB_SIZE = 5000
EMBED_SIZE = 300
NUM_FILTERS = 256
NUM_WORDS = 3
BATCH_SIZE = 64
NUM_EPOCHS = 10
```

The next block extracts the words from the dataset and creates a vocabulary of the most frequent terms, then parses the dataset again to create a list of padded word lists. It also converts the labels to categorical format. Finally, it splits the data into a training and a test set. This block is identical to the previous example and has been explained in depth there:

```
counter = collections.Counter()
fin = open(INPUT_FILE, "rb")
maxlen = 0
for line in fin:
    _, sent = line.strip().split("t")
    words = [x.lower() for x in nltk.word_tokenize(sent)]
    if len(words) > maxlen:
        maxlen = len(words)
    for word in words:
        counter[word] += 1
fin.close()

word2index = collections.defaultdict(int)
for wid, word in enumerate(counter.most_common(VOCAB_SIZE)):
    word2index[word[0]] = wid + 1
vocab_sz = len(word2index) + 1
index2word = {v:k for k, v in word2index.items()}

xs, ys = [], []
fin = open(INPUT_FILE, "rb")
for line in fin:
    label, sent = line.strip().split("t")
    ys.append(int(label))
    words = [x.lower() for x in nltk.word_tokenize(sent)]
    wids = [word2index[word] for word in words]
    xs.append(wids)
fin.close()
X = pad_sequences(xs, maxlen=maxlen)
Y = np_utils.to_categorical(ys)

Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.3,
                                                random_state=42)
```

The next block loads up the word2vec model from a pre-trained model. This model is trained with about 10 billion words of Google News articles and has a vocabulary size of 3 million. We load it and look up embedding vectors from it for words in our vocabulary, and write out the embedding vector into our weight matrix `embedding_weights`. Rows of this weight matrix correspond to words in the vocabulary, and columns of each row constitute the embedding vector for the word.

The dimensions of the `embedding_weights` matrix is `vocab_sz` and `EMBED_SIZE`. The `vocab_sz` is one more than the maximum number of unique terms in the vocabulary, the additional pseudo-token `_UNK_` representing words that are not seen in the vocabulary.

Note that it is possible that some words in our vocabulary may not be there in the Google News word2vec model, so when we encounter such words, the embedding vectors for them remain at the default value of all zeros:

```
# load word2vec model
word2vec = Word2Vec.load_word2vec_format(WORD2VEC_MODEL, binary=True)
embedding_weights = np.zeros((vocab_sz, EMBED_SIZE))
for word, index in word2index.items():
    try:
        embedding_weights[index, :] = word2vec[word]
    except KeyError:
        pass
```

We define our network. The difference in this block from our previous example is that we initialize the weights of the embedding layer with the `embedding_weights` matrix we built in the previous block:

```
model = Sequential()
model.add(Embedding(vocab_sz, EMBED_SIZE, input_length=maxlen,
                    weights=[embedding_weights]))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(Conv1D(filters=NUM_FILTERS, kernel_size=NUM_WORDS,
                 activation="relu"))
model.add(GlobalMaxPooling1D())
model.add(Dense(2, activation="softmax"))
```

We then compile our model with the categorical cross-entropy loss function and the Adam optimizer, and train the network with batch size 64 and for 10 epochs, then evaluate the trained model:

```
model.compile(optimizer="adam", loss="categorical_crossentropy",
              metrics=["accuracy"])
history = model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE,
                     epochs=NUM_EPOCHS,
                     validation_data=(Xtest, Ytest))

score = model.evaluate(Xtest, Ytest, verbose=1)
print("Test score: {:.3f}, accuracy: {:.3f}".format(score[0], score[1]))
```

Output from running the code is shown as follows:

```
((4960, 42), (2126, 42), (4960, 2), (2126, 2))
Train on 4960 samples, validate on 2126 samples
Epoch 1/10
4960/4960 [=====] - 7s - loss: 0.1766 - acc:
0.9369 - val_loss: 0.0397 - val_acc: 0.9854
Epoch 2/10
4960/4960 [=====] - 7s - loss: 0.0725 - acc:
0.9706 - val_loss: 0.0346 - val_acc: 0.9887
Epoch 3/10
4960/4960 [=====] - 7s - loss: 0.0553 - acc:
0.9784 - val_loss: 0.0210 - val_acc: 0.9915
Epoch 4/10
4960/4960 [=====] - 7s - loss: 0.0519 - acc:
0.9790 - val_loss: 0.0241 - val_acc: 0.9934
Epoch 5/10
4960/4960 [=====] - 7s - loss: 0.0576 - acc:
0.9746 - val_loss: 0.0219 - val_acc: 0.9929
Epoch 6/10
4960/4960 [=====] - 7s - loss: 0.0515 - acc:
0.9764 - val_loss: 0.0185 - val_acc: 0.9929
Epoch 7/10
4960/4960 [=====] - 7s - loss: 0.0528 - acc:
0.9790 - val_loss: 0.0204 - val_acc: 0.9920
Epoch 8/10
4960/4960 [=====] - 7s - loss: 0.0373 - acc:
0.9849 - val_loss: 0.0221 - val_acc: 0.9934
Epoch 9/10
4960/4960 [=====] - 7s - loss: 0.0360 - acc:
0.9845 - val_loss: 0.0194 - val_acc: 0.9929
Epoch 10/10
4960/4960 [=====] - 7s - loss: 0.0389 - acc:
0.9853 - val_loss: 0.0254 - val_acc: 0.9915
2126/2126 [=====] - 1s
Test score: 0.025, accuracy: 0.993
```

The model gives us an accuracy of 99.3% on the test set after 10 epochs of training. This is an improvement over the previous example, where we got an accuracy of 98.6% accuracy after 20 epochs.

The source code for this example can be found in `finetune_word2vec_embeddings.py` in the source code download for the chapter.

Fine-tune learned embeddings from GloVe

Fine tuning using pre-trained GloVe embeddings is very similar to fine tuning using pre-trained word2vec embeddings. In fact, all of the code, except for the block that builds the weight matrix for the embedding layer, is identical. Since we have already seen this code twice, I will just focus on the block of code that builds the weight matrix from the GloVe embeddings.

GloVe embeddings come in various flavors. We use the model pre-trained on 6 billion tokens from the English Wikipedia and the gigaword corpus. The vocabulary size for the model is about 400,000, and the download provides vectors of dimensions 50, 100, 200, and 300. We will use embeddings from the 300 dimensional model.

The only thing we need to change in the code for the previous example is to replace the block that instantiated a word2vec model and loaded the embedding matrix using the following block of code. If we use a model with vector size other than 300, then we also need to update EMBED_SIZE.

The vectors are provided in space-delimited text format, so the first step is to read the code into a dictionary, `word2emb`. This is analogous to the line instantiating the Word2Vec model in our previous example:

```
GLOVE_MODEL = "../data/glove.6B.300d.txt"
word2emb = {}
fglove = open(GLOVE_MODEL, "rb")
for line in fglove:
    cols = line.strip().split()
    word = cols[0]
    embedding = np.array(cols[1:], dtype="float32")
    word2emb[word] = embedding
fglove.close()
```

We then instantiate an embedding weight matrix of size (`vocab_sz` and `EMBED_SIZE`) and populate the vectors from the `word2emb` dictionary. Vectors for words that are found in the vocabulary but not in the GloVe model remain set to all zeros:

```
embedding_weights = np.zeros((vocab_sz, EMBED_SIZE))
for word, index in word2index.items():
    try:
        embedding_weights[index, :] = word2emb[word]
    except KeyError:
        pass
```

The full code for this program can be found in `finetune_glove_embeddings.py` in the book's code repository on GitHub. The output of the run is shown as follows:

```
((4960, 42), (2126, 42), (4960, 2), (2126, 2))
Train on 4960 samples, validate on 2126 samples
Epoch 1/10
4960/4960 [=====] - 7s - loss: 0.1748 - acc: 0.9240 - val_loss: 0.0390 - val_acc: 0.9840
Epoch 2/10
4960/4960 [=====] - 7s - loss: 0.0859 - acc: 0.9649 - val_loss: 0.0431 - val_acc: 0.9845
Epoch 3/10
4960/4960 [=====] - 7s - loss: 0.0586 - acc: 0.9754 - val_loss: 0.0528 - val_acc: 0.9779
Epoch 4/10
4960/4960 [=====] - 8s - loss: 0.0565 - acc: 0.9798 - val_loss: 0.0386 - val_acc: 0.9873
Epoch 5/10
4960/4960 [=====] - 8s - loss: 0.0792 - acc: 0.9683 - val_loss: 0.0233 - val_acc: 0.9892
Epoch 6/10
4960/4960 [=====] - 8s - loss: 0.0618 - acc: 0.9746 - val_loss: 0.0247 - val_acc: 0.9911
Epoch 7/10
4960/4960 [=====] - 7s - loss: 0.0569 - acc: 0.9752 - val_loss: 0.0266 - val_acc: 0.9906
Epoch 8/10
4960/4960 [=====] - 8s - loss: 0.0419 - acc: 0.9829 - val_loss: 0.0211 - val_acc: 0.9920
Epoch 9/10
4960/4960 [=====] - 7s - loss: 0.0371 - acc: 0.9849 - val_loss: 0.0206 - val_acc: 0.9920
Epoch 10/10
4960/4960 [=====] - 9s - loss: 0.0422 - acc: 0.9815 - val_loss: 0.0266 - val_acc: 0.9906
2126/2126 [=====] - 1s
Test score: 0.027, accuracy: 0.991
```

This gives us 99.1% accuracy in 10 epochs, which is almost as good as the results we got from fine-tuning the network using word2vec `embedding_weights`.

The source code for this example can be found in `finetune_glove_embeddings.py` in the source code download for this chapter.

Look up embeddings

Our final strategy is to look up embeddings from pre-trained networks. The simplest way to do this with the current examples is to just set the `trainable` parameter of the embedding layer to `False`. This ensures that backpropagation will not update the weights on the embedding layer:

```
model.add(Embedding(vocab_sz, EMBED_SIZE, input_length=maxlen,
                     weights=[embedding_weights],
                     trainable=False))
model.add(SpatialDropout1D(Dropout(0.2)))
```

Setting this value with the word2vec and GloVe examples gave us accuracies of 98.7% and 98.9% respectively after 10 epochs of training.

However, in general, this is not how you would use pre-trained embeddings in your code. Typically, it involves preprocessing your dataset to create word vectors by looking up words in one of the pre-trained models, and then using this data to train some other model. The second model would not contain an Embedding layer, and may not even be a deep learning network.

The following example describes a dense network that takes as its input a vector of size 100, representing a sentence, and outputs a 1 or 0 for positive or negative sentiment. Our dataset is still the one from the UMICH S1650 sentiment classification competition with around 7,000 sentences.

As previously, large parts of the code are repeated, so we only explain the parts that are new or otherwise need explanation.

We begin with the imports, set the random seed for repeatability, and set some constant values. In order to create the 100-dimensional vectors for each sentence, we add up the GloVe 100-dimensional vectors for the words in the sentence, so we choose the `glove.6B.100d.txt` file:

```
from keras.layers.core import Dense, Dropout, SpatialDropout1D
from keras.models import Sequential
from keras.preprocessing.sequence import pad_sequences
from keras.utils import np_utils
from sklearn.model_selection import train_test_split
import collections
import matplotlib.pyplot as plt
import nltk
import numpy as np

np.random.seed(42)

INPUT_FILE = "../data/umich-sentiment-train.txt"
GLOVE_MODEL = "../data/glove.6B.100d.txt"
VOCAB_SIZE = 5000
EMBED_SIZE = 100
BATCH_SIZE = 64
NUM_EPOCHS = 10
```

The next block reads the sentences and creates a word frequency table. From this, the most common 5,000 tokens are selected and lookup tables (from word to word index and back) are created. In addition, we create a pseudo-token `_UNK_` for tokens that do not exist in the vocabulary. Using these lookup tables, we convert each sentence to a sequence of word IDs, padding these sequences so that all sequences are of the same length (the maximum number of words in a sentence in the training set). We also convert the labels to categorical format:

```
counter = collections.Counter()
fin = open(INPUT_FILE, "rb")
maxlen = 0
for line in fin:
    _, sent = line.strip().split("t")
    words = [x.lower() for x in nltk.word_tokenize(sent)]
    if len(words) > maxlen:
        maxlen = len(words)
    for word in words:
        counter[word] += 1
fin.close()

word2index = collections.defaultdict(int)
for wid, word in enumerate(counter.most_common(VOCAB_SIZE)):
    word2index[word[0]] = wid + 1
vocab_sz = len(word2index) + 1
index2word = {v:k for k, v in word2index.items()}
index2word[0] = "_UNK_"

ws, ys = [], []
fin = open(INPUT_FILE, "rb")
for line in fin:
    label, sent = line.strip().split("t")
    ys.append(int(label))
    words = [x.lower() for x in nltk.word_tokenize(sent)]
    wids = [word2index[word] for word in words]
    ws.append(wids)
fin.close()
W = pad_sequences(ws, maxlen=maxlen)
Y = np_utils.to_categorical(ys)
```

We load the GloVe vectors into a dictionary. If we wanted to use word2vec here, all we have to do is replace this block with a `gensim Word2Vec.load_word2vec_format()` call and replace the following block to look up the word2vec model instead of the word2emb dictionary:

```
word2emb = collections.defaultdict(int)
fglove = open(GLOVE_MODEL, "rb")
for line in fglove:
```

```
cols = line.strip().split()
word = cols[0]
embedding = np.array(cols[1:], dtype="float32")
word2emb[word] = embedding
fglove.close()
```

The next block looks up the words for each sentence from the word ID matrix W and populates a matrix E with the corresponding embedding vector. These embedding vectors are then added to create a sentence vector, which is written back into the X matrix. The output of this code block is the matrix X of size (num_records and EMBED_SIZE):

```
X = np.zeros((W.shape[0], EMBED_SIZE))
for i in range(W.shape[0]):
    E = np.zeros((EMBED_SIZE, maxlen))
    words = [index2word[wid] for wid in W[i].tolist()]
    for j in range(maxlen):
        E[:, j] = word2emb[words[j]]
    X[i, :] = np.sum(E, axis=1)
```

We have now preprocessed our data using the pre-trained model and are ready to use it to train and evaluate our final model. Let us split the data into 70/30 training/test as usual:

```
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.3,
random_state=42)
```

The network we will train for doing the sentiment analysis task is a simple dense network. We compile it with a categorical cross-entropy loss function and the Adam optimizer, and train it with the sentence vectors that we built out of the pre-trained embeddings. Finally, we evaluate the model on the 30% test set:

```
model = Sequential()
model.add(Dense(32, input_dim=100, activation="relu"))
model.add(Dropout(0.2))
model.add(Dense(2, activation="softmax"))

model.compile(optimizer="adam", loss="categorical_crossentropy",
metrics=["accuracy"])
history = model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE,
                     epochs=NUM_EPOCHS,
                     validation_data=(Xtest, Ytest))

score = model.evaluate(Xtest, Ytest, verbose=1)
print("Test score: {:.3f}, accuracy: {:.3f}".format(score[0], score[1]))
```

The output for the code using GloVe embeddings is shown as follows:

```
((4960, 100), (2126, 100), (4960, 2), (2126, 2))
Train on 4960 samples, validate on 2126 samples
Epoch 1/10
4960/4960 [=====] - 0s - loss: 1.9577 - acc: 0.5667 - val_loss: 0.4448 - val_acc: 0.8556
Epoch 2/10
4960/4960 [=====] - 0s - loss: 0.5245 - acc: 0.7942 - val_loss: 0.3167 - val_acc: 0.9078
Epoch 3/10
4960/4960 [=====] - 0s - loss: 0.3026 - acc: 0.9002 - val_loss: 0.2456 - val_acc: 0.9473
Epoch 4/10
4960/4960 [=====] - 0s - loss: 0.2338 - acc: 0.9270 - val_loss: 0.2068 - val_acc: 0.9398
Epoch 5/10
4960/4960 [=====] - 0s - loss: 0.1802 - acc: 0.9520 - val_loss: 0.1720 - val_acc: 0.9581
Epoch 6/10
4960/4960 [=====] - 0s - loss: 0.1561 - acc: 0.9552 - val_loss: 0.1561 - val_acc: 0.9610
Epoch 7/10
4960/4960 [=====] - 0s - loss: 0.1396 - acc: 0.9631 - val_loss: 0.1535 - val_acc: 0.9577
Epoch 8/10
4960/4960 [=====] - 0s - loss: 0.1216 - acc: 0.9645 - val_loss: 0.1338 - val_acc: 0.9628
Epoch 9/10
4960/4960 [=====] - 0s - loss: 0.1152 - acc: 0.9641 - val_loss: 0.1273 - val_acc: 0.9643
Epoch 10/10
4960/4960 [=====] - 0s - loss: 0.1044 - acc: 0.9706 - val_loss: 0.1257 - val_acc: 0.9647

1888/2126 [=====>....] - ETA: 0s
Test score: 0.126, accuracy: 0.965
```

The dense network gives us 96.5% accuracy on the test set after 10 epochs of training when preprocessed with the 100-dimensional GloVe embeddings. With preprocessed with the word2vec embeddings (300-dimensional fixed) the network gives us 98.5% on the test set.

The source code for this example can be found in `transfer_glove_embeddings.py` (for the GloVe example) and `transfer_word2vec_embeddings.py` (for the word2vec example) in the source code download for the chapter.

Summary

In this chapter, we learned how to transform words in text into vector embeddings that retain the distributional semantics of the word. We also now have an intuition of why word embeddings exhibit this kind of behavior and why word embeddings are useful for working with deep learning models for text data.

We then looked at two popular word embedding schemes, word2vec and GloVe, and understood how these models work. We also looked at using gensim to train our own word2vec model from data.

Finally, we learned about different ways of using embeddings in our network. The first was to learn embeddings from scratch as part of training our network. The second was to import embedding weights from pre-trained word2vec and GloVe models into our networks and fine-tune them as we train the network. The third was to use these pre-trained weights as is in our downstream applications.

In the next chapter, we will learn about recurrent neural networks, a class of network that is optimized for handling sequence data such as text.

6

Recurrent Neural Network — RNN

In Chapter 3, *Deep Learning with ConvNets*, we learned about **convolutional neural networks** (CNN) and saw how they exploit the spatial geometry of their input. For example, CNNs apply convolution and pooling operations in one dimension for audio and text data along the time dimension, in two dimensions for images along the (height x width) dimensions and in three dimensions, for videos along the (height x width x time) dimensions.

In this chapter, we will learn about **recurrent neural networks** (RNN), a class of neural networks that exploit the sequential nature of their input. Such inputs could be text, speech, time series, and anything else where the occurrence of an element in the sequence is dependent on the elements that appeared before it. For example, the next word in the sentence *the dog...* is more likely to be *barks* than *car*, therefore, given such a sequence, an RNN is more likely to predict *barks* than *car*.

An RNN can be thought of as a graph of RNN cells, where each cell performs the same operation on every element in the sequence. RNNs are very flexible and have been used to solve problems such as speech recognition, language modeling, machine translation, sentiment analysis, and image captioning, to name a few. RNNs can be adapted to different types of problems by rearranging the way the cells are arranged in the graph. We will see some examples of these configurations and how they are used to solve specific problems.

We will also learn about a major limitation of the SimpleRNN cell, and how two variants of the SimpleRNN cell—**long short term memory (LSTM)** and **gated recurrent unit (GRU)**—overcome this limitation. Both LSTM and GRU are drop-in replacements for the SimpleRNN cell, so just replacing the RNN cell with one of these variants can often result in a major performance improvement in your network. While LSTM and GRU are not the only variants, it has been shown empirically (for more information refer to the articles: *An Empirical Exploration of Recurrent Network Architectures*, by R. Jozefowicz, W. Zaremba, and I. Sutskever, JMLR, 2015 and *LSTM: A Search Space Odyssey*, by K. Greff, arXiv:1503.04069, 2015) that they are the best choices for most sequence problems.

Finally, we will also learn about some tips to improve the performance of our RNNs and when and how to apply them.

In this chapter, we will cover the following topics:

- SimpleRNN cell
- Basic RNN implementation in Keras in generating text
- RNN topologies
- LSTM, GRU, and other RNN variants

SimpleRNN cells

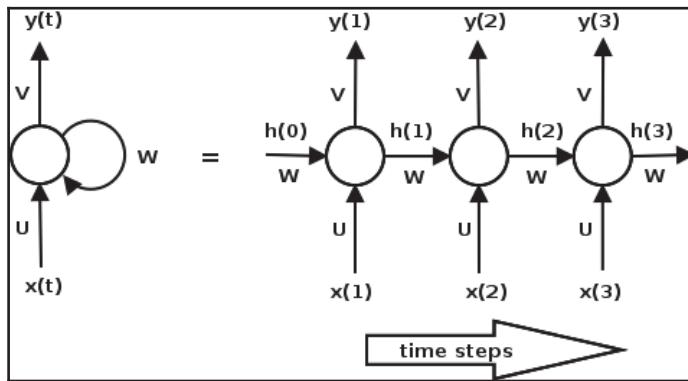
Traditional multilayer perceptron neural networks make the assumption that all inputs are independent of each other. This assumption breaks down in the case of sequence data. You have already seen the example in the previous section where the first two words in the sentence affect the third. The same idea is true of speech—if we are having a conversation in a noisy room, I can make reasonable guesses about a word I may not have understood based on the words I have heard so far. Time series data, such as stock prices or weather, also exhibit a dependence on past data, called the secular trend.

RNN cells incorporate this dependence by having a hidden state, or memory, that holds the essence of what has been seen so far. The value of the hidden state at any point in time is a function of the value of the hidden state at the previous time step and the value of the input at the current time step, that is:

$$\mathbf{h}_t = \phi(\mathbf{h}_{t-1}, \mathbf{x}_t)$$

h_t and h_{t-1} are the values of the hidden states at the time steps t and $t-1$ respectively, and x_t is the value of the input at time t . Notice that the equation is recursive, that is, h_{t-1} can be represented in terms of h_{t-2} and x_{t-1} , and so on, until the beginning of the sequence. This is how RNNs encode and incorporate information from arbitrarily long sequences.

We can also represent the RNN cell graphically as shown in the following diagram on the left. At time t , the cell has an input x_t and an output y_t . Part of the output y_t (the hidden state h_t) is fed back into the cell for use at a later time step $t+1$. Just as a traditional neural network's parameters are contained in its weight matrix, the RNN's parameters are defined by three weight matrices U , V , and W , corresponding to the input, output, and hidden state respectively:



Another way to look at an RNN to *unroll* it, as shown in the preceding diagram on the right. Unrolling means that we draw the network out for the complete sequence. The network shown here is a three-layer RNN, suitable for processing three element sequences. Notice that the weight matrices U , V , and W are shared across the steps. This is because we are applying the same operation on different inputs at each time step. Being able to share these weight vectors across all the time steps greatly reduces the number of parameters that the RNN needs to learn.

We can also describe the computations within an RNN in terms of equations. The internal state of the RNN at a time t is given by the value of the hidden vector h_t , which is the sum of the product of the weight matrix W and the hidden state h_{t-1} at time $t-1$ and the product of the weight matrix U and the input x_t at time t , passed through the *tanh* nonlinearity. The choice of *tanh* over other nonlinearities has to do with its second derivative decaying very slowly to zero. This keeps the gradients in the linear region of the activation function and helps combat the vanishing gradient problem. We will learn more about the vanishing gradient problem later in this chapter.

The output vector y_t at time t is the product of the weight matrix V and the hidden state h_t , with *softmax* applied to the product so the resulting vector is a set of output probabilities:

$$\begin{aligned} h_t &= \tanh(Wh_{t-1} + Ux_t) \\ y_t &= \text{softmax}(Vh_t) \end{aligned}$$

Keras provides the SimpleRNN (for more information refer to: <https://keras.io/layers/recurrent/>) recurrent layer that incorporates all the logic we have seen so far, as well as the more advanced variants such as LSTM and GRU that we will see later in this chapter, so it is not strictly necessary to understand how they work in order to start building with them. However, an understanding of the structure and equations is helpful when you need to compose your own RNN to solve a given problem.

SimpleRNN with Keras — generating text

RNNs have been used extensively by the **natural language processing (NLP)** community for various applications. One such application is building language models. A language model allows us to predict the probability of a word in a text given the previous words. Language models are important for various higher level tasks such as machine translation, spelling correction, and so on.

A side effect of the ability to predict the next word given previous words is a generative model that allows us to generate text by sampling from the output probabilities. In language modeling, our input is typically a sequence of words and the output is a sequence of predicted words. The training data used is existing unlabeled text, where we set the label y_t at time t to be the input x_{t+1} at time $t+1$.

For our first example of using Keras for building RNNs, we will train a character based language model on the text of *Alice in Wonderland* to predict the next character given 10 previous characters. We have chosen to build a character-based model here because it has a smaller vocabulary and trains quicker. The idea is the same as using a word-based language model, except we use characters instead of words. We will then use the trained model to generate some text in the same style.

First we import the necessary modules:

```
from __future__ import print_function
from keras.layers import Dense, Activation
from keras.layers.recurrent import SimpleRNN
from keras.models import Sequential
from keras.utils.visualize_util import plot
import numpy as np
```

We read our input text from the text of *Alice in Wonderland* on the Project Gutenberg website (<http://www.gutenberg.org/files/11/11-0.txt>). The file contains line breaks and non-ASCII characters, so we do some preliminary cleanup and write out the contents into a variable called `text`:

```
fin = open("../data/alice_in_wonderland.txt", 'rb')
lines = []
for line in fin:
    line = line.strip().lower()
    line = line.decode("ascii", "ignore")
    if len(line) == 0:
        continue
    lines.append(line)
fin.close()
text = " ".join(lines)
```

Since we are building a character-level RNN, our vocabulary is the set of characters that occur in the text. There are 42 of them in our case. Since we will be dealing with the indexes to these characters rather than the characters themselves, the following code snippet creates the necessary lookup tables:

```
chars = set([c for c in text])
nb_chars = len(chars)
char2index = dict((c, i) for i, c in enumerate(chars))
index2char = dict((i, c) for i, c in enumerate(chars))
```

The next step is to create the input and label texts. We do this by stepping through the text by a number of characters given by the `STEP` variable (1 in our case) and then extracting a span of text whose size is determined by the `SEQLEN` variable (10 in our case). The next character after the span is our label character:

```
SEQLEN = 10
STEP = 1

input_chars = []
label_chars = []
for i in range(0, len(text) - SEQLEN, STEP):
    input_chars.append(text[i:i + SEQLEN])
    label_chars.append(text[i + SEQLEN])
```

Using the preceding code, the input and label texts for the text `it turned into a pig` would look like this:

```
it turned -> i
t turned i -> n
turned in -> t
turned int -> o
```

```
urned into ->
rned into -> a
ned into a ->
ed into a -> p
d into a p -> i
into a pi -> g
```

The next step is to vectorize these input and label texts. Each row of the input to the RNN corresponds to one of the input texts shown previously. There are SEQLEN characters in this input, and since our vocabulary size is given by nb_chars, we represent each input character as a one-hot encoded vector of size (nb_chars). Thus each input row is a tensor of size (SEQLEN and nb_chars). Our output label is a single character, so similar to the way we represent each character of our input, it is represented as a one-hot vector of size (nb_chars). Thus, the shape of each label is nb_chars:

```
x = np.zeros((len(input_chars), SEQLEN, nb_chars), dtype=np.bool)
y = np.zeros((len(input_chars), nb_chars), dtype=np.bool)
for i, input_char in enumerate(input_chars):
    for j, ch in enumerate(input_char):
        X[i, j, char2index[ch]] = 1
        y[i, char2index[label_chars[i]]] = 1
```

Finally, we are ready to build our model. We define the RNN's output dimension to have a size of 128. This is a hyper-parameter that needs to be determined by experimentation. In general, if we choose too small a size, then the model does not have sufficient capacity for generating good text, and you will see long runs of repeating characters or runs of repeating word groups. On the other hand, if the value chosen is too large, the model has too many parameters and needs a lot more data to train effectively. We want to return a single character as output, not a sequence of characters, so `return_sequences=False`. We have already seen that the input to the RNN is of shape (SEQLEN and nb_chars). In addition, we set `unroll=True` because it improves performance on the TensorFlow backend.

The RNN is connected to a dense (fully connected) layer. The dense layer has (nb_char) units, which emits scores for each of the characters in the vocabulary. The activation on the dense layer is a softmax, which normalizes the scores to probabilities. The character with the highest probability is chosen as the prediction. We compile the model with the categorical cross-entropy loss function, a good loss function for categorical outputs, and the RMSprop optimizer:

```
HIDDEN_SIZE = 128
BATCH_SIZE = 128
NUM_ITERATIONS = 25
NUM_EPOCHS_PER_ITERATION = 1
NUM_PREDs_PER_EPOCH = 100
```

```
model = Sequential()
model.add(SimpleRNN(HIDDEN_SIZE, return_sequences=False,
    input_shape=(SEQLEN, nb_chars),
    unroll=True))
model.add(Dense(nb_chars))
model.add(Activation("softmax"))

model.compile(loss="categorical_crossentropy", optimizer="rmsprop")
```

Our training approach is a little different from what we have seen so far. So far our approach has been to train a model for a fixed number of epochs, then evaluate it against a portion of held-out test data. Since we don't have any labeled data here, we train the model for an epoch (NUM_EPOCHS_PER_ITERATION=1) then test it. We continue training like this for 25 (NUM_ITERATIONS=25) iterations, stopping once we see intelligible output. So effectively, we are training for NUM_ITERATIONS epochs and testing the model after each epoch.

Our test consists of generating a character from the model given a random input, then dropping the first character from the input and appending the predicted character from our previous run, and generating another character from the model. We continue this 100 times (NUM_PREDS_PER_EPOCH=100) and generate and print the resulting string. The string gives us an indication of the quality of the model:

```
for iteration in range(NUM_ITERATIONS):
    print("=" * 50)
    print("Iteration #: %d" % (iteration))
    model.fit(X, y, batch_size=BATCH_SIZE, epochs=NUM_EPOCHS_PER_ITERATION)

    test_idx = np.random.randint(len(input_chars))
    test_chars = input_chars[test_idx]
    print("Generating from seed: %s" % (test_chars))
    print(test_chars, end="")
    for i in range(NUM_PREDS_PER_EPOCH):
        Xtest = np.zeros((1, SEQLEN, nb_chars))
        for i, ch in enumerate(test_chars):
            Xtest[0, i, char2index[ch]] = 1
        pred = model.predict(Xtest, verbose=0)[0]
        ypred = index2char[np.argmax(pred)]
        print(ypred, end="")
        # move forward with test_chars + ypred
        test_chars = test_chars[1:] + ypred
    print()
```

The output of this run is shown as follows. As you can see, the model starts out predicting gibberish, but by the end of the 25th epoch, it has learned to spell reasonably well, although it has trouble expressing coherent thoughts. The amazing thing about this model is that it is character-based and has no knowledge of words, yet it learns to spell words that look like they might have come from the original text:

```
=====
Iteration #: 21
Epoch 1/1
142544/142544 [=====] - 10s - loss: 1.3916
Generating from seed: e with the
e with the white rabbit had no the that the mouse the mouse the mouse the mouse the mouse the mouse
=====
Iteration #: 22
Epoch 1/1
142544/142544 [=====] - 10s - loss: 1.3831
Generating from seed: and an ol
and an ollar the caterpillar the seapped did not a moment the cook of the courter the caterpillar the seapped
=====
Iteration #: 23
Epoch 1/1
142544/142544 [=====] - 10s - loss: 1.3757
Generating from seed: ' the mock
' the mock turtle said the dormouse some of the conce in the dormouse some of the conce in the dormouse some o
=====
Iteration #: 24
Epoch 1/1
142544/142544 [=====] - 10s - loss: 1.3685
Generating from seed: raving mad
raving made to goon of the sord alice could got to the dormouse so they looked at the sord alice could got to
```

Generating the next character or next word of text is not the only thing you can do with this sort of model. This kind of model has been successfully used to make stock predictions (for more information refer to the article: *Financial Market Time Series Prediction with Recurrent Neural Networks*, by A. Bernal, S. Fok, and R. Pidaparthi, 2012) and generate classical music (for more information refer to the article: *DeepBach: A Steerable Model for Bach Chorales Generation*, by G. Hadjeres and F. Pachet, arXiv:1612.01010, 2016), to name a few interesting applications. Andrej Karpathy covers a few other fun examples, such as generating fake Wikipedia pages, algebraic geometry proofs, and Linux source code in his blog post at: *The Unreasonable Effectiveness of Recurrent Neural Networks* at <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>.

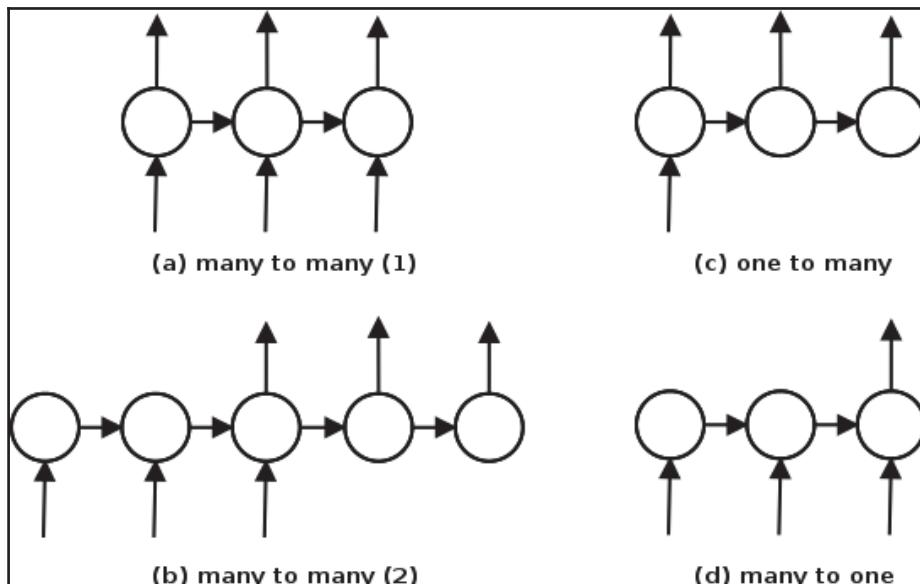
The source code for this example is available in `alice_chargen_rnn.py` in the code download for the chapter. The data is available from Project Gutenberg.

RNN topologies

The APIs for MLP and CNN architectures are limited. Both architectures accept a fixed-size tensor as input and produce a fixed-size tensor as output; and they perform the transformation from input to output in a fixed number of steps given by the number of layers in the model. RNNs don't have this limitation—you can have sequences in the input, the output, or both. This means that RNNs can be arranged in many ways to solve specific problems.

As we have learned, RNNs combine the input vector with the previous state vector to produce a new state vector. This can be thought of as similar to running a program with some inputs and some internal variables. Thus RNNs can be thought of as essentially describing computer programs. In fact, it has been shown that RNNs are turing complete (for more information refer to the article: *On the Computational Power of Neural Nets*, by H. T. Siegelmann and E. D. Sontag, proceedings of the fifth annual workshop on computational learning theory, ACM, 1992.) in the sense that given the proper weights, they can simulate arbitrary programs.

This property of being able to work with sequences gives rise to a number of common topologies, some of which we'll discuss, as follows:



All these different topologies derive from the same basic structure shown in the preceding diagram. In this basic topology, all input sequences are of the same length and an output is produced at each time step. We have already seen an example of this with our character level RNN for generating words in *Alice in Wonderland*.

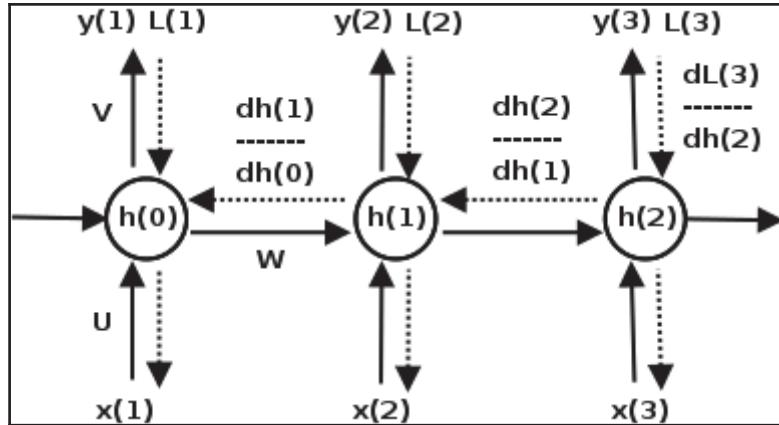
Another example of a many to many RNN could be a machine translation network shown as **(b)**, part of a general family of networks called sequence-to-sequence (for more information refer to: *Grammar as a Foreign Language*, by O. Vinyals, Advances in Neural Information Processing Systems, 2015). These take in a sequence and produces another sequence. In the case of machine translation, the input could be a sequence of English words in a sentence and the output could be the words in a translated Spanish sentence. In the case of a model that uses sequence-to-sequence to do **part-of-speech (POS)** tagging, the input could be the words in a sentence and the output could be the corresponding POS tags. It differs from the previous topology in that at certain time steps there is no input and at others there is no output. We will see an example of such a network later in this chapter.

Other variants are the one-to-many network shown as **(c)**, an example of which could be an image captioning network (for more information refer to the article: *Deep Visual-Semantic Alignments for Generating Image Descriptions*, by A. Karpathy, and F. Li, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015.), where the input is an image and the output a sequence of words.

Similarly, an example of a many-to-one network as shown in **(d)** could be a network that does sentiment analysis of sentences, where the input is a sequence of words and the output is a positive or negative sentiment (for more information refer to the article: *Recursive Deep Models for Semantic Compositionality over a Sentiment Treebank*, by R. Socher, Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP). Vol. 1631, 2013). We will see an (much simplified compared to the cited model) example of this topology as well later in the chapter.

Vanishing and exploding gradients

Just like traditional neural networks, training the RNN also involves backpropagation. The difference in this case is that since the parameters are shared by all time steps, the gradient at each output depends not only on the current time step, but also on the previous ones. This process is called **backpropagation through time (BPTT)** (for more information refer to the article: *Learning Internal Representations by Backpropagating errors*, by G. E. Hinton, D. E. Rumelhart, and R. J. Williams, Parallel Distributed Processing: Explorations in the Microstructure of Cognition 1, 1985):



Consider the small three layer RNN shown in the preceding diagram. During the forward propagation (shown by the solid lines), the network produces predictions that are compared to the labels to compute a loss L_t at each time step. During backpropagation (shown by dotted lines), the gradients of the loss with respect to the parameters U , V , and W are computed at each time step and the parameters are updated with the sum of the gradients.

The following equation shows the gradient of the loss with respect to W , the matrix that encodes weights for the long term dependencies. We focus on this part of the update because it is the cause of the vanishing and exploding gradient problem. The other two gradients of the loss with respect to the matrices U and V are also summed up across all time steps in a similar way:

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}} = \sum_t \frac{\partial \mathbf{L}_t}{\partial \mathbf{W}}$$

Let us now look at what happens to the gradient of the loss at the last time step ($t=3$). As you can see, this gradient can be decomposed to a product of three sub gradients using the chain rule. The gradient of the hidden state h_2 with respect to W can be further decomposed as the sum of the gradient of each hidden state with respect to the previous one. Finally, each gradient of the hidden state with respect to the previous one can be further decomposed as the product of gradients of the current hidden state against the previous one:

$$\begin{aligned}\frac{\partial L_3}{\partial W} &= \frac{\partial L_3}{\partial y_3} \cdot \frac{\partial y_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial W} \\ &= \sum_{t=0}^2 \frac{\partial L_3}{\partial y_3} \cdot \frac{\partial y_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_t} \cdot \frac{\partial h_t}{\partial W} \\ &= \sum_{t=0}^2 \frac{\partial L_3}{\partial y_3} \cdot \frac{\partial y_3}{\partial h_2} \cdot \left(\prod_{j=t+1}^2 \frac{\partial h_j}{\partial h_{j-1}} \right) \cdot \frac{\partial h_t}{\partial W}\end{aligned}$$

Similar calculations are done to compute the gradient of losses L_1 and L_2 (at time steps 1 and 2) with respect to W and to sum them into the gradient update for W . We will not explore the math further in this book. If you want to do so on your own, this WILDML blog post (<https://goo.gl/1061bx>) has a very good explanation of BPTT, including more detailed derivations of the mathematics behind the process.

For our purposes, the final form of the gradient in the equation above tells us why RNNs have the problem of vanishing and exploding gradients. Consider the case where the individual gradients of a hidden state with respect to the previous one is less than one. As we backpropagate across multiple time steps, the product of gradients get smaller and smaller, leading to the problem of vanishing gradients. Similarly, if the gradients are larger than one, the products get larger and larger, leading to the problem of exploding gradients.

The effect of vanishing gradients is that the gradients from steps that are far away do not contribute anything to the learning process, so the RNN ends up not learning long range dependencies. Vanishing gradients can happen for traditional neural networks as well, it is just more visible in case of RNNs, since RNNs tend to have many more layers (time steps) over which back propagation must occur.

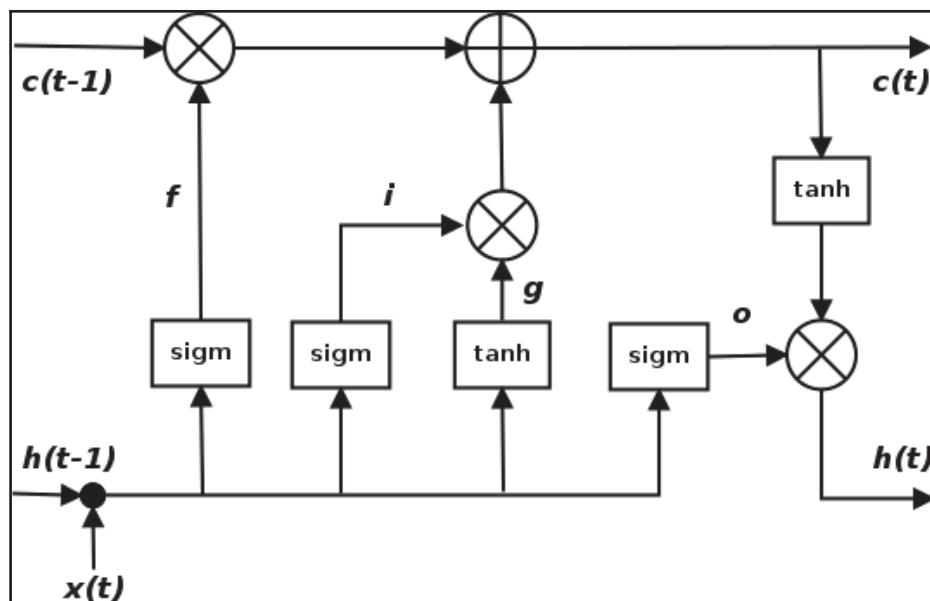
Exploding gradients are more easily detectable, the gradients will become very large and then turn into **not a number** (NaN) and the training process will crash. Exploding gradients can be controlled by clipping them at a predefined threshold as discussed in the paper: *On the Difficulty of Training Recurrent Neural Networks*, by R. Pascanu, T. Mikolov, and Y. Bengio, ICML, Pp 1310-1318, 2013.

While there are a few approaches to minimize the problem of vanishing gradients, such as proper initialization of the W matrix, using a ReLU instead of $tanh$ layers, and pre-training the layers using unsupervised methods, the most popular solution is to use the LSTM or GRU architectures. These architectures have been designed to deal with the vanishing gradient problem and learn long term dependencies more effectively. We will learn more about LSTM and GRU architectures later in this chapter.

Long short term memory — LSTM

The LSTM is a variant of RNN that is capable of learning long term dependencies. LSTMs were first proposed by Hochreiter and Schmidhuber and refined by many other researchers. They work well on a large variety of problems and are the most widely used type of RNN.

We have seen how the SimpleRNN uses the hidden state from the previous time step and the current input in a $tanh$ layer to implement recurrence. LSTMs also implement recurrence in a similar way, but instead of a single $tanh$ layer, there are four layers interacting in a very specific way. The following diagram illustrates the transformations that are applied to the hidden state at time step t :



The diagram looks complicated, but let us look at it component by component. The line across the top of the diagram is the cell state c , and represents the internal memory of the unit. The line across the bottom is the hidden state, and the i, f, o , and g gates are the mechanism by which the LSTM works around the vanishing gradient problem. During training, the LSTM learns the parameters for these gates.

In order to gain a deeper understanding of how these gates modulate the LSTM's hidden state, let us consider the equations that show how it calculates the hidden state h_t at time t from the hidden state h_{t-1} at the previous time step:

$$\begin{aligned} i &= \sigma(W_i h_{t-1} + U_i x_t) \\ f &= \sigma(W_f h_{t-1} + U_f x_t) \\ o &= \sigma(W_o h_{t-1} + U_o x_t) \\ g &= \tanh(W_g h_{t-1} + U_g x_t) \\ c_t &= (c_{t-1} \otimes f) \oplus (g \otimes i) \\ h_t &= \tanh(c_t) \otimes o \end{aligned}$$

Here i, f , and o are the input, forget, and output gates. They are computed using the same equations but with different parameter matrices. The sigmoid function modulates the output of these gates between zero and one, so the output vector produced can be multiplied element-wise with another vector to define how much of the second vector can pass through the first one.

The forget gate defines how much of the previous state h_{t-1} you want to allow to pass through. The input gate defines how much of the newly computed state for the current input x_t you want to let through, and the output gate defines how much of the internal state you want to expose to the next layer. The internal hidden state g is computed based on the current input x_t and the previous hidden state h_{t-1} . Notice that the equation for g is identical to that for the SimpleRNN cell, but in this case we will modulate the output by the output of the input gate i .

Given i, f, o , and g , we can now calculate the cell state c_t at time t in terms of c_{t-1} at time $(t-1)$ multiplied by the forget gate and the state g multiplied by the input gate i . So this is basically a way to combine the previous memory and the new input—setting the forget gate to 0 ignores the old memory and setting the input gate to 0 ignores the newly computed state.

Finally, the hidden state h_t at time t is computed by multiplying the memory c_t with the output gate.

One thing to realize is that an LSTM is a drop-in replacement for a SimpleRNN cell, the only difference is that LSTMs are resistant to the vanishing gradient problem. You can replace an RNN cell in a network with an LSTM without worrying about any side effects. You should generally see better results along with longer training times.

If you would like to know more, WILDML blog post has a very detailed explanation of these LSTM gates and how they work. For a more visual explanation, take a look at Christopher Olah's blog post: *Understanding LSTMs* (<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>) where he walks you step by step through these computations, with illustrations at each step.

LSTM with Keras — sentiment analysis

Keras provides an LSTM layer that we will use here to construct and train a many-to-one RNN. Our network takes in a sentence (a sequence of words) and outputs a sentiment value (positive or negative). Our training set is a dataset of about 7,000 short sentences from UMICH SI650 sentiment classification competition on Kaggle (<https://inclass.kaggle.com/c/si650winter11>). Each sentence is labeled 1 or 0 for positive or negative sentiment respectively, which our network will learn to predict.

We start with the imports, as usual:

```
from keras.layers.core import Activation, Dense, Dropout, SpatialDropout1D
from keras.layers.embeddings import Embedding
from keras.layers.recurrent import LSTM
from keras.models import Sequential
from keras.preprocessing import sequence
from sklearn.model_selection import train_test_split
import collections
import matplotlib.pyplot as plt
import nltk
import numpy as np
import os
```

Before we start, we want to do a bit of exploratory analysis on the data. Specifically we need to know how many unique words there are in the corpus and how many words are there in each sentence:

```
maxlen = 0
word_freqs = collections.Counter()
num_recs = 0
```

```
ftrain = open(os.path.join(DATA_DIR, "umich-sentiment-train.txt"), 'rb')
for line in ftrain:
    label, sentence = line.strip().split("t")
    words = nltk.word_tokenize(sentence.decode("ascii", "ignore").lower())
    if len(words) > maxlen:
        maxlen = len(words)
    for word in words:
        word_freqs[word] += 1
    num_recs += 1
ftrain.close()
```

Using this, we get the following estimates for our corpus:

```
maxlen : 42
len(word_freqs) : 2313
```

Using the number of unique words `len(word_freqs)`, we set our vocabulary size to a fixed number and treat all the other words as **out of vocabulary (OOV)** words and replace them with the pseudo-word UNK (for unknown). At prediction time, this will allow us to handle previously unseen words as OOV words as well.

The number of words in the sentence (`maxlen`) allows us to set a fixed sequence length and zero pad shorter sentences and truncate longer sentences to that length as appropriate. Even though RNNs handle variable sequence length, this is usually achieved either by padding and truncating as above, or by grouping the inputs in different batches by sequence length. We will use the former approach here. For the latter approach, Keras recommends using batches of size one (for more information refer to: <https://github.com/fchollet/keras/issues/40>).

Based on the preceding estimates, we set our `VOCABULARY_SIZE` to 2,000 words from our vocabulary plus the UNK pseudo-word and the PAD pseudo word (used for padding sentences to a fixed number of words), in our case 40 given by

`MAX_SENTENCE_LENGTH`:

```
DATA_DIR = ".../data"
MAX_FEATURES = 2000
MAX_SENTENCE_LENGTH = 40
```

Next we need a pair of lookup tables. Each row of input to the RNN is a sequence of word indices, where the indices are ordered by most frequent to least frequent word in the training set. The two lookup tables allow us to lookup an index given the word and the word given the index. This includes the PAD and UNK pseudo-words as well:

```
vocab_size = min(MAX_FEATURES, len(word_freqs)) + 2
word2index = {x[0]: i+2 for i, x in
enumerate(word_freqs.most_common(MAX_FEATURES)) }
word2index["PAD"] = 0
word2index["UNK"] = 1
index2word = {v:k for k, v in word2index.items() }
```

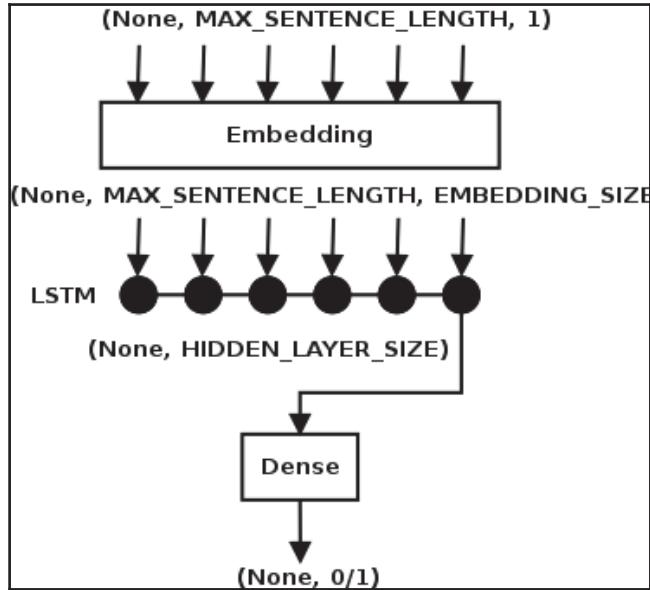
Next, we convert our input sentences to word index sequences, pad them to the MAX_SENTENCE_LENGTH words. Since our output label in this case is binary (positive or negative sentiment), we don't need to process the labels:

```
X = np.empty((num_recs, ), dtype=list)
y = np.zeros((num_recs, ))
i = 0
ftrain = open(os.path.join(DATA_DIR, "umich-sentiment-train.txt"), 'rb')
for line in ftrain:
    label, sentence = line.strip().split("t")
    words = nltk.word_tokenize(sentence.decode("ascii", "ignore").lower())
    seqs = []
    for word in words:
        if word2index.has_key(word):
            seqs.append(word2index[word])
        else:
            seqs.append(word2index["UNK"])
    X[i] = seqs
    y[i] = int(label)
    i += 1
ftrain.close()
X = sequence.pad_sequences(X, maxlen=MAX_SENTENCE_LENGTH)
```

Finally, we split the training set into a 80-20 training test split:

```
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.2,
random_state=42)
```

The following diagram shows the structure of our RNN:



The input for each row is a sequence of word indices. The sequence length is given by `MAX_SENTENCE_LENGTH`. The first dimension of the tensor is set to `None` to indicate that the batch size (the number of records fed to the network each time) is currently unknown at definition time; it is specified during run time using the `batch_size` parameter. So assuming an as-yet undetermined batch size, the shape of the input tensor is $(None, MAX_SENTENCE_LENGTH, 1)$. These tensors are fed into an embedding layer of size `EMBEDDING_SIZE` whose weights are initialized with small random values and learned during training. This layer will transform the tensor to a shape $(None, MAX_SENTENCE_LENGTH, EMBEDDING_SIZE)$. The output of the embedding layer is fed into an LSTM with sequence length `MAX_SENTENCE_LENGTH` and output layer size `HIDDEN_LAYER_SIZE`, so the output of the LSTM is a tensor of shape $(None, HIDDEN_LAYER_SIZE, MAX_SENTENCE_LENGTH)$. By default, the LSTM will output a single tensor of shape $(None, HIDDEN_LAYER_SIZE)$ at its last sequence (`return_sequences=False`). This is fed to a dense layer with output size of 1 with a sigmoid activation function, so it will output either 0 (negative review) or 1 (positive review).

We compile the model using the binary cross-entropy loss function since it predicts a binary value, and the Adam optimizer, a good general purpose optimizer. Note that the hyperparameters EMBEDDING_SIZE, HIDDEN_LAYER_SIZE, BATCH_SIZE and NUM_EPOCHS (set as constants as follows) were tuned experimentally over several runs:

```
EMBEDDING_SIZE = 128
HIDDEN_LAYER_SIZE = 64
BATCH_SIZE = 32
NUM_EPOCHS = 10

model = Sequential()
model.add(Embedding(vocab_size, EMBEDDING_SIZE,
input_length=MAX_SENTENCE_LENGTH))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(LSTM(HIDDEN_LAYER_SIZE, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1))
model.add(Activation("sigmoid"))

model.compile(loss="binary_crossentropy", optimizer="adam",
metrics=["accuracy"])
```

We then train the network for 10 epochs (NUM_EPOCHS) and batch size of 32 (BATCH_SIZE). At each epoch we validate the model using the test data:

```
history = model.fit(Xtrain, ytrain, batch_size=BATCH_SIZE,
epochs=NUM_EPOCHS,
validation_data=(Xtest, ytest))
```

The output of this step shows how the loss decreases and accuracy increases over multiple epochs:

```
Train on 5668 samples, validate on 1418 samples
Epoch 1/10
5668/5668 [=====] - 20s - loss: 0.3316 - acc: 0.8626 - val_loss: 0.0799 - val_acc: 0.9746
Epoch 2/10
5668/5668 [=====] - 19s - loss: 0.0911 - acc: 0.9626 - val_loss: 0.0512 - val_acc: 0.9810
Epoch 3/10
5668/5668 [=====] - 18s - loss: 0.0649 - acc: 0.9730 - val_loss: 0.0553 - val_acc: 0.9859
Epoch 4/10
5668/5668 [=====] - 19s - loss: 0.0642 - acc: 0.9746 - val_loss: 0.0596 - val_acc: 0.9845
Epoch 5/10
5668/5668 [=====] - 20s - loss: 0.0531 - acc: 0.9787 - val_loss: 0.0434 - val_acc: 0.9845
Epoch 6/10
5668/5668 [=====] - 19s - loss: 0.0575 - acc: 0.9762 - val_loss: 0.0396 - val_acc: 0.9852
Epoch 7/10
5668/5668 [=====] - 19s - loss: 0.0494 - acc: 0.9797 - val_loss: 0.0374 - val_acc: 0.9873
Epoch 8/10
5668/5668 [=====] - 19s - loss: 0.0467 - acc: 0.9809 - val_loss: 0.0374 - val_acc: 0.9859
Epoch 9/10
5668/5668 [=====] - 18s - loss: 0.0440 - acc: 0.9811 - val_loss: 0.0425 - val_acc: 0.9852
Epoch 10/10
5668/5668 [=====] - 18s - loss: 0.0464 - acc: 0.9795 - val_loss: 0.0378 - val_acc: 0.9873
1418/1418 [=====] - 0s
```

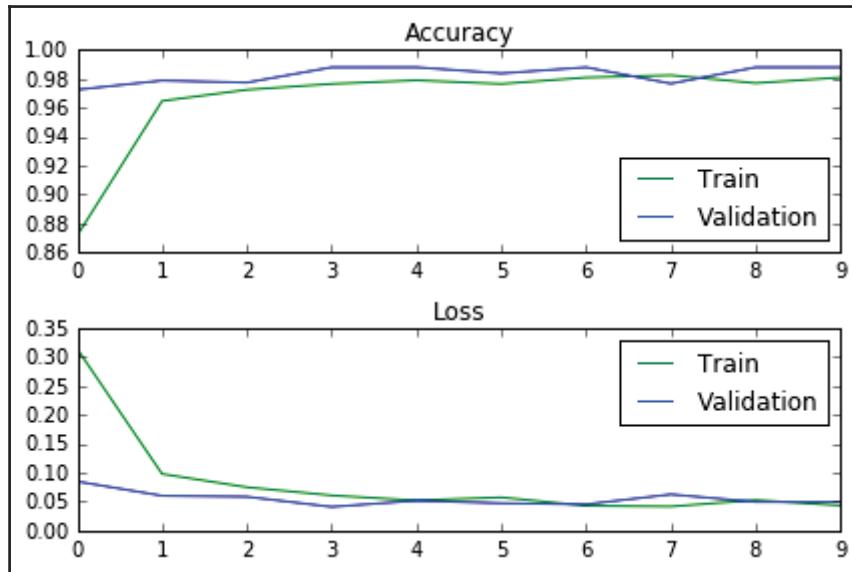
We can also plot the loss and accuracy values over time using the following code:

```
plt.subplot(211)
plt.title("Accuracy")
plt.plot(history.history["acc"], color="g", label="Train")
plt.plot(history.history["val_acc"], color="b", label="Validation")
plt.legend(loc="best")

plt.subplot(212)
plt.title("Loss")
plt.plot(history.history["loss"], color="g", label="Train")
plt.plot(history.history["val_loss"], color="b", label="Validation")
plt.legend(loc="best")

plt.tight_layout()
plt.show()
```

The output of the preceding example is as follows:



Finally, we evaluate our model against the full test set and print the score and accuracy. We also pick a few random sentences from our test set and print the RNN's prediction, the label and the actual sentence:

