Cognitive Technologies

Christian Bauckhage
Rafet Sifa

# Quantum Computing from Hopfield Nets

A Textbook with Python Code Examples

OPEN ACCESS

Springer

Christian Bauckhage · Rafet Sifa

# Quantum Computing from Hopfield Nets

A Textbook with Python Code Examples

Springer

Christian Bauckhage
Department of Hybrid Intelligence
Fraunhofer IAIS
Sankt Augustin, Nordrhein-Westfalen,
Germany

Rafet Sifa
Department of Hybrid Intelligence
Fraunhofer IAIS
Sankt Augustin, Nordrhein-Westfalen,
Germany

ISSN 1611-2482 ISSN 2197-6635 (electronic)
Cognitive Technologies
ISBN 978-3-031-99401-2 ISBN 978-3-031-99402-9 (eBook)
https://doi.org/10.1007/978-3-031-99402-9

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

# Preface

This book is intended for students of computer science who are looking for a gentle introduction to the world of quantum computing.

More specifically, it is written for readers who have basic knowledge of Artificial Intelligence (AI) and Machine Learning (ML) and are passingly familiar with search algorithms, optimization techniques, and neural networks. This is not because we are interested in quantum AI or quantum ML but because our basic premise is that there exists a conceptual bridge between certain AI/ML models and certain quantum computing models. Our purpose in this book is therefore (1) to revisit these AI/ML models and their applications and (2) to build on this familiar foundation to segue into the study of quantum computing and its possible use cases.

Our presentation is technical but pragmatic and practice oriented. We cover theory to the necessary extent but largely proceed in an example-driven manner. Most of our examples are concerned with combinatorial optimization and consider problems that can be cast as quadratic unconstrained binary optimization problems. Numerous Python/NumPy/SciPy codes will support our mathematical discussion and demonstrate how to put theory into practice.

## Why Learn About Quantum Computing

Quantum computers harness quantum mechanical effects such as superposition, entanglement, or quantum tunneling for information processing. When used cleverly, this allows for levels of computational efficiency which are unreachable with digital computers. Indeed, quantum computing algorithms can generally be expected to be at least quadratically faster than their classical counterparts. For certain problems, there can be even up to exponential speedup. This is of course of interest in a world with ever growing demand for efficient number crunching.

The theoretical foundations of quantum computing were first established in the 1980s and substantial theoretical progress happened in the 1990s. Technical progress, on the other hand, was a longer time coming and only became noteworthy in the

2010s. But by now we are living in a world where there exist working quantum computers. Granted, at the time of writing this book, these are of still limited capabilities but they demonstrate that quantum computing is feasible and progress is evident. It therefore comes as little surprise that more and more researchers and industrial players are excited about quantum algorithms and their potential applications.

## How to Learn About Quantum Computing

The above then begs the questions of why quantum computing has not yet received widespread attention in mainstream computer science and why it is not yet widely taught at computer science departments. Well, while quantum computing offers great computational power, it also requires a different kind of thinking than digital computing. Indeed, experience shows that there are at least three major obstacles for a broader engagement with the topic:

1. Quantum mechanical phenomena appear to be weird, abstract, and hard to accept for they mainly manifest on sub-atomic scales and cannot be observed in our daily environments.
2. The mathematical tools used for modeling these phenomena are complex (no pun intended) and again rather abstract.
3. To the uninitiated, the mathematical notation used in the quantum computing literature appears even more abstract and needs getting used to.

With this book, we try to rise to the challenge and go on an extended journey towards understating core concepts of quantum computing. Our journey starts in territory far away from quantum mechanics but first (re)visits areas more familiar to computer scientists before it ends in the realm of applied quantum physics. Along the way, we are going to encounter familiar ideas and models as well as rather abstract mathematical concepts and physical principles. The latter are only rarely considered in mainstream computer science but will turn out to be fundamental to theory and practice of quantum computing.

Put less prosaically, we structure the material in this book into three major parts: *Preliminaries*, *Hopfield Nets*, and *Quantum Computing*.

Hopfield nets are a venerable family of rather simple recurrent neural networks which allow for combinatorial problem solving. Their underlying principles have close ties to models and concepts used in physics but are all in all fairly easy to understand. This motivates our basic idea for this book: By studying Hopfield nets from a variety of uncommon angles, we can familiarize ourselves with mathematical tools and techniques which play a central role in quantum mechanics and thus in quantum computing. To be able to consider these uncommon angles, we need a few preliminaries. Once we have studied those and their connections to Hopfield nets, we should be well prepared to venture into the abstract and at first sight unintuitive worlds of quantum mechanics and quantum computing.

## Overview

The part on preliminaries covers important mathematical concepts and presents a class of problems for which future quantum computers may have an edge over digital computers.

In Chap. 1, we introduce an example of a practically important combinatorial optimization problem, discuss why it is generally difficult to solve, and hint at how quantum computers may circumvent these difficulties. This chapter will recall basic aspects of complexity theory and combinatorial optimization and thus sets the stage for all our upcoming discussions.

In Chap. 2, we study binary and bipolar numbers and vectors. These are central to a deeper understanding of the behavior of Hopfield nets and quantum computing algorithms and allow for a whole spectrum of uncommon perspectives on seemingly mundane objects. In preparation for things to come, we will touch upon Boolean logic, (multi-)linear algebra, probability theory and statistics, geometry, and graph theory.

In Chap. 3, we take things further and study tensor products. These, too, are of fundamental importance in quantum computing and we will segue into understanding their characteristic by relating Boolean functions to multi-linear algebra.

In Chap. 4, we then formally define the notions of quadratic unconstrained binary and bipolar optimization problems (QUBOs) which we first encountered in the introductory chapter. QUBOs are central to our study because they can be solved by running a Hopfield net and any problem that can be solved by running a Hopfield net can be solved on a quantum computer. We will therefore have a closer look at their structure and characteristics and discuss some of their peculiarities.

In Chap. 5, we look at specific QUBO instances. Our application examples will consider problems in areas like combinatorial optimization, constraint satisfaction, data mining, and machine learning and are supposed to illustrate that QUBOs provide a versatile and widely applicable modeling framework. In short, we will demonstrate how to rethink familiar, seemingly unrelated problems in terms of an overarching formalism that allows for running Hopfield nets and quantum implementations.

Part II of this book is dedicated to theory and practice of Hopfield nets as problem solvers.

In Chap. 6, we introduce classical Hopfield nets, the notion of their state spaces, energy functions, and synchronous and asynchronous update mechanisms. Our main take home message will be that Hopfield nets realize an energy minimization process and thus build a bridge between neurocomputing and physics.

In Chap. 7, we study yet another update mechanism for Hopfield nets and discuss how energy gradients can be exploited for better performance. We will also see Hopfield nets in action and practically demonstrate how they can solve constraint satisfaction and combinatorial optimization problems.

In Chap. 8, we begin moving even closer to physics and view Hopfield nets from the point of view of statistical mechanics. We will learn about stochastic Hopfield nets or Boltzmann machines, study their characteristics, and discuss further update mechanisms. In particular, we will learn about stochastic simulated annealing and mean field annealing as alternative ways of running Hopfield nets.

In Chap. 9, we widen our perspective even further and view Hopfield nets from the perspective of quantum mechanics. We will introduce completely different representations of their states and energy functions and relate those to the tensor product spaces studied earlier. Our new perspective on Hopfield nets is of purely theoretical interest as it generally defies implementations on digital computers. However, it will allow us to get to known the fundamental notion of the Hamiltonian of a physical system, its eigenstates, and eigenvalues.

Part III of this book builds on the material studied so far and introduces core concepts from quantum mechanics and quantum computing.

In Chap. 10, we look at the bare essentials of quantum mechanics, its axiomatic mathematical formulation, and its mathematical notation which takes getting used to. This chapter will be dense, abstract, and technical but there is no way around it. However, given all the material studied up until this point, highly abstract tensor state spaces and linear operators acting on them will not be alien to us anymore.

In Chap. 11, we familiarize ourselves with aspects of quantum weirdness and with quantum phenomena such as energy quantization, quantum tunneling, and quantum superposition. We will study three didactic quantum systems and solve their Schrödinger equations to see that quantum systems behave different than classical systems. The kind of computations we perform in this chapter will prepare us for more abstract computations in the following chapters.

In Chap. 12, we finally look at quantum computing. First, we will state axioms of quantum computing and relate them to our discussions of QUBOs and Hopfield nets. Then, we will particularly look at the paradigm of adiabatic quantum computing and how it allows for QUBO solving. Given all our preparations up until this point, the material in this chapter should be easy to understand and will underline that there really is a surprisingly close connection to Hopfield nets.

In Chap. 13, we conclude our overall study by presenting an outlook to the possibilities of quantum gate computing. This paradigm views quantum algorithms from a different perspective than the one we have assumed up until now. However, all the material we studied previously will enable us to understand the gist behind the quantum circuit model of quantum computing and how corresponding algorithms may allow us to harness quantum advantages. Of course, quantum circuit algorithms would merit much deeper study but this chapter is really only meant to create interest in this exciting topic.

Sankt Augustin, Germany
April 2025

Christian Bauckhage
Rafet Sifa

**Acknowledgements** Parts of the material in this book were adopted from our long running lectures on pattern recognition, on the foundations of quantum computing, and on quantum computing algorithms which are taught in the computer science master's program at the University of Bonn.

Other parts were originally developed for the education outreach activities of the Lamarr Institute for Machine Learning and Artificial Intelligence which is funded by the Federal Ministry of Education and Research of Germany and the state of North-Rhine Westphalia. We gratefully acknowledge this support.

Sankt Augustin, Germany  
April 2025

Christian Bauckhage  
Rafet Sifa

**Competing Interests** The authors have no competing interests to declare that are relevant to the content of this manuscript.

# Contents

# Part I
# Preliminaries

# Chapter 1
# Setting the Stage

## 1.1 The Subset Sum Problem

Are you bored and have no idea how to spend your time? Then here is a task that may keep you busy for a while: Consider the following *multiset* of $n = 100$ integers

$$\begin{aligned} \mathcal{X} = \{ \; & 265,\ 453,\ 311,\ 876,\ 158,\ 344,\ 65,\ 411,\ 366,\ 314, \\ & 200,\ 816,\ 305,\ 716,\ 892,\ 787,\ 45,\ 258,\ 967,\ 669, \\ & 525,\ 638,\ 351,\ 438,\ 839,\ 438,\ 732,\ 675,\ 429,\ 278, \\ & 175,\ 192,\ 408,\ 243,\ 733,\ 176,\ 111,\ 570,\ 332,\ 766, \\ & 925,\ 680,\ 305,\ 805,\ 167,\ 162,\ 442,\ 404,\ 14,\ 603, \\ & 279,\ 636,\ 927,\ 757,\ 780,\ 892,\ 178,\ 148,\ 11,\ 766, \\ & 272,\ 520,\ 317,\ 573,\ 89,\ 776,\ 389,\ 444,\ 429,\ 984, \\ & 296,\ 68,\ 415,\ 257,\ 205,\ 409,\ 664,\ 472,\ 888,\ 25, \\ & 641,\ 367,\ 791,\ 687,\ 344,\ 320,\ 936,\ 520,\ 36,\ 494, \\ & 719,\ 362,\ 78,\ 437,\ 841,\ 163,\ 869,\ 945,\ 918,\ 840 \; \} \end{aligned} \tag{1.1}$$

and decide if there is any subset $\mathcal{X}' \subseteq \mathcal{X}$ whose elements sum to the target value of

$$T = 8364 \; . \tag{1.2}$$

While there may be savants who just need a single glance at the numbers in (1.1) to decide if some of them sum to the target in (1.2), most of us will likely agree that this task is not that easy. For those who don't, we could up the ante and list more numbers to be combed through, for instance, twice as many ($n = 200$), qaudradically as many ($n = 10{,}000$), or cubically as many ($n = 1{,}000{,}000$). Eventually, most everybody will admit that computer assistance is called for.

Alas, and this is crucial, even the most powerful computers available today will generally struggle with this kind of problem! But how could this be? Do modern multicore processors not execute many millions of instructions per second? Why

© The Author(s) 2026
C. Bauckhage and R. Sifa, *Quantum Computing from Hopfield Nets*,
Cognitive Technologies, https://doi.org/10.1007/978-3-031-99402-9_1

would it then be difficult for a recent computer to crunch through millions of numbers to determine if some of them sum to yet another number?

To see why, we need to be clear about what we are dealing with. Note therefore that the above task is an instance of the **subset sum problem (SSP)** which is generally specified as follows: Given a finite (multi)set of $n$ integers and a target integer, decide if any subset of the integers sums precisely to the target. More formally, we have

$$\text{Given } \big(\mathcal{X}, T\big) \in \mathbb{Z}^n \times \mathbb{Z}, \text{ decide if there exists } \mathcal{X}' \subseteq \mathcal{X} \text{ such that } \sum_{x \in \mathcal{X}'} x = T \ . \tag{1.3}$$

The remainder of this chapter is largely devoted to the SSP and its characteristics. We will see that it is a member of the broader family of *combinatorial optimization problems* which are highly relevant but generally difficult. The latter has at least three unfortunate consequences: First, when running on a digital computer, *exact algorithms* for solving combinatorial problems will generally take a long time or consume a lot of memory. Second, *approximate algorithms* will run faster and consume less memory but cannot guarantee optimal solutions. Third, even very powerful AI models based on modern machine learning will likely be defeated by this family of problems.

To put all this on solid footing, we will recall core concepts from *complexity theory* and *combinatorics* and have a first look at *quadratic unconstrained binary optimization problems (QUBOs)*. All this is intended to help us to appreciate what *quantum computing* has to offer in regards to combinatorial optimization.

Indeed, the later parts of this chapter will briefly touch upon *Hopfield nets* and quantum computing and hint at how the ones relate to the other. Without going into details yet, we will finally claim that quantum computers have an inbuilt edge over digital computers or mainstream learning-based AI and can therefore be expected to quickly and reliably solve problems like the SSP.

Yet, prior to all of this, we first clarify and expand on the terminology and notation accumulated so far.

## 1.2 Sets, Subsets, Multisets, and Vector Representations

Recall that a (mathematical) **set** such as $\mathcal{S} = \{a, c, e, b, d\}$ is an *unordered* collection of distinct elements. To denote the number of elements or *size* of a set $\mathcal{S}$, we write $|\mathcal{S}|$. If $|\mathcal{S}| < \infty$, then $\mathcal{S}$ is *finite* and we can count its elements. In particular, if a finite set $\mathcal{S}$ has $n \in \mathbb{N}$ elements, we have $|\mathcal{S}| = n$.

If $\mathcal{S}'$ and $\mathcal{S}$ are sets and every element of $\mathcal{S}'$ is also an element of $\mathcal{S}$, then $\mathcal{S}'$ is a **subset** of $\mathcal{S}$ and we write $\mathcal{S}' \subseteq \mathcal{S}$. If we additionally have $|\mathcal{S}'| < |\mathcal{S}|$, then $\mathcal{S}'$ is a *proper* subset of $\mathcal{S}$ and we write $\mathcal{S}' \subset \mathcal{S}$.

Conversely, if $\mathcal{S}' \subseteq \mathcal{S}$, then $\mathcal{S}$ is a **superset** of $\mathcal{S}'$, and if $\mathcal{S}' \subset \mathcal{S}$, then $\mathcal{S}$ is a *proper* superset of $\mathcal{S}'$.

A **multiset** such as $\mathcal{M} = \{a, a, c, e, c, c, b, a, d, c, c, b\}$ is a generalized set in which elements can occur repeatedly.

Given this definition, it becomes clear why we called $\mathcal{X}$ in (1.1) a multiset. You may, for instance, verify that its elements 305 and 344 each occur twice. Speaking of repeated elements, here is another task which poses no problem for computers but keeps most human brains surprisingly busy: There are five more elements in $\mathcal{X}$ which each occur twice. Can you spot them?

Multisets come with their own theory and algebra. For instance, we could next introduce *underlying sets* and *multiplicity functions* which would allow us to define the union, intersection, or difference of any two multisets. Fortunately, we do not have to go that far; our upcoming discussion of the crux behind SSPs just needed the meaning of the term "multiset" to be settled.

Having said this, readers with coding experience might now be wondering why to bother with this term at all and not just talk about *arrays* in the following sense:

A **one-dimensional array (1D array)** such as $\mathcal{A} = [a, a, c, e, c, c, b, a, d, c, c, b]$ is a finite, *ordered* collection or *tuple* of possibly reoccurring elements.

Given this definition, it is clear that (multi)sets and 1D arrays are different kinds of mathematical objects. For example, the two sets

$$\mathcal{S}_1 = \big\{v, a, l, u, e\big\} \qquad \text{and} \qquad \mathcal{S}_2 = \big\{a, e, l, u, v\big\} \tag{1.4}$$

are identical. This is because (multi)sets are unordered collections so that the way in which we arrange their elements in a linear piece of mathematical text does not matter. On the other hand, the two arrays

$$\mathcal{A}_1 = \big[v, a, l, u, e\big] \qquad \text{and} \qquad \mathcal{A}_2 = \big[a, e, l, u, v\big] \tag{1.5}$$

are not identical. This is because arrays are ordered collections in which the position at which we list an array element does matter and can make a difference.

Let us dwell on this some more. Our examples thus far deliberately worked with collections of letters such as $v, a, l, u, e$ for which there exists an inherent ordering, namely the alphabetical one $a, e, l, u, v$. However, such inherent orders are not what we mean when we are talking about (un)ordered collections. We are instead referring to *positional orderings* such as in *lists* or *sequences* or *strings* of objects.

For example, for the five letters $a, e, l, u, v$, there are $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ ways to list them. If we see our letters as elements of a set, then all these lists are but different ways of writing down this set and none of them is special. Yet, seen as 1D arrays, all these lists are distinct and one of them is special because our brains cannot but read something into the particular sequence $v, a, l, u, e$.

While this may seem trivial, it is crucial for the following reason. If we can count the elements of a (multi)set, we can also enumerate or index them. This allows us to write finite (multi)sets more abstractly as

$$\mathcal{S} = \big\{s_1, s_2, \ldots, s_n\big\} \, . \tag{1.6}$$

However, the inherently ordered index values $1, 2, \ldots, n$ used in this notation are not supposed to express any ordering of the $s_j \in \mathcal{S}$. Rather, the right hand side of (1.6) is once again just one out of $n!$ possible ways of writing down set $\mathcal{S}$.

Things are of course different when we apply this notation to one-dimensional arrays

$$\mathcal{A} = \bigl[a_1, a_2, \ldots, a_n\bigr] \tag{1.7}$$

because here the numerically ordered indices are deliberately meant to induce an ordering of the array elements.

Dealing with subset sum problems, the distinction between (multi)sets and one-dimensional arrays is irrelevant as we may arrange the $x \in \mathcal{X}$ in whatever way that suits us and still ask if some of them sum to $T$. In other words, we may just as well specify an SSP in terms of an integer array and ask for a sub-array or slice that sums to a given target. Indeed, when implementing subset sum algorithms on a computer, most programming languages will require us to work with arrays anyway.

Speaking of programming languages, we note that there are some where a growable one-dimensional array is called a vector. However, the mathematical use of this term typically refers to a fixed-sized and positionally ordered collection of numbers. We thus have yet another, for now admittedly pragmatic definition.

A **vector** such as $\boldsymbol{x} = [1, 1, 3, 5, 3, 3, 2, 1, 4, 3, 3, 2]$ is an *ordered* collection or tuple of possibly reoccurring numbers.

Working with this concept, we can again restate the subset sum problem. That is, we may also represent the (multi)set $\mathcal{X}$ of an SSP in terms of a vector $\boldsymbol{x} \in \mathbb{Z}^n$ and ask if we can set some of the entries of $\boldsymbol{x}$ to zero such that its remaining non-zero entries sum to $T$.

The idea of representing (multi)sets of numbers in terms of vectors is profound and we will make extensive use of it throughout. But let us expand on what we just did when we wrote $\boldsymbol{x} \in \mathbb{Z}^n$.

Given a set such as, say, the set $\mathbb{Z}$ of integers, we write $\mathbb{Z}^n$ as a shorthand for the following **Cartesian product**

$$\mathbb{Z}^n = \underbrace{\mathbb{Z} \times \mathbb{Z} \times \cdots \times \mathbb{Z}}_{n \text{ times}} \tag{1.8}$$

and we recall that the Cartesian product of $n$ sets produces a set of *ordered* $n$-tuples

$$\mathcal{S}_1 \times \mathcal{S}_2 \times \cdots \times \mathcal{S}_n = \Bigl\{ \bigl[s_1, s_2, \ldots, s_n\bigr] \Bigm| s_j \in \mathcal{S}_j \Bigr\} . \tag{1.9}$$

Note that we once again emphasized the *ordered* nature of the tuples in a Cartesian product. Indeed, "$\times$" is an example of a product or a multiplication operator that is non-commutative. That is, unless $\mathcal{S}_1 = \mathcal{S}_2$, we always have $\mathcal{S}_1 \times \mathcal{S}_2 \neq \mathcal{S}_2 \times \mathcal{S}_1$ for any two non-empty sets $\mathcal{S}_1$ and $\mathcal{S}_2$.

Since some readers may now begin to wonder about our obsession with unordered and ordered collections, we might as well already tease the following: Later on, a certain non-commutative vector product will serve as a central modeling tool. But based on our discussion so far, it is already clear that when we use tuples to *represent* a given set of objects, we face a modeling choice and must decide for a specific order. This decision may or may not impact our subsequent reasoning or algorithm design but usually requires us to be aware of the difference between ordered and unordered collections.

Speaking about modeling, observe that we already used Cartesian products in our formal definition of the subset sum problem in (1.3) where we wrote $(\mathcal{X}, T) \in \mathbb{Z}^n \times \mathbb{Z}$. Strictly speaking, we therefore defined $\mathcal{X} \in \mathbb{Z}^n$ as a vector of integers instead of as a (multi)set of integers. As we just saw, this makes perfect sense from a practical point of view. But what about the formal point of view? Would it not be better to refer to the subset sum problem as the multi-subset sum problem or the subvector sum problem? It arguably would, but what we are dealing with is a case of established somewhat ambiguous terminology which does not bother experts but may confuse novices. For now, we therefore conclude this section of basic definitions with a disclaimer for the remainder of this book.

**Terminology**

Sets and multisets are closely related but distinct mathematical objects. However, from now on, **we will largely ignore any distinction between sets and multisets**. For ease of discussion, **we will talk about sets even if we refer to collections with repeated elements**. For readers with a background in machine learning, this should be acceptable because they are used to the notion of *data sets* which may contain repeated data points.

Similarly, sets and arrays are related but distinct mathematical objects. Henceforth, however, **we will largely ignore any distinction between sets and arrays** because we can always *represent* finite sets as arrays and any array is but an ordered set. For readers with programming experience, this should be acceptable because most programming language require us to think in terms of arrays anyway. In all our later code examples, we will indeed implement sets in terms of arrays.

## 1.3 Decision- and Combinatorial Optimization Problems

There actually are two points of view on the subset sum problem, namely the *decision problem* point of view and the *combinatorial optimization problem* point of view.

Decision- and combinatorial optimization problems are very general concepts of wide applicability. Since different manifestations or specific instances of such problems will feature prominently later on, we next look at what they are, why they are relevant, and why they are generally difficult to deal with.

A **decision problem** is a kind of computational problem that asks for the answer to a `yes` / `no` question.

For instance, an SSP is a decision problem because we can set it as follows: Given a pair $(\mathcal{X}, T)$, *decide* if there exists any $\mathcal{X}' \subseteq \mathcal{X}$ such that the $x \in \mathcal{X}'$ sum to $T$.

A **combinatorial optimization problem** or a **combinatorial search problem** is a kind of computational problem that asks for an optimal element in a *finite* set. Since the set in which to search for a solution is finite, the set of possible solutions is necessarily finite, too.

For instance, an SSP is a combinatorial optimization problem because we can set it as follows: Given a pair $(\mathcal{X}, T)$, consider the set of all subsets of $\mathcal{X}$ and *find* a subset $\mathcal{X}'$ such that the $x \in \mathcal{X}'$ sum to $T$. If no such subset can be found, report a failure.

Much more prosaically and still with respect to SSPs, we may therefore say that the difference between a decision problem and a combinatorial optimization problem boils down to the difference between the question "is there a solution to an SSP?" and the instruction "find a solution to an SSP!".

At first sight, decision- and combinatorial optimization problems thus only seem to be but two sides of the same coin. After all, if we can find a solution to a problem, we can immediately decide its solvability.

Observe, however, that we carefully restricted our definition of combinatorial optimization problems to search problems over finite sets. Decision problems, on the other hand, are not necessarily restricted thusly. On the contrary, they frequently pertain to infinite sets.

Having said this, all our practical examples in this book will be concerned with finite sets for which decision- and combinatorial optimization problems are on a more equal footing. They are still not equivalent because the latter allow us to address more nuanced tasks than making simple yes/no decisions.

For instance, when dealing with an SSP instance $(\mathcal{X}, T)$, we may also consider more specific combinatorial problems such as the **minimum size SSP** which asks for a subset $\mathcal{X}' \subseteq \mathcal{X}$ of *minimum size* such that the $x \in \mathcal{X}'$ sum to $T$.

Hence, while much of the computer science literature presents the subset sum problem as a decision problem (we did it ourselves on "The Subset Sum Problem" and "Sets, Subsets, Multisets, and Vector Representations"), we actually prefer the point of view of combinatorial optimization and will largely stick with it in all that follows.

### *1.3.1 Applicability*

But are combinatorial optimization problems practically relevant? Yes, they are! In fact, use cases are so omnipresent and diverse that an exhaustive survey would far exceed the scope of any introductory discussion. This will become apparent later when we see many more examples. For now, we continue focusing on the subset sum problem, its role, and characteristics.

Various kinds of nuanced SSPs arise in areas as diverse as cryptography, resource allocation, logistics, or finance where their solutions enable secure communication, efficient work flows, cost reduction, or risk mitigation. For example, if a company needs to store a certain amount of goods in storage facilities of certain capacities but wants to use as few facilities as possible, they are facing a minimum size SSP. The opposite, namely a maximum size SSP, occurs, when, say, stock brokers want to hedge their bets by distributing a certain investment sum over as many given assets as possible.

But even the plain vanilla version of the problem has practical merit. For example and perhaps unsurprisingly, accountants and controllers are well familiar with what we defined as the basic SSP.

Indeed, the authors of this book frequently encounter it in the context of a research cooperation with an auditing firm whose analysts assess financial statements of publicly traded companies. This involves verifying the bottom line in tables of financial figures and often numbers don't add up. Whenever they find such inconsistencies, analysts must figure out what may have been summed to arrive at the bottom line and thus face SSPs. Since modern reporting standards cause tables to grow rapidly, these SSPs become steadily more difficult.

This then leads to the crucial questions of how difficult it really is to solve an SSP and what kind of algorithms there are if any that allow for efficient solutions.

To prepare ourselves for the answers, it may be instructive to briefly look at what modern machine learning and generative artificial intelligence (AI) can contribute in this regard.

Indeed, one of the goals of our cooperation with the auditing firm is to develop reliable solutions for AI assisted consistency checking which harness the capabilities of recent large language models [1]. Now, as of this writing, it is safe to assume that most of us will have had some interaction with chatbots such as ChatGPT or Gemini. Many will thus know that their underlying language models are capable of human-level text analysis and synthesis. These foundational skills can be fine-tuned towards specific domains such as accounting to obtain expert models which then effectively identify and point out inconsistencies in financial documents [2–5].

But what about reliably resolving these inconsistencies? After all, when prompted appropriately, recent language models even exhibit (mathematical) problem solving skills [6–8]. Would this not suggest that data-driven machine learning should be able to solve tedious but easily explained problems such as the SSP?

Alas, being combinatorial, the nature of an SSP is generally such that even the most recent, most comprehensively trained, and thus most capable generative AI models cannot guarantee reliable solutions.

### *1.3.2 Learnability*

To see why combinatorial optimization problems largely defy data-driven learning, we recall that they ask for an *optimal element* in a *finite set*. Their search spaces and sets of feasible solutions are therefore necessarily discrete and, even worse, often only loosely structured. In other words, search spaces and solutions of combinatorial problems do not vary continuously and rarely show regularities. Especially the latter causes difficulties for mainstream machine learning models.

Indeed, current machine learning is almost synonymous with deep learning or the training of deep neural networks [9, 10]. From an abstract point of view, these are just parameterized functions which map certain inputs to certain outputs but the key to their success is that they are universal *approximators* [11]. That is, if a neural network has sufficiently many adjustable parameters, it can learn to *approximate* any continuous mapping from one continuous space to another. This involves training with sufficiently many examples of inputs and outputs which are processed by algorithms like error-backpropagation [12] to iteratively adjust the network's parameters such that it eventually shows the desired I/O behavior.

There thus seem to be three difficulties when applying this framework to learn to solve combinatorial optimization problems. These seem to pertain to the phrases "sufficiently many" and "continuous space" and to the fact that combinatorial problems such as the SSP can involve inputs of arbitrary sizes. Yet, developers of large language models face all these issues, too, and succsessfully address them by working with transformer architectures [13, 14].

Recent transformers come with hundreds of billions of parameters and are trained on staggering amounts of data. This could be replicated when trying to train SSP solvers; large model sizes have become manageable and large amounts of corresponding training data are easy to come by (see Exercise 1.6). Transformers further operate on discrete data, namely sequences of words, which they map to continuous spaces before performing further computations. This, too, could be replicated for the sequences of numbers that occur in SSPs. Finally, transformers are *recurrent* neural networks and can therefore deal with arbitrary input sizes. So why not train an SSP transformer?

This idea is of course conceivable, but there are two more fundamental problems when trying to tackle combinatorial optimization through machine learning models.

First of all, "approximate" solutions may not be good enough. For instance, while hallucinated text produced by a transformer-based language model is largely but a nuisance, hallucinated solutions to a financial SSP are basically useless. Addressing this precision issue would require model sizes which by today's standards are neither economic nor sustainable.

Second of all, and more importantly, we need to ask: What kind of general patterns could be learned from looking at billions of exemplary SSPs? After all, machine learning algorithms adjust model parameters according to statistical objectives such as minimizing the average difference between generated and desired outputs. This works well in settings where there are statistical regularities in the training data. Of course, these regularities may be excruciatingly complex. For instance, in natural language modeling, we encounter an unfathomable amount of possible sequences of words, however, these sequences are not arbitrary but composed according to intricate linguistic or cultural norms. Inputs to combinatorial optimization problems, on the other hand, can, for all intents and purposes, be truly arbitrary.

Of course, there are attempts on transformers for combinatorial optimization but, as of this writing, their performances have not been convincing [15]. Given the current state of technology, it therefore seems unreasonable to try to train reliable universal SSP transformers.

And yet machine learning may still apply. To make a long story short, what could be learned from looking at billions of combinatorial optimization problems and their solutions is not so much an exact solution to any given problem but rather a strategy for how to find such a solution. This puts us squarely into the realm of reinforcement learning which aims at learning what to do when in order to achieve a goal [16]. Indeed, experts in the field acknowledge reinforcement learning as the only reasonable approach towards learning combinatorial optimization [17] and, since it even enabled computers to learn to play GO [18], it would seem that especially deep reinforcement learning should be able to tackle arbitrary SSPs. However, as of this writing, we are not aware of any large scale attempts in this direction.

### *1.3.3 Difficulty*

Given our review of the prospects of data-driven AI for combinatorial optimization, it is not surprising that generally applicable state-of-the-art solvers for problems such as the SSP still rest on other paradigms.

First of all, there are traditional computer scientific appraoches based on carefully designed, problem specific data structures and algorithms [19, 20]. Some of these involve dynamic programming techniques [21] which, as a side node, also occur in reinforcement learning.

Second of all, there are more general and thus often less efficient approaches rooted in knowledge-driven AI [22]. These mainly involve tree-search heuristics or randomized exploration strategies but also include is a wide spectrum of techniques inspired by evolutionary or eusocial optimization processes [23].

Third of all, there are the kind of methods we deal with in this book [24] and study in much detail later on. For now, we note that they can be seen as hybrid approaches wich incorporate some level of problem specific knowledge into problem agnostic optimization techniques [25].

But does it have to be so complicated? Couldn't we just brute force combinatorial optimization problems? That is, couldn't we just exhaustively examine all possible objects in the finite set under consideration to find an optimal one? How much time could this possibly take?

Well, there exist instances of combinatorial problems for which exhaustive search is feasible. But beware, these are typically so simple that most people would not even count them as combinatorial problems.

Consider, for instance, the task of finding the largest number in a set of $n$ numbers. According to our definition, this constitutes a combinatorial optimization problem as it asks for an optimal element (the largest number) in a finite set (of $n$ numbers). To solve it, we may simply iterate over the $n$ given numbers while keeping track of the largest one seen so far.

In other words, we are dealing with a problem of **input size** $n$ and can solve it by executing a number of *operations* or *computational steps* proportional to $n$. We therefore say that the **time complexity** of the algorithm we just sketched is of the order of $n$. Using ***O*** **notation**, we also say that the algorithm takes $O(n)$ time or, even shorter, runs in $O(n)$. A running time like this is *linear in the problem size*.

There even are edge cases of the SSP which we can solve in linearly many steps. For instance, if all the $x \in \mathcal{X}$ are even but $T$ is odd, we only need $O(n)$ computational steps to find that no solution exists.

Another, slightly more cumbersome edge case occurs when the numbers in $\mathcal{X}$ are *super-increasing*. This is to say that every $x \in \mathcal{X}$ is greater than the sum of all the $x' \in \mathcal{X}$ lesser than it. Here is an example: $\mathcal{X} = \{3, 1, 9, 36, 18\}$. To see if numbers like these sum to a certain target, we may first sort them increasingly. This will require $O(n \log n)$ computations but subsequently allows us to run an $O(n)$ algorithm to find a solution. Since the first step is more costly than the second one, the overall time complexity in this situation is therefore *log-linear in the problem size*.

An example of a slightly more demanding combinatorial problem is the task of finding the shortest path between two distinct nodes in a network of $n$ nodes. Dijkstra's algorithm accomplishes this in only $O(n^2)$ computational steps and thus takes time *polynomial in the problem size*.

Apparently, we could have been more specific and said that Dijkstra's algorithm has a running time quadratic in $n$. However, we chose not to which we could because $n^2$ is but a special case of $n^k$ which denotes a polynomial of order $k$. Simultaneously, we could have more generally said that Dijkstra's algorithm runs in $O(\text{poly}(n))$ and thus have used a convenient shorthand common in situations where people choose not to specify the polynomial order $k$.

As in intermediate summary, we note that all the above problems can be solved by algorithms whose running time is at most polynomial in the respective problem size. According to the *Cobham-Edmonds thesis* they are therefore *tractable* in that they can be *feasibly computed* on any computing device. Sure, this may take more time on an off-the-shelf laptop than on the latest supercomputer but, in the end and in either case, we would not have to wait too long until a solution has been found or can be ruled out.

Returning to our question, if we could not just brute force any given combinatorial optimization problem, we note that all the above examples considered problems with very benign properties. Their search spaces were either trivially small (of a size linear in $n$) or inherently structured in a manner that allows for efficient algorithms. For the SSP edge cases, we discussed this special structure. Regarding the path finding example, we observe without going into detail that the connectivity structure of a network can efficiently guide the search for shortest paths. Dijkstra's algorithm exploits this by dynamically growing and shrinking the set of potential solutions which it searches exhaustively.

Alas, the vast majority of combinatorial optimization problems are not so benign. Instead, most problems in this class are known to be *NP-complete* or even *NP-hard*. This makes it generally impossible to exhaustively comb through their search spaces in order to discover a solution and we briefly recall why this is.

### *1.3.4 NP-Completeness and NP-Hardness*

This section delves into *complexity theory* which is a deep and venerable topic. We will not address all its intricacies and subtleties but only roughly recall key concepts and keep our discussion almost informal.

To begin with, we need to clarify the notion of a *computational problem* which occurred in our definitions of decision- and combinatorial optimization problems.

A **computational problem** is a problem for which there exists an algorithm which we can run on a computer to solve it. As this is a seemingly all-encompassing definition, things will get more interesting once we look for different characteristics of different computational problems and their algorithms.

Computational complexity theory is concerned with the resources (running time and memory space) required for computational problem solving and classifies problems and algorithms accordingly. This is mainly an endeavor of theoretical computer science where *resources* are a mathematical concept rather than a technological one. Complexity theory thus abstracts away computer hardware and considers mathematical models of computing machinery instead. One such model is a *deterministic Turing machine* (a theoretical device whose mode of operation we gloss over) which, according to the *Church-Turing thesis*, can compute everything that is computable. In this context, it is noteworthy that Church and Turing also told us that there exist problems such as the *halting problem* which are definable yet not computable.

Within the class of computational problems, there are two important subclasses or computational *complexity classes*. These are called **P** and **NP** which are somewhat unfortunate acronyms for *deterministic polynomial time* and *non-deterministic polynomial time*. Indeed, both classes are defined as follows: Problems in class P can be solved in polynomial time on a deterministic Turing machine. Problems in class NP can be solved in polynomial time on a non-deterministic Turing machine.

The latter is another theoretical computer, a thought experiment really, where an exponential number of computational branches can exist in parallel. As this kind of abstraction easily obfuscates the nature of problems in NP, we note the following.

There actually are two aspects to every computational problem, namely *finding* a solution and *verifying* a purported solution. After all, anybody can claim to have found a solution, as long it is not correct, it is not a solution.

With this, we may say more informally that problems in P can be solved and verified in polynomial time. Problems in NP, on the other hand, can be solved in exponential time *or less* and verified in polynomial time.

Note that the term "or less" expresses that P is believed to be a proper subset of NP. Whether or not this really is the case is still unknown. In fact, the question of whether $P = NP$ is a millennium prize problem whose correct answer is worth a million Dollars. However, as it stands today, it seems that there are problems in NP which take much more time to solve than all the problems in P.

Indeed, there exist problems where finding a solution seems to absolutely require exponential time (to our current knowledge) but verifying it only requires polynomial time. These are the hardest problem in NP and said to be **NP-complete**.

Moreover, a problem is **NP-hard** if it is as least as difficult as any NP-complete problem. Such problems can be outside of NP and we informally say their solution requires exponential time and their verification requires exponential time *or less*.

Given the notions of the class NP and of NP-hard problems, we may also say that a problem is NP-complete if it is in NP and also NP-hard. *All NP-complete problems are therefore NP-hard but not all NP-hard problems are NP-complete*. This can be confusing since many authors tacitly assume NP-hard problems always to be harder than NP-complete problems and we, too, will follow this convention.

The *subset sum problem* can be proven to be NP-complete. When dealing with general settings rather than with edge cases, exact algorithms for finding a solution require time exponential in the problem size but the correctness of a solution can be verified in only polynomial time. Optimists take this to mean that, given enough time, a brute force algorithm could check the sums of all possible subsets to find a solution if one exists. Alas, *enough time* can be a very optimistic assumption indeed (see Exercise 1.2).

The *minimum size subset sum problem* is generally NP-hard. It takes exponential time to find a solution and at least as much time to verify that a found solution is indeed of minimum size. Here, even optimists will acknowledge defeat and admit that heuristic algorithms or approximate solutions are generally called for.

Speaking of heuristic algorithms and approximate solutions, here is another easily understood problem that proves surprisingly difficult. Readers with a background in machine learning will know about *k-means clustering*. Those who don't need not worry as we will (re)introduce it in later chapters.

A lesser known fact is that $k$-means clustering is NP-hard [26]. To some, this may be surprising because Lloyd's famous algorithm [27] (a.k. *the* $k$-means algorithm) *often* solves it quickly and reliably. However, note our emphasis on "often" because the algorithm cannot be guaranteed to yield optimal solutions. In fact, to this day, no such algorithm is known for $k$-means clustering. Rather, Lloyd's algorithm is an example of a **randomized algorithm** and as such involves elements of chance in its mode of operation.

In particular, Lloyd's algorithm starts with a random initial guess of a solution and then refines it iteratively. Ideas like this provide reasonable strategies for many NP-complete or NP-hard problems and we will see more examples later.

Finally, we should mention that the *subset sum problem* is a peculiar instance of an NP-complete problem because it turns out to be only *weakly NP-complete*.

A **weakly NP-complete problem** is an NP-complete problem for which there exist algorithms that are polynomial in *problem size* and *data magnitude* where we assume that both are positive integers.

Accordingly, an NP-complete problem that is not weakly NP-complete is strongly NP-complete and similar notions apply to NP-hard problems. But let us focus on what it means for an algorithm to be polynomial in problem size *and* data magnitude.

Given an SSP $(\mathcal{X}, T)$, its problem size is $n = |\mathcal{X}|$. The data magnitude in this setting is simply the value of $T$. Next, note that there exist exact, dynamic programming-based SSP solvers which run in $O(nT)$. This is good news for practitioners who are dealing with SSPs of up to moderate sizes and magnitudes. Alas, for larger problems, these algorithms lose their edge because $T$ tends to grow exponentially in $n$ so that $nT$ is technically an exponential rate of growth.

Algorithms that are polynomial in problem size and data magnitude are therefore called **pseudo-polynomial algorithms** and we return to them below. But first we need to address the open questions of why it generally takes exponential time to solve an SSP or what it is that makes the search space of an SSP so difficult to deal with.

## 1.4 Power Sets and Indicator Variables

A subset sum problem $(\mathcal{X}, T)$ asks for a subset $\mathcal{X}' \subseteq \mathcal{X}$ whose elements sum to $T$. Its search space therefore is the set of all subsets of $\mathcal{X}$ which is known as the *power set* of $\mathcal{X}$ and usually denoted by $2^{\mathcal{X}}$.

Power sets like this play a fundamental role in most combinatorial optimization problems and will occur all throughout this book. In fact, the concept is so important that we need to get more technical to prepare ourselves for things to come.

**Definition 1.1** Let $\mathcal{S}$ be a finite set. Its **power set** $2^{\mathcal{S}}$ is the set of all subsets of $\mathcal{S}$, namely

$$2^{\mathcal{S}} = \left\{\mathcal{S}' \mid \mathcal{S}' \subseteq \mathcal{S}\right\} .$$

Looking at this definition, we perhaps better reflect on the puzzling notation "$2^{\mathcal{S}}$". Note therefore that the *empty set* $\emptyset$ is, by definition, a subset of any set and that any set is a subset of itself. In other words, we always have $\forall\, \mathcal{S} : \emptyset \subseteq \mathcal{S}$ and $\forall\, \mathcal{S} : \mathcal{S} \subseteq \mathcal{S}$.

But this then means that the sets $\emptyset$ and $\mathcal{S}$ will always be elements of the set $2^{\mathcal{S}}$. For example, the power set of

$$\mathcal{S} = \left\{s_1, s_2, s_3\right\} \tag{1.10}$$

is given by

$$2^{\mathcal{S}} = \Big\{\emptyset, \left\{s_1\right\}, \left\{s_2\right\}, \left\{s_3\right\}, \left\{s_1, s_2\right\}, \left\{s_1, s_3\right\}, \left\{s_2, s_3\right\}, \left\{s_1, s_2, s_3\right\}\Big\} . \tag{1.11}$$

Scrutinizing this example, we realize that the size of $2^{\mathcal{S}}$ is exponential in the size of $\mathcal{S}$. More specifically, we find

$$|\mathcal{S}| = 3 \tag{1.12}$$

$$\left|2^{\mathcal{S}}\right| = 2^{|\mathcal{S}|} = 2^3 = 8 \tag{1.13}$$

and it is this exponential connection between the sizes of $\mathcal{S}$ and $2^{\mathcal{S}}$ which motivates the notation "$2^{\mathcal{S}}$".

Indeed, our example generalizes as follows: If $\mathcal{S}$ is a finite set, the size of its power set is $|2^{\mathcal{S}}| = 2^{|\mathcal{S}|}$, or, equivalently, if $|\mathcal{S}| = n$, then $|2^{\mathcal{S}}| = 2^n$. Since this result is fundamental to our upcoming study, we shall state it as a theorem and give a proof.

**Theorem 1.1** *Let $\mathcal{S}$ be a finite set of size $|\mathcal{S}| = n$. Then the size of its power set $2^{\mathcal{S}}$ amounts to $|2^{\mathcal{S}}| = 2^{|\mathcal{S}|} = 2^n$.*

***Proof*** We base our proof on combinatorics and observe that to choose a subset $\mathcal{S}'$ of size $k$ from a set $\mathcal{S}$ of size $n$ is to choose $k$ out of $n$ objects. Since the number of choices we have for $0 \leq k \leq n$ is given by the **binomial coefficient**

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}\,, \tag{1.14}$$

the size of $2^{\mathcal{S}}$ or total number of distinct subsets of $\mathcal{S}$ can therefore be computed as

$$|2^{\mathcal{S}}| = \sum_{k=0}^{n} \binom{n}{k}\,. \tag{1.15}$$

Next, we resort to the binomial theorem from elementary algebra which provides the following expansion of polynomials of the form $(x + y)^n$

$$(x + y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}\,. \tag{1.16}$$

Given the binomial theorem, we can consider the following very specific special case

$$2^n = (1+1)^n = \sum_{k=0}^{n} \binom{n}{k} \cdot 1^k \cdot 1^{n-k} = \sum_{k=0}^{n} \binom{n}{k} \cdot 1 \cdot 1 = \sum_{k=0}^{n} \binom{n}{k}\,. \tag{1.17}$$

Finally, since the right hand sides of equations (1.15) and (1.17) are the same, their left hand sides must be, too. □

Armed with this result, we can finally appreciate why it is generally hopeless to attempt to brute force combinatorial optimization problems such as the SSP and why an exhaustive search over all possible solutions is generally infeasible. In short, most combinatorial problems suffer from **combinatorial explosion** which we take to mean that the sizes of their search spaces grow *at least* exponentially in the problem size and thus quickly become intractable even for moderate $n$.

For instance, for an SSP of problem size $n$, the worst case running time of an exhaustive search would be $O(n2^n)$ because we would have to examine $2^n$ subsets and perform up to $n$ additions for each of them. For our introductory example with a mere problem size of $n = 100$, this would already translate to up to about $10^{32}$ operations and thus more than max out even the exaFLOP capabilities of current supercomputers which perform $O(10^{18})$ floating point operations per second.

Of course, worst case running time is not average case running time. Brute forcing could be lucky and find a solution after checking only five subsets or fifty subsets or five million subsets but, all in all, there has to be a better way.

We already alluded to the feasibility of the methods studied in this book. In further preparation for this study, we next look at how to characterize a subset $\mathcal{S}'$ of $\mathcal{S}$ other than by listing all its elements.

Given two *finite* sets $\mathcal{S}' \subseteq \mathcal{S}$, an **indicator variable** $z_j$ indicates if an element $s_j$ of the superset $\mathcal{S}$ is also an element of its subset $\mathcal{S}'$. It is most commonly defined as

$$z_j = \begin{cases} 1 & \text{if } s_j \in \mathcal{S}' \\ 0 & \text{otherwise} \end{cases} \tag{1.18}$$

which is to say that an indicator variable can only assume two possible values. If these happen to be 0 and 1 just as above, we henceforth write $z_j \in \{0, 1\}$.

So far so good, but what are the benefits of working with indicator variables? To appreciate these, we restrict our attention to finite sets $\mathcal{X}$ of $n$ real numbers $x_j \in \mathbb{R}$.

Dealing with sets like these, we can, for instance, work with indicator variables to write sums over the elements $x_j$ of a subset $\mathcal{X}' \subseteq \mathcal{X}$ like this

$$\sum_{x_j \in \mathcal{X}'} x_j = \sum_{j=1}^{n} z_j \cdot x_j \ . \tag{1.19}$$

While such expanded sums look cumbersome and involve more additions (of zero-valued terms) than necessary, they often simplify mathematical analysis and reasoning. To substantiate this claim, we recall that we may *represent* sets $\mathcal{X} \subset \mathbb{R}$ of size $n$ as vectors $\boldsymbol{x} \in \mathbb{R}^n$, namely

$$\mathcal{X} = \big\{x_1, x_2, \ldots, x_n\big\} \quad \Leftrightarrow \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \big[x_1, x_2, \ldots, x_n\big]^{\intercal} \tag{1.20}$$

where $\boldsymbol{x}^{\intercal}$ denotes the *transpose* of $\boldsymbol{x}$. Dealing with such a vector representation of a set of numbers, we may introduce an **indicator vector**

$$\boldsymbol{z} = \big[z_1, z_2, \ldots, z_n\big]^{\intercal} \in \big\{0, 1\big\}^n \tag{1.21}$$

where

$$z_j = \begin{cases} 1 & \text{if } x_j \in \mathcal{X}' \\ 0 & \text{otherwise} \end{cases} \tag{1.22}$$

to also obtain a vector representation of a subset of numbers.

For example, given a set $\mathcal{X} = \{3, 5, 1, 2, 4\}$ and one of its subsets $\mathcal{X}' = \{5, 2\}$, we may represent the former in terms of a vector $\boldsymbol{x} = [3, 5, 1, 2, 4]^\intercal$ and the latter in terms of a vector $\boldsymbol{z} = [0, 1, 0, 1, 0]^\intercal$.

Note that the representation of $\mathcal{X}$ in terms of $\boldsymbol{x}$ is *direct* whereas the representation of $\mathcal{X}'$ in terms of $\boldsymbol{z}$ is *indirect*. While vector $\boldsymbol{x}$ contains the elements of $\mathcal{X}$, vector $\boldsymbol{z}$ contains indicators specifying which entries of $\boldsymbol{x}$ belong to $\mathcal{X}'$. Readers who have worked with numerical computing packages before may recognize this as the ideas behind Boolean masks of arrays (see Exercise 1.9).

Given all these preparations, we can now understand the subset sum in (1.19) as an *inner product* of two vectors. This is because

$$\sum_{x_j \in \mathcal{X}'} x_j = \sum_{j=1}^{n} z_j x_j = \boldsymbol{z}^\intercal \boldsymbol{x} \ . \tag{1.23}$$

Indeed, indicator vectors allow us to frame many set theoretic operations in terms of linear algebraic computations and we will make frequent use of this capacity.

Finally, since each entry $z_j$ of a binary indicator vector $\boldsymbol{z}$ can only assume one of two values, namely either 0 or 1, combinatorics tells us that the number of choices of possible indicator vectors over a set $\mathcal{X}$ of size $n$ is given by

$$\underbrace{2 \cdot 2 \cdot \ldots \cdot 2}_{n \text{ times}} = 2^n$$

so that the number of indicator vectors over $\mathcal{X}$ equals the size of the power set $2^{\mathcal{X}}$.

## 1.5 Quadratic Unconstrained Binary Optimization Problems

The above provides us with yet another way of formalizing an SSP $(\mathcal{X}, T)$. Indeed, we may now gather the $n$ integers $x_j \in \mathcal{X}$ in a vector $\boldsymbol{x} \in \mathbb{Z}^n$ and then search for a vector $\boldsymbol{z} \in \{0, 1\}^n$ such that

$$\boldsymbol{z}^\intercal \boldsymbol{x} = T \ . \tag{1.24}$$

This reformulation in terms of vectors changes nothing about the NP-completeness of the SSP. The underlying search space is still discrete and exponentially large. It now just consists of $2^n$ binary vectors $z$ instead of $2^n$ subsets $\mathcal{X}'$. Yet, our reformulation hints at a fundamentally different view on SSPs.