```
score, acc = model.evaluate(Xtest, ytest, batch_size=BATCH_SIZE)
print("Test score: %.3f, accuracy: %.3f" % (score, acc))

for i in range(5):
    idx = np.random.randint(len(Xtest))
    xtest = Xtest[idx].reshape(1, 40)
    ylabel = ytest[idx]
    ypred = model.predict(xtest)[0][0]
    sent = " ".join([index2word[x] for x in xtest[0].tolist() if x != 0])
    print("%.0ft%dt%s" % (ypred, ylabel, sent))
```

As you can see from the results, we get back close to 99% accuracy. The predictions the model makes for this particular set match exactly with the labels, although this is not the case for all predictions:

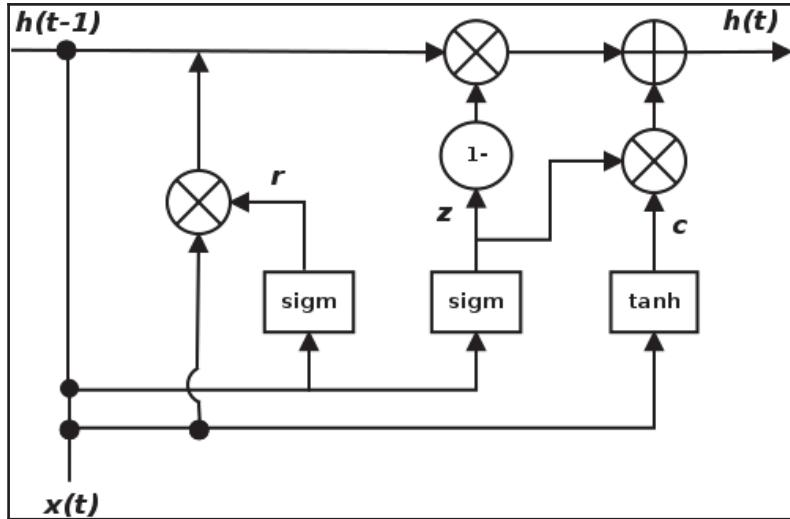
```
Test score: 0.038, accuracy: 0.987  
#pred label sentence  
1    1    i like th mission impossible one ...  
1    1    we 're gon na like watch mission impossible or hoot . (   
1    1    the people who are worth it know how much i love the da vinci code .  
0    0    ok brokeback mountain is such a horrible movie .  
1    1    brokeback mountain is the most amazing / beautiful / romantic /  
        Heartbraking movie i have ever or will ever see in my life
```

If you would like to run this code locally, you need to get the data from the Kaggle website.

The source code for this example is available in the file `umich_sentiment_lstm.py` in the code download for this chapter.

Gated recurrent unit — GRU

The GRU is a variant of the LSTM and was introduced by K. Cho (for more information refer to: *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, by K. Cho, arXiv:1406.1078, 2014). It retains the LSTM's resistance to the vanishing gradient problem, but its internal structure is simpler, and therefore is faster to train, since fewer computations are needed to make updates to its hidden state. The gates for a GRU cell are illustrated in the following diagram:



Instead of the input, forget, and output gates in the LSTM cell, the GRU cell has two gates, an update gate z , and a reset gate r . The update gate defines how much previous memory to keep around and the reset gate defines how to combine the new input with the previous memory. There is no persistent cell state distinct from the hidden state as in LSTM. The following equations define the gating mechanism in a GRU:

$$\begin{aligned} z &= \sigma(W_z h_{t-1} + U_z x_t) \\ r &= \sigma(W_r h_{t-1} + U_r x_t) \\ c &= \tanh(W_c(h_{t-1} \otimes r) + U_c x_t) \\ h_t &= (z \otimes c) \oplus ((1 - z) \otimes h_{t-1}) \end{aligned}$$

According to several empirical evaluations (for more information refer to the articles: *An Empirical Exploration of Recurrent Network Architectures*, by R. Jozefowicz, W. Zaremba, and I. Sutskever, JMLR, 2015 and *Empirical Evaluation of Gated Recurrent Networks on Sequence Modeling*, by J. Chung, arXiv:1412.3555. 2014), GRU and LSTM have comparable performance and there is no simple way to recommend one or the other for a specific task. While GRUs are faster to train and need less data to generalize, in situations where there is enough data, an LSTM's greater expressive power may lead to better results. Like LSTMs, GRUs are drop-in replacements for the SimpleRNN cell.

Keras provides built in implementations of both LSTM and GRU, as well as the SimpleRNN class we saw earlier.

GRU with Keras — POS tagging

Keras provides a GRU implementation, that we will use here to build a network that does POS tagging. A POS is a grammatical category of words that are used in the same way across multiple sentences. Examples of POS are nouns, verbs, adjectives, and so on. For example, nouns are typically used to identify things, verbs are typically used to identify what they do, and adjectives to describe some attribute of these things. POS tagging used to be done manually, but nowadays this is done automatically using statistical models. In recent years, deep learning has been applied to this problem as well (for more information refer to the article: *Natural Language Processing (almost) from Scratch*, by R. Collobert, Journal of Machine Learning Research, Pp. 2493-2537, 2011).

For our training data, we will need sentences tagged with part of speech tags. The Penn Treebank (<https://catalog.ldc.upenn.edu/ldc99t42>) is one such dataset, it is a human annotated corpus of about 4.5 million words of American English. However, it is a non-free resource. A 10% sample of the Penn Treebank is freely available as part of the NLTK (<http://www.nltk.org/>), which we will use to train our network.

Our model will take in a sequence of words in a sentence and output the corresponding POS tags for each word. Thus for an input sequence consisting of the words [*The, cat, sat, on, the, mat, .*], the output sequence emitted would be the POS symbols [*DT, NN, VB, IN, DT, NN*].

We start with the imports:

```
from keras.layers.core import Activation, Dense, Dropout, RepeatVector,  
SpatialDropout1D  
from keras.layers.embeddings import Embedding  
from keras.layers.recurrent import GRU  
from keras.layers.wrappers import TimeDistributed  
from keras.models import Sequential  
from keras.preprocessing import sequence  
from keras.utils import np_utils  
from sklearn.model_selection import train_test_split  
import collections  
import nltk  
import numpy as np  
import os
```

We then download the data from NLTK in a format suitable for our downstream code. Specifically, the data is available in parsed form as part of the NLTK Treebank corpus. We use the following Python code to download this data into two parallel files, one for the words in the sentences and one for the POS tags:

```
DATA_DIR = ".../data"

fedata = open(os.path.join(DATA_DIR, "treebank_sents.txt"), "wb")
ffdata = open(os.path.join(DATA_DIR, "treebank_pos.txt"), "wb")

sents = nltk.corpus.treebank.tagged_sents()
for sent in sents:
    words, poss = [], []
    for word, pos in sent:
        if pos == "-NONE-":
            continue
        words.append(word)
        poss.append(pos)
    fedata.write("{}\n".format(" ".join(words)))
    ffdata.write("{}\n".format(" ".join(poss)))

fedata.close()
ffdata.close()
```

Once again, we want to explore the data a little to find out what vocabulary size to set. This time, we have to consider two different vocabularies, the source vocabulary for the words and the target vocabulary for the POS tags. We need to find the number of unique words in each vocabulary. We also need to find the maximum number of words in a sentence in our training corpus and the number of records. Because of the one-to-one nature of POS tagging, the last two values are identical for both vocabularies:

```
def parse_sentences(filename):
    word_freqs = collections.Counter()
    num_recs, maxlen = 0, 0
    fin = open(filename, "rb")
    for line in fin:
        words = line.strip().lower().split()
        for word in words:
            word_freqs[word] += 1
        if len(words) > maxlen:
            maxlen = len(words)
        num_recs += 1
    fin.close()
    return word_freqs, maxlen, num_recs

s_wordfreqs, s_maxlen, s_numrecs = parse_sentences(
    os.path.join(DATA_DIR, "treebank_sents.txt"))
```

```
t_wordfreqs, t_maxlen, t_numrecs = parse_sentences(  
    os.path.join(DATA_DIR, "treebank_posse.txt"))  
print(len(s_wordfreqs), s_maxlen, s_numrecs, len(t_wordfreqs), t_maxlen,  
t_numrecs)
```

Running this code tells us that there are 10,947 unique words and 45 unique POS tags. The maximum sentence size is 249, and the number of sentences in the 10% set is 3,914. Using this information, we decide to consider only the top 5,000 words for our source vocabulary. Our target vocabulary has 45 unique POS tags, we want to be able to predict all of them, so we will consider all of them in our vocabulary. Finally, we set 250 to be our maximum sequence length:

```
MAX_SEQLEN = 250  
S_MAX_FEATURES = 5000  
T_MAX_FEATURES = 45
```

Just like our sentiment analysis example, each row of the input will be represented as a sequence of word indices. The corresponding output will be a sequence of POS tag indices. So we need to build lookup tables to translate between the words/POS tags and their corresponding indices. Here is the code to do that. On the source side, we build a vocabulary index with two extra slots to hold the PAD and UNK pseudo-words. On the target side, we don't drop any words so there is no need for the UNK pseudo-word:

```
s_vocabsize = min(len(s_wordfreqs), S_MAX_FEATURES) + 2  
s_word2index = {x[0]:i+2 for i, x in  
enumerate(s_wordfreqs.most_common(S_MAX_FEATURES))}  
s_word2index["PAD"] = 0  
s_word2index["UNK"] = 1  
s_index2word = {v:k for k, v in s_word2index.items()}  
  
t_vocabsize = len(t_wordfreqs) + 1  
t_word2index = {x[0]:i for i, x in  
enumerate(t_wordfreqs.most_common(T_MAX_FEATURES))}  
t_word2index["PAD"] = 0  
t_index2word = {v:k for k, v in t_word2index.items()}
```

The next step is to build our datasets to feed into our network. We will use these lookup tables to convert our input sentences into a word ID sequence of length `MAX_SEQLEN` (250). The labels need to be structured as a sequence of one-hot vectors of size `T_MAX_FEATURES + 1` (46), also of length `MAX_SEQLEN` (250). The `build_tensor` function reads the data from the two files and converts them to the input and output tensors. Additional default parameters are passed in to build the output tensor. This triggers the call to `np_utils.to_categorical()` to convert the output sequence of POS tag IDs to one-hot vector representation:

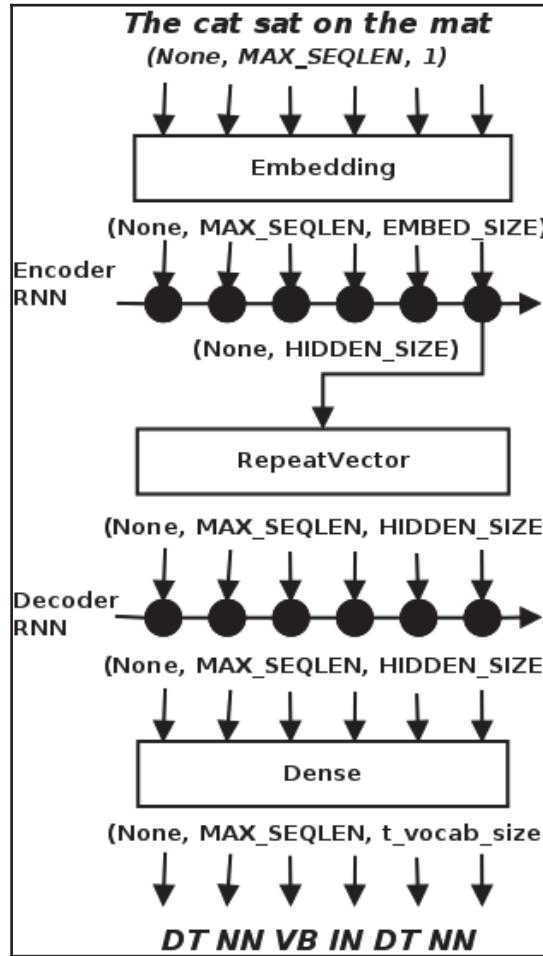
```
def build_tensor(filename, numrecs, word2index, maxlen,
                 make_categorical=False, num_classes=0):
    data = np.empty((numrecs, ), dtype=list)
    fin = open(filename, "rb")
    i = 0
    for line in fin:
        wids = []
        for word in line.strip().lower().split():
            if word2index.has_key(word):
                wids.append(word2index[word])
            else:
                wids.append(word2index["UNK"])
        if make_categorical:
            data[i] = np_utils.to_categorical(wids,
                                              num_classes=num_classes)
        else:
            data[i] = wids
        i += 1
    fin.close()
    pdata = sequence.pad_sequences(data, maxlen=maxlen)
    return pdata

X = build_tensor(os.path.join(DATA_DIR, "treebank_sents.txt"),
                 s_numrecs, s_word2index, MAX_SEQLEN)
Y = build_tensor(os.path.join(DATA_DIR, "treebank_pos.txt"),
                 t_numrecs, t_word2index, MAX_SEQLEN, True, t_vocabsize)
```

We can then split the dataset into a 80-20 train-test split:

```
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.2,
                                                random_state=42)
```

The following figure shows the schematic of our network. It looks complicated, so let us deconstruct it:



As previously, assuming that the batch size is as yet undetermined, the input to the network is a tensor of word IDs of shape $(None, \text{MAX_SEQLEN}, 1)$. This is sent through an embedding layer, which converts each word into a dense vector of shape (EMBED_SIZE) , so the output tensor from this layer has the shape $(None, \text{MAX_SEQLEN}, \text{EMBED_SIZE})$. This tensor is fed to the encoder GRU with an output size of HIDDEN_SIZE . The GRU is set to return a single context vector (`return_sequences=False`) after seeing a sequence of size MAX_SEQLEN , so the output tensor from the GRU layer has shape $(None, \text{HIDDEN_SIZE})$.

This context vector is then replicated using the RepeatVector layer into a tensor of shape (None, MAX_SEQLEN, HIDDEN_SIZE) and fed into the decoder GRU layer. This is then fed into a dense layer which produces an output tensor of shape (None, MAX_SEQLEN, t_vocab_size). The activation function on the dense layer is a softmax. The argmax of each column of this tensor is the index of the predicted POS tag for the word at that position.

The model definition is shown as follows: EMBED_SIZE, HIDDEN_SIZE, BATCH_SIZE, and NUM_EPOCHS are hyperparameters which have been assigned these values after experimenting with multiple different values. The model is compiled with the categorical_crossentropy loss function since we have multiple categories of labels, and the optimizer used is the popular adam optimizer:

```
EMBED_SIZE = 128
HIDDEN_SIZE = 64
BATCH_SIZE = 32
NUM_EPOCHS = 1

model = Sequential()
model.add(Embedding(s_vocabsize, EMBED_SIZE,
input_length=MAX_SEQLEN))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(GRU(HIDDEN_SIZE, dropout=0.2, recurrent_dropout=0.2))
model.add(RepeatVector(MAX_SEQLEN))
model.add(GRU(HIDDEN_SIZE, return_sequences=True))
model.add(TimeDistributed(Dense(t_vocabsize)))
model.add(Activation("softmax"))

model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])
```

We train this model for a single epoch. The model is very rich, with many parameters, and begins to overfit after the first epoch of training. When fed the same data multiple times in the next epochs, the model begins to overfit to the training data and does worse on the validation data:

```
model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE, epochs=NUM_EPOCHS,
validation_data=[Xtest, Ytest])

score, acc = model.evaluate(Xtest, Ytest, batch_size=BATCH_SIZE)
print("Test score: %.3f, accuracy: %.3f" % (score, acc))
```

The output of the training and the evaluation is shown as follows. As you can see, the model does quite well after the first epoch of training:

```
Train on 3131 samples, validate on 783 samples
Epoch 1/1
3131/3131 [=====] - 81s - loss: 0.3013 - acc: 0.8263 - val_loss: 0.2934 - val_acc: 0.9159

783/783 [=====] - 3s
Test score: 0.293, accuracy: 0.916
```

Similar to actual RNNs, the three recurrent classes in Keras (`SimpleRNN`, `LSTM`, and `GRU`) are interchangeable. To demonstrate, we simply replace all occurrences of `GRU` in the previous program with `LSTM` and rerun the program. The model definition and the import statements are the only things that change:

```
from keras.layers.recurrent import GRU

model = Sequential()
model.add(Embedding(s_vocabsize, EMBED_SIZE,
input_length=MAX_SEQLEN))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(GRU(HIDDEN_SIZE, dropout=0.2, recurrent_dropout=0.2))
model.add(RepeatVector(MAX_SEQLEN))
model.add(GRU(HIDDEN_SIZE, return_sequences=True))
model.add(TimeDistributed(Dense(t_vocabsize)))
model.add(Activation("softmax"))
```

As you can see from the output, the results of the `GRU`-based network are quite comparable to our previous `LSTM`-based network.

Sequence-to-sequence models are a very powerful class of model. Its most canonical application is machine translation, but there are many others such as the previous example. Indeed, a lot of NLP tasks further up in the hierarchy, such as named entity recognition (for more information refer to the article: *Named Entity Recognition with Long Short Term Memory*, by J. Hammerton, Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL, Association for Computational Linguistics, 2003) and sentence parsing (for more information refer to the article: *Grammar as a Foreign Language*, by O. Vinyals, Advances in Neural Information Processing Systems, 2015), as well as more complex networks such as those for image captioning (for more information refer to the article: *Deep Visual-Semantic Alignments for Generating Image Descriptions*, by A. Karpathy, and F. Li, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015.), are examples of the sequence-to-sequence compositional model.

The full code for this example can be found in the file `pos_tagging_gru.py` in the the code download for this chapter.

Bidirectional RNNs

At a given time step t , the output of the RNN is dependent on the outputs at all previous time steps. However, it is entirely possible that the output is also dependent on the future outputs as well. This is especially true for applications such as NLP, where the attributes of the word or phrase we are trying to predict may be dependent on the context given by the entire enclosing sentence, not just the words that came before it. Bidirectional RNNs also help a network architecture place equal emphasis on the beginning and end of the sequence, and increase the data available for training.

Bidirectional RNNs are two RNNs stacked on top of each other, reading the input in opposite directions. So in our example, one RNN will read the words left to right and the other RNN will read the words right to left. The output at each time step will be based on the hidden state of both RNNs.

Keras provides support for bidirectional RNNs through a bidirectional wrapper layer. For example, for our POS tagging example, we could make our LSTMs bidirectional simply by wrapping them with this Bidirectional wrapper, as shown in the model definition code as follows:

```
from keras.layers.wrappers import Bidirectional

model = Sequential()
model.add(Embedding(s_vocabsize, EMBED_SIZE,
input_length=MAX_SEQLEN))
model.add(SpatialDropout1D(Dropout(0.2)))
model.add(Bidirectional(LSTM(HIDDEN_SIZE, dropout=0.2,
recurrent_dropout=0.2)))
model.add(RepeatVector(MAX_SEQLEN))
model.add(Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=True)))
model.add(TimeDistributed(Dense(t_vocabsize)))
model.add(Activation("softmax"))
```

This gives us performance comparable to the unidirectional LSTM example shown as follows:

```
Train on 3131 samples, validate on 783 samples
Epoch 1/1
3131/3131 [=====] - 268s - loss: 0.2889 - acc: 0.8226 - val_loss: 0.2788 - val_acc: 0.9036

783/783 [=====] - 12s
Test score: 0.279, accuracy: 0.904
```

Stateful RNNs

RNNs can be stateful, which means that they can maintain state across batches during training. That is, the hidden state computed for a batch of training data will be used as the initial hidden state for the next batch of training data. However, this needs to be explicitly set, since Keras RNNs are stateless by default and resets the state after each batch. Setting an RNN to be stateful means that it can build a state across its training sequence and even maintain that state when doing predictions.

The benefits of using stateful RNNs are smaller network sizes and/or lower training times. The disadvantage is that we are now responsible for training the network with a batch size that reflects the periodicity of the data, and resetting the state after each epoch. In addition, data should not be shuffled while training the network, since the order in which the data is presented is relevant for stateful networks.

Stateful LSTM with Keras — predicting electricity consumption

In this example, we predict electricity consumption for a consumer using a stateful and stateless LSTM network and compare their behaviors. As you will recall, RNNs in Keras are stateless by default. In case of stateful models, the internal states computed after processing a batch of input is reused as initial states for the next batch. In other words, the state computed from element i in a batch will be used as initial state for the element i in the next batch.

The dataset we will use is the electricity load diagram dataset from the UCI Machine Learning Repository (<https://archive.ics.uci.edu/ml/datasets/ElectricityLoadDiagrams20112014>), and contains consumption information about 370 customers, taken at 15 minute intervals over a four year period from 2011 to 2014. We randomly choose customer number 250 for our example.

One thing to remember is that most problems can be solved with stateless RNNs, so if you do use a stateful RNN, make sure you need it. Typically, you would need it when the data has a periodic component. If you think a bit, you will realize that electricity consumption is periodic. Consumption tends to be higher during the day than at night. Let us extract the consumption data for customer number 250 and plot the first 10 days of data. Finally we also save it to a binary NumPy file for our next step:

```
import numpy as np
import matplotlib.pyplot as plt
import os
import re

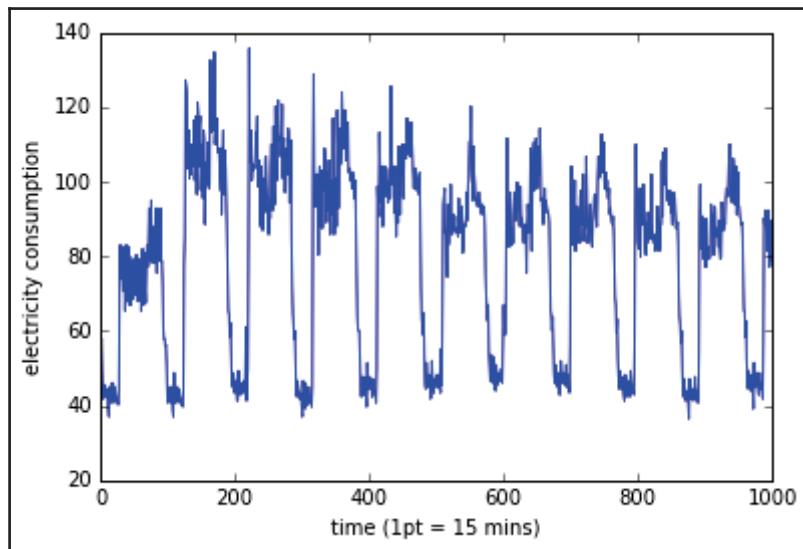
DATA_DIR = "../data"

fld = open(os.path.join(DATA_DIR, "LD2011_2014.txt"), "rb")
data = []
cid = 250
for line in fld:
    if line.startswith("");":
        continue
    cols = [float(re.sub(", ", ".", x)) for x in
            line.strip().split(";") [1:]]
    data.append(cols[cid])
fld.close()

NUM_ENTRIES = 1000
plt.plot(range(NUM_ENTRIES), data[0:NUM_ENTRIES])
plt.ylabel("electricity consumption")
plt.xlabel("time (1pt = 15 mins)")
plt.show()

np.save(os.path.join(DATA_DIR, "LD_250.npy"), np.array(data))
```

The output of the preceding example is as follow:



As you can see, there is clearly a daily periodic trend. So the problem is a good candidate for a stateful model. Also, based on our observation, a `BATCH_SIZE` of 96 (number of 15 minute readings over 24 hours) seems appropriate.

We will show the code for the stateless version of the model simultaneously with the one for the stateful version. Most of the code is identical for both versions, so we will look at both versions simultaneously. I will point out the differences in the code as they arise.

First, as usual, we import the necessary libraries and classes:

```
from keras.layers.core import Dense
from keras.layers.recurrent import LSTM
from keras.models import Sequential
from sklearn.preprocessing import MinMaxScaler
import numpy as np
import math
import os
```

Next we load the data for customer 250 into a long array of size (140256) from the saved NumPy binary file and rescale it to the range (0, 1). Finally, we reshape the input to three dimensions as needed by our network:

```
DATA_DIR = "../data"

data = np.load(os.path.join(DATA_DIR, "LD_250.npy"))
data = data.reshape(-1, 1)
scaler = MinMaxScaler(feature_range=(0, 1), copy=False)
data = scaler.fit_transform(data)
```

Within each batch, the model will take a sequence of 15 minute readings and predict the next one. The length of the input sequence is given by the NUM_TIMESTEPS variable in the code. Based on some experimentation, we get a value of NUM_TIMESTEPS as 20, that is, each input row will be a sequence of length 20, and the output will have length 1. The next step rearranges the input array into X and Y tensors of shapes (None, 4) and (None, 1). Finally, we reshape the input tensor X to three dimensions as required by the network:

```
X = np.zeros((data.shape[0], NUM_TIMESTEPS))
Y = np.zeros((data.shape[0], 1))
for i in range(len(data) - NUM_TIMESTEPS - 1):
    X[i] = data[i:i + NUM_TIMESTEPS].T
    Y[i] = data[i + NUM_TIMESTEPS + 1]

# reshape X to three dimensions (samples, timesteps, features)
X = np.expand_dims(X, axis=2)
```

We then split our X and Y tensors into a 70-30 training test split. Since we are working with time series, we just choose a split point and cut the data into two parts, rather than using the `train_test_split` function, which also shuffles the data:

```
sp = int(0.7 * len(data))
Xtrain, Xtest, Ytrain, Ytest = X[0:sp], X[sp:], Y[0:sp], Y[sp:]
print(Xtrain.shape, Xtest.shape, Ytrain.shape, Ytest.shape)
```

First we define our stateless model. We also set the values of `BATCH_SIZE` and `NUM_TIMESTEPS`, as we discussed previously. Our LSTM output size is given by `HIDDEN_SIZE`, another hyperparameter that is usually arrived at through experimentation. Here, we just set it to 10 since our objective is to compare two networks:

```
NUM_TIMESTEPS = 20
HIDDEN_SIZE = 10
BATCH_SIZE = 96 # 24 hours (15 min intervals)

# stateless
model = Sequential()
model.add(LSTM(HIDDEN_SIZE, input_shape=(NUM_TIMESTEPS, 1),
               return_sequences=False))
model.add(Dense(1))
```

The corresponding definition for the stateful model is very similar, as you can see as follows. In the LSTM constructor, you need to set `stateful=True`, and instead of `input_shape` where the batch size is determined at runtime, you need to set `batch_input_shape` explicitly with the batch size. You also need to ensure that your training and test data sizes are perfect multiples of your batch size. We will see how to do that later when we look at the training code:

```
# stateful
model = Sequential()
model.add(LSTM(HIDDEN_SIZE, stateful=True,
               batch_input_shape=(BATCH_SIZE, NUM_TIMESTEPS, 1),
               return_sequences=False))
model.add(Dense(1))
```

Next we compile the model, which is the same for both stateless and stateful RNNs. Notice that our metric here is mean squared error instead of our usual accuracy. This is because this is really a regression problem; we are interested in knowing how far off our predictions are with respect to the labels rather than knowing whether our prediction matched the label. You can find a full list of Keras built-in metrics on the Keras metrics page:

```
model.compile(loss="mean_squared_error", optimizer="adam",
               metrics=["mean_squared_error"])
```

To train the stateless model, we can use the one liner that we have probably become very familiar with by now:

```
BATCH_SIZE = 96 # 24 hours (15 min intervals)

# stateless
model.fit(Xtrain, Ytrain, epochs=NUM_EPOCHS, batch_size=BATCH_SIZE,
           validation_data=(Xtest, Ytest),
           shuffle=False)
```

The corresponding code for the stateful model is shown as follows. There are three things to be aware of here.

First, you should select a batch size that reflects the periodicity of your data. This is because stateful RNNs align the states from each batch to the next, so selecting the right batch size allows the network to learn faster.

Once you set the batch size, the size of your training and test sets needs to be exact multiples of your batch size. We have ensured this below by truncating the last few records from both our training and test sets.

The second thing is that you need to fit the model manually, training the model in a loop for the required number of epochs. Each iteration trains the model for one epoch, and the state is retained across multiple batches. After each epoch, the state of the model needs to be reset manually.

The third thing is that the data should be fed in sequence. By default, Keras will shuffle the rows within each batch, which will destroy the alignment we need for the stateful RNN to learn effectively. This is done by setting `shuffle=False` in the call to `model.fit()`:

```
BATCH_SIZE = 96 # 24 hours (15 min intervals)

# stateful
# need to make training and test data to multiple of BATCH_SIZE
train_size = (Xtrain.shape[0] // BATCH_SIZE) * BATCH_SIZE
test_size = (Xtest.shape[0] // BATCH_SIZE) * BATCH_SIZE
Xtrain, Ytrain = Xtrain[0:train_size], Ytrain[0:train_size]
Xtest, Ytest = Xtest[0:test_size], Ytest[0:test_size]
print(Xtrain.shape, Xtest.shape, Ytrain.shape, Ytest.shape)
for i in range(NUM_EPOCHS):
    print("Epoch {:d}/{:d}".format(i+1, NUM_EPOCHS))
    model.fit(Xtrain, Ytrain, batch_size=BATCH_SIZE, epochs=1,
               validation_data=(Xtest, Ytest),
               shuffle=False)
    model.reset_states()
```

Finally, we evaluate the model against the test data and print out the scores:

```
score, _ = model.evaluate(Xtest, Ytest, batch_size=BATCH_SIZE)
rmse = math.sqrt(score)
print("MSE: {:.3f}, RMSE: {:.3f}".format(score, rmse))
```

The output for the stateless model, run over five epochs, is as follows:

```
(98179, 20, 1) (42077, 20, 1) (98179, 1) (42077, 1)
Train on 98179 samples, validate on 42077 samples
Epoch 1/5
98179/98179 [=====] - 41s - loss: 0.0086 - mean_squared_error: 0.0086 - val_loss: 0.0040 -
val_mean_squared_error: 0.0040
Epoch 2/5
98179/98179 [=====] - 41s - loss: 0.0045 - mean_squared_error: 0.0045 - val_loss: 0.0039 -
val_mean_squared_error: 0.0039
Epoch 3/5
98179/98179 [=====] - 43s - loss: 0.0041 - mean_squared_error: 0.0041 - val_loss: 0.0038 -
val_mean_squared_error: 0.0038
Epoch 4/5
98179/98179 [=====] - 44s - loss: 0.0039 - mean_squared_error: 0.0039 - val_loss: 0.0040 -
val_mean_squared_error: 0.0040
Epoch 5/5
98179/98179 [=====] - 44s - loss: 0.0038 - mean_squared_error: 0.0038 - val_loss: 0.0038 -
val_mean_squared_error: 0.0038
42077/42077 [=====] - 2s
MSE: 0.004, RMSE: 0.062
```

The corresponding output for the stateful model, also run in a loop five times for one epoch each time, is as follows. Notice the result of the truncating operation in the second line:

```
Train on 98112 samples, validate on 42048 samples
Epoch 1/1
98112/98112 [=====] - 37s - loss: 0.0056 - mean_squared_error: 0.0056 - val_loss: 0.0038 -
val_mean_squared_error: 0.0038
Epoch 2/5
Train on 98112 samples, validate on 42048 samples
Epoch 1/1
98112/98112 [=====] - 36s - loss: 0.0044 - mean_squared_error: 0.0044 - val_loss: 0.0037 -
val_mean_squared_error: 0.0037
Epoch 3/5
Train on 98112 samples, validate on 42048 samples
Epoch 1/1
98112/98112 [=====] - 38s - loss: 0.0043 - mean_squared_error: 0.0043 - val_loss: 0.0038 -
val_mean_squared_error: 0.0038
Epoch 4/5
Train on 98112 samples, validate on 42048 samples
Epoch 1/1
98112/98112 [=====] - 37s - loss: 0.0042 - mean_squared_error: 0.0042 - val_loss: 0.0038 -
val_mean_squared_error: 0.0038
Epoch 5/5
Train on 98112 samples, validate on 42048 samples
Epoch 1/1
98112/98112 [=====] - 37s - loss: 0.0040 - mean_squared_error: 0.0040 - val_loss: 0.0035 -
val_mean_squared_error: 0.0035
41952/42048 [=====>.] - ETA: 0s
MSE: 0.003, RMSE: 0.059
```

As you can see, the stateful model produces results that are slightly better than the stateless model. In absolute terms, since we have scaled our data to the $(0, 1)$ range, this means that the stateless model has about 6.2% error rate and the stateful model has a 5.9% error rate, or conversely, they are about 93.8% and 94.1% accurate respectively. In relative terms, therefore, our stateful model outperforms the stateless model by a slight margin.

The source code for this example is provided in the files `econs_data.py` that parses the dataset, and `econs_stateful.py` that defines and trains the stateless and stateful models, available from the code download for this chapter.

Other RNN variants

We will round up this chapter by looking at some more variants of the RNN cell. RNN is an area of active research and many researchers have suggested variants for specific purposes.

One popular LSTM variant is adding *peephole connections*, which means that the gate layers are allowed to peek at the cell state. This was introduced by Gers and Schmidhuber (for more information refer to the article: *Learning Precise Timing with LSTM Recurrent Networks*, by F. A. Gers, N. N. Schraudolph, and J. Schmidhuber, Journal of Machine Learning Research, pp. 115-43) in 2002.

Another LSTM variant, that ultimately led to the GRU, is to use coupled forget and output gates. Decisions about what information to forget and what to acquire are made together, and the new information replaces the forgotten information.

Keras provides only the three basic variants, namely the SimpleRNN, LSTM, and GRU layers. However, that isn't necessarily a problem. Gref conducted an experimental survey (for more information refer to the article: *LSTM: A Search Space Odyssey*, by K. Greff, arXiv:1503.04069, 2015) of many LSTM variants, and concluded that none of the variants improved significantly over the standard LSTM architecture. So the components provided in Keras are usually sufficient to solve most problems.

In case you do need the capability to construct your own layer, you can build custom Keras layers. We will look at how to build a custom layer in the next chapter. There is also an open source framework called recurrent shop (<https://github.com/datalogai/recurrentshop>) that allows you to build complex recurrent neural networks with Keras.

Summary

In this chapter, we looked at the basic architecture of recurrent neural networks and how they work better than traditional neural networks over sequence data. We saw how RNNs can be used to learn an author's writing style and generate text using the learned model. We also saw how this example can be extended to predicting stock prices or other time series, speech from noisy audio, and so on, as well as generate music that was composed by a learned model.

We looked at different ways to compose our RNN units and these topologies can be used to model and solve specific problems such as sentiment analysis, machine translation, image captioning, and classification, and so on.

We then looked at one of the biggest drawbacks of the SimpleRNN architecture, that of vanishing and exploding gradients. We saw how the vanishing gradient problem is handled using the LSTM (and GRU) architectures. We also looked at the LSTM and GRU architectures in some detail. We also saw two examples of predicting sentiment using an LSTM-based model, and predicting POS tags using a GRU-based sequence-to-sequence architecture.

We then learned about stateful RNNs and how they can be used in Keras. We also saw an example of learning a stateful RNN to predict CO levels in the atmosphere.

Finally, we learned about some RNN variants that are not available in Keras, and briefly explored how to build them.

In the next chapter, we will look at models that don't quite fit into the basic molds we have looked at so far. We will also look at composing these basic models larger and more complex ones using the Keras functional API, as well as look at some examples of customizing Keras to our needs.

7

Additional Deep Learning Models

So far, most of the discussion has been focused around different models that do classification. These models are trained using object features and their labels to predict labels for hitherto unseen objects. The models also had a fairly simple architecture, all the ones we have seen so far have a linear pipeline modeled by the Keras sequential API.

In this chapter, we will focus on more complex architectures where the pipelines are not necessarily linear. Keras provides the functional API to deal with these sorts of architectures. We will learn how to define our networks using the functional API in this chapter. Note that the functional API can be used to build linear architectures as well.

The simplest extension of classification networks are regression networks. The two broad subcategories under supervised machine learning are classification and regression. Instead of predicting a category, the network now predicts a continuous value. You saw an example of a regression network when we discussed stateless versus stateful RNNs. Many regression problems can be solved using classification models with very little effort. We will see an example of such a network to predict atmospheric benzene in this chapter.

Yet another class of models deal with learning the structure of the data from unlabeled data. These are called **unsupervised** (or more correctly, self-supervised) models. They are similar to classification models, but the labels are available implicitly within the data. We have already seen examples of this kind of model; for example, the CBOW and skip-gram word2vec models are self-supervised models. Autoencoders are another example of this type of model. We will learn about autoencoders and describe an example that builds compact vector representations of sentences.

We will then look at how to compose the networks we have seen so far into larger computation graphs. These graphs are often built to achieve some custom objective that is not achievable by a sequential model alone, and may have multiple inputs and outputs and connections to external components. We will see an example of composing such a network for question answering.

We then take a detour to look at the Keras backend API, and how we can use this API to build custom components to extend Keras' functionality.

Going back to models for unlabeled data, another class of models that don't require labels are generative models. These models are trained using a set of existing objects and attempt to learn the distribution these objects come from. Once the distribution is learned, we can draw samples from this distribution that look like the original training data. We have seen an example of this where we trained a character RNN model to generate text similar to *Alice in Wonderland* in the previous chapter. The idea is already covered, so we won't cover this particular aspect of generative models here. However, we will look at how we can leverage the idea of a trained network learning the data distribution to create interesting visual effects using a VGG-16 network pre-trained on ImageNet data.

To summarize, we will learn the following topics in this chapter:

- The Keras functional API
- Regression networks
- Autoencoders for unsupervised learning
- Composing complex networks with the functional API
- Customizing Keras
- Generative networks

Let's get started.

Keras functional API

The Keras functional API defines each layer as a function and provides operators to compose these functions into a larger computational graph. A function is some sort of transformation with a single input and single output. For example, the function $y = f(x)$ defines a function f with input x and output y . Let us consider the simple sequential model from Keras (for more information refer to: <https://keras.io/getting-started/sequential-model-guide/>):

```
from keras.models import Sequential
from keras.layers.core import dense, Activation

model = Sequential([
    dense(32, input_dim=784),
    Activation("sigmoid"),
    dense(10),
    Activation("softmax"),
])

model.compile(loss="categorical_crossentropy", optimizer="adam")
```

As you can see, the sequential model represents the network as a linear pipeline, or list, of layers. We can also represent the network as the composition of the following nested functions. Here x is the input tensor of shape $(None, 784)$ and y is the output tensor of $(None, 10)$. Here $None$ refers to the as-yet undetermined batch size:

$$y = \sigma_K(f(\sigma_2(g(x))))$$

Where:

$$\begin{aligned}g(x) &= W_g x + b_g \\ \sigma_2(x) &= \frac{1}{1 + e^{-x}} \\ f(x) &= W_f x + b_f \\ \sigma_K(x) &= \frac{e^x}{\sum_{k=1}^K e^{x_k}}\end{aligned}$$

The network can be redefined using the Keras functional API as follows. Notice how the predictions variable is a composition of the same functions we defined in equation form previously:

```
from keras.layers import Input
from keras.layers.core import dense
from keras.models import Model
from keras.layers.core import Activation

inputs = Input(shape=(784,))

x = dense(32)(inputs)
x = Activation("sigmoid")(x)
x = dense(10)(x)
predictions = Activation("softmax")(x)

model = Model(inputs=inputs, outputs=predictions)

model.compile(loss="categorical_crossentropy", optimizer="adam")
```

Since a model is a composition of layers that are also functions, a model is also a function. Therefore, you can treat a trained model as just another layer by calling it on an appropriately shaped input tensor. Thus, if you have built a model that does something useful like image classification, you can easily extend it to work with a sequence of images using Keras's TimeDistributed wrapper:

```
sequence_predictions = TimeDistributed(model)(input_sequences)
```

The functional API can be used to define any network that can be defined using the sequential API. In addition, the following types of network can only be defined using the functional API:

- Models with multiple inputs and outputs
- Models composed of multiple submodels
- Models that used shared layers

Models with multiple inputs and outputs are defined by composing the inputs and outputs separately, as shown in the preceding example, and then passing in an array of input functions and an array of output functions in the input and output parameters of the Model constructor:

```
model = Model(inputs=[input1, input2], outputs=[output1, output2])
```

Models with multiple inputs and outputs also generally consist of multiple subnetworks, the results of whose computations are merged into the final result. The merge function provides multiple ways to merge intermediate results such as vector addition, dot product, and concatenation. We will see examples of merging in our question answering example later in this chapter.

Another good use for the functional API are models that use shared layers. Shared layers are defined once, and referenced in each pipeline where their weights need to be shared.

We will use the functional API almost exclusively in this chapter, so you will see quite a few examples of its use. The Keras website has many more usage examples for the functional API.

Regression networks

The two major techniques of supervised learning are classification and regression. In both cases, the model is trained with data to predict known labels. In case of classification, these labels are discrete values such as genres of text or image categories. In case of regression, these labels are continuous values, such as stock prices or human intelligence quotients (IQ).

Most of the examples we have seen show deep learning models being used to perform classification. In this section, we will look at how to perform regression using such a model.

Recall that classification models have a dense layer with a nonlinear activation at the end, the output dimension of which corresponds to the number of classes the model can predict. Thus, an ImageNet image classification model has a dense (1,000) layer at the end, corresponding to 1,000 ImageNet classes it can predict. Similarly, a sentiment analysis model has a dense layer at the end, corresponding to positive or negative sentiment.

Regression models also have a dense layer at the end, but with a single output, that is, an output dimension of one, and no nonlinear activation. Thus the dense layer just returns the sum of the activations from the previous layer. In addition, the loss function used is typically **mean squared error (MSE)**, but some of the other objectives (listed on the Keras objectives page at: <https://keras.io/losses/>) can be used as well.

Keras regression example — predicting benzene levels in the air

In this example, we will predict the concentration of benzene in the atmosphere given some other variables such as concentrations of carbon monoxide, nitrous oxide, and so on in the atmosphere as well as temperature and relative humidity. The dataset we will use is the air quality dataset from the UCI Machine Learning Repository (<https://archive.ics.uci.edu/ml/datasets/Air+Quality>). The dataset contains 9,358 instances of hourly averaged readings from an array of five metal oxide chemical sensors. The sensor array was located in a city in Italy, and the recordings were made from March 2004 to February 2005.

As usual, first we import all our necessary libraries:

```
from keras.layers import Input
from keras.layers.core import dense
from keras.models import Model
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
```

The dataset is provided as a CSV file. We load the input data into a Pandas (for more information refer to: <http://pandas.pydata.org/>) data frame. Pandas is a popular data analysis library built around data frames, a concept borrowed from the R language. We use Pandas here to read the dataset for two reasons. First, the dataset contains empty fields where they could not be recorded for some reason. Second, the dataset uses commas for decimal points, a custom common in some European countries. Pandas has built-in support to handle both situations, along with a few other conveniences, as we will see soon:

```
DATA_DIR = "../data"
AIRQUALITY_FILE = os.path.join(DATA_DIR, "AirQualityUCI.csv")

aqdf = pd.read_csv(AIRQUALITY_FILE, sep=";", decimal=",", header=0)

# remove first and last 2 cols
del aqdf["Date"]
del aqdf["Time"]
del aqdf["Unnamed: 15"]
del aqdf["Unnamed: 16"]

# fill NaNs in each column with the mean value
aqdf = aqdf.fillna(aqdf.mean())

Xorig = aqdf.as_matrix()
```

The preceding example removes the first two columns, which contains the observation date and time, and the last two columns which seem to be spurious. Next we replace the empty fields with the average value for the column. Finally, we export the data frame as a matrix for downstream use.

One thing to note is that each column of the data has different scales since they measure different quantities. For example, the concentration of tin oxide is in the 1,000 range, while non-methanic hydrocarbons is in the 100 range. In many situations our features are homogeneous so scaling is not an issue, but in cases like this it is generally a good practice to scale the data. Scaling here consists of subtracting from each column the mean of the column and dividing by its standard deviation:

$$z = \frac{x - \mu}{\sigma}$$

To do this, we use the `StandardScaler` class provided by the `scikit-learn` library, shown as follows. We store the mean and standard deviations because we will need this later when reporting results or predicting against new data. Our target variable is the fourth column in our input dataset, so we split this scaled data into input variables `x` and target variable `y`:

```
scaler = StandardScaler()
Xscaled = scaler.fit_transform(Xorig)
# store these off for predictions with unseen data
Xmeans = scaler.mean_
Xstds = scaler.scale_

y = Xscaled[:, 3]
X = np.delete(Xscaled, 3, axis=1)
```

We then split the data into the first 70% for training and the last 30% for testing. This gives us 6,549 records for training and 2,808 records for testing:

```
train_size = int(0.7 * X.shape[0])
Xtrain, Xtest, ytrain, ytest = X[0:train_size], X[train_size:],
y[0:train_size], y[train_size:]
```

Next we define our network. This is a simple two layer dense network that takes a vector of 12 features as input and outputs a scaled prediction. The hidden dense layer has eight neurons. We initialize weight matrices for both dense layers with a specific initialization scheme called *glorot uniform*. For a full list of initialization schemes, please refer to the Keras initializations here: <https://keras.io/initializers/>. The loss function used is mean squared error (`mse`) and the optimizer is adam:

```
readings = Input(shape=(12,))
x = dense(8, activation="relu",
kernel_initializer="glorot_uniform") (readings)
benzene = dense(1, kernel_initializer="glorot_uniform") (x)

model = Model(inputs=[readings], outputs=[benzene])
model.compile(loss="mse", optimizer="adam")
```

We train this model for 20 epochs and batch size of 10:

```
NUM_EPOCHS = 20
BATCH_SIZE = 10

history = model.fit(Xtrain, ytrain, batch_size=BATCH_SIZE,
epochs=NUM_EPOCHS,
validation_split=0.2)
```

This results in a model that has a mean squared error of 0.0003 (approximately 2% RMSE) on the training set and 0.0016 (approximately 4% RMSE) on the validation set, as shown in the logs of the training step here:

```
Epoch 8/20
5239/5239 [=====] - 0s - loss: 0.0015 - val_loss: 0.0024
Epoch 9/20
5239/5239 [=====] - 0s - loss: 0.0012 - val_loss: 0.0020
Epoch 10/20
5239/5239 [=====] - 0s - loss: 9.5742e-04 - val_loss: 0.0018
Epoch 11/20
5239/5239 [=====] - 0s - loss: 8.2761e-04 - val_loss: 0.0019
Epoch 12/20
5239/5239 [=====] - 0s - loss: 7.1237e-04 - val_loss: 0.0021
Epoch 13/20
5239/5239 [=====] - 0s - loss: 6.4492e-04 - val_loss: 0.0018
Epoch 14/20
5239/5239 [=====] - 0s - loss: 6.0119e-04 - val_loss: 0.0019
Epoch 15/20
5239/5239 [=====] - 0s - loss: 5.1915e-04 - val_loss: 0.0017
Epoch 16/20
5239/5239 [=====] - 0s - loss: 4.4686e-04 - val_loss: 0.0014
Epoch 17/20
5239/5239 [=====] - 0s - loss: 5.6912e-04 - val_loss: 0.0019
Epoch 18/20
5239/5239 [=====] - 0s - loss: 3.6897e-04 - val_loss: 0.0013
Epoch 19/20
5239/5239 [=====] - 0s - loss: 3.6652e-04 - val_loss: 0.0012
Epoch 20/20
5239/5239 [=====] - 0s - loss: 3.2395e-04 - val_loss: 0.0016
```

We also look at some values of benzene concentrations that were originally recorded and compare them to those predicted by our model. Both actual and predicted values are rescaled from their scaled z-values to actual values:

```
ytest_ = model.predict(Xtest).flatten()
for i in range(10):
    label = (ytest[i] * Xstds[3]) + Xmeans[3]
    prediction = (ytest_[i] * Xstds[3]) + Xmeans[3]
    print("Benzene Conc. expected: {:.3f}, predicted: {:.3f}".format(label,
prediction))
```

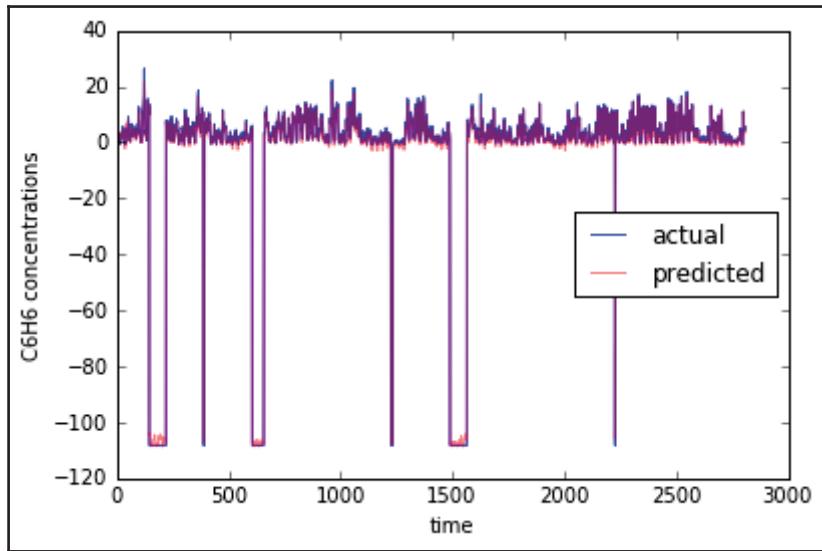
The side-by-side comparison shows that the predictions are quite close to the actual values:

```
Benzene Conc. expected: 4.600, predicted: 5.254
Benzene Conc. expected: 5.500, predicted: 4.932
Benzene Conc. expected: 6.500, predicted: 5.664
Benzene Conc. expected: 10.300, predicted: 8.482
Benzene Conc. expected: 8.900, predicted: 6.705
Benzene Conc. expected: 14.000, predicted: 12.928
Benzene Conc. expected: 9.200, predicted: 7.128
Benzene Conc. expected: 8.200, predicted: 5.983
Benzene Conc. expected: 7.200, predicted: 6.256
Benzene Conc. expected: 5.500, predicted: 5.184
```

Finally, we graph the actual values against the predictions for our entire test set. Once more, we see that the network predicts values that are very close to the expected values:

```
plt.plot(np.arange(ytest.shape[0]), (ytest * Xstds[3]) / Xmeans[3],
         color="b", label="actual")
plt.plot(np.arange(ytest_.shape[0]), (ytest_ * Xstds[3]) / Xmeans[3],
         color="r", alpha=0.5, label="predicted")
plt.xlabel("time")
plt.ylabel("C6H6 concentrations")
plt.legend(loc="best")
plt.show()
```

The output of the preceding example is as follows:



Unsupervised learning — autoencoders

Autoencoders are a class of neural network that attempt to recreate the input as its target using back-propagation. An autoencoder consists of two parts, an encoder and a decoder. The encoder will read the input and compress it to a compact representation, and the decoder will read the compact representation and recreate the input from it. In other words, the autoencoder tries to learn the identity function by minimizing the reconstruction error.

Even though the identity function does not seem like a very interesting function to learn, the way in which this is done makes it interesting. The number of hidden units in the autoencoder is typically less than the number of input (and output) units. This forces the encoder to learn a compressed representation of the input which the decoder reconstructs. If there is structure in the input data in the form of correlations between input features, then the autoencoder will discover some of these correlations, and end up learning a low dimensional representation of the data similar to that learned using **principal component analysis (PCA)**.

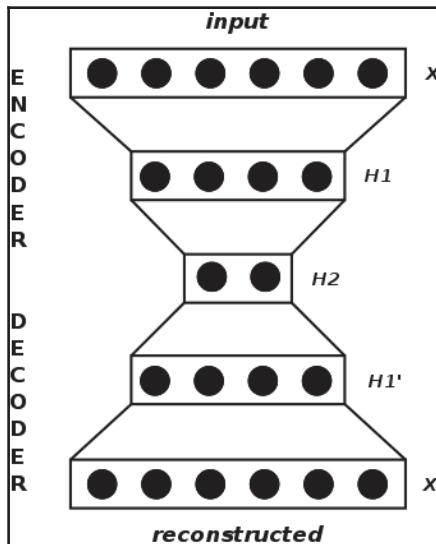
Once the autoencoder is trained, we would typically just discard the decoder component and use the encoder component to generate compact representations of the input.

Alternatively, we could use the encoder as a feature detector that generates a compact, semantically rich representation of our input and build a classifier by attaching a softmax classifier to the hidden layer.

The encoder and decoder components of an autoencoder can be implemented using either dense, convolutional, or recurrent networks, depending on the kind of data that is being modeled. For example, dense networks might be a good choice for autoencoders used to build **collaborative filtering (CF)** models (for more information refer to the articles: *AutoRec: Autoencoders Meet Collaborative Filtering*, by S. Sedhain, Proceedings of the 24th International Conference on World Wide Web, ACM, 2015 and *Wide & Deep Learning for Recommender Systems*, by H. Cheng, Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, ACM, 2016), where we learn a compressed model of user preferences based on actual sparse user ratings. Similarly, convolutional neural networks may be appropriate for the use case covered in the article: *See: Using Deep Learning to Remove Eyeglasses from Faces*, by M. Runfeldt. and recurrent networks a good choice for autoencoders building on text data, such as deep patient (for more information refer to the article: *Deep Patient: An Unsupervised Representation to Predict the Future of Patients from the Electronic Health Records*, by R. Miotto, Scientific Reports 6, 2016) and skip-thought vectors ((for more information refer to the article: *Skip-Thought Vectors*, by R. Kiros, Advances in Neural Information Processing Systems, 2015).

Autoencoders can also be stacked by successively stacking encoders that compress their input to smaller and smaller representations, and stacking decoders in the opposite sequence. Stacked autoencoders have greater expressive power and the successive layers of representations capture a hierarchical grouping of the input, similar to the convolution and pooling operations in convolutional neural networks.

Stacked autoencoders used to be trained layer by layer. For example, in the network shown next, we would first train layer X to reconstruct layer X' using the hidden layer $H1$ (ignoring $H2$). We would then train the layer $H1$ to reconstruct layer $H1'$ using the hidden layer $H2$. Finally, we would stack all the layers together in the configuration shown and fine tune it to reconstruct X' from X. With better activation and regularization functions nowadays, however, it is quite common to train these networks in totality:



The Keras blog post, *Building Autoencoders in Keras* (<https://blog.keras.io/building-autoencoders-in-keras.html>) has great examples of building autoencoders that reconstructs MNIST digit images using fully connected and convolutional neural networks. It also has a good discussion on denoising and variational autoencoders, which we will not cover here.

Keras autoencoder example — sentence vectors

In this example, we will build and train an LSTM-based autoencoder to generate sentence vectors for documents in the Reuters-21578 corpus (<https://archive.ics.uci.edu/ml/datasets/Reuters-21578+Text+Categorization+Collection>). We have already seen in Chapter 5, *Word Embeddings*, how to represent a word using word embeddings to create vectors that represent its meaning in the context of other words it appears with. Here, we will see how to build similar vectors for sentences. Sentences are a sequence of words, so a sentence vector represents the meaning of the sentence.

The easiest way to build a sentence vector is to just add up the word vectors and divide by the number of words. However, this treats the sentence as a bag of words, and does not take the order of words into account. Thus the sentences *The dog bit the man* and *The man bit the dog* would be treated as identical under this scenario. LSTMs are designed to work with sequence input and do take the order of words into consideration thus providing a better and more natural representation for the sentence.

First we import the necessary libraries:

```
from sklearn.model_selection import train_test_split
from keras.callbacks import ModelCheckpoint
from keras.layers import Input
from keras.layers.core import RepeatVector
from keras.layers.recurrent import LSTM
from keras.layers.wrappers import Bidirectional
from keras.models import Model
from keras.preprocessing import sequence
from scipy.stats import describe
import collections
import matplotlib.pyplot as plt
import nltk
import numpy as np
import os
```

The data is provided as a set of SGML files. We have already parsed and consolidated this data into a single text file in Chapter 6, *Recurrent Neural Network — RNN*, for our GRU-based POS tagging example. We will reuse this data to first convert each block of text into a list of sentences, one sentence per line:

```
sents = []
fsent = open(sent_filename, "rb")
for line in fsent:
    docid, sent_id, sent = line.strip().split("t")
    sents.append(sent)
fsent.close()
```

To build up our vocabulary, we read this list of sentences again, word by word. Each word is normalized as it is added. The normalization is to replace any token that looks like a number with the digit 9 and to lowercase them. The result is the word frequency table, `word_freqs`. We also compute the sentence length for each sentence and create a list of parsed sentences by rejoining the tokens with space so it is easier to parse in a subsequent step:

```
def is_number(n):
    temp = re.sub("[.,-/]\"", "", n)
    return temp.isdigit()
```

```
word_freqs = collections.Counter()
sent_lens = []
parsed_sentences = []
for sent in sentences:
    words = nltk.word_tokenize(sent)
    parsed_words = []
    for word in words:
        if is_number(word):
            word = "9"
        word_freqs[word.lower()] += 1
        parsed_words.append(word)
    sent_lens.append(len(words))
    parsed_sentences.append(" ".join(parsed_words))
```

This gives us some information about the corpus that will help us figure out good values for our constants for our LSTM network:

```
sent_lens = np.array(sent_lens)
print("number of sentences: {:d}".format(len(sent_lens)))
print("distribution of sentence lengths (number of words)")
print("min:{:d}, max:{:d}, mean:{:.3f}, med:{:.3f}".format(
    np.min(sent_lens), np.max(sent_lens), np.mean(sent_lens),
    np.median(sent_lens)))
print("vocab size (full): {:d}".format(len(word_freqs)))
```

This gives us the following information about the corpus:

number of sentences: 131545
distribution of sentence lengths (number of words)
min: 1, max: 429, mean: 22.315, median: 21.000
vocab size (full): 50751

Based on this information, we set the following constants for our LSTM model. We choose our VOCAB_SIZE as 5000, that is, our vocabulary covers the most frequent 5,000 words that cover over 93% of the words used in the corpus. The remaining words are treated as **out of vocabulary (OOV)** and replaced with the token UNK. At prediction time, any word that the model hasn't seen will also be assigned the token UNK. SEQUENCE_LEN is set to approximately twice the median length of sentences in the training set, and indeed, approximately 110 million of our 131 million sentences are shorter than this setting. Sentences that are shorter than SEQUENCE_LENGTH will be padded by a special PAD character, and those that are longer will be truncated to fit the limit:

```
VOCAB_SIZE = 5000
SEQUENCE_LEN = 50
```

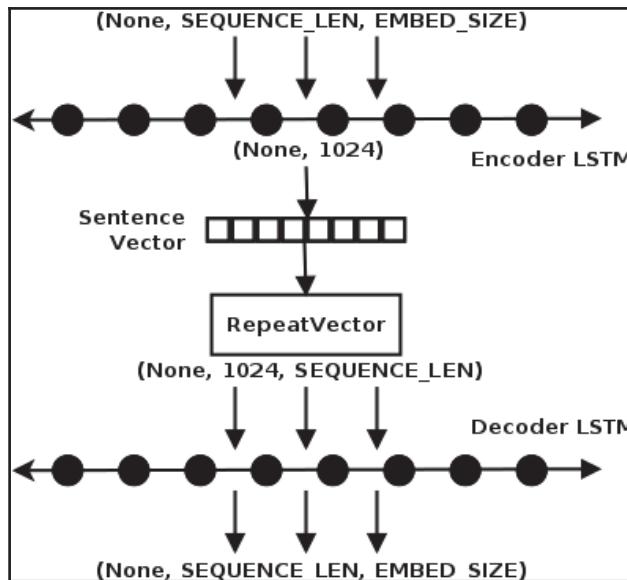
Since the input to our LSTM will be numeric, we need to build lookup tables that go back and forth between words and word IDs. Since we limit our vocabulary size to 5,000 and we have to add the two pseudo-words PAD and UNK, our lookup table contains entries for the most frequently occurring 4,998 words plus PAD and UNK:

```
word2id = {}  
word2id["PAD"] = 0  
word2id["UNK"] = 1  
for v, (k, _) in enumerate(word_freqs.most_common(VOCAB_SIZE - 2)):  
    word2id[k] = v + 2  
id2word = {v:k for k, v in word2id.items() }
```

The input to our network is a sequence of words, where each word is represented by a vector. Simplistically, we could just use a one-hot encoding for each word, but that makes the input data very large. So we encode each word using its 50-dimensional GloVe embeddings. The embedding is generated into a matrix of shape (VOCAB_SIZE, EMBED_SIZE) where each row represents the GloVe embedding for a word in our vocabulary. The PAD and UNK rows (0 and 1 respectively) are populated with zeros and random uniform values respectively:

```
EMBED_SIZE = 50  
  
def lookup_word2id(word):  
    try:  
        return word2id[word]  
    except KeyError:  
        return word2id["UNK"]  
  
def load_glove_vectors(glove_file, word2id, embed_size):  
    embedding = np.zeros((len(word2id), embed_size))  
    fglove = open(glove_file, "rb")  
    for line in fglove:  
        cols = line.strip().split()  
        word = cols[0]  
        if embed_size == 0:  
            embed_size = len(cols) - 1  
        if word2id.has_key(word):  
            vec = np.array([float(v) for v in cols[1:]])  
            embedding[lookup_word2id(word)] = vec  
    embedding[word2id["PAD"]] = np.zeros((embed_size))  
    embedding[word2id["UNK"]] = np.random.uniform(-1, 1, embed_size)  
    return embedding  
  
embeddings = load_glove_vectors(os.path.join(  
    DATA_DIR, "glove.6B.{:d}d.txt".format(EMBED_SIZE)), word2id,  
    EMBED_SIZE)
```

Our autoencoder model takes a sequence of GloVe word vectors and learns to produce another sequence that is similar to the input sequence. The encoder LSTM compresses the sequence into a fixed size context vector, which the decoder LSTM uses to reconstruct the original sequence. A schematic of the network is shown here:



Because the input is quite large, we will use a generator to produce each batch of input. Our generator produces batches of tensors of shape $(BATCH_SIZE, SEQUENCE_LEN, EMBED_SIZE)$. Here $BATCH_SIZE$ is 64, and since we are using 50-dimensional GloVe vectors, $EMBED_SIZE$ is 50. We shuffle the sentences at the beginning of each epoch, and return batches of 64 sentences. Each sentence is represented as a vector of GloVe word vectors. If a word in the vocabulary does not have a corresponding GloVe embedding, it is represented by a zero vector. We construct two instances of the generator, one for training data and one for test data, consisting of 70% and 30% of the original dataset respectively:

```
BATCH_SIZE = 64

def sentence_generator(X, embeddings, batch_size):
    while True:
        # loop once per epoch
        num_recs = X.shape[0]
        indices = np.random.permutation(np.arange(num_recs))
        num_batches = num_recs // batch_size
        for bid in range(num_batches):
            sids = indices[bid * batch_size : (bid + 1) * batch_size]
            Xbatch = embeddings[X[sids, :]]
```