To see what this could possibly mean, we observe the following seemingly trivial but highly consequential equivalencies

$$z^\intercal \boldsymbol{x} = T \quad \Leftrightarrow \quad z^\intercal \boldsymbol{x} - T = 0 \quad \Leftrightarrow \quad \bigl(z^\intercal \boldsymbol{x} - T\bigr)^2 = 0 \,. \tag{1.25}$$

The importance of these equivalencies stems from the fact that they allow us to re-state the SSP as an **optimization problem** of a rather simple form, namely

$$z_* = \underset{z \in \{0,1\}^n}{\operatorname{argmin}} \bigl(z^\intercal \boldsymbol{x} - T\bigr)^2 \,. \tag{1.26}$$

Looking at this problem, we note the following components and characteristics: Vector $z$ is called the **decision variable** whose **optimal value** $z_*$ we wish to find via some kind of optimization procedure. Optimality is to be understood with respect to the **objective function** $f(z) = (z^\intercal \boldsymbol{x} - T)^2$. Since this objective is quadratic in the decision variable, our problem is a **quadratic optimization problem**. Since it involves the argmin operator, we are even more specifically dealing with a **quadratic minimization problem**. We therefore say that $z_*$ is optimal if it globally minimizes the objective such that $f(z_*) \leq f(z)$ for all $z$. Finally, the set $\{0, 1\}^n$ which occurs in the subscript of the argmin operator is the **feasible set** for our problem; valid or acceptable solutions are restricted to this set.

We also note that the objective function in (1.26) is lower bounded by zero in that $0 \leq (z^\intercal \boldsymbol{x} - T)^2$ for all $z$. Hence, if the respective SSP can be solved, that is, if $\mathcal{X}$ contains any subset $\mathcal{X}'$ whose elements sum to $T$, there has to exist a global minimizer $z_*$ with $(z_*^\intercal \boldsymbol{x} - T)^2 = 0$ so that $z_*$ *represents* a solution to the SSP.

For readers with a background in machine learning, all this will be familiar. We could make it likely even more familiar by talking about *model parameters* and *loss functions* instead of *decision variables* and *objective functions*. After all, machine learning is the science of optimizing model parameters which is often achieved by minimizing appropriate loss functions.

What is peculiar about the problem in (1.26) is that it constitutes an instance of a **quadratic unconstrained binary optimization problem** or **QUBO** for short.

As its decision variable is a binary vector, it is a binary optimization problem; as there are no additional constraints on the decision variable other than being binary, it is an unconstrained binary optimization problem; and, as its objective function is quadratic, it is a quadratic unconstrained binary optimization problem.

QUBOs and techniques for their solution are the focal point of this book and we will study their characteristics and applications in more detail later on.

For now, we note that QUBOs are generally NP-hard. This can be deduced from the fact that their decision variables are vectors of integers 0 or 1 so that QUBOs are specific instances of what are called *integer programming problems*. These, in turn, are known to be NP-hard in general.

More specifically, say, with respect to the problem in (1.26), we observe that it may be difficult to find its *global minimizer* $z_*$ because, and this is crucial, the search space or feasible set of the problem is discrete. Standard techniques such as (naïve) gradient descent which we would use to solve the much more widespread continuous versions of such problems will therefore not apply. Moreover, in situations where an SSP cannot be solved exactly, for instance if all the $x_j$ are even but $T$ is odd, it will be just as difficult to verify that a found $z_*$ gives the best approximate solution possible.

So why then would we cast an SSP as a QUBO? Because there exists a versatile natural computing machinery that allows for efficient approximate QUBO solving, namely *Hopfield nets*. Moreover, this machinery operates on principles that translate to principles used in *quantum computing*.

*Hopfield nets* are a venerable class of recurrent neural networks and *quantum computing* is, well, quantum computing. Right now, at this early stage of our study, it would not make sense to introduce either one more carefully. However, we can at least tease why they merit attention.

## 1.6 What About Hopfield Nets and Quantum Computing?

To better motivate our interest in Hopfield nets and quantum computing, we first revisit some of the techniques for SSP solving we mentioned above.

Our following brief review of classical milestone algorithms summarizes several recent accounts [28–31]. It focuses on algorithmic complexity (running time and memory space) and glosses over algorithmic details as they are of minor interest for our overall discussion. Readers interested in those are very much encouraged to consult the cited contributions.

### *1.6.1 Classical SSP Solvers*

Given an SSP $(\mathcal{X}, T)$ of size $|\mathcal{X}| = n$, running times of naïve exhaustive searches for solutions are $O(n2^n)$. Less naïve but still exhaustive searches can be realized via depth-first tree traversal of a binary tree whose levels correspond to the $x_j \in \mathcal{X}$. Left branches represent excluding $x_j$ from the sought after solution $\mathcal{X}'$ and right branches represent including $x_j$ into $\mathcal{X}'$. Such a tree is $O(n)$ deep and $O(2^n)$ wide. Since depth-first searches build trees iteratively and discard invalid nodes and branches, memory requirements of this approach are only $O(n)$, its running time, however, is still $O(n2^n)$ in general. Pruning heuristics known from classical AI can further accelerate such searches but do not fundamentally reduce their running time complexity.

More efficient divide-and-conquer algorithms employ more sophisticated data structures. They date back to the 1970s [32] when it was shown that list-merging can find SSP solutions in $O(n/2\, 2^{n/2})$ time but, alas, comes with exponential memory costs of $O(2^{n/2})$. Introducing heaps can improve this to $O(n/4\, 2^{n/2})$ time and $O(2^{n/4})$ memory [33]. Further notable progress just happened in the 2010s [34] when binary representations were used in a probabilistic algorithm that runs in $O(2^{0.337n})$ time and thus broke the long-standing $2^{n/2}$ barrier. It further requires only $O(2^{0.256n})$ space but can only *decide* solvable SSPs and does nothing for unsolvable ones. Working with ternary representations, this was improved to $O(2^{0.291n})$ time [35] and, to the best of our knowledge, the current record works with quniary representations to achieve $O(2^{0.283n})$ time [30]. Note that the exact algorithms [32, 33] in this category of approaches work much faster than tree-searches [28] but that their exponential memory costs limit them to small problem sizes of up to about $n \approx 100$. The probabilistic algorithms in this category are not limited as severely but only address decision problems.

Pseudo-polynomial algorithms based on dynamic programming were introduced by none other than the godfather of the field, Richard Bellman [36]. The current best improvement over Bellman's original $O(nT)$ time complexity is now at $O(\sqrt{n}T)$ for exact algorithms [29, 31] and it is interesting to note that these methods involve Fourier transforms. Furthermore, under mild conditions on the inputs in $\mathcal{X}$, such dynamic programming techniques allow for parallelization. When implemented on special purpose hardware (GPUs), these algorithms can cope with large problem sizes and moderate data magnitudes or with moderate problem sizes and large data magnitudes [37].

But why would we list so many big $O$ results? And what about QUBOs, Hopfield nets, and quantum computing?

### 1.6.2 Hopfield Nets and Quantum QUBO Solvers

Curiously, none of the cited contributions casts the SSP as a QUBO and attempts to solve it from this direction. However, we shall see that this is easily done once we know much more about how to use Hopfield nets as randomized QUBO solvers.

Indeed, running a Hopfield net on an SSP takes only $O(n^3)$ time, i.e. time polynomial in the problem size, and empirical results suggest that the *probability* for finding an exact solution is $O(1/\sqrt{n})$. In other words, running the Hopfield net $\sqrt{n}$ times will *likely* find a solution if one exists and otherwise at least yield useful approximations. This turns Hopfield nets into competitive SSP solvers for real world settings [24] and it is curious to see that they are not widely recognized as such.

However, although we did discuss all this at length, our focus in this book is not on SSP solving! Instead, we merely considered the SSP as an example of a difficult combinatorial optimization problem which, when viewed from the unconventional perspective of QUBOS, can be tackled using Hopfield nets.

Moreover, our main interest in this book is in Hopfield nets as a stepping stone towards understanding the mechanisms behind quantum computing. Indeed, if a problem can be cast as a QUBO, it can be solved by running a Hopfield net. And if a problem can be solved by running a Hopfield net, we can rest assured that it can be solved on a quantum computer. In case of an *adiabatic quantum computer*, the transition from Hopfield nets to quantum algorithms is almost immediate. For a *quantum gate computer*, the transition is more involved but still straightforward. We therefore believe that once we understand the theory behind Hopfield nets, we can easily familiarize ourselves with the principles behind quantum computing.

For now, all these claims are hard to fathom and obviously need intensive study. But will this study be worth it? Put differently, why would we set the stage for all our upcoming discussions by looking at a combinatorial optimization problem and its difficulty? What could quantum computing contribute in this regard? As a tentative answer to these questions, we will end this chapter on even more claims which are best seen as cliffhangers for things to come.

### *1.6.3 Inherent Advantages of Quantum Computers*

To make a very long story short, quantum computers can explore the elements of a discrete search space of size $2^n$ in parallel. This is good news for combinatorial optimization as we can consequently expect to find a quantum algorithm that reduces search efforts from $O(2^n)$ to $O(2^{n/2}) = O(\sqrt{2^n})$.

Dealing with QUBOs, adiabatic quantum computers do this naturally [38]. On a quantum gate computer, we may invoke Grover's *amplitude amplification* for quadratic running time reduction [39].

Above, we saw that classical algorithms achieve such a reduction, too, but at the cost of exponential memory or exponential parallel processes. Quantum computers have no such needs but simply exploit quantum mechanical phenomena such as *superposition*, *phase flips*, or *entanglement* for which there is no classical analogue.

Since the benefits of seamlessly achieving at least *quadratic speedup*, i.e. a running time reduction from $O(2^n)$ to $O(\sqrt{2^n})$, may be hard to see just from looking at mathematical expressions, we visualize both rates in Fig. 1.1.

To complexity theorists, the quickly growing gap between the two curves seen in this figure is no big deal. After all, both grow exponentially in $n$ and thus belong to the same complexity class.

To business people, on the other hand, the gap can mean billions of Dollars saved or gained. For instance, if you can solve certain investment SSPs quadratically faster than your competition, you can either react to market dynamics much faster than them or you can react as slowly as them but based on many more forecasts than they would have available.

But doesn't quantum computing promise *almost exponential speedup* such as in Shor's famous prime factorization algorithm [40]? Yes, it does, but, given what we know today, only if the solution to the problem at hand involves Fourier transforms at one point or another. Put differently, so far, every known quantum algorithm that is exponentially faster than its classical counterpart makes use of the quantum Fourier transform [41]. This is why we emphasized that there are classical SSP solvers which involve Fourier transforms.

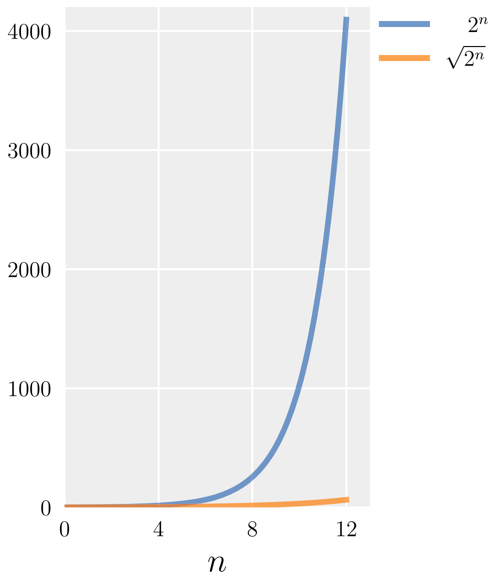


**Fig. 1.1** Visual comparison of the growth rates $2^n$ and $\sqrt{2^n}$

As of this writing, however, we are not aware of any published work where these algorithms would have inspired the design of quantum SSP solvers. While there is intensive research on quantum SSP solving (cf. [30] and the references therein), corresponding methods differ from the ones we discuss in this book. They do not consider QUBO representations and, to be frank, are often impractical in that they rely on theoretical quantum computing machinery with supposed capabilities which real-world quantum computers do not yet have and may even never have.

From time to time, we may return to such hypothetical capabilities in order to critique corresponding quantum computing research where necessary. However, our promise for this book is to avoid overly fanciful theoretical ideas at all costs and instead to focus on actually practical approaches.

## 1.7 Take Home Messages and Further Reading

In this chapter, we went a long way to motivate why quantum computing has disruptive potential in a world where first prototypes of working quantum computers have become available.

We introduced the *subset sum problem (SSP)* and saw that we may understand it as a *decision problem* or as a *combinatorial optimization problem*. We argued that combinatorial optimization problems are of considerable practical importance but generally difficult to solve.

We emphasized that the latter also holds for machine learning-based AI models because recent media coverage has led many to believe tcurrent AI can do everything.

However, combinatorial problem solving requires a different kind of intelligence than encoded in, say, transformers. To better understand the underlying difficulties, we looked at concepts from *computational complexity* theory recalled the notions of *NP-complete* and *NP-hard* problems.

This chapter therefore touched upon broad as well as deep mathematical subject areas, namely *set theory*, *complexity theory*, and *combinatorial optimization*. Those interested in much more in-depth expositions of these topics might want to consult the following resources.

A venerable and easy to read introduction to elementary set theory and its fundamental role in mathematics and the sciences can be found in a classic textbook by Stoll [42]. Interestingly enough, our suggestion for an introductory text on complexity theory also begins with a discussion of sets and their characteristics, namely the book by Bovet and Crescenzi [43] which is an easily accessible classic, too.

Since combinatorial optimization problems will be of major concern later on, we actually have several suggestions on this topic. A recent, gentle, and methodology oriented introduction can be found in the book by Du et al. [44]. Yet another recent book with a specific focus on heuristic approaches is the one by Taillard [45]. Finally, an established and rigorous exposoition emphasizing mathematical fundamentals can be found in the book by Korte and Vygen [46].

## 1.8 Exercises

**1.1** Prove Theorem 1.1 via induction.

**1.2** Assume you were asked to print out all the $2^{100} \approx 1.27 \times 10^{30}$ subsets $\mathcal{X}'$ of set $\mathcal{X}$ in (1.1). Further assume you had access to a current supercomputer which can perform $10^{18}$ floating point operations per second (FLOPS). Finally, assume in an overly optimistic manner that it only took a single FLOP to print a single subset. How many years would you have to wait until your overoptimistic supercomputer has printed all possibles subsets?

**1.3** Consider a set $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ of numbers $x_j = 2^{j-1}$ and show that these numbers are super-increasing.

**1.4** Consider some number $x_0 > 1$ and a set $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ of numbers $x_j = 2x_{j-1} + 1$. Use your result from Exercise 1.3 to show that these numbers are super-increasing.

**1.5** On "Difficulty", we claimed that an SSP can be solved in log-linear time if the set $\mathcal{X}$ under consideration is super-increasing. Invent or search the Web for an algorithm that accomplishes this feat. Implement your algorithm in `python/numpy` and test it. To this end, construct a set $\mathcal{X}$ as in the previous two exercises and randomly pick and sum some of its elements to obtain a target value $T$.

**1.6** Generate your own, fairly general SSP instances. That is, implement python / numpy code that produces (a representation of) a set $\mathcal{X}$ of $n$ integers and a target integer $T$. We suggest to proceed as follows: Use function randint in numpy's random module to create a vector $\boldsymbol{x} \in \mathbb{Z}^n$ of $n$ entries $x_{\min} \leq x_j \leq x_{\max}$ where $x_{\min}, x_{\max} \in \mathbb{Z}$ are parameters you may choose to your liking. Once $\boldsymbol{x}$ is available, use function choice in numpy's random module to create a vector $\boldsymbol{x}' \in \mathbb{Z}^k$ whose $1 \leq k < n$ entries are drawn without replacement from $\boldsymbol{x}$. Once $\boldsymbol{x}'$ is available, sum its entries to obtain $T$.

**1.7** In Exercise 1.6 you just saw how to generate random instances of a set $\mathcal{X}$ of $n$ integers and, in Exercise 1.2, we fancied the idea of printing all the $2^n$ subsets of such a set. So, let's do this for real now. Implement python/numpy code that prints all subsets of a given set of integers. We suggest working with function combinations in python's itertools module. Consider very moderate choices of $n$ to test your code, say, $n = 8$. Once you have ensured that your code is working properly, start increasing $n$ and test your patience. Up until which choice of $n$ are you willing to wait for your code to terminate?

**1.8** Use your code from the previous two exercises to brute force an SSP. That is, given an SSP $(\mathcal{X}, T)$, loop over all subsets $\mathcal{X}' \subseteq \mathcal{X}$ and sum their elements. As soon as one such sum evaluates to $T$, report success and break the loop. If no solution can be found, report failure. Consider different choices of $|\mathcal{X}| = n$ and measure the running time of your code using Timer from python's timeit module. What do you observe?

**1.9** If you have never worked with Boolean array masks before, experiment with the following code snippet and ponder what you observe.

```
import numpy as np

vecX = np.array([1,2,3,4,5])
vecZ = np.array([0,1,0,1,0]).astype(bool)

print (vecX[vecZ], np.sum(vecX[vecZ]))
```

**1.10** Let $\mathcal{X}'$ be a subset of $\mathcal{X} = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}$. In (1.19), we expressed the subset sum

$$S' = \sum_{x_j \in \mathcal{X}'} x_j \tag{1.27}$$

in terms of indicator variables $z_j \in \{0, 1\}$. Now consider the subset product

$$P' = \prod_{x_j \in \mathcal{X}'} x_j \ . \tag{1.28}$$

How would you express this product in terms of indicator variables $z_j \in \{0, 1\}$?

## References

1. Paaß, G., Giesselbach, S.: Foundation Models for Natural Language Processing. Springer (2023)
2. Brito, E., et al.: A Hybrid AI Tool to Extract Key Performance Indicators from Financial Reports for Benchmarking. In: Proc. Symp. on Document Engineering. ACM (2019). https://doi.org/10.1145/3342558.3345420
3. Hillebrand, L., et al.: Towards Automating Numerical Consistency Checks in Financial Reports. In: Proc. Int. Conf. on Big Data. IEEE (2022). https://doi.org/10.1109/BigData55660.2022.10020308
4. Deußer, T., et al.: Uncovering Inconsistencies and Contradictions in Financial Reports Using Large Language Models. In: Proc. Int. Conf. on Big Data. IEEE (2023). https://doi.org/10.1109/BigData59044.2023.10386673
5. Berger, A., et al.: Towards Automated Regulatory Compliance Verification in Financial Auditing with Large Language Models. In: Proc. Int. Conf. on Big Data. IEEE (2023). https://doi.org/10.1109/BigData59044.2023.10386518
6. Bauckhage, C.: Quantum Computing Homework with ChatGPT and Bard. Tech. rep., Lamarr Institute for ML and AI (2023). Available on researchgate.net
7. Frieder, S., et al.: Mathematical Capabilities of ChatGPT. In: Proc. NeurIPS (2023)
8. Plevris, V., Papazafeiropoulos, G., Jimenez Rios, A.: Chatbots Put to the Test in Math and Logic Problems: A Preliminary Comparison and Assessment of ChatGPT-3.5, ChatGPT-4, and Google Bard. AI **4**(4) (2023). https://doi.org/10.3390/ai4040048
9. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016)
10. Bishop, C., Bishop, H.: Deep Learning. Springer (2023)
11. Hornik, K., Stinchcombe, M., White, H.: Multilayer Feedforward Networks Are Universal Approximators. Neural Networks **2**(5) (1989). https://doi.org/10.1016/0893-6080(89)90020-8
12. Rumelhart, D., Hinton, G., Williams, R.: Learning Representations by Back-Propagating Errors. Nature **323**(6088) (1986). https://doi.org/10.1038/323533a0
13. Vaswani, A., et al.: Attention Is All You Need. In: Proc. NeurIPS (2017)
14. Brown, T., et al.: Language Models Are Few-Shot Learners. In: Proc. NeurIPS (2020)
15. Garmendia, A., Ceberio, J., Mendiburu, A.: Applicability of Neural Combinatorial Optimization: A Critical View. ACM Trans. Evolutionary Learning and Optimization **4**(3) (2024). https://doi.org/10.1145/3647644
16. Sutton, R., Barto, A.: Reinforcement Learning, 2nd edn. MIT Press (2018)
17. Bengio, Y., Lodi, A., Prouvost, A.: Machine Learning for Combinatorial Optimization: A Methodological Tour d'Horizon. European J. of Operational Research **290**(2) (2021). https://doi.org/10.1016/j.ejor.2020.07.063
18. Silver, D., et al.: Mastering the Game of Go with Deep Neural Networks and Tree Search. Nature **529**(7587) (2016). https://doi.org/10.1038/nature16961
19. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009)
20. Kregher, D., Stinson, D.: Combinatorial Algorithms: Generation, Enumeration, and Search. CRC Press (2020)
21. Bertsekas, D.: Dynamic Programming and Optimal Control, 4th edn. Athena Scientific (2017)
22. Russel, S., Norvig, P.: Artificial Intelligence, 4th edn. Pearson (2021)
23. Neumann, F., Witt, C.: Bioinspired Computation in Combinatorial Optimization. Springer (2010)
24. Biesner, D., et al.: Solving Subset Sum Problems Using Quantum Inspired Optimization Algorithms with Applications in Auditing and Financial Data Analysis. In: Proc. Int. Conf. Machine Learning and Applications. IEEE (2022). https://doi.org/10.1109/ICMLA55696.2022.00150
25. von Rueden, L., et al.: Informed Machine Learning – A Taxonomy and Survey of Integrating Prior Knowledge into Learning Systems. IEEE Trans. on Knowledge and Data Engineering **35**(1) (2023). https://doi.org/10.1109/TKDE.2021.3079836

26. Aloise, D., Deshapande, A., Hansen, P., Popat, P.: NP-Hardness of Euclidean Sum-of-Squares Clustering. Machine Learning **75**(2) (2009). https://doi.org/10.1007/s10994-009-5103-0
27. Lloyd, S.: Least Squares Quantization in PCM. IEEE Trans. on Information Theory **28**(2) (1982). https://doi.org/10.1109/TIT.1982.1056489
28. Schreiber, E., Korf, R., Moffit, M.: Optimal Multi-Way Number Partitioning. J. of the ACM **65**(4) (2018). https://doi.org/10.1145/318440
29. Koiliaris, K., Xu, C.: Subset Sum Made Simle. arXiv:1807.08248 [cs.DS] (2018)
30. Bonnetain, X., Bricout, R., Schrottenloher, A., Shen, Y.: Improved Classical and Quantum Algorithms for Subset-Sum. In: S. Morain, H. Wang (eds.) Advances in Cryptography, *LNCS*, vol. 12492. Springer (2020). https://doi.org/10.1007/978-3-030-64834-3_22
31. Bringmann, K., Fischer, N., Nakos, V.: Beating Bellman's Algorithm for Subset Sum. arXiv:2410.21942 [cs.DS] (2024)
32. Horowitz, E., Sahni, S.: Computing Partitions with Applications to the Knapsack Problem. J. of the ACM **21**(2) (1974). https://doi.org/10.1145/321812.321823
33. Schroeppel, R., Shamir, A.: A $T = O(2^{n/2})$, $S = O(2^{n/4})$ Algorithm for Certain NP-Complete Problems. SIAM J. of Computing **10**(3) (1981). https://doi.org/10.1137/0210033
34. Howgrave-Graham, N., Joux, A.: New Generic Algorithms for Hard Knapsacks. In: H. Gilbert (ed.) EUROCRYPT, *LNCS*, vol. 6110. Springer (2010). https://doi.org/10.1007/978-3-642-13190-5_12
35. Becker, A., Coron, J., Joux, A.: Improved Generic Algorithms for Hard Knapsacks. In: K. Paterson (ed.) EUROCRYPT, *LNCS*, vol. 6632. Springer (2011). https://doi.org/10.1007/978-3-642-20465-4_21
36. Bellman, R.: Dynamic Programming. Princeton University Press (1957)
37. Curtis, V., Sanches, C.: A Low-Space Algorithm for the Subset-Sum Problem on GPU. Computers & Operations Research **83** (2017). https://doi.org/10.1016/j.cor.2017.02.006
38. Johnson, M., et al.: Quantum Annealing with Manufactured Spins. Nature **473**(7346) (2011). https://doi.org/10.1038/nature10012
39. Grover, L.: A Fast Quantum Mechanical Algorithm for Database Search. In: Proc. Symp. on Theory of Computing. ACM (1996). https://doi.org/10.1145/237814.237866
40. Shor, P.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: Proc. Symp. on Foundations of Computer Science. IEEE (1994). https://doi.org/10.1109/SFCS.1994.365700
41. Coppersmith, D.: An Approximate Fourier Transform Useful in Quantum Factoring. arXiv:quant-ph/0201067 (2002)
42. Stoll, R.: Set Theory and Logic. Dover (1979)
43. Bovet, D., Crescenzi, P.: Introduction to the Theory of Complexity. Prentice Hall (1994)
44. Du, D.Z., Pardalos, P., Hu, X., Wu, W.: Introduction to Combinatorial Optimization. Springer (2022)
45. Taillard, E.: Design of Heuristic Algorithms for Hard Optimization. Springer (2023)
46. Korte, B., Vygen, J.: Combinatorial Optimization, 6th edn. Springer (2018)

# Chapter 2
# Boolean Domains, Numbers, and Vectors

## 2.1 Introduction

To firmly understand the behavior of Hopfield nets and many quantum computing algorithms requires us to understand properties and characteristics of *bivalent vectors* whose entries can only assume one of two values.

In Chap. 1, we already came across such vectors. We were concerned with subset sums and saw that we may represent a set $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ of $n$ numbers $x_j \in \mathbb{R}$ as a vector $\boldsymbol{x} \in \mathbb{R}^n$. We also saw that we may represent any subset $\mathcal{X}' \subseteq \mathcal{X}$ in terms of a binary vector $\boldsymbol{z} \in \{0, 1\}^n$ whose entries $z_j$ indicate which elements of $\mathcal{X}$ are contained in $\mathcal{X}'$. Following the convention of letting $z_j = 1$ if $x_j \in \mathcal{X}'$ and $z_j = 0$ if $x_j \notin \mathcal{X}'$, all this allowed us to compute the sum of the elements of $\mathcal{X}' \subseteq \mathcal{X}$ as the inner product of the indicator vector representing $\mathcal{X}'$ and the vector representing $\mathcal{X}$, because

$$\boldsymbol{z}^\mathsf{T}\boldsymbol{x} = \sum_{j=1}^{n} z_j x_j = \sum_{x_j \notin \mathcal{X}'} 0 \cdot x_j + \sum_{x_j \in \mathcal{X}'} 1 \cdot x_j = \sum_{x_j \in \mathcal{X}'} x_j \ . \tag{2.1}$$

Binary indicator vectors can thus specify which $k$ objects out of $n$ objects to select for further processing. But applications of binary- or, more generally, bivalent vectors extend far beyond subset selection. They can represent numbers, allow for Boolean logic computations, play a central role in QUBO problems, and, last but not least, characterize the states of Hopfield nets and (certain) quantum systems.

In this chapter, we will therefore have a closer look at sets of bivalent numbers and vectors and study their characteristics from quite a wide range of different angles. These include Boolean logic, ((multi-)liner) algebra, probability theory and statistics, geometry, and graph theory. The content of this chapter is therefore dense and fairly technical. Yet, its seemingly unrelated topics are actually interlinked and altogether fundamental to our upcoming discussions. In short, the following equips us with a

C. Bauckhage and R. Sifa, *Quantum Computing from Hopfield Nets*,
Cognitive Technologies, https://doi.org/10.1007/978-3-031-99402-9_2

very broad perspective on the apparently innocuous ideas of bivalent numbers and vectors and introduces important terminology and concepts which we will need in later chapters.

## 2.2 Boolean Domains

Let us widen our perspective on bivalent variables and relate them to *propositional calculus* or *Boolean logic*. To motivate this widening, we return to binary indicator vectors and emphasize the meaning of their entries in terms of *logical equivalencies*, namely

$$z_j = 0 \Leftrightarrow x_j \notin \mathcal{X}' \tag{2.2}$$
$$z_j = 1 \Leftrightarrow x_j \in \mathcal{X}' \,. \tag{2.3}$$

Note that the two *propositions* $x_j \notin \mathcal{X}'$ and $x_j \in \mathcal{X}'$ on the right hand sides are mutually exclusive because $x_j$ either *is* or *is not* an element of $\mathcal{X}'$. If the former is `true`, the latter must be `false` and vice versa. The binary variable $z_j \in \{0, 1\}$ on the left thus *represents* the truth value of the proposition "$x_j$ is an element of $\mathcal{X}'$". If it is `false`, then $z_j = 0$, and if it is `true`, then $z_j = 1$.

Given this connection between the numbers 0 and 1 and the truth values `false` and `true`, we now begin to widen our point of view.

**Definition 2.1** A **Boolean domain** $\mathbb{B}$ is a set of only two elements which can be interpreted as representations of the truth values `false` and `true`.

Looking at this definition, several remarks appear to be in order. First of all, we must be clear about what it means to *represent* the truth values `false` and `true`. We therefore note that their defining property is that they are *logical opposites* in that

$$\texttt{NOT(false)} = \texttt{true} \tag{2.4}$$
$$\texttt{NOT(true)} = \texttt{false} \,. \tag{2.5}$$

Any two objects which reasonably represent `false` and `true` should thus be able to replicate this crucial characteristic.

Second of all, we already saw the most canonical example of a Boolean domain, namely the set of numbers {0, 1}.

Third of all, it is indeed common to work with Boolean domains composed of numbers since this allows for arithmetic with logical opposites. This is useful because there are numerous real world entities which exist in one of two mutually exclusive *states* such as `on/off`, `up/down`, or `selected/unselected`. Dealing with such *two state systems*, we often need to reason about their behavior or dynamics and corresponding computations might be simpler if the underlying states were amenable to operations such as addition, subtraction, or multiplication.

The two most prominent and arguably most important Boolean domains composed of numbers are

$$\bigl\{0, 1\bigr\} \quad \text{and} \quad \bigl\{-1, +1\bigr\} \tag{2.6}$$

where 1 and $+1$ denote the same number; the latter is but a more elaborate way of writing the former.

**Notation**

We henceforth write $z$ to denote a variable with values in $\{0, 1\}$ and $s$ to denote a variable with values in $\{-1, +1\}$. As a mnemonic, read these variable names as

$$\begin{aligned} z &\Leftrightarrow \text{“\underline{z}ero or one”} \\ s &\Leftrightarrow \text{“\underline{s}igned one”}. \end{aligned}$$

Moreover, we henceforth write the set $\{-1, +1\}$ in a more compact manner and let

$$\bigl\{\pm 1\bigr\} \equiv \bigl\{-1, +1\bigr\} .$$

To see that the two numbers 0 and 1 can be interpreted as logical opposites which allow for arithmetic, we simply define negation as a function $\texttt{NOT} : \{0, 1\} \to \{0, 1\}$ such that $\texttt{NOT}(z) = 1 - z$. We then have $\texttt{NOT}(0) = 1$ and $\texttt{NOT}(1) = 0$ which replicates the characteristics in Eqs. (2.4) and (2.5).

Similarly, when working with the two numbers $-1$ and $+1$, we may simply define $\texttt{NOT} : \{\pm 1\} \to \{\pm 1\}$ such that $\texttt{NOT}(s) = -s$. With this, we then have $\texttt{NOT}(-1) = +1$ as well as $\texttt{NOT}(+1) = -1$ which again replicates the behavior of logical negation.

We thus just demonstrated that the sets $\{0, 1\}$ and $\{\pm 1\}$ form Boolean domains. This only required us to define appropriate negations which turn their elements into logical opposites. What it decidedly did not require us to do was to declare which of the respective elements should stand for `false` and which for `true`. While it seems natural or intuitive to identify 0 or $-1$ with `false` and 1 or $+1$ with `true`, we remark that it can be reasonable to do it the other way around, for instance in the analysis of Boolean functions [1].

Next, we further substantiate our claim that the numerical Boolean domains $\{0, 1\}$ and $\{\pm 1\}$ allow for logic in terms of arithmetic. We have already seen this for the *univariate* Boolean function NOT but, to be truly general, we also need to verify this for *bivariate* Boolean functions. To keep things simple, we restrict our attention to the functions AND, OR, XOR, and NAND whose definitions we recall in Table 2.1.

This restriction is perfectly fine because, if we wanted to, we could show that the sets {NOT, AND, OR}, {AND, XOR}, and {NAND} are examples of *functionally*

**Table 2.1** Definitions of the bivariate Boolean functions AND, OR, XOR, and NAND

| Inputs | | Outputs | | | |
|---|---|---|---|---|---|
| $x_1$ | $x_2$ | AND$(x_1, x_2)$ | OR$(x_1, x_2)$ | XOR$(x_1, x_2)$ | NAND$(x_1, x_2)$ |
| false | false | false | false | false | true |
| false | true | false | true | true | true |
| true | false | false | true | true | true |
| true | true | true | true | false | false |

*complete* or *universal* sets of Boolean functions [2, 3]. This means that every Boolean function of arbitrarily many arguments can be written only in terms of the functions gathered in these sets and we will use this later on.

If we represent `false` in terms of 0 and `true` in terms of 1, it is an easy exercise (left to the reader) to verify that the following are analytical representations of the Boolean functions in Table 2.1:

$$\mathrm{AND}_z(z_1, z_2) = z_1 \cdot z_2 \tag{2.7}$$

$$\mathrm{OR}_z(z_1, z_2) = z_1 + z_2 - z_1 \cdot z_2 \tag{2.8}$$

$$\mathrm{XOR}_z(z_1, z_2) = z_1 + z_2 - 2 \cdot z_1 \cdot z_2 \tag{2.9}$$

$$\mathrm{NAND}_z(z_1, z_2) = 1 - z_1 \cdot z_2 \ . \tag{2.10}$$

All these functions map $(z_1, z_2) \in \{0, 1\}^2$ to $z \in \{0, 1\}$ and are analytical because they only involve arithmetic operations on numbers. Curiously, the expression for $\mathrm{XOR}_z(z_1, z_2)$ involves the number 2 which is not an element of $\{0, 1\}$. But this is again perfectly fine and reflects the fact that XOR is a quadratic in its arguments

$$\mathrm{XOR}_z(z_1, z_2) = (z_1 - z_2)^2 = z_1^2 - 2\, z_1\, z_2 + z_2^2 \ . \tag{2.11}$$

This latent quadratic nature of XOR is just not immediately apparent from (2.9) where we simplified the quadratic terms by exploiting a rather remarkable property of the numbers 0 and 1, namely that $\forall z \in \{0, 1\} : z^2 = z$.

We could now similarly show that the Boolean domain $\{\pm 1\}$ also allows for analytical representations of the functions in Table 2.1. However, we will postpone this step to the next section where we will get to a know a tool which we can apply to automatically transform the above mappings from $\{0, 1\}^2$ into $\{0, 1\}$ to mappings from $\{\pm 1\}^2$ into $\{\pm 1\}$.

## 2.3 Binary and Bipolar Numbers

From now on and throughout, we will work with specific, distinct names for the *Boolean numbers* $z \in \{0, 1\}$ and $s \in \{\pm 1\}$. Following common conventions, we will refer to the former as *binary numbers* and to the latter as *bipolar numbers*.

**Definition 2.2** A number $z \in \{0, 1\}$ is called a **binary number**.

**Definition 2.3** A number $s \in \{\pm 1\}$ is called a **bipolar number**.

Looking at our definition of the binary numbers, some readers may now interject that it clashes with another use of the term. Indeed, it is common to say that a binary number is a natural number written in the base-two numeral system. To disambiguate between these two meanings, we therefore say that a **binary string** (such as `1001`) is not a binary number but a **binary representation of a number** (such as 9).

The binary numbers are truly special and have noteworthy *algebraic properties* which other numbers simply do not have. For instance, 0 is *neutral with respect to addition* and 1 is *neutral with respect to multiplication*. That is, for any $x \in \mathbb{R}$, we have

$$0 + x = x + 0 = x \tag{2.12}$$

$$1 \cdot x = x \cdot 1 = x \ . \tag{2.13}$$

Moreover, 0 and 1 are said to be *idempotent* under multiplication because both *square to themselves*. That is $0^2 = 0$ and $1^2 = 1$ or, in short

$$\forall z \in \big\{0, 1\big\} : z^2 = z \ . \tag{2.14}$$

As a corollary we therefore have $z^3 = z^2 \cdot z = z \cdot z = z$ which we can inductively extend to $z^k = z$ for any $2 \leq k \in \mathbb{N}$.

Furthermore, since any product that involves a 0 evaluates to 0 and since 1 is a neutral multiplier, the set of binary numbers as a whole is *closed under multiplication*

$$\forall z_1, z_2 \in \big\{0, 1\big\} : z_1 \cdot z_2 \in \big\{0, 1\big\} \ . \tag{2.15}$$

Last but certainly not least, the binary numbers behave extraordinarily special when used as exponents. Indeed, for or any $x \in \mathbb{R}$, we have the crucial identities

$$x^0 = 1 \tag{2.16}$$

$$x^1 = x \ . \tag{2.17}$$

The bipolar numbers are not quite so remarkable. While $+1$ has all the properties we listed for 1, its opposite $-1$ behaves not as interestingly as 0.

However, we note that the bipolar numbers are their own multiplicative inverses. In other words, they are *involutory under multiplication* since both square to $+1$. That is $(-1)^2 = +1$ and $(+1)^2 = +1$ or, in short

$$\forall s \in \big\{\pm 1\big\} : s^2 = +1 \; . \tag{2.18}$$

We further observe that, for any $x \in \mathbb{R} \setminus \{0\}$, we have $x^{-1} \cdot x^{+1} = +1$ and that the set of bipolar numbers is *closed under multiplication*, too, because

$$\forall s_1, s_2 \in \big\{\pm 1\big\} : s_1 \cdot s_2 \in \big\{\pm 1\big\} \; . \tag{2.19}$$

An overarching crucial property of binary- and bipolar numbers is that we can easily map back and forth between them. This will be so important later on that it merits emphasis in form of a theorem.

**Theorem 2.1** *The sets* $\{0, 1\}$ *and* $\{\pm 1\}$ *of binary- and bipolar numbers are* affinely isomorphic*. There exists an affine transformation from* $\{0, 1\}$ *to* $\{\pm 1\}$ *whose inverse is an affine transformation from* $\{\pm 1\}$ *to* $\{0, 1\}$*. In particular, we have the implications*

$$z \in \big\{0, 1\big\} \quad \Rightarrow \quad s = 2\,z - 1 \in \big\{\pm 1\big\} \tag{2.20}$$

$$s \in \big\{\pm 1\big\} \quad \Rightarrow \quad z = \tfrac{1}{2} s + \tfrac{1}{2} \in \big\{0, 1\big\} \; . \tag{2.21}$$

***Proof*** All three sentences in this theorem are but different ways of expressing the same fact. For the proof, we therefore only need to verify the two implications which can be done via direct computation. □

An immediate application of this theorem consists in a procedure for translating the binary Boolean functions in (2.7)–(2.10) into bipolar Boolean functions. For brevity, we only demonstrate the general mechanism for the function AND and leave the other cases as exercises.

First, we transform the two inputs $(z_1, z_2)$ from binary- to bipolar variables. Letting

$$z_j = \frac{s_j + 1}{2} \tag{2.22}$$

we can write

$$\text{AND}_z(z_1, z_2) = \frac{s_1 + 1}{2} \cdot \frac{s_2 + 1}{2} = \tfrac{1}{4}\big(s_1 \cdot s_2 + s_1 + s_2 + 1\big) = \text{AND}_z(s_1, s_2) \tag{2.23}$$

Since the resulting function still produces outputs $z \in \{0, 1\}$, we next transform those to $s \in \{\pm 1\}$. Using $s = 2z - 1$, we accomplish this as follows

$$\begin{aligned} 2\,\texttt{AND}_z(s_1, s_2) - 1 &= \tfrac{2}{4}\big(s_1 \cdot s_2 + s_1 + s_2 + 1\big) - 1 && (2.24)\\ &= \tfrac{1}{2}\big(s_1 \cdot s_2 + s_1 + s_2 - 1\big) && (2.25)\\ &\equiv \texttt{AND}_s(s_1, s_2) && (2.26)\end{aligned}$$

and thus obtain a function $\texttt{AND}_s(s_1, s_2)$ with two inputs $(s_1, s_2) \in \{\pm 1\}^2$ and an output $s \in \{\pm 1\}$. Readers are encouraged to verify its characteristics and that these characteristics consistently represent the behavior of the Boolean AND in Table 2.1.

## 2.4 Bernoulli Random Variables

Next, we assume another point of view on Boolean domains and consider them to be representations of the outcomes of random events. This allows us to recall important concepts from probability theory and statistics which we will need in later chapters.

To begin with, we recall that a **random variable (r.v.)** is a variable whose value is subject to chance. Throughout, we will mainly be concerned with *discrete random variables* $X$ whose possible values $x$ come from a *discrete set* $\mathcal{X}$.

We henceforth write $p(X = x)$ to denote the **probability** for a discrete r.v. $X$ to assume a value of $x \in \mathcal{X}$. By definition, these probabilities are real numbers with

$$0 \leq p(X = x) \leq 1 \tag{2.27}$$

for which the **sum rule** of probability theory demands that

$$\sum_{x \in \mathcal{X}} p(X = x) = 1\ . \tag{2.28}$$

The **probability distribution** of a random variable is a function which maps the possible values $x \in \mathcal{X}$ of $X$ to their probabilities. Since distributions of discrete- and continuous variables have different characteristics, they are further distinguished into *probability mass functions* and *probability density functions*. In what follows, we focus on the former. The **probability mass function** of a discrete random variable $X$ over a discrete set $\mathcal{X}$ is a mapping $f_X$ from $\mathcal{X}$ into the *interval* $[0, 1]$ such that

$$f_X\big(x \mid \Theta\big) = p(X = x) \tag{2.29}$$

where $\Theta = \{\theta_1, \ldots, \theta_m\}$ denotes a generic set of possible parameters. Details as to the exact form of probability mass functions and their parameters always depend on application contexts. However, once they have been specified, it is common to write

$$X \sim f_X\big(x \mid \Theta\big) \tag{2.30}$$

in order to express that *the value of r.v. $X$ is distributed according to* $f_X(x \mid \Theta)$.

A **Bernoulli random variable** is the simplest kind of random variable as it can only take on two different values. A canonical example is the outcome of coin toss which may result in `heads` or `tails`. However, these are non-numeric quantities which do not immediately allow for computations. To compute with Bernoulli r.v.s, it is therefore common to *represent* their two possible values in terms of the binary numbers $\{0, 1\}$.

Having said this, we emphasize the following: Whenever we consider a Bernoulli variable $Z$ with possible values $z \in \{0, 1\}$, we have

$$0 \leq p(Z = z) \leq 1 \tag{2.31}$$

where the 0 and 1 to the left and right of the less-than-or-equal signs have a different meaning than the 0 and 1 which occur in the expression $z \in \{0, 1\}$. The former are the lower- and upper bound of a probability; the latter are potential instantiations of a random variable.

The **Bernoulli distribution** is a probability mass function which models the distribution of a Bernoulli variable $Z$ whose **success probability** is

$$p(Z = 1) = \mu \, . \tag{2.32}$$

Since a Bernoulli variable $Z$ can only assume two values, the sum rule implies that $p(Z = 0) + p(Z = 1) = 1$. This, in turn, implies that $Z$ has a **failure probability** of

$$p(Z = 0) = 1 - p(Z = 1) = 1 - \mu \, . \tag{2.33}$$

The Bernoulli distribution can therefore be defined as a function $f_{\text{Ber}}$ with an argument $z \in \{0, 1\}$ and a single (success) parameter $0 \leq \mu \leq 1$ such that

$$f_{\text{Ber}}\big(z \mid \mu\big) = \begin{cases} \mu & \text{if } z = 1 \\ 1 - \mu & \text{if } z = 0 \, . \end{cases} \tag{2.34}$$

An equivalent but more compact definition resorts to the exponentiation properties of binary numbers in (2.16) and (2.17). It reads

$$f_{\text{Ber}}\big(z \mid \mu\big) = \mu^{z} \cdot \big(1 - \mu\big)^{1-z} \tag{2.35}$$

and can be verified by plugging in $z = 1$ and $z = 0$. Readers are very much encouraged to do this and thus to recover the behavior of case-based definition in (2.34).

### 2.4.1 Expected Value and Variance of Bernoulli Variables

To explain our seemingly curious choice of writing "$\mu$" for the success probability of a Bernoulli variable, we could simply argue that it adopts common practices in the machine learning literature [4].

Yet, to be more rigorous, we observe that the **expected value** of a discrete random variable $X$ with possible values $x \in \mathcal{X}$ is defined to be

$$\mathbb{E}\big[X\big] = \sum_{x \in \mathcal{X}} x \cdot p(X = x) \ . \tag{2.36}$$

We further observe that $\mathbb{E}[X]$ is not the only way of referring to this value. In fact, the following are but a few examples of commonly found alternative notations

$$\mathbb{E}\big[X\big] \equiv \big\langle X \big\rangle \equiv \mu_X \equiv \mu \ . \tag{2.37}$$

If we now evaluate (2.36) for a Bernoulli variable $Z$ with possible values $z \in \{0, 1\}$ and success probability $\mu$, we curiously find that

$$\mathbb{E}\big[Z\big] = \sum_{z=0}^{1} z \cdot p(Z = z) = 0 \cdot \big(1 - \mu\big) + 1 \cdot \mu = \mu \ . \tag{2.38}$$

In other words, we find that the expected value of a Bernoulli variable coincides with its success probability. Together with (2.37), this justifies our notation.

To better understand what it means for a Bernoulli variable $Z$ to have an expected value of $\mu$, we return to the example of a coin toss with outcomes `heads` or `tails` which we associate with success and failure, respectively. Assuming a slightly loaded coin with a success probability of, say, $\mu = 0.6$, it will come up `heads` 60% of the time and `tails` 40% of the time. A Bernoulli variable $Z \in \{0, 1\}$ representing this coin would therefore have an expected value of $\mathbb{E}[Z] = 0.6$.

But isn't $Z$ supposed to be a binary variable? How then can its expectation be a real number between 0 and 1?

Well, the important thing to note is that $\mathbb{E}[Z]$ is a *statistical characterization* of $Z$ and not $Z$ itself. In machine learning parlance, we could say that $\mathbb{E}[Z]$ is a *feature* we could use to reason about $Z$.

There are many more statistical features. For instance, the **variance** of a random variable $X$ is defined as the expected value of the squared difference or deviation between $X$ and its expectation $\mathbb{E}[X]$. Formally, we have

$$\mathbb{V}\big[X\big] = \mathbb{E}\Big[\big(X - \mathbb{E}[X]\big)^2\Big] = \mathbb{E}\Big[X^2 - 2\,X\,\mathbb{E}[X] + \mathbb{E}^2[X]\Big] \tag{2.39}$$

and observe that Exercise 2.4 allows for the following quite convenient simplification

$$\mathbb{V}\big[X\big] = \mathbb{E}\big[X^2\big] - \mathbb{E}^2\big[X\big] \,. \tag{2.40}$$

With respect to notation, we remark that $\mathbb{V}[X]$ is again not the only way of referring to the variance of $X$; the following are common alternatives: $\mathbb{V}[X] \equiv \text{var}[X] \equiv \sigma^2$.

For a Bernoulli random variable $Z \in \{0, 1\}$ whose success probability $p(Z = 1)$ and expectation $\mathbb{E}[Z]$ are both equal to $\mu$, the above translates to

$$\mathbb{V}\big[Z\big] = \mathbb{E}\big[Z^2\big] - \mathbb{E}^2\big[Z\big] = \left[\sum_{z=0}^{1} z^2 \cdot p(Z = z)\right] - \mu^2 \tag{2.41}$$

$$= \left[\sum_{z=0}^{1} z \cdot p(Z = z)\right] - \mu^2 \tag{2.42}$$

$$= \mu - \mu^2 = \big(1 - \mu\big) \cdot \mu \tag{2.43}$$

where we once again have used that $z^2 = z$ for all $z \in \{0, 1\}$. In other words, we find the curious fact that the variance of a Bernoulli variable amounts to the product of its success- and failure probability.

All in all, our insights so far can be summarized in a short lemma about Bernoulli random variables.

**Lemma 2.1** *Let Z be a Bernoulli variable with possible values* $z \in \{0, 1\}$ *and success probability* $0 \leq \mu \leq 1$ *such that*

$$p(Z = z) = f_{Ber}(z \mid \mu) = \mu^z \cdot \big(1 - \mu\big)^{1-z} \,. \tag{2.44}$$

*Then its expected value and variance are given by*

$$\mathbb{E}\big[Z\big] = \mu \tag{2.45}$$

$$\mathbb{V}\big[Z\big] = \mu \cdot \big(1 - \mu\big) \,. \tag{2.46}$$

## 2.4.2 Products and Sums of Bernoulli Variables

Next, we let $Z_1 \sim f_{\text{Ber}}(z \mid \mu_1)$ and $Z_2 \sim f_{\text{Ber}}(z \mid \mu_2)$ be two *independent* Bernoulli variables and ask for the probability distribution of their product $Z_1 \cdot Z_2 = Z_3$.

We can immediately infer that $Z_3$ will be yet another Bernoulli random variable: First of all, since $Z_1$ and $Z_2$ are random variables, their product will be yet another random variable. Second of all, since the values of $Z_1$ and $Z_2$ are restricted to $\{0, 1\}$, the value of their product will be, too.

To determine the specific form of the distribution of $Z_3$, we now bring together combinatorics, logic, and probability theory and first observe the equivalencies:

$$
\begin{aligned}
Z_3 = 0 \;&\Leftrightarrow\; \big(Z_1 = 0 \texttt{ AND } Z_2 = 0\big) \texttt{ OR} \\
&\phantom{\Leftrightarrow}\;\; \big(Z_1 = 0 \texttt{ AND } Z_2 = 1\big) \texttt{ OR} \\
&\phantom{\Leftrightarrow}\;\; \big(Z_1 = 1 \texttt{ AND } Z_2 = 0\big) \\
Z_3 = 1 \;&\Leftrightarrow\; \big(Z_1 = 1 \texttt{ AND } Z_2 = 1\big)\ .
\end{aligned}
$$

We further observe that the **joint probability** $p(Z_1 = z_1, Z_2 = z_2)$ characterizes the probability of the event $(Z_1 = z_1 \texttt{ AND } Z_2 = z_2)$ and that the sum rule demands

$$\sum_{z_1=0}^{1} \sum_{z_2=0}^{1} p(Z_1 = z_1, Z_2 = z_2) = 1\ . \tag{2.47}$$

For our current scenario where $Z_1 \cdot Z_2 = Z_3$ and $Z_1, Z_2, Z_3 \in \{0, 1\}$, we therefore realize that the success probability for $Z_3$ is a joint probability in disguise, namely

$$p(Z_3 = 1) = p(Z_1 = 1, Z_2 = 1) \tag{2.48}$$

for which (2.47) implies that its failure probability is a sum of joint probabilities

$$p(Z_3 = 0) = p(Z_1 = 0, Z_2 = 0) + p(Z_1 = 0, Z_2 = 1) + p(Z_1 = 1, Z_2 = 0)\ . \tag{2.49}$$

Because we assumed $Z_1$ and $Z_2$ to be **independent** random variables, their joint probability factors as

$$p(Z_1 = z_1, Z_2 = z_2) = p(Z_1 = z_1) \cdot p(Z_2 = z_2)\ . \tag{2.50}$$

We also assumed the success probabilities for $Z_1$ and $Z_2$ to be $\mu_1$ and $\mu_2$, respectively. For $j \in \{1, 2\}$, we therefore have $p(Z_j = 1) = \mu_j$ and $p(Z_j = 0) = 1 - \mu_j$. Plugging these into (2.50) and the results into (2.48) and (2.49), we find

$$
\begin{aligned}
p(Z_3 = 1) &= \mu_1 \mu_2 && (2.51)\\
p(Z_3 = 0) &= \big(1 - \mu_1\big)\big(1 - \mu_2\big) + \big(1 - \mu_1\big)\,\mu_2 + \mu_1\,\big(1 - \mu_2\big) && (2.52)\\
&= 1 - \mu_1 \mu_2\ . && (2.53)
\end{aligned}
$$

This is quite a profound result because it tells us that the product $Z_1 Z_2 = Z_3$ of two Bernoulli variables is yet another Bernoulli variable whose success probability $\mu_1 \mu_2 = \mu_3$ is the product of two success probabilities. In short, we just proved:

**Lemma 2.2** *Let $Z_1 \sim f_{Ber}(z \mid \mu_1)$ and $Z_2 \sim f_{Ber}(z \mid \mu_2)$ be two independent Bernoulli variables. Then their product is a Bernoulli variable such that*

$$Z_1 Z_2 \sim f_{Ber}(z \mid \mu_1 \mu_2)\ . \tag{2.54}$$

We could next generalize this result to products of $n$ independent Bernoulli variables $Z_1, Z_2, \ldots, Z_n$. However, as this is easily done, we leave it to Exercise 2.6.