```
yield Xbatch, Xbatch

train_size = 0.7
Xtrain, Xtest = train_test_split(sent_wids, train_size=train_size)
train_gen = sentence_generator(Xtrain, embeddings, BATCH_SIZE)
test_gen = sentence_generator(Xtest, embeddings, BATCH_SIZE)
```

Now we are ready to define the autoencoder. As we have shown in the diagram, it is composed of an encoder LSTM and a decoder LSTM. The encoder LSTM reads a tensor of shape (BATCH_SIZE, SEQUENCE_LEN, EMBED_SIZE) representing a batch of sentences. Each sentence is represented as a padded fixed-length sequence of words of size SEQUENCE_LEN. Each word is represented as a 300-dimensional GloVe vector. The output dimension of the encoder LSTM is a hyperparameter LATENT_SIZE, which is the size of the sentence vector that will get out of the encoder part of the trained autoencoder later. The vector space of dimensionality LATENT_SIZE represents the latent space that encodes the meaning of the sentence. The output of the LSTM is a vector of size (LATENT_SIZE) for each sentence, so for the batch the shape of the output tensor is (BATCH_SIZE, LATENT_SIZE). This is now fed to a RepeatVector layer, which replicates this across the entire sequence, that is., the output tensor from this layer has the shape (BATCH_SIZE, SEQUENCE_LEN, LATENT_SIZE). This tensor is now fed into the decoder LSTM, whose output dimension is the EMBED_SIZE, so the output tensor has shape (BATCH_SIZE, SEQUENCE_LEN, EMBED_SIZE), that is, the same shape as the input tensor.

We compile this model with the SGD optimizer and the mse loss function. The reason we use MSE is that we want to reconstruct a sentence that has a similar meaning, that is, something that is close to the original sentence in the embedded space of dimension LATENT_SIZE:

```
inputs = Input(shape=(SEQUENCE_LEN, EMBED_SIZE), name="input")
encoded = Bidirectional(LSTM(LATENT_SIZE), merge_mode="sum",
                       name="encoder_lstm")(inputs)
decoded = RepeatVector(SEQUENCE_LEN, name="repeater")(encoded)
decoded = Bidirectional(LSTM(EMBED_SIZE, return_sequences=True),
                       merge_mode="sum",
                       name="decoder_lstm")(decoded)

autoencoder = Model(inputs, decoded)

autoencoder.compile(optimizer="sgd", loss="mse")
```

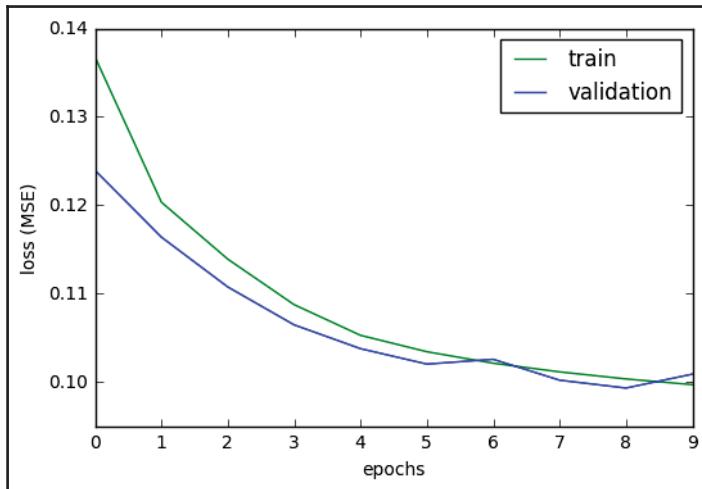
We train the autoencoder for 10 epochs using the following code. 10 epochs were chosen because the MSE loss converges within this time. We also save the best model retrieved so far based on the MSE loss:

```
num_train_steps = len(Xtrain) // BATCH_SIZE
num_test_steps = len(Xtest) // BATCH_SIZE
checkpoint = ModelCheckpoint(filepath=os.path.join(DATA_DIR,
    "sent-thoughts-autoencoder.h5"), save_best_only=True)
history = autoencoder.fit_generator(train_gen,
    steps_per_epoch=num_train_steps,
    epochs=NUM_EPOCHS,
    validation_data=test_gen,
    validation_steps=num_test_steps,
    callbacks=[checkpoint])
```

The results of the training are shown as follows. As you can see, the training MSE reduces from 0.14 to 0.1 and the validation MSE reduces from 0.12 to 0.1:

```
Epoch 1/10
92032/92032 [=====] - 542s - loss: 0.1368 - val_loss: 0.1239
Epoch 2/10
92032/92032 [=====] - 540s - loss: 0.1203 - val_loss: 0.1164
Epoch 3/10
92032/92032 [=====] - 546s - loss: 0.1139 - val_loss: 0.1107
Epoch 4/10
92032/92032 [=====] - 547s - loss: 0.1087 - val_loss: 0.1064
Epoch 5/10
92032/92032 [=====] - 542s - loss: 0.1053 - val_loss: 0.1038
Epoch 6/10
92032/92032 [=====] - 543s - loss: 0.1034 - val_loss: 0.1020
Epoch 7/10
92032/92032 [=====] - 544s - loss: 0.1021 - val_loss: 0.1025
Epoch 8/10
92032/92032 [=====] - 545s - loss: 0.1011 - val_loss: 0.1002
Epoch 9/10
92032/92032 [=====] - 545s - loss: 0.1003 - val_loss: 0.0993
Epoch 10/10
92032/92032 [=====] - 545s - loss: 0.0997 - val_loss: 0.1009
```

Or, graphically it shows as follows:



Since we are feeding in a matrix of embeddings, the output will also be a matrix of word embeddings. Since the embedding space is continuous and our vocabulary is discrete, not every output embedding will correspond to a word. The best we can do is to find a word that is closest to the output embedding in order to reconstruct the original text. This is a bit cumbersome, so we will evaluate our autoencoder in a different way.

Since the objective of the autoencoder is to produce a good latent representation, we compare the latent vectors produced from the encoder using the original input versus the output of the autoencoder. First, we extract the encoder component into its own network:

```
encoder = Model(autoencoder.input,  
                 autoencoder.get_layer("encoder_lstm").output)
```

Then we run the autoencoder on the test set to return the predicted embeddings. We then send both the input embedding and the predicted embedding through the encoder to produce sentence vectors from each, and compare the two vectors using *cosine* similarity. Cosine similarities close to one indicate high similarity and those close to zero indicate low similarity. The following code runs against a random subset of 500 test sentences and produces some sample values of cosine similarities between the sentence vectors generated from the source embedding and the corresponding target embedding produced by the autoencoder:

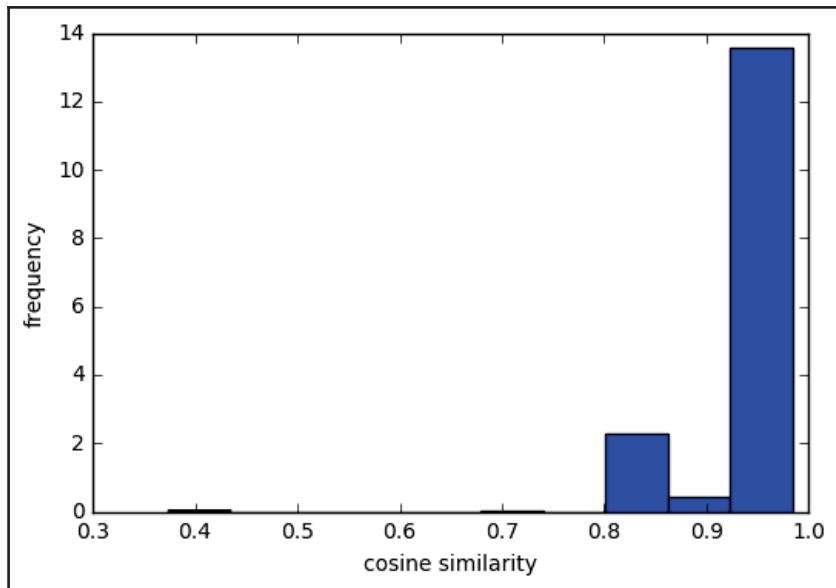
```
def compute_cosine_similarity(x, y):  
    return np.dot(x, y) / (np.linalg.norm(x, 2) * np.linalg.norm(y, 2))  
  
k = 500
```

```
cosims = np.zeros((k))
i = 0
for bid in range(num_test_steps):
    xtest, ytest = test_gen.next()
    ytest_ = autoencoder.predict(xtest)
    Xvec = encoder.predict(xtest)
    Yvec = encoder.predict(ytest_)
    for rid in range(Xvec.shape[0]):
        if i >= k:
            break
        cosims[i] = compute_cosine_similarity(Xvec[rid], Yvec[rid])
        if i <= 10:
            print(cosims[i])
        i += 1
    if i >= k:
        break
```

The first 10 values of cosine similarities are shown as follows. As we can see, the vectors seem to be quite similar:

```
0.982818722725
0.970908224583
0.98131018877
0.974798440933
0.968060493469
0.976065933704
0.96712064743
0.949920475483
0.973583400249
0.980291545391
0.817819952965
```

A histogram of the distribution of values of cosine similarities for the sentence vectors from the first 500 sentences in the test set are shown as follows. As previously, it confirms that the sentence vectors generated from the input and output of the autoencoder are very similar, showing that the resulting sentence vector is a good representation of the sentence:



Composing deep networks

We have looked extensively at these three basic deep learning networks—the **fully connected network (FCN)**, the CNN and the RNN models. While each of these have specific use cases for which they are most suited, you can also compose larger and more useful models by combining these models as Lego-like building blocks and using the Keras functional API to glue them together in new and interesting ways.

Such models tend to be somewhat specialized to the task for which they were built, so it is impossible to generalize about them. Usually, however, they involve learning from multiple inputs or generating multiple outputs. One example could be a question answering network, where the network learns to predict answers given a story and a question.

Another example could be a siamese network that calculates similarity between a pair of images, where the network is trained to predict either a binary (similar/not similar) or categorical (gradations of similarity) label using a pair of images as input. Yet another example could be an object classification and localization network where it learns to predict the image category as well as where the image is located in the picture jointly from the image. The first two examples are examples of composite networks with multiple inputs, and the last is an example of a composite network with multiple outputs.

Keras example — memory network for question answering

In this example, we will build a memory network for question answering. Memory networks are a specialized architecture that consist of a memory unit in addition to other learnable units, usually RNNs. Each input updates the memory state and the final output is computed by using the memory along with the output from the learnable unit. This architecture was suggested in 2014 via the paper (for more information refer to: *Memory Networks*, by J. Weston, S. Chopra, and A. Bordes, arXiv:1410.3916, 2014). A year later, another paper (for more information refer to: *Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks*, by J. Weston, arXiv:1502.05698, 2015) put forward the idea of a synthetic dataset and a standard set of 20 question answering tasks, each with a higher degree of difficulty than the previous one, and applied various deep learning networks to solve these tasks. Of these, the memory network achieved the best results across all the tasks. This dataset was later made available to the general public through Facebook's bAbI project (<https://research.fb.com/projects/babi/>). The implementation of our memory network resembles most closely the one described in this paper (for more information refer to: *End-To-End Memory Networks*, by S. Sukhbaatar, J. Weston, and R. Fergus, Advances in Neural Information Processing Systems, 2015), in that all the training happens jointly in a single network. It uses the bAbI dataset to solve the first question answering task.

First, we will import the necessary libraries:

```
from keras.layers import Input
from keras.layers.core import Activation, dense, Dropout, Permute
from keras.layers.embeddings import Embedding
from keras.layers.merge import add, concatenate, dot
from keras.layers.recurrent import LSTM
from keras.models import Model
from keras.preprocessing.sequence import pad_sequences
from keras.utils import np_utils
import collections
import itertools
import nltk
import numpy as np
import matplotlib.pyplot as plt
import os
```

The bAbI data for the first question answering task consists of 10,000 short sentences each for the training and the test sets. A story consists of two to three sentences, followed by a question. The last sentence in each story has the question and the answer appended to it at the end. The following block of code parses each of the training and test files into a list of triplets of story, question and answer:

```
DATA_DIR = ".../data"
TRAIN_FILE = os.path.join(DATA_DIR, "qa1_single-supporting-fact_train.txt")
TEST_FILE = os.path.join(DATA_DIR, "qa1_single-supporting-fact_test.txt")

def get_data(infile):
    stories, questions, answers = [], [], []
    story_text = []
    fin = open(TRAIN_FILE, "rb")
    for line in fin:
        line = line.decode("utf-8").strip()
        lno, text = line.split(" ", 1)
        if "t" in text:
            question, answer, _ = text.split("t")
            stories.append(story_text)
            questions.append(question)
            answers.append(answer)
            story_text = []
        else:
            story_text.append(text)
    fin.close()
    return stories, questions, answers

data_train = get_data(TRAIN_FILE)
data_test = get_data(TEST_FILE)
```

Our next step is to run through the texts in the generated lists and build our vocabulary. This should be quite familiar to us by now, since we have used a similar idiom a few times already. Unlike the previous time, our vocabulary is quite small, only 22 unique words, so we will not have any out of vocabulary words:

```
def build_vocab(train_data, test_data):
    counter = collections.Counter()
    for stories, questions, answers in [train_data, test_data]:
        for story in stories:
            for sent in story:
                for word in nltk.word_tokenize(sent):
                    counter[word.lower()] += 1
            for question in questions:
                for word in nltk.word_tokenize(question):
                    counter[word.lower()] += 1
        for answer in answers:
```

```
        for word in nltk.word_tokenize(answer):
            counter[word.lower()] += 1
    word2idx = {w:(i+1) for i, (w, _) in enumerate(counter.most_common())}
    word2idx["PAD"] = 0
    idx2word = {v:k for k, v in word2idx.items()}
    return word2idx, idx2word

word2idx, idx2word = build_vocab(data_train, data_test)

vocab_size = len(word2idx)
```

The memory network is based on RNNs, where each sentence in the story and question is treated as a sequence of words, so we need to find out the maximum length of the sequence for our story and question. The following block of code does this. We find that the maximum length of a story is 14 words and the maximum length of a question is just four words:

```
def get_maxlens(train_data, test_data):
    story maxlen, question maxlen = 0, 0
    for stories, questions, _ in [train_data, test_data]:
        for story in stories:
            story_len = 0
            for sent in story:
                swords = nltk.word_tokenize(sent)
                story_len += len(swords)
            if story_len > story maxlen:
                story maxlen = story_len
        for question in questions:
            question_len = len(nltk.word_tokenize(question))
            if question_len > question maxlen:
                question maxlen = question_len
    return story maxlen, question maxlen

story maxlen, question maxlen = get_maxlens(data_train, data_test)
```

As previously, the input to our RNNs is a sequence of word IDs. So we need to use our vocabulary dictionary to convert the (story, question, and answer) triplet into a sequence of integer word IDs. The next block of code does this and zero pads the resulting sequences of story and answer to the maximum sequence lengths we computed previously. At this point, we have lists of padded word ID sequences for each triplet in the training and test sets:

```
def vectorize(data, word2idx, story maxlen, question maxlen):
    Xs, Xq, Y = [], [], []
    stories, questions, answers = data
    for story, question, answer in zip(stories, questions, answers):
        xs = [[word2idx[w.lower()]] for w in nltk.word_tokenize(s)]
              for s in story]
```

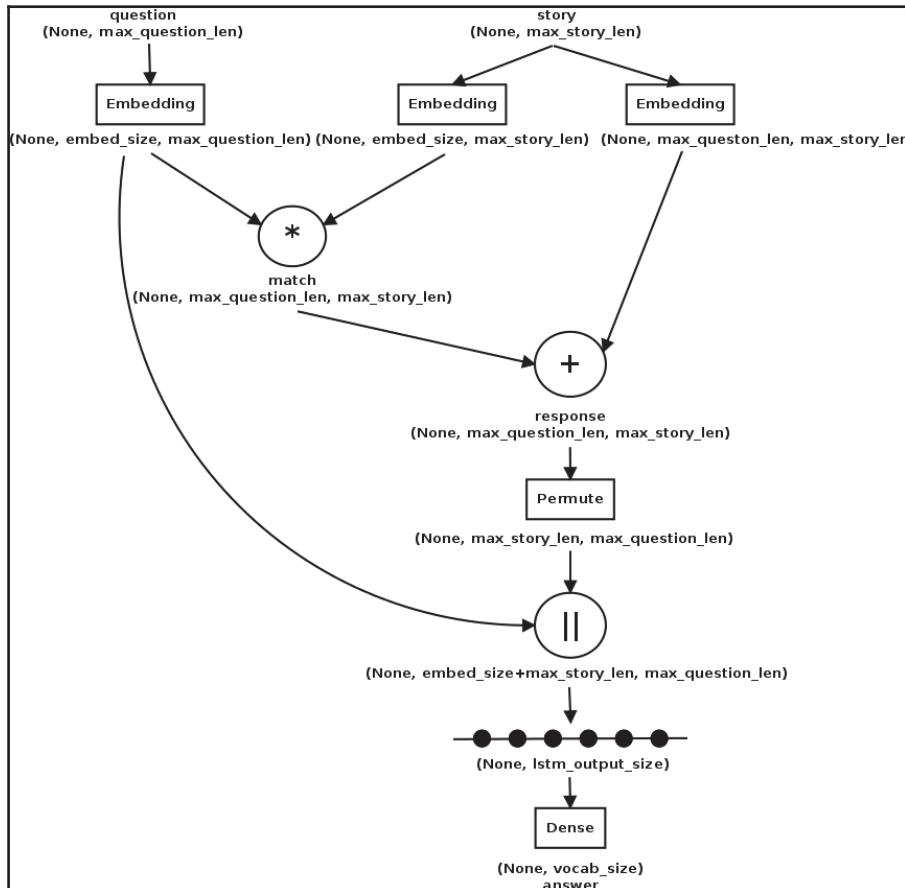
```

xs = list(itertools.chain.from_iterable(xs))
xq = [word2idx[w.lower()] for w in nltk.word_tokenize(question)]
Xs.append(xs)
Xq.append(xq)
Y.append(word2idx[answer.lower()])
return pad_sequences(Xs, maxlen=story_maxlen),
pad_sequences(Xq, maxlen=question_maxlen),
np_utils.to_categorical(Y, num_classes=len(word2idx))

Xstrain, Xqtrain, Ytrain = vectorize(data_train, word2idx, story_maxlen,
question_maxlen)
Xtest, Xqtest, Ytest = vectorize(data_test, word2idx, story_maxlen,
question_maxlen)

```

We want to define the model. The definition is longer than we have seen previously, so it may be convenient to refer to the diagram as you look through the definition:



There are two inputs to our model, the sequence of word IDs for the question and that for the sentence. Each of these is passed into an Embedding layer to convert the word IDs to a vector in the 64-dimensional embedding space. Additionally the story sequence is passed through an additional embedding that projects it to an embedding of size `max_question_length`. All these embedding layers start with random weights and are trained jointly with the rest of the network.

The first two embeddings (story and question) are merged using a dot product to form the network's memory. These represent words in the story and question that are identical or close to each other in the embedding space. The output of the memory is merged with the second story embedding and summed to form the network response, which is once again merged with the embedding for the question to form the response sequence. This response sequence is sent through an LSTM, the context vector of which is sent to a dense layer to predict the answer, which can be one of the words in the vocabulary.

The model is trained using the RMSprop optimizer and categorical cross-entropy as the loss function:

```
EMBEDDING_SIZE = 64
LATENT_SIZE = 32

# inputs
story_input = Input(shape=(story_maxlen,))
question_input = Input(shape=(question_maxlen,))

# story encoder memory
story_encoder = Embedding(input_dim=vocab_size,
                           output_dim=EMBEDDING_SIZE,
                           input_length=story_maxlen)(story_input)
story_encoder = Dropout(0.3)(story_encoder)

# question encoder
question_encoder = Embedding(input_dim=vocab_size,
                             output_dim=EMBEDDING_SIZE,
                             input_length=question_maxlen)(question_input)
question_encoder = Dropout(0.3)(question_encoder)

# match between story and question
match = dot([story_encoder, question_encoder], axes=[2, 2])

# encode story into vector space of question
story_encoder_c = Embedding(input_dim=vocab_size,
                           output_dim=question_maxlen,
                           input_length=story_maxlen)(story_input)
story_encoder_c = Dropout(0.3)(story_encoder_c)
```

```
# combine match and story vectors
response = add([match, story_encoder_c])
response = Permute((2, 1))(response)

# combine response and question vectors
answer = concatenate([response, question_encoder], axis=-1)
answer = LSTM(LATENT_SIZE)(answer)
answer = Dropout(0.3)(answer)
answer = dense(vocab_size)(answer)
output = Activation("softmax")(answer)

model = Model(inputs=[story_input, question_input], outputs=output)
model.compile(optimizer="rmsprop", loss="categorical_crossentropy",
metrics=["accuracy"])
```

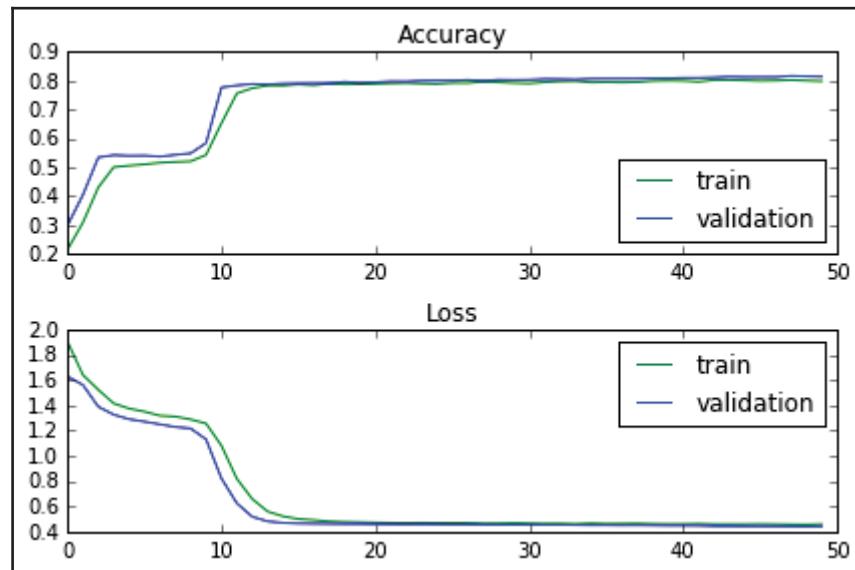
We train this network for 50 epochs with a batch size of 32 and achieve an accuracy of over 81% on the validation set:

```
BATCH_SIZE = 32
NUM_EPOCHS = 50
history = model.fit([Xstrain, Xqtrain], [Ytrain], batch_size=BATCH_SIZE,
epochs=NUM_EPOCHS,
validation_data=([Xtest, Xqtest], [Ytest]))
```

Here is the trace of the training logs:

```
Epoch 38/50
10000/10000 [=====] - 5s - loss: 0.4636 - acc: 0.7952 - val_loss: 0.4499 - val_acc: 0.8071
Epoch 39/50
10000/10000 [=====] - 5s - loss: 0.4603 - acc: 0.7993 - val_loss: 0.4489 - val_acc: 0.8083
Epoch 40/50
10000/10000 [=====] - 5s - loss: 0.4590 - acc: 0.8003 - val_loss: 0.4475 - val_acc: 0.8086
Epoch 41/50
10000/10000 [=====] - 5s - loss: 0.4592 - acc: 0.7997 - val_loss: 0.4472 - val_acc: 0.8099
Epoch 42/50
10000/10000 [=====] - 5s - loss: 0.4611 - acc: 0.7966 - val_loss: 0.4466 - val_acc: 0.8099
Epoch 43/50
10000/10000 [=====] - 5s - loss: 0.4577 - acc: 0.8025 - val_loss: 0.4437 - val_acc: 0.8114
Epoch 44/50
10000/10000 [=====] - 5s - loss: 0.4576 - acc: 0.8023 - val_loss: 0.4431 - val_acc: 0.8136
Epoch 45/50
10000/10000 [=====] - 5s - loss: 0.4575 - acc: 0.8013 - val_loss: 0.4422 - val_acc: 0.8127
Epoch 46/50
10000/10000 [=====] - 5s - loss: 0.4587 - acc: 0.7998 - val_loss: 0.4420 - val_acc: 0.8127
Epoch 47/50
10000/10000 [=====] - 6s - loss: 0.4574 - acc: 0.8005 - val_loss: 0.4412 - val_acc: 0.8126
Epoch 48/50
10000/10000 [=====] - 5s - loss: 0.4559 - acc: 0.8023 - val_loss: 0.4408 - val_acc: 0.8168
Epoch 49/50
10000/10000 [=====] - 6s - loss: 0.4550 - acc: 0.8003 - val_loss: 0.4395 - val_acc: 0.8154
Epoch 50/50
10000/10000 [=====] - 5s - loss: 0.4577 - acc: 0.7985 - val_loss: 0.4407 - val_acc: 0.8139
```

The change in training and validation loss and accuracy for this training run is shown graphically in this graph:



We ran the model against the first 10 stories from our test set to verify how good the predictions were:

```
ytest = np.argmax(Ytest, axis=1)
Ytest_ = model.predict([Xtest, Xqtest])
ytest_ = np.argmax(Ytest_, axis=1)

for i in range(NUM_DISPLAY):
    story = " ".join([idx2word[x] for x in Xtest[i].tolist() if x != 0])
    question = " ".join([idx2word[x] for x in Xqtest[i].tolist()])
    label = idx2word[ytest[i]]
    prediction = idx2word[ytest_[i]]
    print(story, question, label, prediction)
```

As you can see, the predictions were mostly correct:

Story	Question	Answer	Predicted
mary moved to the bathroom . john went to the hallway .	where is mary ?	bathroom	bathroom
daniel went back to the hallway . sandra moved to the garden .	where is daniel ?	hallway	hallway
john moved to the office . sandra journeyed to the bathroom .	where is daniel ?	hallway	kitchen
mary moved to the hallway . daniel travelled to the office .	where is daniel ?	office	office
john went back to the garden . john moved to the bedroom .	where is sandra ?	bathroom	bedroom
sandra travelled to the office . sandra went to the bathroom .	where is sandra ?	bathroom	bathroom
mary went to the bedroom . daniel moved to the hallway .	where is sandra ?	bathroom	garden
john went to the garden . john travelled to the office .	where is sandra ?	bathroom	bathroom
daniel journeyed to the bedroom . daniel travelled to the hallway .	where is john ?	office	kitchen
john went to the bedroom . john travelled to the office .	where is daniel ?	hallway	kitchen

Customizing Keras

Just as composing our basic building blocks into larger architectures enables us to build interesting deep learning models, sometimes we need to look at the other end of the spectrum. Keras has a lot of functionality built in already, so it is very likely that you can build all your models with the provided components and not feel the need for customization at all. In case you do need customization, Keras has you covered.

As you will recall, Keras is a high level API that delegates to either a TensorFlow or Theano backend for the computational heavy lifting. Any code you build for your customization will call out to one of these backends. In order to keep your code portable across the two backends, your custom code should use the Keras backend API (<https://keras.io/backend/>), which provides a set of functions that act like a facade over your chosen backend. Depending on the backend selected, the call to the backend facade will translate to the appropriate TensorFlow or Theano call. The full list of functions available and their detailed descriptions can be found on the Keras backend page.

In addition to portability, using the backend API also results in more maintainable code, since Keras code is generally more high-level and compact compared to equivalent TensorFlow or Theano code. In the unlikely case that you do need to switch to using the backend directly, your Keras components can be used directly inside TensorFlow (not Theano though) code as described in this Keras blog (<https://blog.keras.io/keras-as-a-simplified-interface-to-tensorflow-tutorial.html>).

Customizing Keras typically means writing your own custom layer or custom distance function. In this section, we will demonstrate how to build some simple Keras layers. You will see more examples of using the backend functions to build other custom Keras components, such as objectives (loss functions), in subsequent sections.

Keras example — using the lambda layer

Keras provides a lambda layer; it can wrap a function of your choosing. For example, if you wanted to build a layer that squares its input tensor element-wise, you can say simply:

```
model.add(lambda(x: x ** 2))
```

You can also wrap functions within a lambda layer. For example, if you want to build a custom layer that computes the element-wise euclidean distance between two input tensors, you would define the function to compute the value itself, as well as one that returns the output shape from this function, like so:

```
def euclidean_distance(vecs):
    x, y = vecs
    return K.sqrt(K.sum(K.square(x - y), axis=1, keepdims=True))

def euclidean_distance_output_shape(shapes):
    shape1, shape2 = shapes
    return (shape1[0], 1)
```

You can then call these functions using the lambda layer shown as follows:

```
lhs_input = Input(shape=(VECTOR_SIZE,))
lhs = dense(1024, kernel_initializer="glorot_uniform",
activation="relu")(lhs_input)

rhs_input = Input(shape=(VECTOR_SIZE,))
rhs = dense(1024, kernel_initializer="glorot_uniform",
activation="relu")(rhs_input)

sim = lambda(euclidean_distance,
output_shape=euclidean_distance_output_shape)([lhs, rhs])
```

Keras example — building a custom normalization layer

While the lambda layer can be very useful, sometimes you need more control. As an example, we will look at the code for a normalization layer that implements a technique called **local response normalization**. This technique normalizes the input over local input regions, but has since fallen out of favor because it turned out not to be as effective as other regularization methods such as dropout and batch normalization, as well as better initialization methods.

Building custom layers typically involves working with the backend functions, so it involves thinking about the code in terms of tensors. As you will recall, working with tensors is a two step process. First, you define the tensors and arrange them in a computation graph, and then you run the graph with actual data. So working at this level is harder than working in the rest of Keras. The Keras documentation has some guidelines for building custom layers (<https://keras.io/layers/writing-your-own-keras-layers/>), which you should definitely read.

One of the ways to make it easier to develop code in the backend API is to have a small test harness that you can run to verify that your code is doing what you want it to do. Here is a small harness I adapted from the Keras source to run your layer against some input and return a result:

```
from keras.models import Sequential
from keras.layers.core import Dropout, Reshape

def test_layer(layer, x):
    layer_config = layer.get_config()
    layer_config["input_shape"] = x.shape
    layer = layer.__class__.from_config(layer_config)
    model = Sequential()
    model.add(layer)
    model.compile("rmsprop", "mse")
    x_ = np.expand_dims(x, axis=0)
    return model.predict(x_) [0]
```

And here are some tests with `layer` objects provided by Keras to make sure that the harness runs okay:

```
from keras.layers.core import Dropout, Reshape
from keras.layers.convolutional import ZeroPadding2D
import numpy as np

x = np.random.randn(10, 10)
```

```
layer = Dropout(0.5)
y = test_layer(layer, x)
assert(x.shape == y.shape)

x = np.random.randn(10, 10, 3)
layer = ZeroPadding2D(padding=(1,1))
y = test_layer(layer, x)
assert(x.shape[0] + 2 == y.shape[0])
assert(x.shape[1] + 2 == y.shape[1])

x = np.random.randn(10, 10)
layer = Reshape((5, 20))
y = test_layer(layer, x)
assert(y.shape == (5, 20))
```

Before we begin building our local response normalization layer, we need to take a moment to understand what it really does. This technique was originally used with Caffe, and the Caffe documentation (<http://caffe.berkeleyvision.org/tutorial/layers/lrn.html>) describes it as a kind of *lateral inhibition* that works by normalizing over local input regions. In ACROSS_CHANNEL mode, the local regions extend across nearby channels but have no spatial extent. In WITHIN_CHANNEL mode, the local regions extend spatially, but are in separate channels. We will implement the WITHIN_CHANNEL model as follows. The formula for local response normalization in the WITHIN_CHANNEL model is given by:

$$LRN(x_i) = \frac{x_i}{(k + \frac{\alpha}{n} \sum_i x_i)^\beta}$$

The code for the custom layer follows the standard structure. The `__init__` method is used to set the application specific parameters, that is, the hyperparameters associated with the layer. Since our layer only does a forward computation and doesn't have any learnable weights, all we do in the build method is to set the input shape and delegate to the superclass's build method, which takes care of any necessary book-keeping. In layers where learnable weights are involved, this method is where you would set the initial values.

The call method does the actual computation. Notice that we need to account for dimension ordering. Another thing to note is that the batch size is usually unknown at design times, so you need to write your operations so that the batch size is not explicitly invoked. The computation itself is fairly straightforward and follows the formula closely. The sum in the denominator can also be thought of as average pooling over the row and column dimension with a padding size of (n, n) and a stride of $(1, 1)$. Because the pooled data is averaged already, we no longer need to divide the sum by n .

The last part of the class is the `get_output_shape_for` method. Since the layer normalizes each element of the input tensor, the output size is identical to the input size:

```
from keras import backend as K
from keras.engine.topology import Layer, InputSpec

class LocalResponseNormalization(Layer):

    def __init__(self, n=5, alpha=0.0005, beta=0.75, k=2, **kwargs):
        self.n = n
        self.alpha = alpha
        self.beta = beta
        self.k = k
        super(LocalResponseNormalization, self).__init__(**kwargs)

    def build(self, input_shape):
        self.shape = input_shape
        super(LocalResponseNormalization, self).build(input_shape)

    def call(self, x, mask=None):
        if K.image_dim_ordering == "th":
            _, f, r, c = self.shape
        else:
            _, r, c, f = self.shape
        squared = K.square(x)
        pooled = K.pool2d(squared, (n, n), strides=(1, 1),
                          padding="same", pool_mode="avg")
        if K.image_dim_ordering == "th":
            summed = K.sum(pooled, axis=1, keepdims=True)
            averaged = self.alpha * K.repeat_elements(summed, f, axis=1)
        else:
            summed = K.sum(pooled, axis=3, keepdims=True)
            averaged = self.alpha * K.repeat_elements(summed, f, axis=3)
        denom = K.pow(self.k + averaged, self.beta)
        return x / denom

    def get_output_shape_for(self, input_shape):
        return input_shape
```

You can test this layer during development using the test harness we described here. It is easier to run this instead of trying to build a whole network to put this into, or worse, waiting till you have fully specified the layer before running it:

```
x = np.random.randn(225, 225, 3)
layer = LocalResponseNormalization()
y = test_layer(layer, x)
assert(x.shape == y.shape)
```

While building custom Keras layers seems to be fairly commonplace among experienced Keras developers, there are not too many examples available on the Internet. This is probably because custom layers are usually built to serve a specific narrow purpose and may not be widely useful. The variability also means that one single example cannot demonstrate all the possibilities of what you can do with the API. Now that you have a good idea of how to build a custom Keras layer, you might find it instructive to look at Keunwoo Choi's `melspectrogram` (<https://keunwoochoi.wordpress.com/2016/11/18/for-beginners-writing-a-custom-keras-layer/>) and Shashank Gupta's `NodeEmbeddingLayer` (<http://shashankg7.github.io/2016/10/12/Custom-Layer-In-Keras-Graph-Embedding-Case-Study.html>).

Generative models

Generative models are models that learn to create data similar to data it is trained on. We saw one example of a generative model that learns to write prose similar to *Alice in Wonderland* in Chapter 6, *Recurrent Neural Network — RNN*. In that example, we trained a model to predict the 11th character of text given the first 10 characters. Yet another type of generative model is **generative adversarial models (GAN)** that have recently emerged as a very powerful class of models—you saw examples of GANs in Chapter 4, *Generative Adversarial Networks and WaveNet*. The intuition for generative models is that it learns a good internal representation of its training data, and is therefore able to generate similar data during the *prediction* phase.

Another perspective on generative models is the probabilistic one. A typical classification or regression network, also called a discriminative model, learns a function that maps the input data X to some label or output y , that is, these models learn the conditional probability $P(y|X)$. On the other hand, a generative model learns the joint probability and labels simultaneously, that is, $P(x, y)$. This knowledge can then be used to create probable new (X, y) samples. This gives generative models the ability to explain the underlying structure of input data even when there are no labels. This is a very important advantage in the real world, since unlabeled data is more abundant than labeled data.

Simple generative models such as the example mentioned above can be extended to audio as well, for example, models that learn to generate and play music. One interesting one is described in the WaveNet paper (for more information refer to: *WaveNet: A Generative Model for Raw Audio*, by A. van den Oord, 2016.) which describes a network built using atrous convolutional layers and provides a Keras implementation on GitHub (<https://github.com/basveeling/wavenet>).

Keras example — deep dreaming

In this example, we will look at a slightly different generative network. We will see how to take a pre-trained convolutional network and use it to generate new objects in an image. Networks trained to discriminate between images learn enough about the images to generate them as well. This was first demonstrated by Alexander Mordvintsev of Google and described in this Google Research blog post (<https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html>). It was originally called *inceptionism* but the term *deep dreaming* became more popular to describe the technique.

Deep dreaming takes the backpropagated gradient activations and adds it back to the image, running the same process over and over in a loop. The network optimizes the loss function in the process, but we get to see how it does so in the input image (three channels) rather than in a high dimensional hidden layer that cannot easily be visualized.

There are many variations to this basic strategy, each of which leads to new and interesting effects. Some variations are blurring, adding constraints on the total activations, decaying the gradient, infinitely zooming into the image by cropping and scaling, adding jitter by randomly moving the image around, and so on. In our example, we will show the simplest approach—we will optimize the gradient of the mean of the selected layer's activation for each of the pooling layers of a pre-trained VGG-16 and observe the effect on our input image.

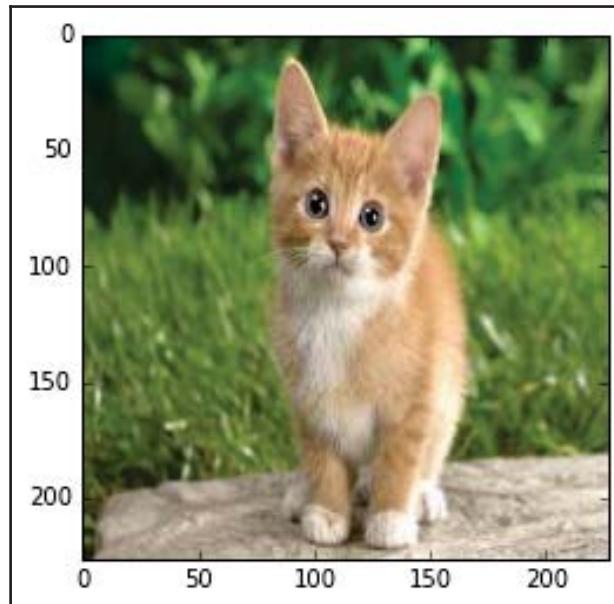
First, as usual, we will declare our imports:

```
from keras import backend as K
from keras.applications import vgg16
from keras.layers import Input
import matplotlib.pyplot as plt
import numpy as np
import os
```

Next we will load up our input image. This image may be familiar to you from blog posts about deep learning. The original image is from here (<https://www.flickr.com/photos/billgarrett-newagecrap/14984990912>):

```
DATA_DIR = "../data"
IMAGE_FILE = os.path.join(DATA_DIR, "cat.jpg")
img = plt.imread(IMAGE_FILE)
plt.imshow(img)
```

The output of the preceding example is as follows:



Next we define a pair of functions to preprocess and deprocess the image to and from a four-dimensional representation suitable for input to a pre-trained VGG-16 network:

```
def preprocess(img):
    img4d = img.copy()
    img4d = img4d.astype("float64")
    if K.image_dim_ordering() == "th":
        # (H, W, C) -> (C, H, W)
        img4d = img4d.transpose((2, 0, 1))
        img4d = np.expand_dims(img4d, axis=0)
        img4d = vgg16.preprocess_input(img4d)
    return img4d

def deprocess(img4d):
    img = img4d.copy()
    if K.image_dim_ordering() == "th":
        # (B, C, H, W)
        img = img.reshape((img4d.shape[1], img4d.shape[2],
img4d.shape[3]))
        # (C, H, W) -> (H, W, C)
        img = img.transpose((1, 2, 0))
    else:
        # (B, H, W, C)
```

```
        img = img.reshape((img4d.shape[1], img4d.shape[2], img4d.shape[3]))
        img[:, :, 0] += 103.939
        img[:, :, 1] += 116.779
        img[:, :, 2] += 123.68
        # BGR -> RGB
        img = img[:, :, ::-1]
        img = np.clip(img, 0, 255).astype("uint8")
    return img
```

These two functions are inverses of each other, that is, passing the image through preprocess and then through deprocess will return the original image.

Next, we load up our pre-trained VGG-16 network. This network has been pre-trained on ImageNet data and is available from the Keras distribution. You already learned how to work with pre-trained models in Chapter 3, *Deep Learning with ConvNets*. We select the version whose fully connected layers have been removed already. Apart from saving us the trouble of having to remove them ourselves, this also allows us to pass in any shape of image, since the reason we need to specify the image width and height in our input is because this determines the size of the weight matrices in the fully connected layers. Because CNN transformations are local in nature, the size of the image doesn't affect the sizes of the weight matrices for the convolutional and pooling layers. So the only constraint on image size is that it must be constant within the batch:

```
img_copy = img.copy()
print("Original image shape:", img.shape)
p_img = preprocess(img_copy)
batch_shape = p_img.shape
dream = Input(batch_shape=batch_shape)
model = vgg16.VGG16(input_tensor=dream, weights="imagenet",
                     include_top=False)
```

We will need to refer to the CNN's layer objects by name in our following calculations, so let us construct a dictionary. We also need to understand the layer naming convention, so we dump it out:

```
layer_dict = {layer.name : layer for layer in model.layers}
print(layer_dict)
```

The output of the preceding example is as follows:

```
{'block1_conv1': <keras.layers.convolutional.Convolution2D at 0x11b847690>,
 'block1_conv2': <keras.layers.convolutional.Convolution2D at 0x11b847f90>,
 'block1_pool': <keras.layers.pooling.MaxPooling2D at 0x11c45db90>,
 'block2_conv1': <keras.layers.convolutional.Convolution2D at 0x11c45ddd0>,
 'block2_conv2': <keras.layers.convolutional.Convolution2D at 0x11b88f810>,
 'block2_pool': <keras.layers.pooling.MaxPooling2D at 0x11c2d2690>,
```

```
'block3_conv1': <keras.layers.convolutional.Convolution2D at 0x11c47b890>,
'block3_conv2': <keras.layers.convolutional.Convolution2D at 0x11c510290>,
'block3_conv3': <keras.layers.convolutional.Convolution2D at 0x11c4afa10>,
'block3_pool': <keras.layers.pooling.MaxPooling2D at 0x11c334a10>,
'block4_conv1': <keras.layers.convolutional.Convolution2D at 0x11c345b10>,
'block4_conv2': <keras.layers.convolutional.Convolution2D at 0x11c345950>,
'block4_conv3': <keras.layers.convolutional.Convolution2D at 0x11d52c910>,
'block4_pool': <keras.layers.pooling.MaxPooling2D at 0x11d550c90>,
'block5_conv1': <keras.layers.convolutional.Convolution2D at 0x11d566c50>,
'block5_conv2': <keras.layers.convolutional.Convolution2D at 0x11d5b1910>,
'block5_conv3': <keras.layers.convolutional.Convolution2D at 0x11d5b1710>,
'block5_pool': <keras.layers.pooling.MaxPooling2D at 0x11fd68e10>,
'input_1': <keras.engine.topology.InputLayer at 0x11b847410>}
```

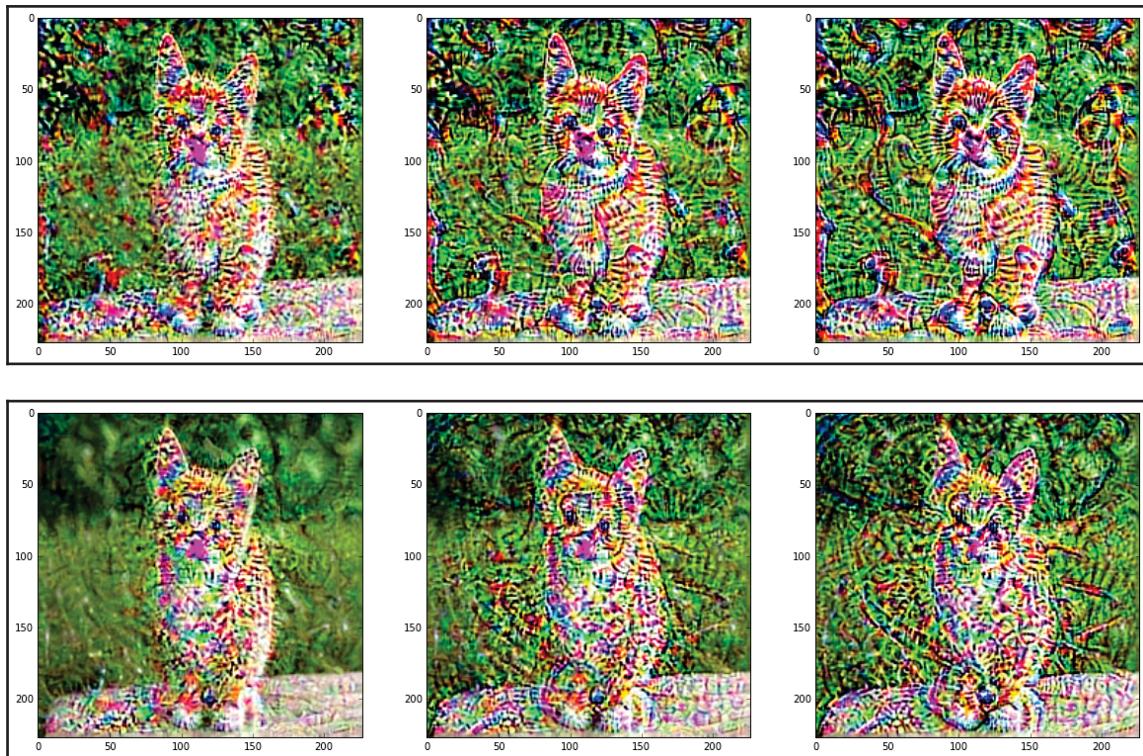
We then compute the loss at each of the five pooling layers and compute the gradient of the mean activation for three steps each. The gradient is added back to the image and the image displayed at each of the pooling layers for each step:

```
num_pool_layers = 5
num_iters_per_layer = 3
step = 100

for i in range(num_pool_layers):
    # identify each pooling layer
    layer_name = "block{:d}_pool".format(i+1)
    # build loss function that maximizes the mean activation in layer
    layer_output = layer_dict[layer_name].output
    loss = K.mean(layer_output)
    # compute gradient of image wrt loss and normalize
    grads = K.gradients(loss, dream)[0]
    grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)
    # define function to return loss and grad given input image
    f = K.function([dream], [loss, grads])
    img_value = p_img.copy()
    fig, axes = plt.subplots(1, num_iters_per_layer, figsize=(20, 10))
    for it in range(num_iters_per_layer):
        loss_value, grads_value = f([img_value])
        img_value += grads_value * step
        axes[it].imshow(deprocess(img_value))
    plt.show()
```

The resulting images are shown as follows:



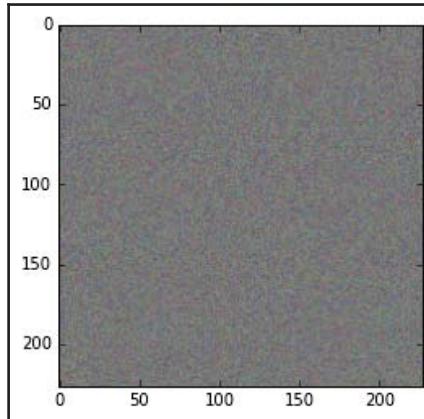


As you can see, the process of deep dreaming amplifies the effect of the gradient on the chosen layer, resulting in images that are quite surreal. Later layers backpropagate gradients that result in more distortion, reflecting their larger receptive fields and their capacity to recognize more complex features.

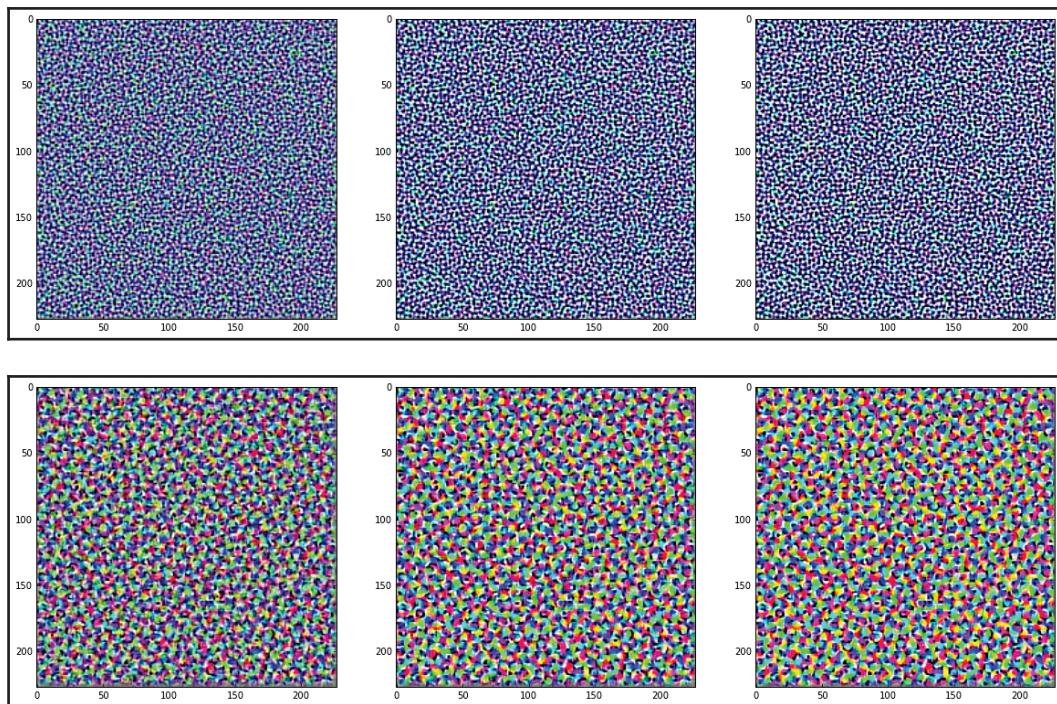
To convince ourselves that a trained network really learns a representation of the various categories of the image it was trained on, let us consider a completely random image, shown next, and pass it through the pre-trained network:

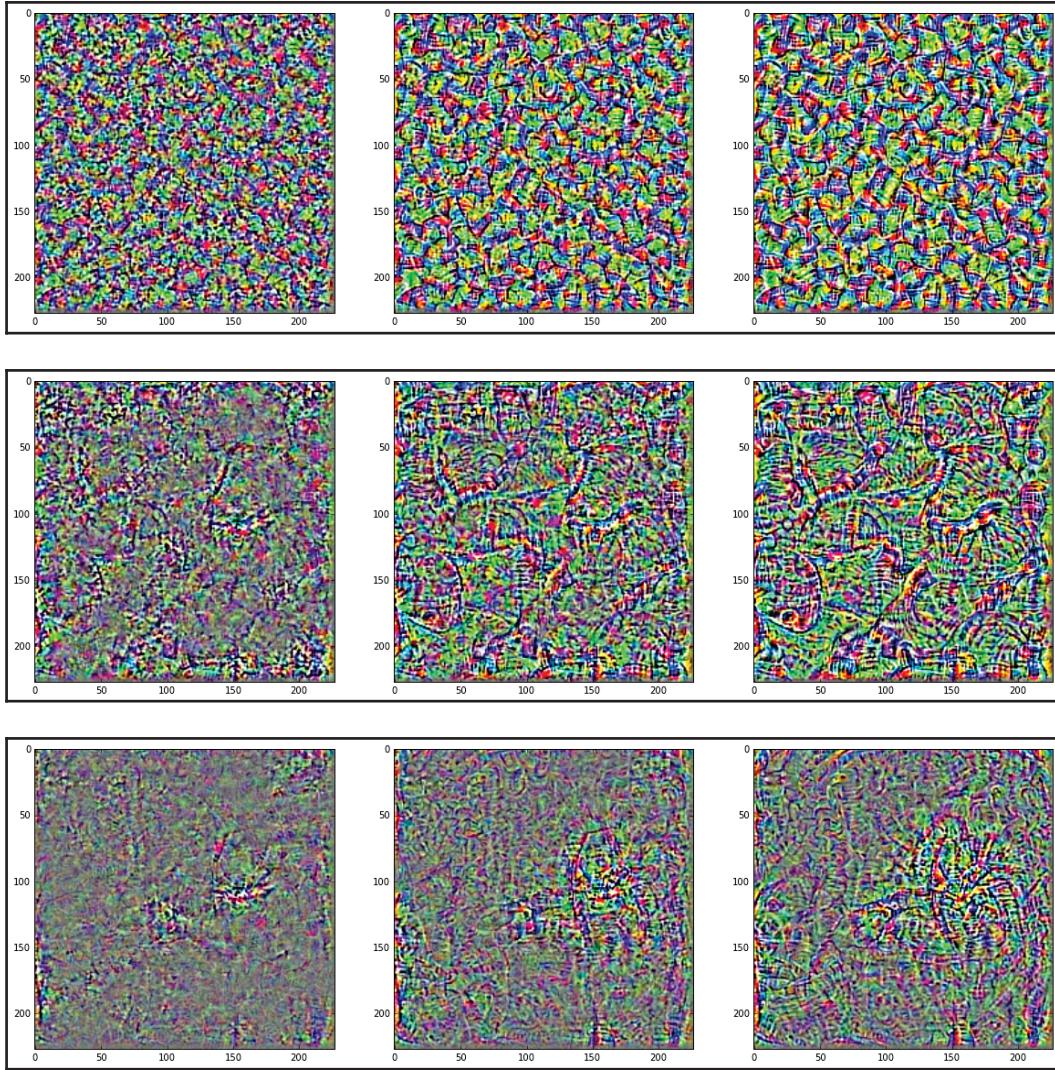
```
img_noise = np.random.randint(100, 150, size=(227, 227, 3), dtype=np.uint8)
plt.imshow(img_noise)
```

The output of the preceding example is as follows:



Passing this image through the preceding code results in very specific patterns at each layer, as shown next, showing that the network is trying to find a structure in the random data:

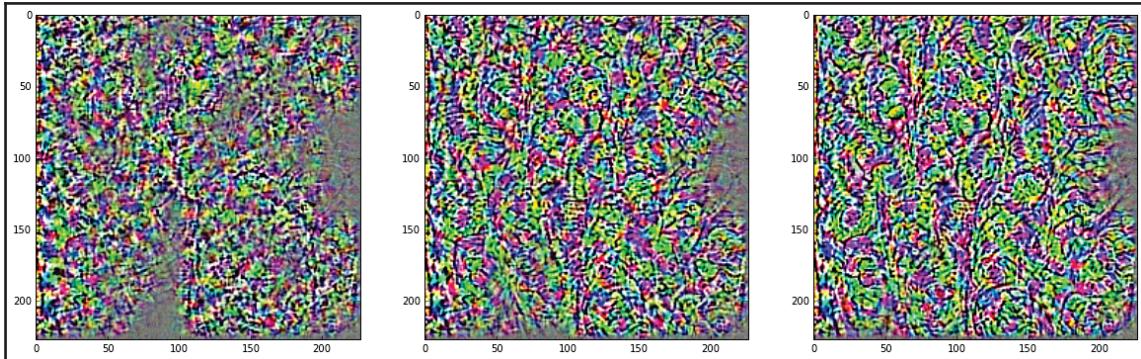




We can repeat our experiment with the noise image as input and compute the loss from a single filter instead of taking the mean across all the filters. The filter we choose is for the ImageNet label African elephant (24). Thus, we replace the value of the loss in the previous code with the following. So instead of computing the mean across all filters, we calculate the loss as the output of the filter representing the African elephant class:

```
loss = layer_output[:, :, :, 24]
```

We get back what looks very much like repeating images of the trunk of an elephant in the `block4_pool` output, as shown here:



Keras example — style transfer

An extension of deep dreaming was described in this paper (for more information refer to: *Image Style Transfer Using Convolutional Neural Networks*, by L. A. Gatys, A. S. Ecker, and M. Bethge, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016), which showed that trained neural networks, such as the VGG-16, learn both content and style, and these two can be manipulated independently. Thus an image of an object (content) could be styled to look like a painting by combining it with the image of a painting (style).

Let us start, as usual, by importing our libraries:

```
from keras.applications import vgg16
from keras import backend as K
from scipy.misc import imresize
import matplotlib.pyplot as plt
import numpy as np
import os
```

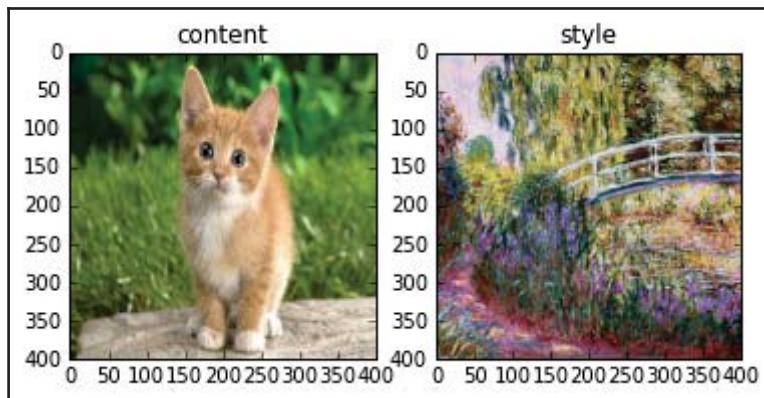
Our example will demonstrate styling our image of a cat with this image of a reproduction of Claude Monet's *The Japanese Bridge* by Rosalind Wheeler (<https://goo.gl/0VXC39>):

```
DATA_DIR = "../data"
CONTENT_IMAGE_FILE = os.path.join(DATA_DIR, "cat.jpg")
STYLE_IMAGE_FILE = os.path.join(DATA_DIR, "JapaneseBridgeMonetCopy.jpg")
RESIZED_WH = 400

content_img_value = imresize(plt.imread(CONTENT_IMAGE_FILE), (RESIZED_WH,
RESIZED_WH))
style_img_value = imresize(plt.imread(STYLE_IMAGE_FILE), (RESIZED_WH,
```

```
RESIZED_WH) )  
  
plt.subplot(121)  
plt.title("content")  
plt.imshow(content_img_value)  
  
plt.subplot(122)  
plt.title("style")  
plt.imshow(style_img_value)  
  
plt.show()
```

The output of the preceding example is as follows:



As previously, we declare our two functions to convert back and forth from the image and the four-dimensional tensor that the CNN expects:

```
def preprocess(img):  
    img4d = img.copy()  
    img4d = img4d.astype("float64")  
    if K.image_dim_ordering() == "th":  
        # (H, W, C) -> (C, H, W)  
        img4d = img4d.transpose((2, 0, 1))  
    img4d = np.expand_dims(img4d, axis=0)  
    img4d = vgg16.preprocess_input(img4d)  
    return img4d  
  
def deprocess(img4d):  
    img = img4d.copy()  
    if K.image_dim_ordering() == "th":  
        # (B, C, H, W)  
        img = img.reshape((img4d.shape[1], img4d.shape[2], img4d.shape[3]))  
        # (C, H, W) -> (H, W, C)
```

```
        img = img.transpose((1, 2, 0))
    else:
        # (B, H, W, C)
        img = img.reshape((img4d.shape[1], img4d.shape[2], img4d.shape[3]))
    img[:, :, 0] += 103.939
    img[:, :, 1] += 116.779
    img[:, :, 2] += 123.68
    # BGR -> RGB
    img = img[:, :, ::-1]
    img = np.clip(img, 0, 255).astype("uint8")
    return img
```

We declare tensors to hold the content image and the style image, and another tensor to hold the combined image. The content and style images are then concatenated into a single input tensor. The input tensor will be fed to the pre-trained VGG-16 network:

```
content_img = K.variable(preprocess(content_img_value))
style_img = K.variable(preprocess(style_img_value))
if K.image_dim_ordering() == "th":
    comb_img = K.placeholder((1, 3, RESIZED_WH, RESIZED_WH))
else:
    comb_img = K.placeholder((1, RESIZED_WH, RESIZED_WH, 3))

# concatenate images into single input
input_tensor = K.concatenate([content_img, style_img, comb_img], axis=0)
```

We instantiate an instance of a pre-trained VGG-16 network, pre-trained with the ImageNet data, and with the fully connected layers excluded:

```
model = vgg16.VGG16(input_tensor=input_tensor, weights="imagenet",
include_top=False)
```

As previously, we construct a layer dictionary to map the layer name to the output layer of the trained VGG-16 network:

```
layer_dict = {layer.name : layer.output for layer in model.layers}
```

The next block defines the code for computing the `content_loss`, the `style_loss`, and the `variational_loss`. Finally, we define our loss as a linear combination of these three losses:

```
def content_loss(content, comb):
    return K.sum(K.square(comb - content))

def gram_matrix(x):
    if K.image_dim_ordering() == "th":
        features = K.batch_flatten(x)
    else:
```

```
        features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
        gram = K.dot(features, K.transpose(features))
        return gram

def style_loss_per_layer(style, comb):
    S = gram_matrix(style)
    C = gram_matrix(comb)
    channels = 3
    size = RESIZED_WH * RESIZED_WH
    return K.sum(K.square(S - C)) / (4 * (channels ** 2) * (size ** 2))

def style_loss():
    stl_loss = 0.0
    for i in range(NUM_LAYERS):
        layer_name = "block{:d}_conv1".format(i+1)
        layer_features = layer_dict[layer_name]
        style_features = layer_features[1, :, :, :]
        comb_features = layer_features[2, :, :, :]
        stl_loss += style_loss_per_layer(style_features, comb_features)
    return stl_loss / NUM_LAYERS

def variation_loss(comb):
    if K.image_dim_ordering() == "th":
        dx = K.square(comb[:, :, :RESIZED_WH-1, :RESIZED_WH-1] -
                      comb[:, :, 1:, :RESIZED_WH-1])
        dy = K.square(comb[:, :, :RESIZED_WH-1, :RESIZED_WH-1] -
                      comb[:, :, :, :RESIZED_WH-1, 1:])
    else:
        dx = K.square(comb[:, :, :RESIZED_WH-1, :RESIZED_WH-1, :] -
                      comb[:, :, 1:, :RESIZED_WH-1, :])
        dy = K.square(comb[:, :, :RESIZED_WH-1, :RESIZED_WH-1, :] -
                      comb[:, :, :, :RESIZED_WH-1, 1:])
    return K.sum(K.pow(dx + dy, 1.25))