But what about sums of $n$ Bernoulli variables $Z_1, Z_2, \ldots, Z_n$? Well, for these we would or better should have to distinguish two cases, namely the case where the

$$Z_j \sim f_{\text{Ber}}(z \mid \mu_j) \tag{2.55}$$

are **independent *and* identically distributed (i.i.d.)** and the case where they are independent but not necessarily identically distributed.

We already know that independence means that the joint probability of the $Z_j$ factors as $p(Z_1, Z_2, \ldots, Z_n) = p(Z_1)\, p(Z_2) \cdots p(Z_n)$. To say that the $Z_j$ are identically distributed simply means that they all follow the same distribiution which is to say that $\mu_j = \mu$ for all $j$.

Now, to make a long story short, it is immediately clear that the sum

$$K = \sum_{j=1}^{n} Z_j \tag{2.56}$$

of $n$ Bernoulli variables $Z_j$ with values in $z_j \in \{0, 1\}$ will be another random variable whose value $k \in \mathbb{N}$ will have to obey $0 \leq k \leq n$.

Just as we did above, we could now bring together combinatorics, logic, and probability theory to derive an expression for the probability $p(K = k)$. This would be fairly easy for the case of i.i.d. Bernoulli variables and not quite so easy for merely independent Bernoulli variables. However, we won't go down this road but instead simply state the following result without proof.

**Lemma 2.3** *Let $Z_1, \ldots, Z_n$ be i.i.d. Bernoulli variables with $Z_j \sim f_{Ber}(z \mid \mu)$. Then their sum $K$ is a* ***binomial random variable*** *such that*

$$p(K = k) = f_{Bin}(k \mid n, \mu) = \binom{n}{k} \mu^k \cdot \big(1 - \mu\big)^{n-k} . \tag{2.57}$$

*Moreover, the expected value and variance of $K \sim f_{Bin}(k \mid n, \mu)$ are given by*

$$\mathbb{E}\big[K\big] = n \cdot \mu \tag{2.58}$$

$$\mathbb{V}\big[K\big] = n \cdot \mu \cdot \big(1 - \mu\big) . \tag{2.59}$$

For the more general case where the Bernoulli variables are independent but not necessarily identically distributed, we even simpler claim that their sum follows a *Poisson binomial distribution* and leave it at that.

A interpretation of the **binomial distribution** $f_{\text{Bin}}(k \mid n, \mu)$ in Lemma 2.3 is to say that it models the probability for $0 \leq k \leq n$ successes in $n$ independent Bernoulli trials each with success rate $\mu$. For example, we may toss a coin $n$ times and count the number $k$ of success. If we repeat this procedure again and again (tossing $n$ times and counting $k$), we likely get different values of $k$. The binomial distribution tells us how these values are distributed.

## 2.5 Binary and Bipolar Vectors

Given our preparations up to this point, we can finally look at *bivalent vectors* and their properties. As we already did a lot of heavy lifting with binary- and bipolar numbers, it will be no surprise that we shall focus on *binary*- and *bipolar vectors*. These are defines as follows.

**Definition 2.4** A vector $\boldsymbol{z} \in \{0, 1\}^n$ is called a **binary vector**.

**Definition 2.5** A vector $\boldsymbol{s} \in \{\pm 1\}^n$ is called a **bipolar vector**.

What will be of considerable importance later on is that we can easily map back and forth between binary- and bipolar vectors, for instance, by means of simple affine transformations. Indeed, the following is a straightforward generalization of Theorem 2.1.

**Theorem 2.2** *The sets* $\{0, 1\}^n$ *and* $\{\pm 1\}^n$ *of binary- and bipolar vectors are* affinely isomorphic. *Letting* $\mathbf{1} \in \mathbb{R}^n$ *denote the vector of all ones, we have the implications*

$$\boldsymbol{z} \in \big\{0, 1\big\}^n \quad \Rightarrow \quad \boldsymbol{s} = 2\,\boldsymbol{z} - \mathbf{1} \in \big\{\pm 1\big\}^n$$

$$\boldsymbol{s} \in \big\{\pm 1\big\}^n \quad \Rightarrow \quad \boldsymbol{z} = \tfrac{1}{2}\boldsymbol{s} + \tfrac{1}{2} \in \big\{0, 1\big\}^n\,.$$

***Proof*** Both statements in this theorem are again equivalent so that we only have to verify the implications. These simply follow from a component-wise application of Theorem 2.1 to the entries $z_1, z_2, \ldots, z_n$ and $s_1, s_2, \ldots, s_n$ of vectors $\boldsymbol{z}$ and $\boldsymbol{s}$, respectively. □

### *2.5.1 Geometry and Topology of Binary and Bipolar Vectors*

If we think of binary and bipolar vectors $\boldsymbol{z} \in \{0, 1\}^n$ and $\boldsymbol{s} \in \{\pm 1\}^n$ as points embedded in a Euclidean space $\mathbb{R}^n$, we realize that they form the corners of hypercubes.

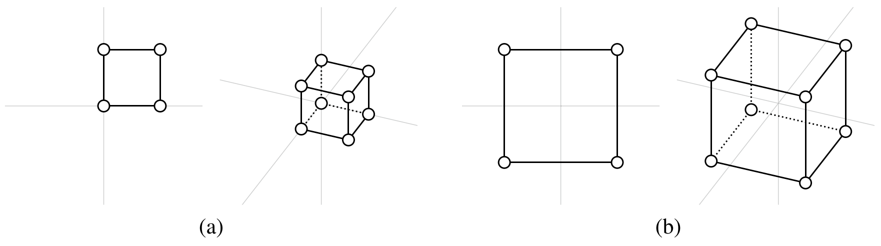


**Fig. 2.1** Binary- and bipolar cubes embedded in two- and three-dimensional Euclidean spaces. **a** Binary cubes in $\mathbb{R}^2$ and $\mathbb{R}^3$. **b** Bipolar cubes in $\mathbb{R}^2$ and $\mathbb{R}^3$

Figure 2.1 illustrates this for the easily visualized cases where $n \in \{2, 3\}$. For higher dimensional settings, visualizations are still possible albeit not with respect to coordinate axes and we shall return to this point shortly.

While corners of a binary cube all reside in the positive orthant of the respective coordinate system, corners of a bipolar cube are arranged symmetrically about the origin of the system. This symmetry or lack thereof entails further characteristics of binary- and bipolar vectors.

Recall, for instance, that the squared Euclidean norm $\|\boldsymbol{x}\|^2$ of a vector $\boldsymbol{x} \in \mathbb{R}^n$ is but the inner product $\boldsymbol{x}^\mathsf{T}\boldsymbol{x}$. For a binary vector $\boldsymbol{z} \in \{0, 1\}^n$, we therefore have

$$\|\boldsymbol{z}\|^2 = \boldsymbol{z}^\mathsf{T}\boldsymbol{z} = \sum_{j=1}^{n} z_j^2 = \sum_{j=1}^{n} z_j \tag{2.60}$$

where we again used the idempotency of binary numbers. But this guarantees the following bounds for (squared) norms of $n$-dimensional binary vectors

$$0 \leq \|\boldsymbol{z}\|^2 \leq n \quad \Leftrightarrow \quad 0 \leq \|\boldsymbol{z}\| \leq \sqrt{n}\;. \tag{2.61}$$

For bipolar vectors $\boldsymbol{s} \in \{\pm 1\}^n$, we have an arguably more interesting result. Since their entries are involutory, we find

$$\|\boldsymbol{s}\|^2 = \boldsymbol{s}^\mathsf{T}\boldsymbol{s} = \sum_{j=1}^{n} s_j^2 = \sum_{j=1}^{n} 1 = n \tag{2.62}$$

and realize that (squared) norms of $n$-dimensional bipolar vectors always amount to

$$\|\boldsymbol{s}\|^2 = n \quad \Leftrightarrow \quad \|\boldsymbol{s}\| = \sqrt{n}\;. \tag{2.63}$$

Next, we explore (a)symmetries of binary- and bipolar vectors in regards to inner products. We shall argue statistically and consider expected values of inner products of pairs of random binary- and bipolar vectors.

To begin with, we let $z, z' \in \{0, 1\}^n$ be two i.i.d. Bernoulli vectors and thus assume that $p(z_j = z) = p(z'_j = z) = f_{\text{Ber}}(z \mid \mu)$ and $p(z_j, z'_j) = p(z_j)\, p(z'_j)$. For the expected value of their inner product, we therefore have

$$\mathbb{E}\big[z^\intercal z'\big] = \mathbb{E}\left[\sum_{j=1}^{n} z_j z'_j\right] = \sum_{j=1}^{n} \mathbb{E}\big[z_j z'_j\big] = \sum_{j=1}^{n} \mathbb{E}\big[z_j\big]\,\mathbb{E}\big[z'_j\big] = \sum_{j=1}^{n} \mu\mu = n\mu^2 \ . \tag{2.64}$$

Observe that this agrees with Lemmas 2.2 and 2.3: Each of the $n$ products $z_j z'_j$ is a Bernoulli random variable with success rate $\mu\mu$ so that their sum is binomially distributed with expected value $n \cdot \mu\mu$.

To obtain a corresponding estimate for the expectation of the inner product of two random bipolar vectors $s, s' \in \{\pm 1\}^n$, we consider them to be transformed versions of i.i.d. Bernoulli vectors such that $p(s_j, s'_j) = p(s_j)\, p(s'_j)$ and, crucially

$$p(s_j = s) = p(s'_j = s) = 2 \cdot f_{\text{Ber}}(z \mid \mu) - 1 \ . \tag{2.65}$$

It is then a straightforward exercise for the reader to verify the following result

$$\mathbb{E}\big[s^\intercal s'\big] = \sum_{j=1}^{n} \mathbb{E}\big[s_j\big]\,\mathbb{E}\big[s'_j\big] = \sum_{j=1}^{n} \big(2\,\mu - 1\big)^2 = n\,\big(2\,\mu - 1\big)^2 \ . \tag{2.66}$$

Now, to compare the behavior of $n\mu^2$ and $n(2\,\mu - 1)^2$, we note that truly random binary- and bipolar vectors should not have any preference for their entries to be 0 or 1 and $-1$ or $+1$, respectively. Put differently, entries of truly random binary- and bipolar vectors should be independent (transformed) Bernoulli variables with success probabilities $\mu = \frac{1}{2}$. For these, we find

$$\mathbb{E}\big[z^\intercal z'\big] = n\,\big(\tfrac{1}{2}\big)^2 = \frac{n}{4} > 0 \tag{2.67}$$

$$\mathbb{E}\big[s^\intercal s'\big] = n\,\big(2\tfrac{1}{2} - 1\big)^2 = 0 \tag{2.68}$$

and therefore realize that two random bipolar vectors can always be expected to be orthogonal regardless of their dimensionality. Two random binary vectors, on the other hand, will become less orthogonal the more their dimensionality grows.

Next, we return to the issue of visualizing binary- and bipolar hypercubes. Without loss of generality, we shall focus on binary hypercubes which we could always affinely transform into bipolar ones.

Observe that we may think of a vector $z = [z_1, z_2, \ldots, z_n]^\intercal \in \{0, 1\}^n$ as a 1D array or ordered sequence of the numbers 0 and 1. We can thus equivalently represent it as a **bit string** $\mathtt{z}_1\mathtt{z}_2 \ldots \mathtt{z}_n$ composed of the digits `0` and `1`. In Fig. 2.2, we use such bit strings to label the $2^3 = 8$ corners of the binary cube in $\mathbb{R}^3$.

Figure 2.3 shows the same three-dimensional cube undergoing a sequence of two rotations leading to a different perspective on the cube. Letting $e_1, e_2, e_3$ denote the

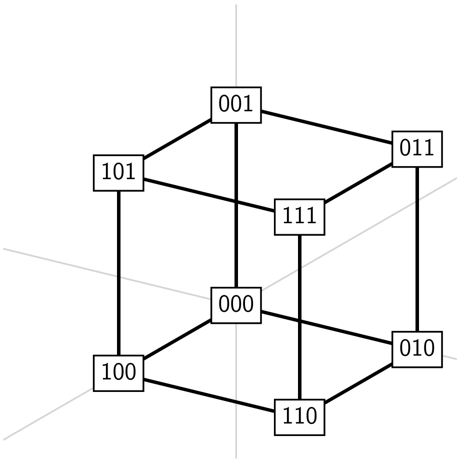


**Fig. 2.2** Another visualization of the binary hypercube in $\mathbb{R}^3$

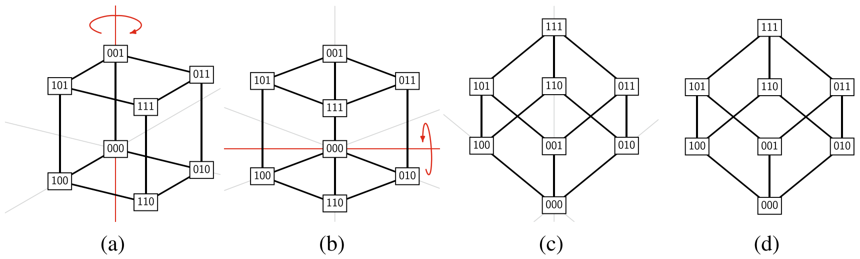


**Fig. 2.3** Two consecutive rotations produce a different view of the binary cube in $\mathbb{R}^3$. This new view immediately leads to a more abstract representation in terms of a graph. **a** Rotation about axis $\boldsymbol{e}_3$. **b** Rotation about axis $\frac{\boldsymbol{e}_1+\boldsymbol{e}_2}{\sqrt{2}}$. **c** Resulting view. **d** Corresponding graph representation

three standard basis vectors of $\mathbb{R}^3$, we first rotate the cube by an angle of $\pi/4$ about the axis spanned by $\boldsymbol{e}_3$ and second by an angle of $\pi/2 - \arccos\left(2/3\right)^{1/2}$ about the axis spanned by $1/\sqrt{2}\,[\boldsymbol{e}_1 + \boldsymbol{e}_2]$. The result is a view of the cube where its corners 000 and 111 are now both aligned with the $\boldsymbol{e}_3$-axis and reside at the bottom and top of the picture, respectively.

If we now drop the coordinate axes from this visualization, we obtain a more abstract representation of the $n = 3$-dimensional binary cube, namely a representation in terms of a *graph*.

This graph consists of $2^n$ labeled *vertices* and $n\,2^{n-1}$ *edges*. For each vertex, there are $n$ incident edges so that each vertex has $n$ neighbors. We further observe that the **Hamming distance** between the labels of adjacent vertices is always 1. That is, the two bit strings which label two adjacent vertices always differ in exactly only one position. Without going into further details, we finally remark that a graph with all these properties is also called a **Boolean lattice**.

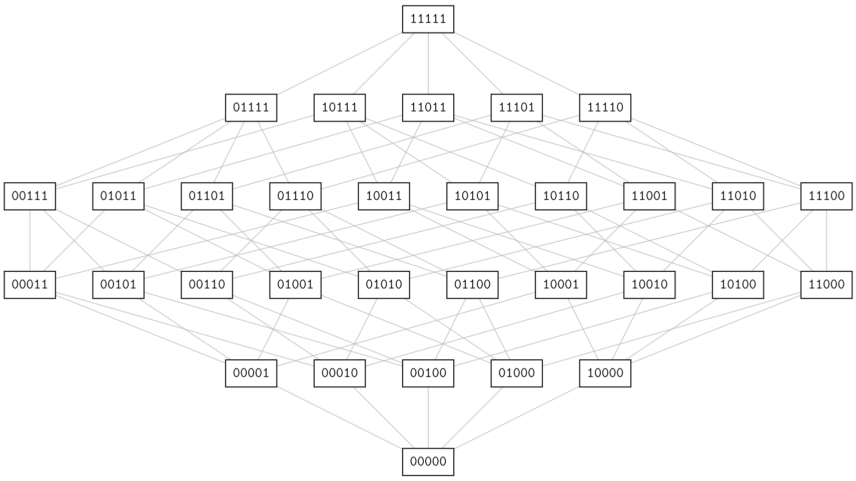


**Fig. 2.4** Lattice representation of the $n = 5$-dimensional binary hypercube

Working with graph representations like these, we assume a *topological* rather than a *geometrical* view on binary hypercubes. That is, instead of focusing on *where* binary vectors $z \in \{0, 1\}^n$ are situated in $\mathbb{R}^n$, we focus on *how* they are interrelated. An upshot of this relational point of view is that it allows for visualizing higher dimensional cubes. Figure 2.4 exemplifies this by showing a lattice visualization of the $n = 5$-dimensional binary hypercube.

In fact, these kind of representations or visualizations are not restricted to binary cubes but also apply to their bipolar analogues with corners $s \in \{\pm 1\}^n \subset \mathbb{R}^n$. This can be seen in Fig. 2.5 which shows a visualization of the $n = 5$-dimensional bipolar cube. Observe that we labeled the vertices of this graph using strings over the two symbols **-** and **+** which we henceforth often use as an alternative notation of the numbers $-1$ and $+1$.

## 2.5.2 *Enumeration of Binary and Bipolar Vectors*

There is yet another view on binary- and bipolar vectors. Without loss of generality, we next again focus on binary vectors.

Above, we emphasized that we can identify binary vectors $z = [z_1, z_2, \ldots, z_n]^\intercal$ with bit strings $\mathtt{z}_1\mathtt{z}_2 \ldots \mathtt{z}_n$. Computer scientists know that these are commonly used as binary representations of natural numbers $0 \leq x \leq 2^n - 1$. But if binary vectors can be identified with bit strings which can be identified with natural numbers, then binary vectors can be directly identified with natural numbers.

Note that there are different possible ways for how to associate a binary vector $z \in \{0, 1\}^n$ with a natural number $x \in \mathbb{N}$.

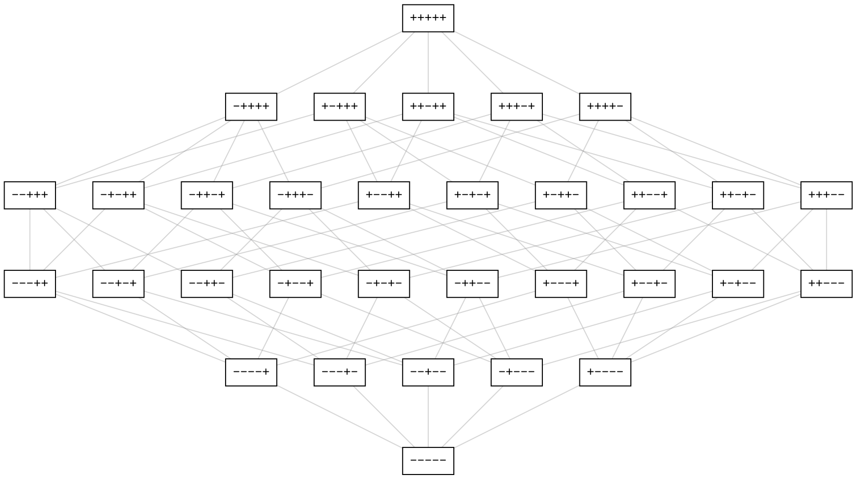
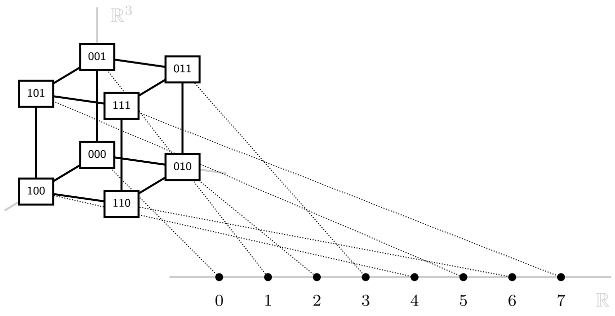


**Fig. 2.5** Lattice representation of the $n = 5$-dimensional bipolar hypercube

| $z_1$ | $z_2$ | $z_3$ | $x$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 2 |
| 0 | 1 | 1 | 3 |
| 1 | 0 | 0 | 4 |
| 1 | 0 | 1 | 5 |
| 1 | 1 | 0 | 6 |
| 1 | 1 | 1 | 7 |

(a)



(b)

**Fig. 2.6** Correspondence between vectors $\mathbf{z} \in \{0, 1\}^3$ and numbers $x = 2^2 z_1 + 2^1 z_2 + 2^0 z_3 \in \mathbb{N}$. **a** Big-endian codes. **b** A mapping between the binary hypercube in $\mathbb{R}^3$ and the real number line

For instance, we may resort to the **big-endian convention** where entries $z_1$ to $z_n$ of $\mathbf{z}$ represent the most to least significant bits of the binary representation of $x$. We thus have

$$x = \sum_{j=1}^{n} 2^{n-j} z_j \tag{2.69}$$

and the table in Fig. 2.6a illustrates this conversion from binary vectors to natural numbers for the case where $n = 3$.

An interesting aspect of this view of binary vectors is that it allows us to arrange the vertices of hypercubes or lattice graphs along the real number line. This provides an ordering of the elements of these discrete structures which arises "naturally"

**Table 2.2** Examples of 3 bit and 8 bit codes for the natural numbers $0 \leq x \leq 7$

| $x$ | Binary code | Gray code | One-Hot Code | Unary code |
|---|---|---|---|---|
| 0 | 000 | 000 | 10000000 | 10000000 |
| 1 | 001 | 001 | 01000000 | 11000000 |
| 2 | 010 | 011 | 00100000 | 11100000 |
| 3 | 011 | 010 | 00010000 | 11110000 |
| 4 | 100 | 110 | 00001000 | 11111000 |
| 5 | 101 | 111 | 00000100 | 11111100 |
| 6 | 110 | 101 | 00000010 | 11111110 |
| 7 | 111 | 100 | 00000001 | 11111111 |

from their relation to natural numbers. Note, however, that corresponding mappings between $\{0, 1\}^n$ and $\mathbb{N} \subset \mathbb{R}$ are anything but linear as can be seen in Fig. 2.6b.

For completeness, we finally mention that there are many other ideas for how to encode natural numbers $0 \leq x \leq 2^n - 1$ as binary vectors or bit strings. Such codes commonly involve either $n$ or $2^n$ vector entries or string elements and Table 2.2 illustrates but a few examples for the case where $n = 3$. Since these examples are rather self-explanatory, we will not discuss them any further. Nevertheless, we point out that especially *one-hot encodings* will be of interest later on.

### 2.5.3 Logic with Binary and Bipolar Vectors

Next, we return to the Boolean functions in Table 2.1 and (re)write them in terms of binary- and bipolar vectors.

In Eqs. (2.7)–(2.10), we already expressed AND, OR, XOR, and NAND as analytic functions $f : \{0, 1\}^2 \to \{0, 1\}$. Here are these expressions again, this time in a more verbose or more elaborate form:

$$\texttt{AND}_z(z_1, z_2) = 0 \cdot 1 + 0 \cdot z_1 + 0 \cdot z_2 + 1 \cdot z_1 z_2 \tag{2.70}$$

$$\texttt{OR}_z(z_1, z_2) = 0 \cdot 1 + 1 \cdot z_1 + 1 \cdot z_2 - 1 \cdot z_1 z_2 \tag{2.71}$$

$$\texttt{XOR}_z(z_1, z_2) = 0 \cdot 1 + 1 \cdot z_1 + 1 \cdot z_2 - 2 \cdot z_1 z_2 \tag{2.72}$$

$$\texttt{NAND}_z(z_1, z_2) = 1 \cdot 1 + 0 \cdot z_1 + 0 \cdot z_2 - 1 \cdot z_1 z_2 \,. \tag{2.73}$$

We further claimed that it is possible to express AND, OR, XOR, and NAND as analytic functions $f : \{\pm 1\}^2 \to \{\pm 1\}$ but only showed this for the case of AND. Here, we now show this for all four functions using elaborate forms similar to the above:

$$\begin{aligned}
\mathtt{AND}_s(s_1, s_2) &= -\tfrac{1}{2} \cdot 1 + \tfrac{1}{2} \cdot s_1 + \tfrac{1}{2} \cdot s_2 + \tfrac{1}{2} \cdot s_1 s_2 && (2.74)\\
\mathtt{OR}_s(s_1, s_2) &= +\tfrac{1}{2} \cdot 1 + \tfrac{1}{2} \cdot s_1 + \tfrac{1}{2} \cdot s_2 - \tfrac{1}{2} \cdot s_1 s_2 && (2.75)\\
\mathtt{XOR}_s(s_1, s_2) &= 0 \cdot 1 + 0 \cdot s_1 + 0 \cdot s_2 - 1 \cdot s_1 s_2 && (2.76)\\
\mathtt{NAND}_s(s_1, s_2) &= +\tfrac{1}{2} \cdot 1 - \tfrac{1}{2} \cdot s_1 - \tfrac{1}{2} \cdot s_2 - \tfrac{1}{2} \cdot s_1 s_2 \, . && (2.77)
\end{aligned}$$

The fundamental insight we can glean from all the above expressions is as follows: Regardless of whether we consider $\mathbb{B} = \{0, 1\}$ or $\mathbb{B} = \{\pm 1\}$, every bivariate Boolean logic function $f : \mathbb{B}^2 \to \mathbb{B}$ can be expressed as a sum of four products.

Put differently, every bivariate Boolean logic function can be written as an inner product of two $2^2 = 4$ dimensional vectors. Regardless of whether we are dealing with the Boolean domain $\mathbb{B} = \{0, 1\}$ or with the Boolean domain $\mathbb{B} = \{\pm 1\}$, we may consider Boolean input vectors

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{B}^2 \tag{2.78}$$

and transform them into, for example, the following Boolean *feature vectors*

$$\boldsymbol{\varphi}(\boldsymbol{x}) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1 x_2 \end{bmatrix} \in \mathbb{B}^4 \, . \tag{2.79}$$

Working with this representation, any Boolean function $f : \mathbb{B}^2 \to \mathbb{B}$ can be written

$$f(\boldsymbol{x}) = \boldsymbol{c}^{\intercal} \boldsymbol{\varphi}(\boldsymbol{x}) \tag{2.80}$$

where the content of the coefficient vector $\boldsymbol{c} \in \mathbb{R}^4$ will depend on the function $f$ at hand. For instance, for $\mathtt{XOR}_z$, we would have $\boldsymbol{c} = [0, 1, 1, -2]^{\intercal}$ and, for $\mathtt{XOR}_s$, we would have $\boldsymbol{c} = [0, 0, 0, -1]^{\intercal}$.

Note that we may enumerate the entries of vector $\boldsymbol{c}$ like this $\boldsymbol{c} = [c_1, c_2, c_3, c_3]^{\intercal}$ or, much more unconventionally, like this $\boldsymbol{c} = [c, c_1, c_2, c_{12}]^{\intercal}$. Working with the latter, we may thus expand the inner product in (2.80) as

$$f(\boldsymbol{x}) = c + c_1\, x_1 + c_2\, x_2 + c_{12}\, x_1 x_2 \, . \tag{2.81}$$

At this point, we can therefore rest assured that everything we are doing here is legitimate because we recognize (2.81) as a *Boolean Fourier expansion*. In fact, if we wanted to, we could prove that every multivariate Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ has such an expansion [1]. What we are currently considering is but the special case of bivariate Boolean functions where $n = 2$.

Dealing with this special case, there is more to be said about (2.81) because we may reorganize the entries of the 4 dimensional vectors $\boldsymbol{c}$ and $\boldsymbol{\varphi}(\boldsymbol{x})$ into $2 \times 2$ matrices, say

$$\boldsymbol{C} = \begin{bmatrix} c & c_2 \\ c_1 & c_{12} \end{bmatrix} \quad \text{and} \quad \boldsymbol{\Phi}(\boldsymbol{x}) = \begin{bmatrix} 1 & x_2 \\ x_1 & x_1 x_2 \end{bmatrix}. \tag{2.82}$$

We may then refer to the entries of these matrices as $C_{jk}$ and $\Phi_{jk}$ to alternatively write the Boolean Fourier expansion in (2.81) as

$$f(\boldsymbol{x}) = \sum_{j=1}^{2} \sum_{k=1}^{2} C_{jk}\, \Phi_{jk} = \big\langle \boldsymbol{C}, \boldsymbol{\Phi}(\boldsymbol{x}) \big\rangle \tag{2.83}$$

which we recognize as the *Frobenius inner product* of the matrices $\boldsymbol{C}$ and $\boldsymbol{\Phi}(\boldsymbol{x})$.

But what is all of this supposed to mean? Aren't these different representations $\boldsymbol{\varphi}(\boldsymbol{x})$ and $\boldsymbol{\Phi}(\boldsymbol{x})$ of the input $\boldsymbol{x}$ of a bivariate Boolean function mere mathematical oddities?

No! They allow us to perform logic within the framework of linear algebra. This is another profound insight for later chapters because quantum computing is all about linear algebra. Both representations further indicate what it generally means to "rethink" computations which are seemingly unrelated to linear algebra in terms of vectors, matrices, or higher order *tensors*. In particular, both exemplify that this generally means to work with much higher dimensional vector spaces. Granted, for what we just discussed, these vector spaces are not that high dimensional but still exponentially larger than the original input space $\mathbb{B}^2 \subset \mathbb{R}^2$.

To be specific, dealing with 2-variate Boolean functions with inputs $\boldsymbol{x} \in \mathbb{B}^2$, we had to consider vectors $\boldsymbol{\varphi}(\boldsymbol{x}) \in \mathbb{B}^{2^2}$ or matrices $\boldsymbol{\Phi}(\boldsymbol{x}) \in \mathbb{B}^{2\times 2}$. But had we considered $n$-variate functions with inputs $\boldsymbol{x} \in \mathbb{B}^n$, we would have had to work with vectors $\boldsymbol{\varphi}(\boldsymbol{x}) \in \mathbb{B}^{2^n}$ or tensors $\boldsymbol{\Phi}(\boldsymbol{x}) \in \mathbb{B}^{2\times 2\times\cdots\times 2}$. This kind of combinatorial explosion is foreshadowing and we will return to it in the next chapter.

## 2.6 Code Examples

In later chapters, we will often and extensively work with the affine transformations

$$s = 2z - 1 \tag{2.84}$$

$$z = \tfrac{1}{2}s + \tfrac{1}{2} \tag{2.85}$$

from Theorem 2.1 which map binary numbers to bipolar numbers and vice versa. They will be so central to most our application examples and corresponding `python` codes that we should look at how to implement them.

Code snippet 2.1 therefore presents two `python` functions `bin2bip` and `bip2bin` which realize these mappings. Both implementations are straightforward and do not need elaboration. We should mention though that this snippet and all out upcoming ones assume *responsible users*. For instance, function `bin2bip` takes

```python
def bin2bip(z):
    '''convert binary z in {0,1} to bipolar s in {-1,+1}'''
    return 2*z - 1

def bip2bin(s):
    '''convert bipolar s in {-1,+1} to binary z in {0,1}'''
    return (s+1) / 2
```

**Code 2.1** Plain vanilla python implementations of the two affine transformations $s = 2z - 1$ and $z = \frac{s+1}{2}$ which map from $\{0, 1\}$ to $\{\pm 1\}$ and from $\{\pm 1\}$ to $\{0, 1\}$, respectively.

```python
import numpy as np
rng = np.random.default_rng()

def bernoulli(mu, N):
    return rng.binomial(1, mu, N)
```

**Code 2.2** A Simple numpy function for drawing a random sample of $N$ independent Bernoulli variables, each with an identical success rate $\mu$.

an argument z which we suppose to be either 0 or 1 of type int or float. We neither perform type checking nor exception handling but simply trust that users pass appropriate values.

Many of our upcoming application examples will also require us to sample from Bernoulli distributions. Let us therefore look at how to produce a random *sample*

$$\mathcal{Z} = \left\{z_1, z_2, \ldots, z_N\right\} \tag{2.86}$$

of $N$ binary numbers $z_j \sim f_B\bigl(z \mid \mu\bigr)$ where $\mu$ is a given, constant success parameter.

Code snippet 2.2 defines a function bernoulli which accomplishes this using functionalities available in numpy's random module. We therefore point out that, for this code and all that follows to work, we have to import numpy as np.

Although np.random does not provide a function for immediately sampling from Bernoulli distributions, it ships with a function binomial which we can harness for our purpose. To see why and how this might work, we resort to Exercise 2.7 which establishes that the binomial distribution $f_{\text{Bin}}(k \mid n, \mu)$ with $n = 1$ coincides with the Bernoulli distribution $f_{\text{Ber}}(z \mid \mu)$.

In code snippet 2.2, we thus call binomial with its first parameter set to 1 to turn it into a Bernoulli sampler. Its second and third parameters mu and N indicate the desired success probability $\mu$ and number $N$ of samples to be drawn. We further note that we invoke binomial according to the currently suggested numpy dialect. That is, we instantiate a random number generator rng and use binomial as a method of this generator.

Hence, we may now use, say, smplZ = bernoulli (mu=0.8,N=10) to obtain a 1D array of $N = 10$ randomly drawn integers 0 or 1. Using print (smplZ), we should then see results like this:

```
[1 0 1 1 1 1 0 0 1 1]
```

Note that the fractions of `1`s in this example is 0.7 whereas we would *expect* it to coincide with the successes rate $\mu = 0.8$.

We therefore emphasize that our code involves (computer generated) randomness so that different runs will likely produce different results in which the fraction of `1`s will fluctuate about the expected value. We also emphasize that sample sizes matter in applied statistics. Indeed, if we repeat our sampling procedure with larger sample sizes, we should find the fluctuations to become smaller on average.

To verify this empirically, we may work with `numpy`'s function `mean`. For instance, the following snippet considers different sample sizes $N$. For each $N$, it draws 100 samples and prints the mean of the 100 sample means.

```
for N in [10, 100, 1000, 10000]:
    print (np.mean( [np.mean(bernoulli(0.8, N)) for _ in range(100)] ))
```

While this snippet again involves (computer generated) randomness, a typical run should show that, the larger $N$, the closer the printed number to $\mu = 0.8$. Readers are encouraged to experiment with this code and to consider even larger samples sizes or to replace `mean` by `var` to compute sample variances instead of sample means.

One more thing: Array `smplZ` as produced above is an array of binary numbers. We may actually use the following to turn it into an array `smplS` of bipolar numbers.

```
smplS = bin2bip(smplZ)
```

This works because of `numpy`'s broadcasting capabilities. Although our function `bin2bip` only involves plain vanilla `python` code, we defined it in a manner which can handle `numpy` arrays in its argument. That is, if `numpy` has been imported and an array is passed to `bin2bip`, then the operation $2z - 1$ will be applied to each individual array element and a correspondingly transformed array will be returned. Indeed, if we issue `print (smplS)`, we get:

```
[ 1 -1  1  1  1  1 -1 -1  1  1]
```

Later on, we will frequently use this capability to create random bipolar vectors $\boldsymbol{s} = [s_1, s_2, \ldots, s_n]^\intercal \in \{\pm 1\}^n$ whose entries are Bernoulli distributed.

## 2.7 Exercises

**2.1** Consider a complex number $x \in \mathbb{C}$. Define a negation `NOT` such that $\{-x, +x\}$ becomes a Boolean domain.

**2.2** Consider a complex number $x \in \mathbb{C} \setminus \{0\}$ and its complex conjugate $x^*$. Define a negation `NOT` such that $\{x^*, x\}$ becomes a Boolean domain.

**2.3** Let $f : \mathbb{R} \to \mathbb{R}$ be a function of a discrete random variable $X$ whose possible values $x \in \mathbb{R}$ are distributed according to a probability $p(x)$. The *expected value* of such a function under such a distribution is defined as

$$\mathbb{E}\big[f(x)\big] = \sum_x f(x)\, p(x) \;. \tag{2.87}$$

Show that the expected value of a constant function $f(x) = c$ simply amounts to

$$\mathbb{E}\big[c\big] = c \;. \tag{2.88}$$

Let $g$ be yet another function of $X$ and $a, b \in \mathbb{R}$ be two constants. Show that the expectation operator $\mathbb{E}[\,]$ in (2.87) is a *linear operator*. That is, show that

$$\mathbb{E}\big[a \cdot f(x) + b \cdot g(x)\big] = a \cdot \mathbb{E}\big[f(x)\big] + b \cdot \mathbb{E}\big[g(x)\big] \;. \tag{2.89}$$

**2.4** Let $f(x)$ and $p(x)$ be as above. Show that the expectation operator $\mathbb{E}[\,]$ in (2.87) is an *idempotent operator*. That is, show that

$$\mathbb{E}\Big[\mathbb{E}\big[f(x)\big]\Big] = \mathbb{E}\big[f(x)\big] \;. \tag{2.90}$$

**2.5** For a function $f$ of two discrete random variables $X$ and $Y$ with possible values $x, y \in \mathbb{R}$ and joint probability $p(x, y)$, the definition in (2.87) generalizes as follows

$$\mathbb{E}\big[f(x, y)\big] = \sum_x \sum_y f(x, y)\, p(x, y) \;. \tag{2.91}$$

Consider the specific function $f(x, y) = x \cdot y$. Prove that, if $X$ and $Y$ are independent such that $p(x, y) = p(x) \cdot p(y)$, then

$$\mathbb{E}\big[x \cdot y\big] = \mathbb{E}\big[x\big] \cdot \mathbb{E}\big[y\big] \;. \tag{2.92}$$

**2.6** Show that the product of $n > 2$ independent Bernoulli variables is yet another Bernoulli variable.

**2.7** Consider the binomial distribution $f_{\text{Bin}}(k \mid n, \mu)$ for the case where $n = 1$, i.e. consider $f_{\text{Bin}}(k \mid 1, \mu)$. Show that this is the Bernoulli distribution $f_{\text{Ber}}(z \mid \mu)$.

**2.8** Empirically verify the theoretical results in Eqs. (2.67) and (2.68). That is write `numpy` code that creates random samples $\big\{(\boldsymbol{z}_j, \boldsymbol{z}'_j)\big\}_{j=1}^N$ and $\big\{(\boldsymbol{s}_j, \boldsymbol{s}'_j)\big\}_{j=1}^N$ of pairs of binary vectors $\boldsymbol{z}_j, \boldsymbol{z}'_j \in \{0, 1\}^n$ and bipolar vectors $\boldsymbol{s}_j, \boldsymbol{s}'_j \in \{\pm 1\}^n$. For each sample, compute the average value of the inner products of its pairs. Do this for $n \in \{10, 20, \ldots, 100, 200, \ldots, 1000, 2000, \ldots 10000\}$ and plot your results. Experiment with different choices of $N$, say, 100, 1000, or 10000. Ponder what you observe.

**2.9** Function `bernoulli` in code snippet 2.2 produces a sample of $N$ i.i.d. Bernoulli numbers $z_j$. Extend it such that it can also draw independent but not necessarily identically distributed numbers. That is, extend it such that each $z_j$ can have its own success rate $\mu_j$.

**2.10** Implement a `numpy` function that takes a number $x \in \mathbb{N}$ as input and returns the number $n \in \mathbb{N}$ of bits required for its binary representation according to the big-endian convention in (2.69).

**2.11** Implement a `numpy` function that takes a natural number $x \in \mathbb{N}$ as input and returns a binary vector $\boldsymbol{z} \in \{0, 1\}^n$ that represents $x$ according to the big-endian convention in (2.69).

**2.12** Table 2.2 shows 3 bit Gray codes of the natural numbers $0, 1, \ldots, 7$. Think about the principle behind these codes and implement a `numpy` function that takes an arbitrary natural number $x$ as input and returns a vector $\boldsymbol{z} \in \{0, 1\}^n$ representing the corresponding Gray code.

**2.13** The *Frobenius inner product* of two real matrices $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m \times n}$ is defined as

$$\langle \boldsymbol{A}, \boldsymbol{B} \rangle = \sum_{j=1}^{m} \sum_{k=1}^{n} A_{jk} B_{jk} = \sum_{j=1}^{m} \sum_{k=1}^{n} B_{jk} A_{jk} = \langle \boldsymbol{B}, \boldsymbol{A} \rangle \, . \tag{2.93}$$

Further observe that the operator tr[] computes the *trace* of a square matrix $\boldsymbol{M} \in \mathbb{R}^{l \times l}$

$$\mathrm{tr}\big[\boldsymbol{M}\big] = \sum_{j=1}^{l} M_{jj} \, . \tag{2.94}$$

Use the above to prove the following identities

$$\langle \boldsymbol{A}, \boldsymbol{B} \rangle = \mathrm{tr}\big[\boldsymbol{A}^\intercal \boldsymbol{B}\big] = \mathrm{tr}\big[\boldsymbol{A} \boldsymbol{B}^\intercal\big] \, . \tag{2.95}$$

**2.14** Let $\boldsymbol{u} \in \mathbb{R}^m$ and $\boldsymbol{v} \in \mathbb{R}^n$ be two vectors. Their *outer product* $\boldsymbol{u}\boldsymbol{v}^\intercal = \boldsymbol{W}$ produces a matrix whose entries are given by $u_j \cdot v_k = w_{jk}$. Now, consider yet another matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and prove the following identity

$$\langle \boldsymbol{A}, \boldsymbol{u}\boldsymbol{v}^\intercal \rangle = \boldsymbol{u}^\intercal \boldsymbol{A} \, \boldsymbol{v} \, . \tag{2.96}$$

**2.15** Reconsider Eq. (2.83) and use the result in Eq. (2.96) of the previous exercise to implement two `numpy` functions which compute the following binary- and bipolar `XOR` functions

$$\mathtt{XOR}_z\big(z_1, z_2\big) = \boldsymbol{z}_1^\intercal \boldsymbol{C} \boldsymbol{z}_2 \tag{2.97}$$

$$\mathtt{XOR}_s\big(s_1, s_2\big) = \boldsymbol{s}_1^\intercal \boldsymbol{C} \boldsymbol{s}_2 \, . \tag{2.98}$$

Note that the entries of the respective matrices $\boldsymbol{C}$ follow from (2.72) and (2.76) and that the vectors $\boldsymbol{z}_j$ and $\boldsymbol{s}_j$ are given by

$$\boldsymbol{z}_j = \left[1, z_j\right]^\intercal \in \{0, 1\}^2 \tag{2.99}$$

$$\boldsymbol{s}_j = \left[1, s_j\right]^\intercal \in \{\pm 1\}^2 \,. \tag{2.100}$$

## References

1. O'Donnell, R.: Analysis of Boolean Functions. Cambridge University Press (2014)
2. Post, E.: The Two-Valued Iterative Systems of Mathematical Logic. Princeton University Press (1942)
3. Smith, P.: An Introduction to Formal Logic. Cambridge University Press (2003)
4. Bishop, C.: Pattern Recognition and Machine Learning. Springer (2006)

# Chapter 3
# Kronecker Products

## 3.1 Introduction

In this chapter, we introduce the notion of *Kronecker products* and discuss their algebraic properties. Kronecker products play a pivotal role in quantum computing but they are of even broader interest to us as we will also use them for modeling.

For now, we focus on the modeling aspect and note that we tacitly already worked with vectors resulting from Kronecker products. At the end of Chap. 2, we were concerned with bivariate Boolean operators and modeled them as binary- or bipolar functions $f : \mathbb{B}^2 \to \mathbb{B}$ where

$$\mathbb{B} \in \Bigl\{ \{0, 1\}, \{\pm 1\} \Bigr\} . \tag{3.1}$$

We saw that the functions we considered there can be computed as inner products

$$f(\boldsymbol{x}) = \boldsymbol{c}^\intercal \boldsymbol{\varphi}(\boldsymbol{x}) \tag{3.2}$$

which involved function-dependent coefficients $\boldsymbol{c} \in \mathbb{R}^4$ and a function-independent feature vector representation of the input variable $\boldsymbol{x} = [x_1, x_2]^\intercal \in \mathbb{B}^2$, namely

$$\boldsymbol{\varphi}(\boldsymbol{x}) = \bigl[1, x_1, x_2, x_1 x_2\bigr]^\intercal \in \mathbb{B}^4 . \tag{3.3}$$

In other words, we saw how to compute lower dimensional functions in terms of simple linear operations in higher dimensional spaces. For readers with a background in machine learning, this will be reminiscent of working with reproducing kernel Hilbert spaces [1]. However, the higher dimensional spaces we are concerned with in this book are different in that they are *tensor product spaces*.

Indeed, the "high" dimensional vector in (3.3) results from a tensor product or, more specifically, from a Kronecker product of two lower dimensional vectors.

C. Bauckhage and R. Sifa, *Quantum Computing from Hopfield Nets*,
Cognitive Technologies, https://doi.org/10.1007/978-3-031-99402-9_3

We next substantiate this statement and its implications. To guide our discussion, we stick with the use case of modeling Boolean functions. We begin by widening our perspective once again and generalize what we already know about those.

## 3.2 Pseudo Boolean Functions

The following definition generalizes the notion of Boolean functions and will be of interest throughout.

**Definition 3.1** An $n$-variate or $n$-ary **pseudo Boolean function**

$$f : \mathbb{B}^n \rightarrow \mathbb{R} \tag{3.4}$$

maps from an $n$-dimensional Boolean domain to the real numbers.

Looking at this definition, we observe that the family of pseudo Boolean functions $f : \mathbb{B}^n \rightarrow \mathbb{R}$ subsumes the family $f : \mathbb{B}^n \rightarrow \mathbb{B}$ of Boolean functions because $\mathbb{B} \subset \mathbb{R}$.

Second, we shall simply claim without proof that every pseudo Boolean function $f(\boldsymbol{x}) = f(x_1, \ldots, x_n)$ has a unique *Boolean Fourier expansion* [2]. A more user friendly way of stating this result is to say that every pseudo Boolean function can be written as a multi-linear polynomial

$$f(x_1, \ldots, x_n) = c + \sum_{j} c_j x_j + \sum_{j<k} c_{jk} x_j x_k + \sum_{j<k<l} c_{jkl} x_j x_k x_l + \cdots \tag{3.5}$$

For instance, for the simple case where $n = 2$, the above expression simplifies to

$$f(x_1, x_2) = c + c_1 x_1 + c_2 x_2 + c_{12} x_1 x_2 \,. \tag{3.6}$$

Next, we will rewrite the right hand side of this last equation using an idea that drops out of the blue sky. Alas, it is what it is; this idea applies an *inspired* observation and does not logically follow from anything we have discussed so far.

Consider this: Since $x^0 = 1$ and $x^1 = x$ for all $x \in \mathbb{R}$, we may first rename the four coefficients in (3.6) and then rewrite the whole equation as

$$f(x_1, x_2) = c_{00}\, x_1^0 x_2^0 + c_{01}\, x_1^0 x_2^1 + c_{10}\, x_1^1 x_2^0 + c_{11}\, x_1^1 x_2^1 = \sum_{j=0}^{1} \sum_{k=0}^{1} c_{jk}\, x_1^j x_2^k \,. \tag{3.7}$$

Readers are very much encouraged to ponder this “index heavy” representation of bivariate pseudo Boolean functions and to convince themselves that right hand sides of (3.6) and (3.7) are indeed equivalent.

For the slightly more involved case with $n = 3$ variables, we would have similarly found

$$f(x_1, x_2, x_3) = \sum_{j=0}^{1} \sum_{k=0}^{1} \sum_{l=0}^{1} c_{jkl}\, x_1^j x_2^k x_3^l \tag{3.8}$$

and, for a general $n$, we would have obtained the following rather unwieldy expression

$$f(x_1, x_2, \ldots, x_n) = \sum_{j_1=0}^{1} \sum_{j_2=0}^{1} \cdots \sum_{j_n=0}^{1} c_{j_1 j_2 \cdots j_n}\, x_1^{j_1} x_2^{j_2} \cdots x_n^{j_n} \,. \tag{3.9}$$

### 3.2.1 Tensors

The fundamental observation at this point is that our admittedly involved way of rewriting pseudo Boolean polynomials involves two *arrays* of numbers.

We shall call them $\boldsymbol{C}$ and $\boldsymbol{\Phi}$ to signify *coefficients* and *features*. Their entries are

$$\big[\boldsymbol{C}\big]_{j_1 j_2 \cdots j_n} = c_{j_1 j_2 \cdots j_n} \quad \text{and} \quad \big[\boldsymbol{\Phi}\big]_{j_1 j_2 \cdots j_n} = x_1^{j_1} x_2^{j_2} \cdots x_n^{j_n} \tag{3.10}$$

and consist of real numbers for the former and (products) of Boolean numbers for the latter which of course are real numbers, too. Since all the $n$ indices of both arrays range from 0 to 1 and thus can only assume two values, we further realize that both arrays are of size

$$\underbrace{2 \times 2 \times \cdots \times 2}_{n \text{ times}} \tag{3.11}$$

In other words, we are dealing with

$$\boldsymbol{C} \in \mathbb{R}^{2 \times 2 \times \cdots \times 2} \tag{3.12}$$

$$\boldsymbol{\Phi} \in \mathbb{B}^{2 \times 2 \times \cdots \times 2} \tag{3.13}$$

and observe that both arrays contain $2^n$ numbers and are therefore exponentially large in the number $n$ of parameters of the function $f$ they represent.

In mathematics and physics, such multi-indexed arrays of numbers are commonly called **higher-order tensors**. Tensors with one index are *first order tensors* or *vectors*, tensors with two indices are *second order tensors* or *matrices*, tensors with three indices are *third order tensors*, etc. While this terminology may sound intimidating, there is nothing to worry about. For example, we just saw that instances of higher order tensors naturally arise when modeling pseudo Boolean functions. In a sense tensors are really nothing but numerical $n$-way arrays and might be familiar to those who have coding experience with numerical computing packages such as `numpy` [3] or with machine learning toolboxes such as `TensorFlow`, or `PyTorch` [4, 5].

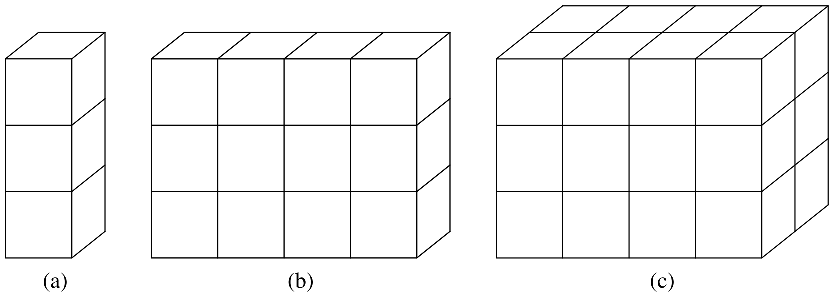


**Fig. 3.1** Illustrations of $n$-way arrays or $n$-th order tensors. **a** 1-way array or 1st order tensor of size 3. **b** 2-way array or 2nd order tensor of size $3 \times 4$. **c** 3-way array or 3rd order tensor of size $3 \times 4 \times 2$. Elements of these tensors can be indexed using one, two, and three indices, respectively

Our next crucial observation might also be familiar to those who have coded with $n$-way arrays before. We simply point out that higher order arrays can be flattened or, equivalently, that higher order tensors can be vectorized. To illustrate this, we return to the simple case in (3.7) where $n = 2$ (Fig. 3.1b).

In Eq. (3.7), we are dealing with two second order tensors whose matrix representations read

$$\boldsymbol{C} = \begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} \quad \text{and} \quad \boldsymbol{\Phi} = \begin{bmatrix} x_1^0 x_2^0 & x_1^0 x_2^1 \\ x_1^1 x_2^0 & x_1^1 x_2^1 \end{bmatrix} = \begin{bmatrix} 1 & x_2 \\ x_1 & x_1 x_2 \end{bmatrix}. \tag{3.14}$$

Given these $2 \times 2$ matrices, we can now, first of all, easily recognize the polynomial in (3.7) as a Frobenius inner product $f(x_1, x_2) = \langle \boldsymbol{C}, \boldsymbol{\Phi} \rangle$.