CONTENT_WEIGHT = 0.1
STYLE_WEIGHT = 5.0
VAR_WEIGHT = 0.01
NUM_LAYERS = 5

c_loss = content_loss(content_img, comb_img)
s_loss = style_loss()
v_loss = variation_loss(comb_img)
loss = (CONTENT_WEIGHT * c_loss) + (STYLE_WEIGHT * s_loss) + (VAR_WEIGHT *
v_loss)
```

Here the content loss is the root mean square distance (also known as **L2 distance**) between the features of the content image extracted from the target layer and the combination image. Minimizing this has the effect of keeping the styled image close to the original one.

The style loss is the L2 distance between the gram matrices of the base image representation and the style image. A gram matrix of a matrix M is the transpose of M multiplied by M , that is, $MT * M$. This loss measures how often features appear together in the content image representation and the style image. One practical implication of this is that the content and style matrices must be square.

The total variation loss measures the difference between neighboring pixels. Minimizing this has the effect that neighboring pixels will be similar so the final image is smooth rather than *jumpy*.

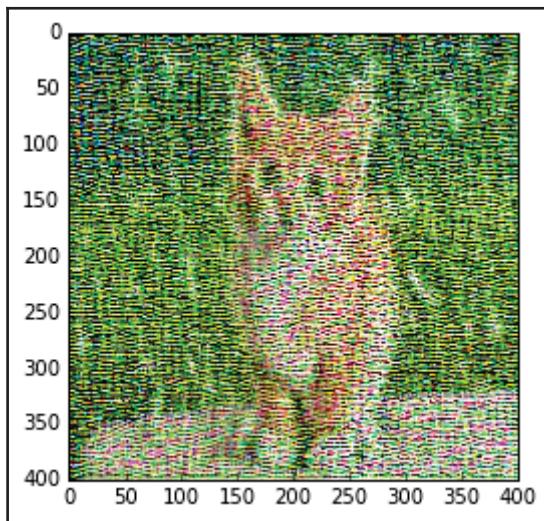
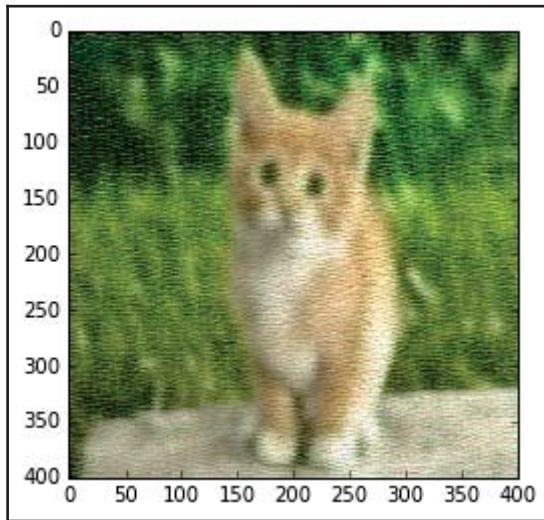
We calculate the gradient and the loss function, and run our network in reverse for five iterations:

```
grads = K.gradients(loss, comb_img)[0]
f = K.function([comb_img], [loss, grads])

NUM_ITERATIONS = 5
LEARNING_RATE = 0.001

content_img4d = preprocess(content_img_value)
for i in range(NUM_ITERATIONS):
    print("Epoch {:d}/{:d}".format(i+1, NUM_ITERATIONS))
    loss_value, grads_value = f([content_img4d])
    content_img4d += grads_value * LEARNING_RATE
    plt.imshow(deprocess(content_img4d))
    plt.show()
```

The output from the last two iterations is shown as follows. As you can see, it has picked up the impressionistic fuzziness and even the texture of the canvas in the final images:



Summary

In this chapter, we covered some deep learning networks that were not covered in earlier chapters. We started with a brief look into the Keras functional API, which allows us to build networks that are more complex than the sequential networks we have seen so far. We then looked at regression networks, which allow us to do predictions in a continuous space, and opens up a whole new range of problems we can solve. However, a regression network is really a very simple modification of a standard classification network. The next area we looked at was autoencoders, which are a style of network that allows us to do unsupervised learning and make use of the massive amount of unlabeled data that all of us have access to nowadays. We also learned how to compose the networks we had already learned about as giant Lego-like building blocks into larger and more interesting networks. We then moved from building large networks using smaller networks, to learning how to customize individual layers in a network using the Keras backend layer. Finally, we looked at generative models, another class of models that learn to mimic the input it is trained on, and looked at some novel uses for this kind of model.

In the next chapter, we will turn our attention to another learning style called reinforcement learning, and explore its concepts by building and training a network in Keras to play a simple computer game.

8

AI Game Playing

In previous chapters, we looked at supervised learning techniques such as regression and classification, and unsupervised learning techniques such as GANs, autoencoders and generative models. In the case of supervised learning, we train the network with the expected input and output and expect it to predict the output given a new input. In the case of unsupervised learning, we show the network some input and expect it to learn the structure of the data so that it can apply this knowledge to a new input.

In this chapter, we will learn about reinforcement learning, or more specifically deep reinforcement learning, that is, the application of deep neural networks to reinforcement learning. Reinforcement learning has its roots in behavioral psychology. An agent is trained by rewarding it for correct behavior and punishing it for incorrect behavior. In the context of deep reinforcement learning, a network is shown some input and is given a positive or negative reward based on whether it produces the correct output from that input. Thus, in reinforcement learning, we have sparse and time-delayed labels. Over many iterations, the network learns to produce the correct output.

The pioneer in the deep reinforcement learning space was a small British company called DeepMind, which in 2013 published a paper (for more information refer to: *Playing Atari with Deep Reinforcement Learning*, by V. Mnih, arXiv:1312.5602, 2013.) describing how a **convolutional neural network (CNN)** could be taught to play Atari 2600 video games by showing it screen pixels and giving it a reward when the score increases. The same architecture was used to learn seven different Atari 2600 games, in six of which the model outperformed all previous approaches, and it outperformed a human expert in three.

Unlike the learning strategies we learned about previously, where each network learns about a single discipline, reinforcement learning seems to be a general learning algorithm that can be applied to a variety of environments; it may even be the first step to general artificial intelligence. DeepMind has since been acquired by Google, and the group has been on the forefront of AI research. A subsequent paper (for more information refer to: *Human-Level Control through Deep Reinforcement Learning*, by V. Mnih, Nature 518.7540, 2015: 529-533.) was featured in the prestigious Nature journal in 2015, where they applied the same model to 49 different games.

In this chapter, we will explore the theoretical framework that underlies deep reinforcement learning. We'll then apply this framework to build a network using Keras that learns to play a game of catch. We'll briefly look at some ideas that can make this network better as well as some promising new areas of research in this space.

To sum up, we will learn the following core concepts around reinforcement learning in this chapter:

- Q-learning
- Exploration versus exploitation
- Experience replay

Reinforcement learning

Our objective is to build a neural network to play the game of catch. Each game starts with a ball being dropped from a random position from the top of the screen. The objective is to move a paddle at the bottom of the screen using the left and right arrow keys to catch the ball by the time it reaches the bottom. As games go, this is quite simple. At any point in time, the state of this game is given by the (x, y) coordinates of the ball and paddle. Most arcade games tend to have many more moving parts, so a general solution is to provide the entire current game screen image as the state. The following screenshot shows four consecutive screenshots of our catch game:



Astute readers might note that our problem could be modeled as a classification problem, where the input to the network are the game screen images and the output is one of three actions--move left, stay, or move right. However, this would require us to provide the network with training examples, possibly from recordings of games played by experts. An alternative and simpler approach might be to build a network and have it play the game repeatedly, giving it feedback based on whether it succeeds in catching the ball or not. This approach is also more intuitive and is closer to the way humans and animals learn.

The most common way to represent such a problem is through a **markov decision process (MDP)**. Our game is the environment within which the agent is trying to learn. The state of the environment at time step t is given by s_t (and contains the location of the ball and paddle). The agent can perform certain actions (such as moving the paddle left or right). These actions can sometimes result in a reward r_t , which can be positive or negative (such as an increase or decrease in the score). Actions change the environment and can lead to a new state s_{t+1} , where the agent can perform another action a_{t+1} , and so on. The set of states, actions and rewards, together with the rules for transitioning from one state to the other, make up a markov decision process. A single game is one episode of this process, and is represented by a finite sequence of states, actions, and rewards:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \dots, s_{n-1}, a_{n-1}, r_n, s_n$$

Since, this is a markov decision process, the probability of state s_{t+1} depends only on current state s_t and action a_t .

Maximizing future rewards

As an agent, our objective is to maximize the total reward from each game. The total reward can be represented as follows:

$$R = \sum_{i=1}^n r_i$$

In order to maximize the total reward, the agent should try to maximize the total reward from any time point t in the game. The total reward at time step t is given by R_t and is represented as:

$$R_t = \sum_{i=t}^n r_i = r_t + r_{t+1} + \dots + r_n$$

However, it is harder to predict the value of the rewards the further we go into the future. In order to take this into consideration, our agent should try to maximize the total discounted future reward at time t instead. This is done by discounting the reward at each future time step by a factor γ over the previous time step. If γ is 0, then our network does not consider future rewards at all, and if γ is 1, then our network is completely deterministic. A good value for γ is around 0.9. Factoring the equation allows us to express the total discounted future reward at a given time step recursively as the sum of the current reward and the total discounted future reward at the next time step:

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^{n-t} r_n \\ &= r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \dots)) \\ &= r_t + \gamma R_{t+1} \end{aligned}$$

Q-learning

Deep reinforcement learning utilizes a model-free reinforcement learning technique called **Q-learning**. Q-learning can be used to find an optimal action for any given state in a finite markov decision process. Q-learning tries to maximize the value of the Q-function which represents the maximum discounted future reward when we perform action a in state s :

$$Q(s_t, a_t) = \max(R_{t+1})$$

Once we know the Q-function, the optimal action a at a state s is the one with the highest Q-value. We can then define a policy $\tilde{\epsilon}(s)$ that gives us the optimal action at any state:

$$\Pi(s) = \operatorname{argmax}_a Q(s, a)$$

We can define the Q-function for a transition point (s_t, a_t, r_t, s_{t+1}) in terms of the Q-function at the next point $(s_{t+1}, a_{t+1}, r_{t+1}, s_{t+2})$ similar to how we did with the total discounted future reward. This equation is known as the **Bellman equation**:

$$Q(s_t, a_t) = r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$

The Q-function can be approximated using the Bellman equation. You can think of the Q-function as a lookup table (called a **Q-table**) where the states (denoted by s) are rows and actions (denoted by a) are columns, and the elements (denoted by $Q(s, a)$) are the rewards that you get if you are in the state given by the row and take the action given by the column. The best action to take at any state is the one with the highest reward. We start by randomly initializing the Q-table, then carry out random actions and observe the rewards to update the Q-table iteratively according to the following algorithm:

```
initialize Q-table Q
observe initial state s
repeat
    select and carry out action a
    observe reward r and move to new state s'
    Q(s, a) = Q(s, a) + α(r + γ max_a' Q(s', a') - Q(s, a))
    s = s'
until game over
```

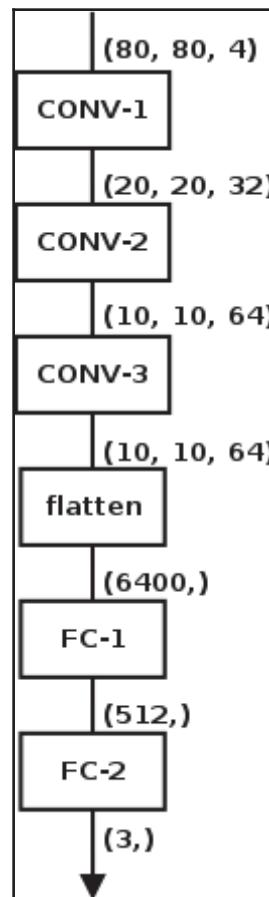
You will realize that the algorithm is basically doing stochastic gradient descent on the Bellman equation, backpropagating the reward through the state space (or episode) and averaging over many trials (or epochs). Here α is the learning rate that determines how much of the difference between the previous Q-value and the discounted new maximum Q-value should be incorporated.

The deep Q-network as a Q-function

We know that our Q-function is going to be a neural network, the natural question is: what kind? For our simple example game, each state is represented by four consecutive black and white screen images of size $(80, 80)$, so the total number of possible states (and the number of rows of our Q-table) is $2^{80 \times 80 \times 4}$. Fortunately, many of these states represent impossible or highly improbable pixel combinations. Since convolutional neural networks have local connectivity (that is, each neuron is connected to only a local region of its input), it avoids these impossible or improbable pixel combinations. In addition, neural networks are generally very good at coming up with good features for structured data such as images. Hence a CNN can be used to model a Q-function very effectively.

The DeepMind paper (for more information refer to: *Playing Atari with Deep Reinforcement Learning*, by V. Mnih, arXiv:1312.5602, 2013.), also uses three layers of convolutions followed by two fully connected layers. Unlike traditional CNNs used for image classification or recognition, there are no pooling layers. This is because pooling layers makes the network less sensitive to the location of specific objects in the image. In case of games this information is likely to be required to compute the reward, and thus cannot be discarded.

The following diagram, shows the structure of the deep Q-network that is used for our example. It follows the same structure as the original DeepMind paper except for the input and output layer shapes. The shape for each of our inputs is $(80, 80, 4)$: four black and white consecutive screenshots of the game console, each 80×80 pixels in size. Our output shape is (3) , corresponding to the Q-value for each of three possible actions (move left, stay, move right):



Since our output are the three Q-values, this is a regression task, and we can optimize this by minimizing the difference of the squared error between the current value of $Q(s, a)$ and its computed value in terms of the sum of the reward and the discounted Q-value $Q(s', a')$ one step into the future. The current value is already known at the beginning of the iteration and the future value is computed based on the reward returned by the environment:

$$L = \frac{1}{2} [r + \gamma \max Q(s', a') - Q(s, a)]^2$$

Balancing exploration with exploitation

Deep reinforcement learning is an example of online learning, where the training and prediction steps are interspersed. Unlike batch learning techniques where the best predictor is generated by learning on the entire training data, a predictor trained with online learning is continuously improving as it trains on new data.

Thus in the initial epochs of training, a deep Q-network gives random predictions which can give rise to poor Q-learning performance. To alleviate this, we can use a simple exploration method such as ϵ -greedy. In case of ϵ -greedy exploration, the agent chooses the action suggested by the network with probability $1-\epsilon$; or an action uniformly at random otherwise. That is why this strategy is called exploration/exploitation.

As the number of epochs increases and the Q-function converges, it begins to return more consistent Q-values. The value of ϵ can be attenuated to account for this, so as the network begins to return more consistent predictions, the agent chooses to exploit the values returned by the network over choosing random actions. In case of DeepMind, the value of ϵ decreases over time from 1 to 0.1, and in our example it decreases from 0.1 to 0.001.

Thus, ϵ -greedy exploration ensures that in the beginning the system balances the unreliable predictions made from the Q-network with completely random moves to explore the state space, and then settles down to less aggressive exploration (and more aggressive exploitation) as the predictions made by the Q-network improve.

Experience replay, or the value of experience

Based on the equations that represent the Q-value for a state action pair (s_t, a_t) in terms of the current reward r_t and the discounted maximum Q-value for the next time step (s_{t+1}, a_{t+1}) , our strategy would logically be to train the network to predict the best next state s' given the current state (s, a, r) . It turns out that this tends to drive the network into a local minimum. The reason for this is that consecutive training samples tend to be very similar.

To counter this, during game play, we collect all the previous moves (s, a, r, s') into a large fixed size queue called the **replay memory**. The replay memory represents the experience of the network. When training the network, we generate random batches from the replay memory instead of the most recent (batch of) transactions. Since the batches are composed of random experience tuples (s, a, r, s') that are out of order, the network trains better and avoids getting stuck in local minima.

Experiences could be collected from human gameplay as well instead of (or in addition to) from previous moves during game play by the network. Yet another approach is to collect experiences by running the network in *observation* mode for a while in the beginning, when it generates completely random actions ($\epsilon = 1$) and extracts the reward and next state from the game and collects them into its experience replay queue.

Example - Keras deep Q-network for catch

The objective of our game is to catch a ball released from a random location from the top of the screen with a paddle at the bottom of the screen by moving the paddle horizontally using the left and right arrow keys. The player wins if the paddle can catch the ball and loses if the ball falls off the screen before the paddle gets to it. The game has the advantage of being very simple to understand and build, and is modeled after the game of catch described by Eder Santana in his blog post (for more information refer to: *Keras Plays Catch, a Single File Reinforcement Learning Example*, by Eder Santana, 2017.) on deep reinforcement learning. We built the original game using Pygame (<https://www.pygame.org/news>), a free and open source library for building games. This game allows the player to move the paddle using the left and right arrow keys. The game is available as `game.py` in the code bundle for this chapter in case you want to get a feel for it.

Installing Pygame:

Pygame runs on top of Python, and is available for Linux (various flavors), macOS, Windows, as well as some phone operating systems such as Android and Nokia. The full list of distributions can be found at: <http://www.pygame.org/download.shtml>. Pre-built versions are available for 32-bit and 64-bit versions of Linux and Windows and 64-bit version of macOS. On these platforms, you can install Pygame with `pip install pygame` command.



If a pre-built version does not exist for your platform, you can also build it from source using instructions available at: <http://www.pygame.org/wiki/GettingStarted>.

Anaconda users can find pre-built Pygame versions on the conda-forge:

```
conda install binstar
conda install anaconda-client
conda install -c https://conda.binstar.org/tlatorre
pygame # Linux
conda install -c https://conda.binstar.org/quasiben
pygame # Mac
```

In order to train our neural network, we need to make some changes to the original game so the network can play instead of the human player. We want to wrap the game to allow the network to communicate with it via an API instead of the keyboard left and right arrow keys. Let us look at the code for this wrapped game.

As usual, we start with the imports:

```
from __future__ import division, print_function
import collections
import numpy as np
import pygame
import random
import os
```

We define our class. Our constructor can optionally set the wrapped version of the game to run in *headless* mode, that is, without needing to display a Pygame screen. This is useful where you have to run on a GPU box in the cloud and only have access to a text based terminal. You can comment this line out if you are running the wrapped game locally where you have access to a graphics terminal. Next we call the `pygame.init()` method to initialize all Pygame components. Finally, we set a bunch of class level constants:

```
class MyWrappedGame(object):

    def __init__(self):
        # run pygame in headless mode
```

```
os.environ["SDL_VIDEODRIVER"] = "dummy"

pygame.init()

# set constants
self.COLOR_WHITE = (255, 255, 255)
self.COLOR_BLACK = (0, 0, 0)
self.GAME_WIDTH = 400
self.GAME_HEIGHT = 400
self.BALL_WIDTH = 20
self.BALL_HEIGHT = 20
self.PADDLE_WIDTH = 50
self.PADDLE_HEIGHT = 10
self.GAME_FLOOR = 350
self.GAME_CEILING = 10
self.BALL_VELOCITY = 10
self.PADDLE_VELOCITY = 20
self.FONT_SIZE = 30
self.MAX_TRIES_PER_GAME = 1
self.CUSTOM_EVENT = pygame.USEREVENT + 1
self.font = pygame.font.SysFont("Comic Sans MS", self.FONT_SIZE)
```

The `reset()` method defines the operations that need to be called at the start of each game, such as clearing out the state queue, setting the ball, and paddle to their starting positions, initializing the scores, and so on:

```
def reset(self):
    self.frames = collections.deque(maxlen=4)
    self.game_over = False
    # initialize positions
    self.paddle_x = self.GAME_WIDTH // 2
    self.game_score = 0
    self.reward = 0
    self.ball_x = random.randint(0, self.GAME_WIDTH)
    self.ball_y = self.GAME_CEILING
    self.num_tries = 0

    # set up display, clock, etc
    self.screen = pygame.display.set_mode((self.GAME_WIDTH,
    self.GAME_HEIGHT))
    self.clock = pygame.time.Clock()
```

In the original game, there is a Pygame event queue into which the left and right arrow key events raised by the player as he moves the paddle, as well as internal events raised by Pygame components are written to. The central part of the game code is basically a loop (called the **event loop**), that reads the event queue and reacts to it.

In the wrapped version, we have moved the event loop to the caller. The `step()` method describes what happens in a single pass in the loop. The method takes an integer 0, 1, or 2 representing an action (respectively move left, stay, and move right), and then it sets variables that control the position of the ball and paddle at this time step. The `PADDLE_VELOCITY` variable represents a *speed* that moves the paddle that many pixels to the left or right when the move left and move right actions are sent. If the ball has dropped past the paddle, it checks whether there is a collision. If there is, the paddle *catches* the ball and the player (the neural network) wins, otherwise the player loses. The method then redraws the screen and appends it to the fixed length `deque` that contains the last four frames of the game screen. Finally, it returns the state (given by the last four frames), the reward for the current action and a flag that tells the caller if the game is over:

```
def step(self, action):
    pygame.event.pump()

    if action == 0: # move paddle left
        self.paddle_x -= self.PADDLE_VELOCITY
        if self.paddle_x < 0:
            # bounce off the wall, go right
            self.paddle_x = self.PADDLE_VELOCITY
    elif action == 2: # move paddle right
        self.paddle_x += self.PADDLE_VELOCITY
        if self.paddle_x > self.GAME_WIDTH - self.PADDLE_WIDTH:
            # bounce off the wall, go left
            self.paddle_x = self.GAME_WIDTH - self.PADDLE_WIDTH -
self.PADDLE_VELOCITY
    else: # don't move paddle
        pass

    self.screen.fill(self.COLOR_BLACK)
    score_text = self.font.render("Score: {:d}/{:d}, Ball: {:d}"
        .format(self.game_score, self.MAX_TRIES_PER_GAME,
                self.num_tries), True, self.COLOR_WHITE)
    self.screen.blit(score_text,
        ((self.GAME_WIDTH - score_text.get_width()) // 2,
        (self.GAME_FLOOR + self.FONT_SIZE // 2)))

    # update ball position
    self.ball_y += self.BALL_VELOCITY
    ball = pygame.draw.rect(self.screen, self.COLOR_WHITE,
        pygame.Rect(self.ball_x, self.ball_y, self.BALL_WIDTH,
                    self.BALL_HEIGHT))
    # update paddle position
    paddle = pygame.draw.rect(self.screen, self.COLOR_WHITE,
        pygame.Rect(self.paddle_x, self.GAME_FLOOR,
                    self.PADDLE_WIDTH, self.PADDLE_HEIGHT))
```

```
# check for collision and update reward
self.reward = 0
if self.ball_y >= self.GAME_FLOOR - self.BALL_WIDTH // 2:
    if ball.colliderect(paddle):
        self.reward = 1
    else:
        self.reward = -1

self.game_score += self.reward
self.ball_x = random.randint(0, self.GAME_WIDTH)
self.ball_y = self.GAME_CEILING
self.num_tries += 1

pygame.display.flip()

# save last 4 frames
self.frames.append(pygame.surfarray.array2d(self.screen))

if self.num_tries >= self.MAX_TRIES_PER_GAME:
    self.game_over = True

self.clock.tick(30)
return np.array(list(self.frames)), self.reward, self.game_over
```

We will look at the code to train our network to play the game.

As usual, first we import the libraries and objects that we need. In addition to third-party components from Keras and SciPy, we also import the `wrapped_game` class we described previously:

```
from __future__ import division, print_function
from keras.models import Sequential
from keras.layers.core import Activation, Dense, Flatten
from keras.layers.convolutional import Conv2D
from keras.optimizers import Adam
from scipy.misc import imresize
import collections
import numpy as np
import os

import wrapped_game
```

We define two convenience functions. The first converts the set of four input images to a form suitable for use by the network. The input comes in a set of four 800×800 images, so the shape of the input is $(4, 800, 800)$. However, the network expects its input as a four-dimensional tensor of shape $(batch\ size, 80, 80, 4)$. At the very beginning of the game, we don't have four frames, so we fake it by stacking the first frame four times. The shape of the output tensor returned from this function is $(80, 80, 4)$.

The `get_next_batch()` function samples `batch_size` state tuples from the experience replay queue, and gets the reward and predicted next state from the neural network. It then calculates the value of the Q-function at the next time step and returns it:

```
def preprocess_images(images):
    if images.shape[0] < 4:
        # single image
        x_t = images[0]
        x_t = imresize(x_t, (80, 80))
        x_t = x_t.astype("float")
        x_t /= 255.0
        s_t = np.stack((x_t, x_t, x_t, x_t), axis=2)
    else:
        # 4 images
        xt_list = []
        for i in range(images.shape[0]):
            x_t = imresize(images[i], (80, 80))
            x_t = x_t.astype("float")
            x_t /= 255.0
            xt_list.append(x_t)
        s_t = np.stack((xt_list[0], xt_list[1], xt_list[2], xt_list[3]),
                       axis=2)
    s_t = np.expand_dims(s_t, axis=0)
    return s_t

def get_next_batch(experience, model, num_actions, gamma, batch_size):
    batch_indices = np.random.randint(low=0, high=len(experience),
                                       size=batch_size)
    batch = [experience[i] for i in batch_indices]
    X = np.zeros((batch_size, 80, 80, 4))
    Y = np.zeros((batch_size, num_actions))
    for i in range(len(batch)):
        s_t, a_t, r_t, s_tp1, game_over = batch[i]
        X[i] = s_t
        Y[i] = model.predict(s_t)[0]
        Q_sa = np.max(model.predict(s_tp1)[0])
        if game_over:
            Y[i, a_t] = r_t
        else:
```

```
    Y[i, a_t] = r_t + gamma * Q_sa  
return X, Y
```

We define our network. This is the network that models the Q-function for our game. Our network is very similar to the one proposed in the DeepMind paper. The only difference is the size of the input and the output. Our input shape is $(80, 80, 4)$ while theirs was $(84, 84, 4)$ and our output is (3) corresponding to the three actions for which the value of the Q-function needs to be computed, whereas their was (18) , corresponding to the actions possible from Atari.

There are three convolutional layers and two fully connected (dense) layers. All layers, except the last have the ReLU activation unit. Since we are predicting values of Q-functions, it is a regression network and the last layer has no activation unit:

```
# build the model  
model = Sequential()  
model.add(Conv2D(32, kernel_size=8, strides=4,  
                kernel_initializer="normal",  
                padding="same",  
                input_shape=(80, 80, 4)))  
model.add(Activation("relu"))  
model.add(Conv2D(64, kernel_size=4, strides=2,  
                kernel_initializer="normal",  
                padding="same"))  
model.add(Activation("relu"))  
model.add(Conv2D(64, kernel_size=3, strides=1,  
                kernel_initializer="normal",  
                padding="same"))  
model.add(Activation("relu"))  
model.add(Flatten())  
model.add(Dense(512, kernel_initializer="normal"))  
model.add(Activation("relu"))  
model.add(Dense(3, kernel_initializer="normal"))
```

As we have described previously, our loss function is the squared difference between the current value of $Q(s, a)$ and its computed value in terms of the sum of the reward and the discounted Q-value $Q(s', a')$ one step into the future, so the mean squared error (MSE) loss function works very well. For the optimizer, we choose Adam, a good general-purpose optimizer, instantiated with a low learning rate:

```
model.compile(optimizer=Adam(lr=1e-6), loss="mse")
```

We define some constants for our training. The `NUM_ACTIONS` constant defines the number of output actions that the network can send to the game. In our case, these actions are 0, 1, and 2, corresponding to move left, stay, and move right. The `GAMMA` value is the discount factor γ for future rewards. The `INITIAL_EPSILON` and `FINAL_EPSILON` refer to starting and ending values for the ϵ parameter in ϵ -greedy exploration. The `MEMORY_SIZE` is the size of the experience replay queue. The `NUM_EPOCHS_OBSERVE` refer to the number of epochs where the network is allowed to explore the game by sending it completely random actions and seeing the rewards. The `NUM_EPOCHS_TRAIN` variable refers to the number of epochs the network will undergo online training. Each epoch corresponds to a single game or episode. The total number of games played for a training run is the sum of the `NUM_EPOCHS_OBSERVE` and `NUM_EPOCHS_TRAIN` values. The `BATCH_SIZE` is the size of the mini-batch that we will use for training:

```
# initialize parameters
DATA_DIR = "../data"
NUM_ACTIONS = 3 # number of valid actions (left, stay, right)
GAMMA = 0.99 # decay rate of past observations
INITIAL_EPSILON = 0.1 # starting value of epsilon
FINAL_EPSILON = 0.0001 # final value of epsilon
MEMORY_SIZE = 50000 # number of previous transitions to remember
NUM_EPOCHS_OBSERVE = 100
NUM_EPOCHS_TRAIN = 2000

BATCH_SIZE = 32
NUM_EPOCHS = NUM_EPOCHS_OBSERVE + NUM_EPOCHS_TRAIN
```

We instantiate the game and the experience replay queue. We also open up a log file and initialize some variables in preparation for training:

```
game = wrapped_game.MyWrappedGame()
experience = collections.deque(maxlen=MEMORY_SIZE)

fout = open(os.path.join(DATA_DIR, "rl-network-results.tsv"), "wb")
num_games, num_wins = 0, 0
epsilon = INITIAL_EPSILON
```

Next up, we set up the loop that controls the number of epochs of training. As noted previously, each epoch corresponds to a single game, so we reset the game state at this point. A game corresponds to a single episode of a ball falling from the ceiling and either getting caught by the paddle or being missed. The loss is the squared difference between the predicted and actual Q-value for the game.

We start the game off by sending it a dummy action (in our case, a *stay*) and get back the initial state tuple for the game:

```
for e in range(NUM_EPOCHS):
    game.reset()
    loss = 0.0

    # get first state
    a_0 = 1 # (0 = left, 1 = stay, 2 = right)
    x_t, r_0, game_over = game.step(a_0)
    s_t = preprocess_images(x_t)
```

The next block is the main loop of the game. This is the event loop in the original game that we moved to the calling code. We save the current state because we will need that for our experience replay queue, then decide what action signal to send the wrapped game. If we are in observation mode, we will just generate a random number corresponding to one of our actions, otherwise we will use ϵ -greedy exploration to either select a random action or use our neural network (which we are also training) to predict the action we should send:

```
while not game_over:
    s_tm1 = s_t

    # next action
    if e <= NUM_EPOCHS_OBSERVE:
        a_t = np.random.randint(low=0, high=NUM_ACTIONS, size=1)[0]
    else:
        if np.random.rand() <= epsilon:
            a_t = np.random.randint(low=0, high=NUM_ACTIONS, size=1)[0]
        else:
            q = model.predict(s_t)[0]
            a_t = np.argmax(q)
```

Once we know our action, we send it to the game by calling `game.step()`, which returns the new state, the reward and a Boolean flag indicating the game is over. If the reward is positive (indicating that the ball was caught), we increment the number of wins, and we store this (*state, action, reward, new state, game over*) tuple in our experience replay queue:

```
# apply action, get reward
x_t, r_t, game_over = game.step(a_t)
s_t = preprocess_images(x_t)
# if reward, increment num_wins
if r_t == 1:
    num_wins += 1
# store experience
experience.append((s_tm1, a_t, r_t, s_t, game_over))
```

We then draw a random mini-batch from our experience replay queue and train our network. For each session of training, we compute the loss. The sum of the losses for all the trainings in each epoch is the loss for the entire epoch:

```
if e > NUM_EPOCHS_OBSERVE:  
    # finished observing, now start training  
    # get next batch  
    X, Y = get_next_batch(experience, model, NUM_ACTIONS, GAMMA,  
BATCH_SIZE)  
    loss += model.train_on_batch(X, Y)
```

When the network is relatively untrained, its predictions are not very good, so it makes sense to explore the state space more in an effort to reduce the chances of getting stuck in a local minima. However, as the network gets more and more trained, we reduce the value of ϵ gradually so the model gets to predict more and more of the actions the network sends to the game:

```
# reduce epsilon gradually  
if epsilon > FINAL_EPSILON:  
    epsilon -= (INITIAL_EPSILON - FINAL_EPSILON) / NUM_EPOCHS
```

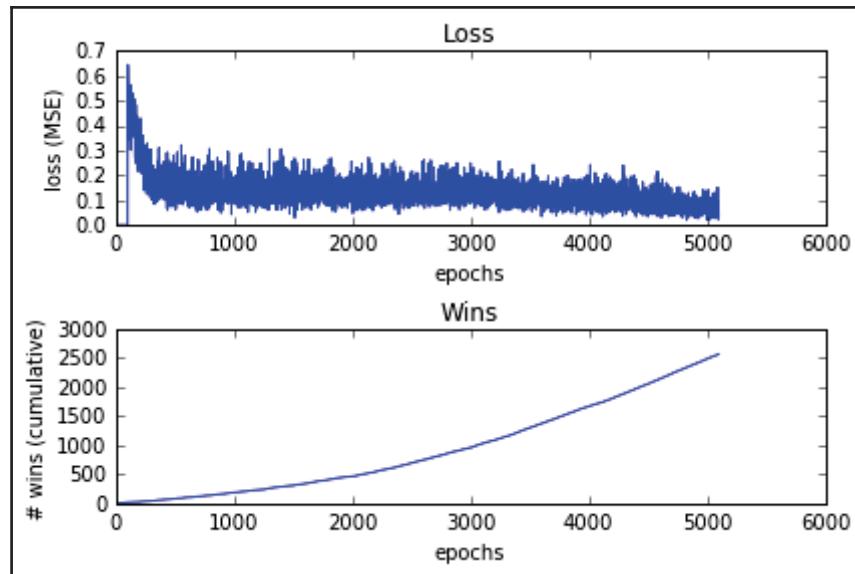
We write out a per epoch log both on console and into a log file for later analysis. After 100 epochs of training, we save the current state of the model so that we can recover in case we decide to stop training for any reason. We also save our final model so that we can use it to play our game later:

```
print("Epoch {:04d}/{:d} | Loss {:.5f} | Win Count {:d}"  
    .format(e + 1, NUM_EPOCHS, loss, num_wins))  
fout.write("{}{:04d}{:d}{:.5f}{:d}\n".format(e + 1, loss, num_wins))  
  
if e % 100 == 0:  
    model.save(os.path.join(DATA_DIR, "rl-network.h5"), overwrite=True)  
  
fout.close()  
model.save(os.path.join(DATA_DIR, "rl-network.h5"), overwrite=True)
```

We trained the game by making it observe 100 games, followed by playing 1,000, 2,000, and 5,000 games respectively. The last few lines of the log file for the 5,000 game run are shown next. As you can see, towards the end of the training, the network gets quite skilled at playing the game:

```
Epoch 5075/5100 | Loss 0.02603 | Win Count 2548
Epoch 5076/5100 | Loss 0.06248 | Win Count 2549
Epoch 5077/5100 | Loss 0.09836 | Win Count 2550
Epoch 5078/5100 | Loss 0.05955 | Win Count 2551
Epoch 5079/5100 | Loss 0.07357 | Win Count 2552
Epoch 5080/5100 | Loss 0.05425 | Win Count 2553
Epoch 5081/5100 | Loss 0.05961 | Win Count 2553
Epoch 5082/5100 | Loss 0.05737 | Win Count 2553
Epoch 5083/5100 | Loss 0.06699 | Win Count 2554
Epoch 5084/5100 | Loss 0.04265 | Win Count 2555
Epoch 5085/5100 | Loss 0.06579 | Win Count 2556
Epoch 5086/5100 | Loss 0.06825 | Win Count 2557
Epoch 5087/5100 | Loss 0.09329 | Win Count 2557
Epoch 5088/5100 | Loss 0.06124 | Win Count 2558
Epoch 5089/5100 | Loss 0.15128 | Win Count 2559
Epoch 5090/5100 | Loss 0.03769 | Win Count 2560
Epoch 5091/5100 | Loss 0.06348 | Win Count 2560
Epoch 5092/5100 | Loss 0.03817 | Win Count 2561
Epoch 5093/5100 | Loss 0.05225 | Win Count 2562
Epoch 5094/5100 | Loss 0.04986 | Win Count 2563
Epoch 5095/5100 | Loss 0.06316 | Win Count 2564
Epoch 5096/5100 | Loss 0.07558 | Win Count 2564
Epoch 5097/5100 | Loss 0.04027 | Win Count 2565
Epoch 5098/5100 | Loss 0.03801 | Win Count 2566
Epoch 5099/5100 | Loss 0.02446 | Win Count 2567
Epoch 5100/5100 | Loss 0.04321 | Win Count 2568
```

The plot of loss and win count over epoch, shown in the following graph, also tells a similar story. While it does look like the loss could converge further with more training, it has gone down from 0.6 to around 0.1 in 5000 epochs of training. Similarly, the plot of the number of wins curve upward, showing that the network is learning faster as the number of epochs increases:



Finally, we evaluate the skill of our trained model by making it play a fixed number of games (100 in our case) and seeing how many it can win. Here is the code to do this. As previously, we start with our imports:

```
from __future__ import division, print_function
from keras.models import load_model
from keras.optimizers import Adam
from scipy.misc import imresize
import numpy as np
import os
import wrapped_game
```

We load up the model we had saved at the end of training and compile it. We also instantiate our `wrapped_game`:

```
DATA_DIR = "../data"
model = load_model(os.path.join(DATA_DIR, "rl-network.h5"))
model.compile(optimizer=Adam(lr=1e-6), loss="mse")

game = wrapped_game.MyWrappedGame()
```

We then loop over 100 games. We instantiate each game by calling its `reset()` method, and start it off. Then, for each game, until it is over, we call on the model to predict the action with the best Q-function. We report a running total of how many games it won.

We ran the test with each of our models. The first one that was trained for 1,000 games won 42 of 100 games, the one trained for 2,000 games won 74 of 100 games, and the one trained for 5,000 games won 87 of 100 games. This clearly shows that the network is improving with training:

```
num_games, num_wins = 0, 0
for e in range(100):
    game.reset()

    # get first state
    a_0 = 1 # (0 = left, 1 = stay, 2 = right)
    x_t, r_0, game_over = game.step(a_0)
    s_t = preprocess_images(x_t)

    while not game_over:
        s_tm1 = s_t
        # next action
        q = model.predict(s_t) [0]
        a_t = np.argmax(q)
        # apply action, get reward
        x_t, r_t, game_over = game.step(a_t)
        s_t = preprocess_images(x_t)
        # if reward, increment num_wins
        if r_t == 1:
            num_wins += 1

    num_games += 1
    print("Game: {:03d}, Wins: {:03d}".format(num_games, num_wins),
          end="r")
    print("")
```

If you run the evaluation code with the call to run it in headless mode commented out, you can watch the network playing the game and it's quite amazing to watch. Given that the Q-value predictions start off as random values and that it's mainly the sparse reward mechanism that provides the guidance to the network during training, it is almost unreasonable that the network learns to play the game this effectively. But as with other areas of deep learning, the network does in fact learn to play quite well.

The example presented previously is fairly simple, but it illustrates the process by which deep reinforcement learning models work, and hopefully has helped create a mental model using which you can approach more complex implementations. One implementation you might find interesting is Ben Lau's implementation of FlappyBird (for more information refer to: *Using Keras and Deep Q-Network to Play FlappyBird*, by Ben Lau, 2016. and GitHub page: <https://github.com/yanpanlau/Keras-FlappyBird>) using Keras. The Keras-RL project (<https://github.com/matthiasplappert/keras-rl>), a Keras library for deep reinforcement learning, also has some very good examples.

Since the original proposal from DeepMind, there have been other improvements suggested, such as double Q-learning (for more information refer to: *Deep Reinforcement Learning with Double Q-Learning*, by H. Van Hasselt, A. Guez, and D. Silver, AAAI. 2016), prioritized experience replay (for more information refer to: *Prioritized Experience Replay*, by T. Schaul, arXiv:1511.05952, 2015), and dueling network architectures (for more information refer to: *Dueling Network Architectures for Deep Reinforcement Learning*, by Z. Wang, arXiv:1511.06581, 2015). Double Q-learning uses two networks - the primary network chooses the action and the target network chooses the target Q-value for the action. This reduces possible overestimation of Q-values by the single network, and allows the network to train quicker and better. Prioritized experience replay increases the probability of sampling experience tuples with a higher expected learning progress. Dueling network architectures decompose the Q-function into state and action components and combine them back separately.

All of the code discussed in this section, including the base game that can be played by a human player, is available in the code bundle accompanying this chapter.

The road ahead

In January 2016, DeepMind announced the release of AlphaGo (for more information refer to: *Mastering the Game of Go with Deep Neural Networks and Tree Search*, by D. Silver, Nature 529,7587, pp. 484-489, 2016), a neural network to play the game of Go. Go is regarded as a very challenging game for AIs to play, mainly because at any point in the game, there are an average of approximately 10^{170} possible (for more information refer to: <http://ai-depot.com/LogicGames/Go-Complexity.html>) moves (compared with approximately 10^{50} for chess). Hence determining the best move using brute force methods is computationally infeasible. At the time of publication, AlphaGo had already won 5-0 in a 5-game competition against the current European Go champion, Fan Hui. This was the first time that any computer program had defeated a human player at Go. Subsequently, in March 2016, AlphaGo won 4-1 against Lee Sedol, the world's second professional Go player.

There were several notable new ideas that went into AlphaGo. First, it was trained using a combination of supervised learning from human expert games and reinforcement learning by playing one copy of AlphaGo against another. You have seen applications of both these ideas in previous chapters.

Second, AlphaGo was composed of a value network and a policy network. During each move, AlphaGo uses Monte Carlo simulation, a process used to predict the probability of different outcomes in the future in the presence of random variables, to imagine many alternative games starting from the current position. The value network is used to reduce the depth of the tree search to estimate win/loss probability without having to compute all the way to the end of the game, sort of like an intuition about how good the move is. The policy network is used to reduce the breadth of the search by guiding the search towards actions that promise the maximum immediate reward (or Q-value). For a more detailed description, please refer to the blog post: *AlphaGo: Mastering the ancient game of Go with Machine Learning*, Google Research Blog, 2016.

While AlphaGo was a major improvement over the original DeepMind network, it was still playing a game where all the players can see all the game pieces, that is, they are still games of perfect information. In January, 2017, researchers at Carnegie Mellon University announced Libratus (for more information refer to: *AI Takes on Top Poker Players*, by T. Revel, New Scientist 223.3109, pp. 8, 2017), an AI that plays Poker. Simultaneously, another group comprised of researchers from the University of Alberta, Charles University of Prague, and Czech Technical University (also from Prague), have proposed the DeepStack architecture (for more information refer to: *DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker*, by M. Moravaak, arXiv:1701.01724, 2017) to do the same thing. Poker is a game of imperfect information, since a player cannot see the opponent's cards. So, in addition to learning how to play the game, the Poker playing AI also needs to develop an intuition about the opponent's game play.

Rather than use a built-in strategy for its intuition, Libratus has an algorithm that computes this strategy by trying to achieve a balance between risk and reward, also known as the Nash equilibrium. From January 11, 2017 to January 31, 2017, Libratus was pitted against four top human Poker players (for more information refer to: *Upping the Ante: Top Poker Pros Face Off vs. Artificial Intelligence*, Carnegie Mellon University, January 2017), and beat them resoundingly.

DeepStack's intuition is trained using reinforcement learning, using examples generated from random Poker situations. It has played 33 professional Poker players from 17 countries and has a win rating that makes it an *order of magnitude* better than a good player rating (for more information refer to: *The Uncanny Intuition of Deep Learning to Predict Human Behavior*, by C. E. Perez, Medium corporation, Intuition Machine, February 13, 2017).

As you can see, these are very exciting times indeed. Advances that started with deep learning networks able to play arcade games have led to networks that can effectively read your mind, or at least anticipate (sometimes non-rational) human behavior and win at games of bluffing. The possibilities with deep learning seem to be just limitless.

Summary

In this chapter, we have learned the concepts behind reinforcement learning, and how it can be used to build deep learning networks with Keras that learn how to play arcade games based on reward feedback. From there, we moved on to briefly discuss advances in this field, such as networks that have been taught to play harder games such as Go and Poker at a superhuman level. While game playing might seem like a frivolous application, these ideas are the first step towards general artificial intelligence, where a network learns from experience rather than large amounts of training data.

Conclusion

Congratulations on making it to the end of the book! Let us take a moment and see how far we have come since we started.

If you are like most readers, you started with some knowledge of Python and some background in machine learning, but you were interested in learning more about deep learning and wanted to be able to apply these deep learning skills using Python.

You learned how to install Keras on your machine and started using it to build simple deep learning models. You then learned about the original deep learning model, the multi-layer perceptron, also called the **fully connected network (FCN)**. You learned how to build this network using Keras.

You also learned about the many tunable parameters that you need to tweak to get good results from your network. With Keras, a lot of the hard work has been done for you since it comes with sensible defaults, but there are occasions where this knowledge will be helpful to you.

Continuing on from there, you were introduced to **convolutional neural network (CNN)**, originally built to exploit feature locality of images, although you can also use them for other types of data such as text, audio or video. Once again, you saw how to build a CNN using Keras. You also saw the functionality that Keras provides to build CNNs easily and intuitively. You saw how to use pre-trained image networks to make predictions about your own images, via the process of transfer learning and fine-tuning.

From there, you learned about **generative adversarial network (GAN)**, which are a pair of networks (usually CNN) that attempt to work against each other and, in the process, make each other stronger. GANs are a cutting-edge technology in the deep learning space; a lot of recent work is going on around GANs.

From there, we turned our attention to text and we learned about **word embeddings**, which have become the most common technology used for the vector representation of text in the last couple of years. We looked at various popular word embedding algorithms and saw how to use pre-trained word embeddings to represent collections of words, as well as support for word embeddings in Keras and gensim.

We then looked at **recurrent neural network (RNN)**, a class of neural network optimized for handing sequence data such as text or time series. We learned about the shortcomings of the basic RNN model and how these are alleviated in the more powerful variants such as the **long short term model (LSTM)** and **gated recurrent unit (GRU)**. We looked at a few examples where these components are used. We also looked briefly at Stateful RNN models and where they might be used.

Next up, we looked at a few additional models that don't quite fit the molds of the models we have spoken so far. Among them are **autoencoders**, a model for unsupervised learning—**regression networks** that predict a continuous value rather than a discrete label. We introduced the **Keras functional API**, which allows us to build complex networks with multiple inputs and outputs and share components among multiple pipelines. We looked at ways to customize Keras to add functionality that doesn't currently exist.

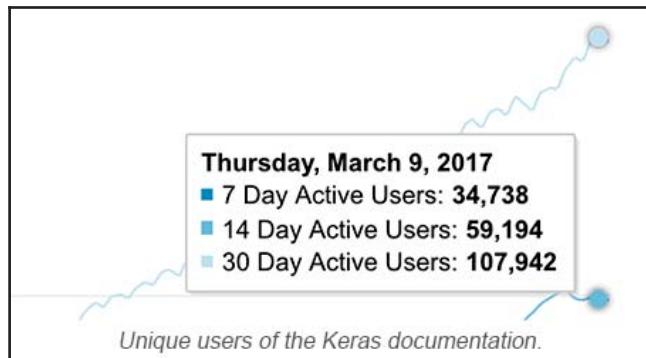
Finally, we looked at training deep learning networks using **reinforcement learning** in the context of playing arcade games, which many consider a first step toward a general artificial intelligence. We provided a Keras example of training a simple game. We then briefly described advances in this field in the context of networks playing even harder games such as Go and Poker at a superhuman level.

We believe you are now equipped with the skills to solve new machine learning problems using deep learning and Keras. This is an important and valuable skill in your journey to becoming a deep learning expert.

We would like to thank you for letting us help you on your journey to deep learning mastery.

Keras 2.0 — what is new

According to Francois Chollet, Keras was released two years ago, in March, 2015. It then proceeded to grow from one user to one hundred thousand. The following image, taken from the Keras blog, shows the growth of number of Keras users over time.



One important update with Keras 2.0 is that the API will now be a part of TensorFlow, starting with TensorFlow 1.2. Indeed, Keras is becoming more and more the *lingua franca* for deep learning, a *spec* used in an increasing number of deep learning contexts. For instance, Skymind is implementing Keras spec in Scala for ScalNet, and Keras.js is doing the same for JavaScript for running of deep learning directly in the browser. Efforts are also underway to provide a Keras API for MXNET and CNTK deep learning toolkits.

Installing Keras 2.0

Installing Keras 2.0 is very simple via the `pip install keras --upgrade` followed by `pip install tensorflow --upgrade`.

API changes

The Keras 2.0 changes implied the need to rethink some APIs. For full details, please refer to the release notes (<https://github.com/fchollet/keras/wiki/Keras-2.0-release-notes>). This module `legacy.py` summarizes the most impactful changes and prevents warnings when using Keras 1.x calls:

```
"""
Utility functions to avoid warnings while testing both Keras 1 and 2.
"""

import keras
keras_2 = int(keras.__version__.split(".")[0]) > 1 # Keras > 1

def fit_generator(model, generator, epochs, steps_per_epoch):
    if keras_2:
        model.fit_generator(generator, epochs=epochs,
        steps_per_epoch=steps_per_epoch)
    else:
        model.fit_generator(generator, nb_epoch=epochs,
        samples_per_epoch=steps_per_epoch)

def fit(model, x, y, nb_epoch=10, *args, **kwargs):
    if keras_2:
        return model.fit(x, y, *args, epochs=nb_epoch, **kwargs)
    else:
        return model.fit(x, y, *args, nb_epoch=nb_epoch, **kwargs)

def l1l2(l1=0, l2=0):
    if keras_2:
        return keras.regularizers.L1L2(l1, l2)
    else:
        return keras.regularizers.l1l2(l1, l2)

def Dense(units, W_regularizer=None, W_initializer='glorot_uniform',
          **kwargs):
    if keras_2:
        return keras.layers.Dense(units, kernel_regularizer=W_regularizer,
        kernel_initializer=W_initializer, **kwargs)
    else:
        return keras.layers.Dense(units, W_regularizer=W_regularizer,
                                 init=W_initializer, **kwargs)

def BatchNormalization(mode=0, **kwargs):
    if keras_2:
        return keras.layers.BatchNormalization(**kwargs)
    else:
        return keras.layers.BatchNormalization(mode=mode, **kwargs)
```

```
def Convolution2D(units, w, h, W_regularizer=None,
W_initializer='glorot_uniform', border_mode='same', **kwargs):
    if keras_2:
        return keras.layers.Conv2D(units, (w, h), padding=border_mode,
                                  kernel_regularizer=W_regularizer,
                                  kernel_initializer=W_initializer,
                                  **kwargs)
    else:
        return keras.layers.Conv2D(units, w, h, border_mode=border_mode,
                                  W_regularizer=W_regularizer, init=W_initializer, **kwargs)

def AveragePooling2D(pool_size, border_mode='valid', **kwargs):
    if keras_2:
        return keras.layers.AveragePooling2D(pool_size=pool_size,
                                             padding=border_mode, **kwargs)
    else:
        return keras.layers.AveragePooling2D(pool_size=pool_size,
                                             border_mode=border_mode,
                                             **kwargs)
```

There are also a number of breaking changes. In particular:

- The maxout dense, time distributed dense, and highway legacy layers have been removed
- The batch normalization layer no longer supports the mode argument, because Keras internals have changed
- Custom layers have to be updated
- Any undocumented Keras functionality could have broken

In addition, the Keras code base has been instrumented to detect the use of the Keras 1.x API calls and show deprecation warnings that show how to change the call to conform to the Keras 2 API. If you have some volume of Keras 1.x code already and are hesitant to try Keras 2 because of the fear of non-breaking changes, these deprecation warnings from the Keras 2 code base can be very helpful in making the transition.

Index

A

activation function
about 15
reference link 16
ReLU 15
sigmoid function 14
Adam 30
AlphaGo 284
Amazon AWS
Keras, installing 55
references 57
art forgery
reference link 106
autoencoders
about 224, 288
Keras autoencoder example 226, 230, 232, 233
AWS Spot instance
reference link 67

B

bAbI project
URL 236
backpropagation 40
backpropagation through time (BPTT) 184
baseline
establishing 21
batch computation
size, increasing 37
batch normalization
about 63
reference link 63
Bellman equation 267
bias 73
bidirectional RNNs 205

C

Caffe
reference link 96, 246
callbacks
for training process, customizing 66
CBOW word2vec model 147
checkpointing 67
CIFAR-10 images
forging, Keras adversarial GANs 122
performance, improving with data augmentation 91
performance, improving with deep network 89
predicting with 95
recognizing, with deep learning 84
reference link 84, 94
collaborative filtering (CF) 225
concatenative TTS 130
convolutional neural network (CNN) 43, 71, 75, 159, 264, 287
custom layer
guidelines, for building 245

D

data augmentation
CIFAR-10 images, performance improving 91
deep convolutional generative adversarial networks (DCGAN)
about 112
reference link 112
deep convolutional neural network (DCNN)
about 72
bias 73
image recognition 96
LeNet, example 76
pooling layers 74
receptive fields 72

weights 73
deep dreaming 249
deep learning
about 42, 83
CIFAR-10 images, recognizing 84
reference link 105
deep network
CIFAR-10 images, performance improving 89
composing 235
memory network, for question answering 236, 237, 240, 243

DeepMind
reference link 138
URL 130

dependencies
installing 45
distributed representations 141

Docker
Keras, installing 49
references 49
dropout
Keras net, improving 25

E

epochs
count, increasing 34
event loop 273

F

features
extraction, for recycling pre-built deep learning models 100
fully connected network (FCN) 287

G

GAN applications 108
gated recurrent unit (GRU) 176, 196, 197, 288
generative adversarial network (GAN) 105, 106, 248, 288
generative models
about 248
deep dreaming 249, 251, 252, 255, 257
style transfer 257, 259, 262
GloVe-Python project
reference link 158

GloVe
about 156
exploring 156
Google Cloud ML
Keras, installing 52
reference link 52
gradient descent (GD) 28
gradients
exploding 186
vanishing 184
GRU, with Keras
POS tagging 198, 200, 202, 204

H

handwritten digits recognizing example
about 16
baseline, establishing 21
batch computation, size increasing 37
epochs, count increasing 34
experiments run, summarizing 37
hyperparameter, tuning 40
internal hidden neurons, count increasing 35
Keras net, executing 21
Keras net, improving with dropout 25
Keras net, improving with hidden layers 22
neural network, defining in Keras 17
OHE 17
optimizer learning rate, controlling 34
optimizers, testing in Keras 28
output, predicting 40
regularization, adopting to avoid overfit 38

HDF5
reference link 67

hidden layers
Keras net, improving 22
hyperparameter
tuning 40

I

image recognition
DCNN 96
inception-v3 net, used for transfer learning 101
Keras built-in VGG-16 net module, utilizing 98
pre-built deep learning models, recycling to extract features 100

with VGG-16 net 97
ImageNet ILSVRC-2012
reference link 96
inception-v3 net
used, for transfer learning 101
information retrieval (IR) 140
initializers
reference link 11
intelligence quotients (IQ) 219
internal hidden neurons
count, increasing 35

J

Jupyter Notebooks
reference link 51

K

Keras 2.0
about 289
API 290
installing 289
Keras adversarial GANs
CIFAR-10 images, forging 122
MNIST, forging 115
Keras adversarial
references 115
Keras API
about 59
Keras architecture 59
losses functions 64
metric function 65
model architectures, loading 66
model architectures, saving 66
optimizers 65
predefined activation functions 63
predefined neural network layers 60
utility operations 65
weights, loading 66
weights, saving 66
Keras architecture
about 59
tensor 59
Keras built-in VGG-16 net module
utilizing 98
Keras customization

about 244
custom normalization layer, building 245, 248
lambda layer, using 244
Keras deep Q-network
for catch 271, 273, 276, 279, 281, 284
Keras functional API 217, 288
Keras metrics
about 20
reference link 20
Keras net
executing 21
improving, with dropout 25
improving, with hidden layers 22
Keras objective functions
reference link 19
Keras
configuring 48
customizing 243
dependencies, installing 45
installing 45, 47
installing, on Amazon AWS 55
installing, on Docker 49
installing, on Google Cloud ML 52
installing, on Microsoft Azure 57
LeNet code 76
neural network, defining 17
optimizers, testing 28
reference link 54, 99
TensorFlow, installing 46
TensorFlow, testing 47
testing 47
Theano, installing 46
Theano, testing 47
using 68, 69

L

latent semantic analysis (LSA) 140
LeNet
about 76
code, in Keras 76
deep learning 83
Libratus 285
local response normalization 245
long short term memory (LSTM) 176, 187, 288
losses functions

about 64
reference link 64
LSTM, with Keras
sentiment analysis 189, 191, 193, 195

M

markov decision process (MDP) 266
mean squared error (MSE) 219
metric function
about 65
reference link 65
Microsoft Azure
Keras, installing 57
MNIST
forging, Keras adversarial GANs 115
reference link 16, 84
model architectures
loading 66
saving 66
models, Keras
composing 60
functional composition 60
sequential composition 60
multilayer perceptron (MLP)
activation function 15
activation function, rectified linear unit (ReLU)
function 15
activation function, sigmoid function 14
example, of network 12
perceptron, training issues 13
perceptron, training solution 13

N

natural language processing (NLP) 140
natural language toolkit (NLTK) 159
neural network
defining, in Keras 17

O

objective function
binary cross-entropy 19
categorical cross-entropy 19
MSE 19
one-hot encoding (OHE) 17, 140
OpenAI

URL 105
optimizer learning rate
controlling 34
optimizers
about 65
reference link 30, 65
testing, in Keras 28
out of vocabulary (OOV) 190, 228

P

packages
reference link 45
parametric TTS 130
part-of-speech (POS) tagging 184
perceptron
about 11
sequential Keras model 11
training, issues 13
training, solution 13
pooling layers
about 74
average pooling 75
max-pooling 74
reference link 75
pre-built deep learning models
recycling, to extract features 100
pre-trained embeddings
learned embeddings from GloVe, fine-tuning
168, 169
learned embeddings from word2vec, fine-tuning
164, 167
learning, from scratch 159, 161, 164
look up embeddings 169, 171, 173
using 158
predefined activation functions 63
predefined neural network layers
about 60
batch normalization 63
convolution layers 62
dense model 61
pooling layers 62
recurrent neural network 61
regularization 62
preemptible virtual machine
reference link 67

principal component analysis (PCA) 224

Pygame

installing 271

URL 271

Q

Q-learning 267

Q-table 268

Quiver

reference link 69

using 69

R

receptive fields 72

rectified linear unit (ReLU) function 15

recurrent neural network (RNN)

about 159, 288

GRU 61

LSTM 61

simple 61

regression networks

about 219

Keras regression example 220, 223

regularization, types

Elastic net regularization 39

L1 regularization 39

L2 regularization 39

regularization

about 62

adopting, to avoid overfit 38

regularizers

reference link 39

reinforcement learning

about 265, 288

deep Q-network, as Q-function 268

experience replay 271

exploration, balancing with exploitation 270

future rewards, maximizing 266

Q-learning 267

replay memory 271

RMSprop 30

RNN topologies 183, 184

RNN variants 213

S

sequential Keras model 11

sigmoid function 14

SimpleRNN cells 176

SimpleRNN, with Keras

text, generating 178, 179, 181, 182

skip-gram word2vec model 143, 146

stateful LSTM, with Keras

electricity consumption, predicting 206, 209, 211, 213

stateful RNNs 206

stochastic gradient descent (SGD) 30

stride length 73

style transfer 257

T

tensor 59

TensorBoard

using 68

TensorFlow

installing 46

reference link 59

references 10

testing 47

URL, for installing 46

text-to-speech (TTS) systems

about 130

concatenative TTS 130

parametric TTS 130

Theano

installing 46

reference link 59

testing 47

third-party implementations, word2vec

using 152

training process

checkpointing 67

customizing, callbacks 66

Keras, using 68, 69

Quiver, using 69

TensorBoard, using 68

transfer learning

Inception-v3 net, used 101

U

unsupervised learning 224
utility operations 65

V

VGG-16 net
 used, for image recognition 97
visual cortex V1 10

W

WaveNet
 about 130

reference link 130, 131
weights
 about 73
 loading 66
 saving 66
word2vec
 about 142
 CBOW architecture 142
 CBOW word2vec model 147, 149
 embeddings, extracting from model 149, 151
 skip-gram word2vec model 143, 147
 skip-skip-gram word2vec model 146, 147
 third-party implementations, using 152, 156