Second of all, we can flatten both these matrices either row- or column-wise. For matrix $\boldsymbol{\Phi}$, for example, this will produce the vectors

$$\boldsymbol{\varphi}_r = \left[1, x_2, x_1, x_1 x_2\right]^\intercal \tag{3.15}$$

$$\boldsymbol{\varphi}_c = \left[1, x_1, x_2, x_1 x_2\right]^\intercal . \tag{3.16}$$

Note the *ambiguity*! The two first order tensors $\boldsymbol{\varphi}_r$ and $\boldsymbol{\varphi}_c$ are *both* lower-order representations of the second order tensor $\boldsymbol{\Phi}$. Both vectors $\boldsymbol{\varphi}_r$ and $\boldsymbol{\varphi}_c$ gather the entries of matrix $\boldsymbol{\Phi}$ but do so in a (slightly) different order. Both vectors $\boldsymbol{\varphi}_r$ and $\boldsymbol{\varphi}_c$ are thus permuted versions of each other and therefore *different*.

However, and this is crucial, with respect to the task of modeling a pseudo Boolean function $f(x_1, x_2)$, they are *equivalent*.

This is because both vectors $\boldsymbol{\varphi}_r$ and $\boldsymbol{\varphi}_c$ allow us to compute $f(x_1, x_2)$ in terms of inner products

$$f(x_1, x_2) = \boldsymbol{c}_r^\intercal \boldsymbol{\varphi}_r = \boldsymbol{c}_c^\intercal \boldsymbol{\varphi}_c \tag{3.17}$$

where $\boldsymbol{c}_r$ and $\boldsymbol{c}_c$ are correspondingly vectorized versions of the respective coefficient matrix $\boldsymbol{C}$.

Note that *the ambiguity runs deeper*! What we are currently looking at is the idea of expressing polynomials in terms of inner products between (higher-order) tensors. While polynomials do not care about the order in which we write their terms, tensors do care about the order in which we arrange their entries. That is, if we gather polynomial coefficients and variables in (higher-order) tensors to represent polynomials in terms of (higher-order) inner products, then order matters because numerical $n$-way arrays are *ordered* or specifically arranged collections of numbers.

For example, the following are all different permutations of the above matrix $\boldsymbol{\Phi}$

$$\begin{bmatrix} 1 & x_2 \\ x_1 & x_1x_2 \end{bmatrix} \neq \begin{bmatrix} 1 & x_1 \\ x_2 & x_1x_2 \end{bmatrix} \neq \begin{bmatrix} x_1 & x_2 \\ 1 & x_1x_2 \end{bmatrix} \neq \cdots \tag{3.18}$$

In fact, there are $4! = 24$ different permutations of $\boldsymbol{\Phi}$ and of $\boldsymbol{C}$ correspondingly. All of these contain all the numbers required to model $f(x_1, x_2)$ as an inner product between matrices. All of these permuted matrices could be flattened to model $f(x_1, x_2)$ as an inner product between vectors.

In short, whenever we opt for mathematical modeling in terms of (higher-order) tensors, we are facing a modeling choice since different models may be *different but equivalent* with respect to the task at hand. Crucially, once we have decided for a certain tensor structure or layout, we must adhere to it in all subsequent mathematical manipulations and downstream processing steps.

How then do we decide for a certain tensor structure? Well, common criteria are “aesthetics”, convenience, or established conventions. Consider, for example the above vectors $\boldsymbol{\varphi}_r$ and $\boldsymbol{\varphi}_c$ and note that the latter is identical to vector $\boldsymbol{\varphi}(\boldsymbol{x})$ from the introductory Eqs. (3.2) and (3.3). Why did these equations involve $\boldsymbol{\varphi}_c$ rather than $\boldsymbol{\varphi}_r$? To be honest, simply because we felt that vector $[1, x_1, x_2, x_1x_2]^\intercal$ looks nicer than vector $[1, x_2, x_1, x_1x_2]^\intercal$. Next, we will see that a decision more in line with common conventions would have been to work with $\boldsymbol{\varphi}_r$ instead; examples of tensor structures motivated by convenience will follow in the next chapter.

We emphasize all this so emphatically because mathematical models of quantum computing involve *tensor product spaces* which leads us to the next subsection.

### 3.2.2 Tensor Products

While there is generally not much to be said about the coefficient tensors of Boolean polynomials, the tensors which represent their variables turn out to have interesting structural properties. To ease into their study, we once again focus on the simple case of bivariate pseudo Boolean functions $f(x_1, x_2)$. For these, we just saw that the following matrix and vector contain all the variables which occur in their polynomial expansion

$$\boldsymbol{\Phi} = \begin{bmatrix} 1 & x_2 \\ x_1 & x_1x_2 \end{bmatrix} \quad \text{and} \quad \boldsymbol{\varphi} = \left[1, x_2, x_1, x_1x_2\right]^\intercal . \tag{3.19}$$

Note that these two objects are *different but equivalent* with respect to the task of modeling $n = 2$ variate Boolean polynomials. The one is an $n \times n$ matrix with entries $[\boldsymbol{\Phi}]_{jk}$ indexed by $j$ and $k$. The other is a $2^n$ vector with entries $[\boldsymbol{\varphi}]_l$ indexed by $l$. But both contain all the variables required for expressing a general polynomial over $x_1, x_2 \in \mathbb{B}$.

Now, consider the following inspired *feature vector representations* $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{B}^2$ of the univariate Boolean variables $x_1$ and $x_2$, namely

$$\boldsymbol{u} = \begin{bmatrix} 1 \\ x_1 \end{bmatrix} \quad \text{and} \quad \boldsymbol{v} = \begin{bmatrix} 1 \\ x_2 \end{bmatrix} . \tag{3.20}$$

While the entries of these vectors are obviously related to the entries of the matrix and vector in (3.19), the relation runs deeper than it may appear at first sight. Indeed, scrutinizing $\boldsymbol{\Phi}$ and $\boldsymbol{\varphi}$, we realize that their entries can be systematically expressed as products of the entries of $\boldsymbol{u}$ and $\boldsymbol{v}$. More specifically we observe

$$\big[\boldsymbol{\Phi}\big]_{jk} = \big[\boldsymbol{u}\big]_j \cdot \big[\boldsymbol{v}\big]_k \qquad \Leftrightarrow \qquad \boldsymbol{\Phi} = \begin{bmatrix} u_1 v_1 & u_1 v_2 \\ u_2 v_1 & u_2 v_2 \end{bmatrix} \tag{3.21}$$

$$\big[\boldsymbol{\varphi}\big]_{2(j-1)+k} = \big[\boldsymbol{u}\big]_j \cdot \big[\boldsymbol{v}\big]_k \qquad \Leftrightarrow \qquad \boldsymbol{\varphi} = \big[u_1 v_1,\ u_1 v_2,\ u_2 v_1,\ u_2 v_2\big]^{\intercal} . \tag{3.22}$$

But this is to say that the second order tensor $\boldsymbol{\Phi}$ is an *outer product* of the two first order tensors $\boldsymbol{u}$ and $\boldsymbol{v}$ and that the first order tensor $\boldsymbol{\varphi}$ is a *Kronecker product* of the two first order tensors $\boldsymbol{u}$ and $\boldsymbol{v}$. In other words, $\boldsymbol{\Phi}$ and $\boldsymbol{\varphi}$ are both tensors which result from products of two other tensors.

Since outer products and Kronecker products both are products of tensors, they both are *tensor products*. However, they are *different instantiations* of a the idea of a tensor product.

We emphasize this because, in much of the literature, it is common to denote all instantiations of tensor products by "$\otimes$". Using this generic notation, we would write the gist of equations (3.21) and (3.22) as

$$\boldsymbol{\Phi} = \boldsymbol{u} \otimes \boldsymbol{v} \tag{3.23}$$

$$\boldsymbol{\varphi} = \boldsymbol{u} \otimes \boldsymbol{v} . \tag{3.24}$$

This is of course confusing. Since the objects on the left hand sides are different kinds of mathematical objects, the objects on the right hand sides must be different, too. But they look alike because we are facing a case of ambiguous notation. This is not uncommon and has usually to be resolved from context. Indeed, while the right hand sides of (3.23) and (3.24) look identical, we know from our preceding discussion that they signify different operations. In (3.23), the operator "$\otimes$" denotes a mapping

$$\otimes : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^{2\times 2} \tag{3.25}$$

whereas in (3.24) it denotes a mapping

$$\otimes : \mathbb{R}^2 \times \mathbb{R}^2 \rightarrow \mathbb{R}^4 . \tag{3.26}$$

For theory minded people, this ambiguity of "$\otimes$" is not a big deal because the most important aspect of the objects on the left of (3.23) and (3.24) is that they are tensors; their internal structure is often of minor concern to theorists. Alas, for practitioners who want to turn mathematics into computer code, such a lofty point of view may have dire consequences. Since we count ourselves among the latter, we therefore need a way to nationally distinguish between outer products and Kronecker products to minimize the risk of faulty implementations of mathematical equations. Fortunately, there already exists another established notation for outer products which we will use from now on.

**Notation**

To denote the *outer product* of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ which produces a matrix, we henceforth write

$$\boldsymbol{u}\boldsymbol{v}^\intercal .$$

To denote the *Kronecker product* of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ which produces a vector, we henceforth write

$$\boldsymbol{u} \otimes \boldsymbol{v} .$$

Given these notational conventions, we better point out immediately that neither product is commutative. That is, we generally have $\boldsymbol{u}\boldsymbol{v}^\intercal \neq \boldsymbol{v}\boldsymbol{u}^\intercal$ as well as $\boldsymbol{u} \otimes \boldsymbol{v} \neq \boldsymbol{v} \otimes \boldsymbol{u}$.

In the remainder of this book, we will sporadically work with outer products and quite frequently consider Kronecker products. We therefore need not look extensively at properties of outer products but shall familiarize ourselves more closely with the characteristics of Kronecker products.

The latter will happen in the subsequent sections. With respect to the former, we simply state the following two theorems whose proofs we omit. They are not that difficult and mainly require careful manipulations of matrix-matrix, matrix-vector, and vector-vector multiplications expressed in terms of the individual components of the respective objects.

**Theorem 3.1** *Let $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ be a matrix and let $\boldsymbol{u} \in \mathbb{R}^m$ and $\boldsymbol{v} \in \mathbb{R}^n$ be two vectors. Then the the following equality holds for the Frobenius inner product of $\boldsymbol{A}$ and $\boldsymbol{u}\boldsymbol{v}^\intercal$*

$$\left\langle \boldsymbol{A}, \boldsymbol{u}\boldsymbol{v}^\intercal \right\rangle = \boldsymbol{u}^\intercal \boldsymbol{A}\, \boldsymbol{v} \,. \tag{3.27}$$

**Theorem 3.2** *Let $\boldsymbol{u} \in \mathbb{R}^m$, $\boldsymbol{v} \in \mathbb{R}^n$, and $\boldsymbol{w} \in \mathbb{R}^n$ be three vectors. Then the expression $\boldsymbol{u}\boldsymbol{v}^\intercal\boldsymbol{w}$ can be read as an outer product matrix acting on a vector or as a vector scaled by an inner product since the following associations hold*

$$\boldsymbol{u}\boldsymbol{v}^\intercal\boldsymbol{w} = \left[\boldsymbol{u}\boldsymbol{v}^\intercal\right]\boldsymbol{w} = \boldsymbol{u}\left[\boldsymbol{v}^\intercal\boldsymbol{w}\right] . \tag{3.28}$$

## 3.3 Kronecker Products of Vectors

Next, we discuss general algebraic properties and specific examples of Kronecker products of vectors. If we wanted to be most general, our discussion would consider complex valued vectors. Indeed, the most common mathematical frameworks behind quantum computing involve Kronecker products of complex vectors. However, for now and for simplicity, we will focus on real valued vectors. Nevertheless, all we say below either applies to complex valued vectors, too, or generalizes correspondingly with only little additional theoretical overheads to be spent.

For our practical examples, we will again stick with Boolean vectors over $\mathbb{B} \subset \mathbb{R}$ and we note that these examples, too, easily extend to more general settings.

**Definition 3.2** The **Kronecker product** of two vectors $\boldsymbol{u} \in \mathbb{R}^m$ and $\boldsymbol{v} \in \mathbb{R}^n$ is a mapping

$$\otimes : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^{m\,n} \tag{3.29}$$

such that

$$\boldsymbol{u} \otimes \boldsymbol{v} \equiv \begin{bmatrix} u_1\, \boldsymbol{v} \\ u_2\, \boldsymbol{v} \\ \vdots \\ u_m\, \boldsymbol{v} \end{bmatrix} = \begin{bmatrix} u_1\, v_1 \\ \vdots \\ u_1\, v_n \\ u_2\, v_1 \\ \vdots \\ u_m\, v_n \end{bmatrix} . \tag{3.30}$$

Given this definition, it makes sense to call the operation $\boldsymbol{u} \otimes \boldsymbol{v}$ a product of vectors as it produces another vector with entries $u_j v_k$ which are products of the entries $u_j$ of $\boldsymbol{u}$ and $v_k$ of $\boldsymbol{v}$. But we must pay attention to the fact that the entries $u_j v_k$ are defined to occur in a specific order. This will necessarily impact the algebra of the Kronecker product and we must be aware of what this means.

Letting $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ be any three vectors and $c$ be a scalar, the following algebraic properties of Kronecker products are easily verified [6].

Most importantly, the peculiar ordering of the $u_j v_k$ causes the Knocker product to be *not commutative*. Unless $\boldsymbol{u} = \boldsymbol{0}$ or $\boldsymbol{v} = \boldsymbol{0}$ or $\boldsymbol{u} = \boldsymbol{v}$, we therefore generally have

$$\boldsymbol{u} \otimes \boldsymbol{v} \neq \boldsymbol{v} \otimes \boldsymbol{u} \ . \tag{3.31}$$

On the other hand, the Kronecker product is guaranteed to be *associative*

$$\boldsymbol{u} \otimes \boldsymbol{v} \otimes \boldsymbol{w} = \boldsymbol{u} \otimes \big[\boldsymbol{v} \otimes \boldsymbol{w}\big] = \big[\boldsymbol{u} \otimes \boldsymbol{v}\big] \otimes \boldsymbol{w} \tag{3.32}$$

as well as *homogeneous under scaling* by which we mean the following

$$c\big[\boldsymbol{u} \otimes \boldsymbol{v}\big] = \big[c\boldsymbol{u}\big] \otimes \boldsymbol{v} = \boldsymbol{u} \otimes \big[c\boldsymbol{v}\big] \ . \tag{3.33}$$

Regarding distributivity, we note that *the Knocker product distributes over addition*

$$\boldsymbol{u} \otimes \big[\boldsymbol{v} + \boldsymbol{w}\big] = \boldsymbol{u} \otimes \boldsymbol{v} + \boldsymbol{u} \otimes \boldsymbol{w} \tag{3.34}$$

$$\big[\boldsymbol{u} + \boldsymbol{v}\big] \otimes \boldsymbol{w} = \boldsymbol{u} \otimes \boldsymbol{w} + \boldsymbol{v} \otimes \boldsymbol{w} \tag{3.35}$$

and that the operation of *vector transposition distributes over the Kronecker product*

$$\big[\boldsymbol{u} \otimes \boldsymbol{v}\big]^\intercal = \boldsymbol{u}^\intercal \otimes \boldsymbol{v}^\intercal \ . \tag{3.36}$$

Based on the above definition and algebra, it is another easy exercise to convince ourselves that the following makes sense.

**Definition 3.3** Let $U$ and $V$ be two vector spaces over the same number field. Then the **tensor product space**

$$U \otimes V = W \tag{3.37}$$

is a vector space of dimension $\dim(U) \cdot \dim(V) = \dim(W)$ whose elements are

$$W = \big\{\boldsymbol{u} \otimes \boldsymbol{v} \;\big|\; \boldsymbol{u} \in U, \boldsymbol{v} \in V\big\} \ . \tag{3.38}$$

Armed with Definition 3.2, we can moreover finally verify our claim from the introduction of this chapter where we said that vector

$$\boldsymbol{\varphi}(\boldsymbol{x}) = \big[1, x_1, x_2, x_1 x_2\big]^{\intercal} \tag{3.39}$$

can be thought of as the result of a Kronecker product of two vectors. To this end, we revisit the vectors from (3.20) for which a more overarching view and more general notation is the following.

Whenever we are dealing with a set $\{x_1, x_2, \ldots, x_n\}$ of Boolean variables $x_j \in \mathbb{B}$, we may *represent* them in terms of two-dimensional vectors

$$\boldsymbol{x}_j = \begin{bmatrix} 1 \\ x_j \end{bmatrix} \in \mathbb{B}^2 \subset \mathbb{R}^2 \,. \tag{3.40}$$

If we now consider the two specific vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ and their specific Kronecker product $\boldsymbol{x}_2 \otimes \boldsymbol{x}_1$, we do indeed find

$$\boldsymbol{x}_2 \otimes \boldsymbol{x}_1 = \begin{bmatrix} 1 \cdot \boldsymbol{x}_1 \\ x_2 \cdot \boldsymbol{x}_1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 \\ 1 \cdot x_1 \\ x_2 \cdot 1 \\ x_2 \cdot x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1 x_2 \end{bmatrix} = \boldsymbol{\varphi}(\boldsymbol{x}) \tag{3.41}$$

and thus confirm our claim. However, note the peculiar order of the factors in this Kronecker product. While the order of the symbolic, indexed entries of the resulting vector looks "nice", the order of the indexed factors looks "strange". Would it not be more conventional to arrange them according to increasing index values? After all, if we write down a function $f$ of two arguments, we usually write $f(x_1, x_2)$ as we did throughout this chapter so far. Hence, let us compute

$$\boldsymbol{x}_1 \otimes \boldsymbol{x}_2 = \begin{bmatrix} 1 \cdot \boldsymbol{x}_2 \\ x_1 \cdot \boldsymbol{x}_2 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 \\ 1 \cdot x_2 \\ x_1 \cdot 1 \\ x_1 \cdot x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ x_2 \\ x_1 \\ x_1 x_2 \end{bmatrix} \neq \boldsymbol{\varphi}(\boldsymbol{x}) \,. \tag{3.42}$$

Now, the order of the factors looks "nice" but the order of the entries of the result looks "strange". Alas, there is no way around this. Since Kronecker products do not commute, we generally have $\boldsymbol{x}_1 \otimes \boldsymbol{x}_2 \neq \boldsymbol{x}_2 \otimes \boldsymbol{x}_1$ and must therefore either live with strange looking factor sequences or with strange looking results. Observe, however, that $\boldsymbol{x}_1 \otimes \boldsymbol{x}_2$ and $\boldsymbol{x}_2 \otimes \boldsymbol{x}_1$ are *different but equivalent* representations of the variable terms of a Boolean polynomial.

We emphasize this ambiguity because, whenever we use Kronecker products as a modeling tool, we face another modeling choice and must decide how to order or to sequence their factors. For most applications, different orders will work equally well with regard to the overall goal. However, once we have decided for a specific order, we must adhere to it in subsequent mathematical reasoning and downstream processing.

Let us further illustrate our "factor sequencing dilemma" by looking at Kronecker products of several factors. Consider, for example, three vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3 \in \mathbb{B}^2$ of the form in (3.40). If we were aiming at a symbolically nice looking result, we would likely work with the product $\boldsymbol{x}_3 \otimes \boldsymbol{x}_2 \otimes \boldsymbol{x}_1$. However, if we were aiming at a conventional or canonical factor sequence, we would work with $\boldsymbol{x}_1 \otimes \boldsymbol{x}_2 \otimes \boldsymbol{x}_3$. For both these choices, readers are very much encouraged to verify the following symbolic computation

$$\boldsymbol{x}_3 \otimes \boldsymbol{x}_2 \otimes \boldsymbol{x}_1 = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1x_2 \\ x_3 \\ x_1x_3 \\ x_2x_3 \\ x_1x_2x_3 \end{bmatrix} \neq \begin{bmatrix} 1 \\ x_3 \\ x_2 \\ x_2x_3 \\ x_1 \\ x_1x_3 \\ x_1x_2 \\ x_1x_2x_3 \end{bmatrix} = \boldsymbol{x}_1 \otimes \boldsymbol{x}_2 \otimes \boldsymbol{x}_3 \; . \tag{3.43}$$

Looking at these results, we may again say that they are *different but equivalent* with respect to the task of representing Boolean functions (of three variables).

What is arguably more interesting about this latest example is that we observe another instance of the phenomenon of exponential growth. In other words, what we first pointed out with respect to Eqs. (3.10)–(3.13) now reoccurs quite visibly in the context of Kronecker products of vectors. While the 3 vectors $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, and $\boldsymbol{x}_3$ are all 2-dimensional, their Kronecker product produces a $2^3 = 8$-dimensional vector. In fact, the following generalized result is easily verified: For $n$ vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ with $\boldsymbol{x}_j \in \mathbb{R}^2$, we have

$$\bigotimes_{j=1}^{n} \boldsymbol{x}_j \in \mathbb{R}^{2^n} \; . \tag{3.44}$$

## 3.4 Kronecker Products of Matrices

Kronecker products generalize from products of first order tensors (vectors) to products of higher order tensors. The most interesting use cases for us involve second order tensors (matrices). For these we may state the following simple definition.

**Definition 3.4** The Kronecker product $\boldsymbol{A} \otimes \boldsymbol{B}$ of two matrices $\boldsymbol{A} \in \mathbb{R}^{k \times l}$ and $\boldsymbol{B} \in \mathbb{R}^{m \times n}$ produces a matrix of size $km \times ln$, namely

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} A_{11}\,\boldsymbol{B} & A_{12}\,\boldsymbol{B} & \cdots & A_{1l}\,\boldsymbol{B} \\ A_{21}\,\boldsymbol{B} & A_{22}\,\boldsymbol{B} & \cdots & A_{2l}\,\boldsymbol{B} \\ \vdots & \vdots & \ddots & \vdots \\ A_{k1}\,\boldsymbol{B} & A_{k2}\,\boldsymbol{B} & \cdots & A_{kl}\,\boldsymbol{B} \end{bmatrix}. \tag{3.45}$$

In addition to all the algebraic properties we listed above, this matrix Kronecker product (sometimes) has another noteworthy characteristic called the *mixed product property*.

In preparation for its formal statement, we note that the matrix product $\boldsymbol{AB}$ of two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ *exists*, if $\boldsymbol{A}$ has as many columns as $\boldsymbol{B}$ has rows.

Given this clarification, we can say: If $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$ are matrices such that the matrix products $\boldsymbol{AC}$ and $\boldsymbol{BD}$ exist, then the following *mixed product property* holds

$$\big[\boldsymbol{A} \otimes \boldsymbol{B}\big]\big[\boldsymbol{C} \otimes \boldsymbol{D}\big] = \boldsymbol{AC} \otimes \boldsymbol{BD}\,. \tag{3.46}$$

Noting that we may view a vector as a matrix of just one column, we have the following special case of the above. If $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{u}$, and $\boldsymbol{v}$ are matrices and vectors such that the two matrix-vector products $\boldsymbol{Au}$ and $\boldsymbol{Bv}$ exists, then the following *mixed product property* holds

$$\big[\boldsymbol{A} \otimes \boldsymbol{B}\big]\big[\boldsymbol{u} \otimes \boldsymbol{v}\big] = \boldsymbol{Au} \otimes \boldsymbol{Bv}\,. \tag{3.47}$$

## 3.5 Kronecker Products of Matrices and Vectors

Kronecker products can furthermore be computed among tensors of different orders. For us, the most interesting use cases are tensor products which involve matrices and vectors. We need not formally define them because we can consider them to be special cases of matrix tensor products. To see why, we note again that we may think of a vector $\boldsymbol{v} \in \mathbb{R}^n$ as a matrix $\boldsymbol{V} \in \mathbb{R}^{n \times 1}$ of just one column. By the same token, we may think of a transposed vector as a matrix of just one row.

Considering a matrix $\boldsymbol{M} \in \mathbb{R}^{l \times m}$ and a vector $\boldsymbol{v} \in \mathbb{R}^n$, we then for instance have

$$\boldsymbol{v} \otimes \boldsymbol{M} = \begin{bmatrix} v_1\,\boldsymbol{M} \\ \vdots \\ v_n\,\boldsymbol{M} \end{bmatrix} \quad \text{and} \quad \boldsymbol{M} \otimes \boldsymbol{v} = \begin{bmatrix} M_{11}\,\boldsymbol{v} & \cdots & M_{1m}\,\boldsymbol{v} \\ \vdots & \ddots & \vdots \\ M_{l1}\,\boldsymbol{v} & \cdots & M_{lm}\,\boldsymbol{v} \end{bmatrix} \tag{3.48}$$

which are both matrices of size $ln \times m$. Yet, because of non-commutativity, we will generally have $\boldsymbol{v} \otimes \boldsymbol{M} \neq \boldsymbol{M} \otimes \boldsymbol{v}$.

We may furthermore and just as well compute

$$\boldsymbol{v}^\intercal \otimes \boldsymbol{M} = \begin{bmatrix} v_1 \boldsymbol{M} & \cdots & v_n \boldsymbol{M} \end{bmatrix} \quad \text{and} \quad \boldsymbol{M} \otimes \boldsymbol{v}^\intercal = \begin{bmatrix} M_{11}\, \boldsymbol{v}^\intercal & \cdots & M_{1m}\, \boldsymbol{v}^\intercal \\ \vdots & \ddots & \vdots \\ M_{l1}\, \boldsymbol{v}^\intercal & \cdots & M_{lm}\, \boldsymbol{v}^\intercal \end{bmatrix} \tag{3.49}$$

to obtain two matrices of the same size $l \times mn$ but with $\boldsymbol{v}^\intercal \otimes \boldsymbol{M} \neq \boldsymbol{M} \otimes \boldsymbol{v}^\intercal$ in general.

Interestingly, given this idea of Kronecker products which involve transposed vectors, we now realize that outer products are nothing but Kronecker products in disguise. Simply consider this example:

$$\boldsymbol{x}_1 \boldsymbol{x}_2^\intercal = \begin{bmatrix} 1 \\ x_1 \end{bmatrix} \begin{bmatrix} 1 & x_2 \end{bmatrix} = \begin{bmatrix} 1 & x_2 \\ x_1 & x_1 x_2 \end{bmatrix} = \boldsymbol{x}_1 \otimes \boldsymbol{x}_2^\intercal \,. \tag{3.50}$$

## 3.6 Code Examples

Outer products and Kronecker products play an important role in many practical applications so that it is nor surprise that `numpy` provides the convenience functions `outer` and `kron` for their computation.

In order to briefly demonstrate how to apply these functions in practice, we first of all need to

```
import numpy as np
```

and, second of all, should define at least two vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ to have objects to work with.

Staying with the spirit of this chapter, we thus consider binary- or bipolar vectors $\boldsymbol{x}_j$ as in (3.40) which we may, for example, implement as 1D row arrays like this

```
x1,x2 = 1,0
vecX1 = np.array([1, x1])
vecX2 = np.array([1, x2])
```

Given these instances of vectors $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \{0, 1\}^2$, we can numerically verify our claim that Kronecker products do generally not commute so that $\boldsymbol{x}_1 \otimes \boldsymbol{x}_2 \neq \boldsymbol{x}_2 \otimes \boldsymbol{x}_1$. Indeed, invoking `kron` like this

```
print (np.kron(vecX1, vecX2))
print (np.kron(vecX2, vecX1))
```

results in two distinct vectors in $\{0, 1\}^4$, namely

```
[1 0 1 0]
[1 1 0 0]
```

Just as easily, we can also verify that outer products do generally not commute so that $\boldsymbol{x}_1\boldsymbol{x}_2^\intercal \neq \boldsymbol{x}_2\boldsymbol{x}_1^\intercal$. Indeed, invoking `outer` like this

```
print (np.outer(vecX1, vecX2))
print (np.outer(vecX2, vecX1))
```

results in two distinct matrices in $\{0, 1\}^{2\times 2}$, namely

```
[[1 0]
 [1 0]]
[[1 1]
 [0 0]]
```

This basically already is all there is to say about the use of `kron` and `outer` and readers are encouraged to repeat the above with bipolar variables `x1,x2 = +1,-1` or to experiment with Kronecker products of matrices or matrices and vectors.

However, just for the fun of it, we will conclude our discussion by verifying claims from Chap. 2, namely that Boolean logic functions $f : \mathbb{B}^2 \to \mathbb{B}$ can be computed using either one of the following inner products between vectors or matrices

$$f(x_1, x_2) = \boldsymbol{c}^\intercal \boldsymbol{\varphi}(x_1, x_2) \tag{3.51}$$

$$f(x_1, x_2) = \big\langle \boldsymbol{C}, \boldsymbol{\Phi}(x_1, x_2)\big\rangle \tag{3.52}$$

where $\boldsymbol{\varphi}(x_1, x_2) = \boldsymbol{x}_1 \otimes \boldsymbol{x}_2$ and $\boldsymbol{\Phi}(x_1, x_2) = \boldsymbol{x}_1\boldsymbol{x}_2^\intercal$. Since vector $\boldsymbol{c}$ and matrix $\boldsymbol{C}$ gather function dependent coefficients, we shall focus on only one such function, say, the binary $\texttt{XOR}_z(x_1, x_2)$ with $x_1, x_2 \in \{0, 1\}$ for which $\boldsymbol{c} = [0, 1, 1, -2]^\intercal$.

Working with (3.51), we may then produce an `XOR` truth table using this snippet

```
vecC = np.array([0, 1, 1, -2])
for x1 in [0,1]:
    for x2 in [0,1]:
        vecX1 = np.array([1, x1])
        vecX2 = np.array([1, x2])
        print (x1, x2, vecC @ np.kron(vecX1, vecX2))
```

which, when executed, prints the following

```
0 0 0
0 1 1
1 0 1
1 1 0
```

Similarly, when working with (3.52), the following ever so slightly more involved snippet

```
matC = vecC.reshape(2,2)
for x1 in [0,1]:
    for x2 in [0,1]:
        vecX1 = np.array([1, x1])
        vecX2 = np.array([1, x2])
        print (x1, x2, np.sum(matC * np.outer(vecX1, vecX2)))
```

produces this output

```
0 0 0
0 1 1
1 0 1
1 1 0
```

Both these results are indeed easily recognized as as the truth tables of the binary $\texttt{XOR}_z$ and readers are very much encouraged to repeat the above for the bipolar $\texttt{XOR}_s$.

## 3.7 Exercises

**3.1** Let $x_j \in \mathbb{B} \in \{\{0, 1\}, \{\pm 1\}\}$ be a Boolean number and represent it as a Boolean vector $\boldsymbol{x}_j = [1, x_j]^\intercal \in \mathbb{B}^2$. Show that every pseudo Boolean function $f : \mathbb{B}^2 \to \mathbb{R}$ can be computed as

$$f(x_1, x_2) = \boldsymbol{x}_1^\intercal \, \boldsymbol{C} \, \boldsymbol{x}_2 \tag{3.53}$$

where $\boldsymbol{C} \in \mathbb{R}^{2\times 2}$ is an appropriate coefficient matrix.

**3.2** Let $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^m$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ be four vectors. Prove the following identity

$$\big[\boldsymbol{u} \otimes \boldsymbol{x}\big]^\intercal \big[\boldsymbol{v} \otimes \boldsymbol{y}\big] = \boldsymbol{u}^\intercal \boldsymbol{v} \cdot \boldsymbol{x}^\intercal \boldsymbol{y} \; . \tag{3.54}$$

**3.3** Let $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{x}, \boldsymbol{y}$ be four vectors as above and prove the following identity

$$\big[\boldsymbol{u} \otimes \boldsymbol{x}\big]^\intercal \big[\boldsymbol{v} \otimes \boldsymbol{y}\big] = \boldsymbol{u}^\intercal \big[\, \boldsymbol{v} \boldsymbol{x}^\intercal \big] \, \boldsymbol{y} \; . \tag{3.55}$$

**3.4** Reconsider equation (3.54). How many operations does it take to compute its right hand side? How many operations does it take to compute its left hand side? Give appropriate big $O$ expressions.

**3.5** Reconsider equation (3.55). How many operations does it take to compute its right hand side? How many operations does it take to compute its left hand side? Give appropriate big $O$ expressions.

**3.6** Implement a `python` / `numpy` function that takes a set $\mathcal{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ of vectors as input and returns the Kronecker product

$$\boldsymbol{x}_1 \otimes \boldsymbol{x}_2 \otimes \cdots \otimes \boldsymbol{x}_n \; . \tag{3.56}$$

**Note:** This can be done without using `for` loops if you work with function `reduce` available in the `functools` module of the `python` standard library.

**3.7** Answer this: Does your above `python` / `numpy` function also work with an input set $\mathcal{X} = \{\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_n\}$ of matrices?

**3.8** *Hadamard matrices* $\boldsymbol{H}_n$ of size $2^n \times 2^n$ are incredibly important operators in quantum computing and have a very special structure; for instance, in the particular case where $n = 2$, we have

$$\boldsymbol{H}_2 = \frac{1}{\sqrt{2^2}} \begin{bmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{bmatrix} . \tag{3.57}$$

For any $n \geq 1$, they can be computed using the following recursion

$$\boldsymbol{H}_1 = \tfrac{1}{\sqrt{2}} \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} \tag{3.58}$$

$$\boldsymbol{H}_n = \boldsymbol{H}_1 \otimes \boldsymbol{H}_{n-1}\,. \tag{3.59}$$

Implement a `numpy` function that takes a number $n \in \mathbb{N}$ as input and returns the corresponding Hadamard matrix $\boldsymbol{H}_n$.

**3.9** Compute the expression $\boldsymbol{H}_n^\intercal \boldsymbol{H}_n$ for several choices of $n$. What do you observe. What does your observation imply for the columns and/or rows of matrix $\boldsymbol{H}_n$?

**3.10** Similarly, compute the expression $\boldsymbol{H}_n^2$ for several choices of $n$. What do you observe. What does your observation imply for the columns and/or rows of matrix $\boldsymbol{H}_n$?

**3.11** *Vector logic* was developed by the Uruguayan mathematician Eduardo Mizraji [7–9]. Following his ideas, we represent the two truth values `false` and `true` in terms of orthonormal vectors, for instance

$$\texttt{false} \equiv \boldsymbol{n} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad (\text{“no”})$$

$$\texttt{true} \equiv \boldsymbol{s} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad (\text{“si”})$$

Implement `numpy` code that computes the two matrices

$$\boldsymbol{C} = \boldsymbol{n}\big[\boldsymbol{n} \otimes \boldsymbol{n}\big]^\intercal + \boldsymbol{n}\big[\boldsymbol{n} \otimes \boldsymbol{s}\big]^\intercal + \boldsymbol{n}\big[\boldsymbol{s} \otimes \boldsymbol{n}\big]^\intercal + \boldsymbol{s}\big[\boldsymbol{s} \otimes \boldsymbol{s}\big]^\intercal \tag{3.60}$$

$$\boldsymbol{D} = \boldsymbol{n}\big[\boldsymbol{n} \otimes \boldsymbol{n}\big]^\intercal + \boldsymbol{s}\big[\boldsymbol{n} \otimes \boldsymbol{s}\big]^\intercal + \boldsymbol{s}\big[\boldsymbol{s} \otimes \boldsymbol{n}\big]^\intercal + \boldsymbol{s}\big[\boldsymbol{s} \otimes \boldsymbol{s}\big]^\intercal \tag{3.61}$$

and print the resulting arrays. What do you observe? Do you already see a connection between matrices $\boldsymbol{C}$ and $\boldsymbol{D}$ and certain Boolean logic functions?

**3.12** Given the above matrices, consider two vectors $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \big\{\boldsymbol{n}, \boldsymbol{s}\big\}$ and feed them into the following functions

$$f_{\boldsymbol{C}}\big(\boldsymbol{x}_1, \boldsymbol{x}_2\big) = \boldsymbol{C}\big[\boldsymbol{x}_1 \otimes \boldsymbol{x}_2\big] \tag{3.62}$$

$$f_{\boldsymbol{D}}\big(\boldsymbol{x}_1, \boldsymbol{x}_2\big) = \boldsymbol{D}\big[\boldsymbol{x}_1 \otimes \boldsymbol{x}_2\big] \tag{3.63}$$

Scrutinize your results. Which Boolean logic functions did you just compute?

**3.13** In order to explore whether your results are dependent on our particular choices for $\boldsymbol{n}$ and $\boldsymbol{s}$, repeat the last two exercise but this time with the vectors

$$\boldsymbol{n} = \big[1, 0, 0\big]^\intercal \qquad \text{and} \qquad \boldsymbol{s} = \big[0, 0, 1\big]^\intercal \tag{3.64}$$

$$\boldsymbol{n} = \tfrac{1}{\sqrt{2}}\big[+1, -1\big]^\intercal \qquad \text{and} \qquad \boldsymbol{s} = \tfrac{1}{\sqrt{2}}\big[+1, +1\big]^\intercal \tag{3.65}$$

$$\boldsymbol{n} = \tfrac{1}{\sqrt{2}}\big[+i\,, -i\,\big]^\intercal \qquad \text{and} \qquad \boldsymbol{s} = \tfrac{1}{\sqrt{2}}\big[+i\,, +i\,\big]^\intercal \tag{3.66}$$

where $i = \sqrt{-1}$ denotes the *imaginary unit*.

What do you observe? Does the representation of `false` and `true` impact the logic of the results produced by this method? What would happen if $\boldsymbol{n}$ and $\boldsymbol{s}$ were *not* orthonormal?

**3.14** Let $\boldsymbol{N} = \boldsymbol{n}\boldsymbol{s}^\intercal + \boldsymbol{s}\boldsymbol{n}^\intercal$ and $\boldsymbol{C}$ and $\boldsymbol{D}$ as in (3.60) and (3.61), respectively. Prove the following: If matrices $\boldsymbol{N}$ and $\boldsymbol{D}$ are given, we can compute matrix $\boldsymbol{C}$ as

$$\boldsymbol{C} = \boldsymbol{N}\boldsymbol{D}\big[\boldsymbol{N} \otimes \boldsymbol{N}\big] \tag{3.67}$$

and, vice versa, if matrices $\boldsymbol{N}$ and $\boldsymbol{C}$ are given, we can compute matrix $\boldsymbol{D}$ as

$$\boldsymbol{D} = \boldsymbol{N}\boldsymbol{C}\big[\boldsymbol{N} \otimes \boldsymbol{N}\big]. \tag{3.68}$$

**Note:** Symbolically, these proofs require very careful thinking so you may just do them computationally.

## References

1. Schölkopf, B., Smola, A.: Lerning with Kernels. MIT Press (2001)
2. O'Donnell, R.: Analysis of Boolean Functions. Cambridge University Press (2014)
3. Oliphant, T.: Python for Scientific Computing. Computing in Science & Engineering **9**(3) (2007). https://doi.org/10.1109/MCSE.2007.58
4. Abadi, M., et al.: TensorFlow: A System for Large-Scale Machine Learning. In: Proc. USENIX Symp. on Operating Systems Design and Implementation (2016)
5. Paszke, A., et al.: PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Proc. NeurIPS (2019)
6. Graham, A.: Kronecker Products and Matrix Calculus. Ellis Horwood Ltd. (1981)
7. Mizraji, E.: Vector Logics: The Matrix-vector Representation of Logical Calculus. Fuzzy Sets and Systems **50**(2) (1992). https://doi.org/10.1016/0165-0114(92)90216-Q
8. Mizraji, E.: The Operators of Vector Logic. Mathematical Logic Quarterly **42**(1) (1996). https://doi.org/10.1002/malq.19960420104
9. Mizraji, E.: Vector Logic: A Natural Algebraic Representation of the Fundamental Logical Gates. J. of Logic and Computation **18**(1) (2008). https://doi.org/10.1093/logcom/exm057

# Chapter 4
# QUBOs

## 4.1 Introduction

In Chap. 1, we already came across a *quadratic unconstrained binary optimization problem* or *QUBO* for short. We considered the subset sum problem (SSP) where we are given a set $\mathcal{X}$ of $n$ integers $x_j \in \mathbb{Z}$ and a target integer $T \in \mathbb{Z}$ and need to search for a subset $\mathcal{X}' \subseteq \mathcal{X}$ whose elements sum to $T$. Representing $\mathcal{X}$ as a vector $\boldsymbol{x} \in \mathbb{R}^n$ and any subset $\mathcal{X}' \subseteq \mathcal{X}$ in terms of a binary indicator vector $\boldsymbol{z} \in \{0, 1\}^n$ with entries $z_j = 0$ if $x_j \notin \mathcal{X}'$ and $z_j = 1$ if $x_j \in \mathcal{X}'$, we were able to express the operation of summing the $x_j \in \mathcal{X}' \subseteq \mathcal{X}$ as an inner product $\boldsymbol{z}^\intercal \boldsymbol{x}$. This linear algebraic view on subset sums finally allowed us to formalize the SPP as a minimization problem

$$\boldsymbol{z}_* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} \left(\boldsymbol{z}^\intercal \boldsymbol{x} - T\right)^2 . \tag{4.1}$$

Note that this problem is an *optimization problem* because it asks for an *optimal value* $\boldsymbol{z}_*$ of a (vector valued) *decision variable* $\boldsymbol{z}$. More specifically, it is a *quadratic optimization problem* because its *objective function* $f(\boldsymbol{z}) = (\boldsymbol{z}^\intercal \boldsymbol{x} - T)^2$ is quadratic in the decision variable. It is also a *binary optimization problem* because its *feasible set* $\{0, 1\}^n$ is the set of all $n$-dimensional binary vectors. It is finally an *unconstrained optimization problem* because, other than being binary, there are no constraints on the decision variable. All in all, (4.1) thus indeed constitutes an instance of a QUBO.

QUBOs like this will serve as catalysts for much of what we study later because our major plot line in this book is that any QUBO can be solved by running a Hopfield net and that any problem that can be solved by running a Hopfield net can be solved on a quantum computer [1–5].

In this chapter, we will therefore have a closer look at QUBOs and familiarize ourselves with their characteristics. We will assume a more general view on their structure and discuss some of their peculiarities. However, we will not yet look at further practical examples nor will we discuss how to actually solve QUBOs. The former will happen in the next chapter and the latter will be left to the chapters on Hopfield nets and quantum computing.

## 4.2 Binary and Bipolar QUBOs

To begin with, we need to scrutinize the term *binary* in quadratic unconstrained *binary* optimization problem. This is because there actually exist two predominant interpretations of QUBOs, namely one that involves *binary* variables $\boldsymbol{z} \in \{0, 1\}^n$ and one that involves *bipolar* variables $\boldsymbol{s} \in \{\pm 1\}^n$.

Would it then not be better to talk about quadratic unconstrained *bivalent* optimization or quadratic unconstrained *Boolean* optimization? It arguably would, but we are once again dealing with a case of established somewhat ambiguous terminology which does not bother experts but might confuse novices. In an attempt to mitigate the risk of confusion, we will therefore break with tradition and henceforth distinguish between *binary* QUBOs and *bipolar* QUBOs.

Next, we formally define the general forms of these two flavors of QUBOs and then immediately comment on our definitions as they may raise several questions.

**Definition 4.1** A **binary QUBO** is an optimization problem of the form

$$\boldsymbol{z}_* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} \ \boldsymbol{z}^\intercal \boldsymbol{Q} \, \boldsymbol{z} + \boldsymbol{q}^\intercal \boldsymbol{z} \tag{4.2}$$

where matrix $\boldsymbol{Q} \in \mathbb{R}^{n \times n}$ and vector $\boldsymbol{q} \in \mathbb{R}^n$ are given problem parameters.

**Definition 4.2** A **bipolar QUBO** is an optimization problem of the form

$$\boldsymbol{s}_* = \underset{\boldsymbol{s} \in \{\pm 1\}^n}{\operatorname{argmin}} \ -\tfrac{1}{2} \, \boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s} \tag{4.3}$$

where matrix $-\frac{1}{2} \boldsymbol{W} \in \mathbb{R}^{n \times n}$ and vector $\boldsymbol{b} \in \mathbb{R}^n$ are given problem parameters.

Given these definitions, it stand outs that they demand the two objective functions

$$f_z(\boldsymbol{z}) = \boldsymbol{z}^\intercal \boldsymbol{Q} \, \boldsymbol{z} + \boldsymbol{q}^\intercal \boldsymbol{z} \tag{4.4}$$

$$f_s(\boldsymbol{s}) = -\tfrac{1}{2} \, \boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s} \tag{4.5}$$

to be quadratic forms augmented by linear terms. The former involve a parameter matrix ($\boldsymbol{Q}$ or $\boldsymbol{W}$), and the latter involve a parameter vector ($\boldsymbol{q}$ or $\boldsymbol{b}$).

Right now, this may seem odd as the only specific instance of a (binary) QUBO we have seen so far, namely the subset sum QUBO in (4.1), does not appear to be of this form. However, we will soon see (in Sect. 4.2.2) that there is no conundrum! That is, we will soon see that the specific objective in (4.1) can be rewritten in terms of the general objective in (4.4).

With respect to the bipolar QUBO objective in (4.5), we remark that physicists know it as an *Ising model* named after Ernst Ising who introduced these models to the study of magnetic spin systems [6]. This is foreshadowing because magnetic spin is a quantum mechanical phenomenon. We also remark that the terms *QUBO* and *Ising model* are almost synonymous and that much of the quantum computing literature talks about Ising models rather than about QUBOs. We further emphasize that the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ we used in (4.5) are rarely used in the physics literature. Instead, they are more common in the neurocomputing literature where they occur in *Hopfield energy functions* named after [7]. This is, of course, foreshadowing, too.

There is more to be said about the objectives of binary- and bipolar QUBOs. For instance, when taking a step back and letting

$$\mathbb{B}^n \in \Big\{\{0,1\}^n, \{\pm 1\}^n\Big\}, \tag{4.6}$$

we realize that both minimization objectives in (4.4) and (4.5) are $n$-ary pseudo Boolean functions $f : \mathbb{B}^n \to \mathbb{R}$. This is interesting as it means that everything we know about pseudo Boolean functions directly applies to QUBO objectives.

We may also already point out a fact we cannot fully fathom yet: When setting up a QUBO model for a practical application in combinatorics, constraint satisfaction, or data science, it is usually more intuitive to think in terms of binary QUBOs. Yet, when using Hopfield nets or quantum computers for QUBO solving, it is usually preferable to work with bipolar QUBOs. While this seems to cause a schism between modeling and solving, our next observation will immediately resolve this issue.

Finally and most importantly, we can map back and forth between binary- and bipolar QUBOs. As this is of vital importance for theory and application of QUBO modeling and solving, we next demonstrate how to turn a binary objective as in (4.4) into a bipolar objective as in (4.5). The other direction is left to Exercise 4.1.

### 4.2.1 Binary and Bipolar QUBOs are Equivalent

Working with the affine transformation

$$\boldsymbol{z} = \tfrac{1}{2}\big[\boldsymbol{s} + \boldsymbol{1}\big] \tag{4.7}$$

from Theorem 2.2 in Chap. 2, we have the following equalities

$$z^\intercal Q\, z + z^\intercal q = \tfrac{1}{4}\big[s+\mathbf{1}\big]^\intercal Q\,\big[s+\mathbf{1}\big] + \tfrac{1}{2}\big[s+\mathbf{1}\big]^\intercal q \tag{4.8}$$

$$= \tfrac{1}{4}\, s^\intercal Q\, s + \tfrac{1}{4}\, s^\intercal Q\,\mathbf{1} + \tfrac{1}{4}\,\mathbf{1}^\intercal Q\, s + \tfrac{1}{4}\,\mathbf{1}^\intercal Q\,\mathbf{1} + \tfrac{1}{2}\, s^\intercal q + \tfrac{1}{2}\,\mathbf{1}^\intercal q\ . \tag{4.9}$$

On the right of (4.8), we simply substituted $\frac{1}{2}\big[s+\mathbf{1}\big]$ for every occurrence of $z$ on the left, and, in (4.9), we distributed matrix-vector and vector-vector products over sums of vectors.

Observe that the expression in (4.9) includes terms which involve vectors $s$ and terms which do not. To make this more visible, we separate the ones from the others and further emphasize that the latter are constants independent of $s$. This way, we obtain the following equality

$$z^\intercal Q\, z + z^\intercal q = \tfrac{1}{4}\, s^\intercal Q\, s + \tfrac{1}{4}\, s^\intercal Q\,\mathbf{1} + \tfrac{1}{4}\,\mathbf{1}^\intercal Q\, s + \tfrac{1}{2}\, s^\intercal q + \underbrace{\tfrac{1}{4}\,\mathbf{1}^\intercal Q\,\mathbf{1} + \tfrac{1}{2}\,\mathbf{1}^\intercal q}_{const}\ . \tag{4.10}$$

The next step is conceptually important. If we understand the right hand side of (4.10) as a minimization objective $f_s(s)$, then the constant terms will impact the value of the minimum but not its location. In other words, they will impact the result of computing

$$f_s\big(s_*\big) = \min_{s} f_s\big(s\big) \tag{4.11}$$

but not the result of computing

$$s_* = \underset{s}{\operatorname{argmin}}\ f_s\big(s\big). \tag{4.12}$$

Regarding our current goal of establishing the equivalence of binary- and bipolar QUBOs which ask for either $z_*$ or $s_*$, we can therefore drop the constant terms to get the following proportionality

$$z^\intercal Q\, z + z^\intercal q \propto \tfrac{1}{4}\, s^\intercal Q\, s + \tfrac{1}{2}\, s^\intercal Q\,\mathbf{1} + \tfrac{1}{2}\, s^\intercal q\ . \tag{4.13}$$

At this point, the structure on the right hand side of (4.13) is already of the form of the objective in (4.5). To make this clearly visible, we first factor out common factors in the linear terms and then perform a substitution to get

$$z^\intercal Q\, z + z^\intercal q \propto \tfrac{1}{4}\, s^\intercal Q\, s + s^\intercal \Big[\tfrac{1}{2}\big[Q\,\mathbf{1} + q\big]\Big] \tag{4.14}$$

$$\equiv -\tfrac{1}{2}\, s^\intercal W s + s^\intercal b \tag{4.15}$$

where the two newly introduced parameters $W$ and $b$ are given by

$$W = -\tfrac{1}{2}\, Q \tag{4.16}$$

$$b = \ \tfrac{1}{2}\big[Q\,\mathbf{1} + q\big]. \tag{4.17}$$

Note what we just did: We went through a series of purely algebraic manipulations to establish that, for every binary QUBO objective, the exists an equivalent bipolar QUBO objective. Neither did this require us to specify the content of the parameter matrix $\boldsymbol{Q}$ nor to specify the content of the parameter vector $\boldsymbol{q}$.

Since our derivation was therefore general, we can summarize the gist of it by means of the following theorem.

**Theorem 4.1** *For every binary QUBO of the general form*

$$\boldsymbol{z}_* = \underset{\boldsymbol{z} \in \{0,1\}^n}{argmin} \ \boldsymbol{z}^\intercal \boldsymbol{Q} \, \boldsymbol{z} + \boldsymbol{q}^\intercal \boldsymbol{z} \, , \tag{4.18}$$

*there exists an affinely equivalent bipolar QUBO of the form*

$$\boldsymbol{s}_* = \underset{\boldsymbol{s} \in \{\pm 1\}^n}{\operatorname{argmin}} \ -\tfrac{1}{2} \, \boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s} \tag{4.19}$$

*whose parameter matrix and parameter vector are given by*

$$\boldsymbol{W} = -\tfrac{1}{2} \, \boldsymbol{Q} \tag{4.20}$$

$$\boldsymbol{b} = \ \tfrac{1}{2} \big[ \boldsymbol{Q} \, \mathbf{1} + \boldsymbol{q} \big] \, . \tag{4.21}$$

*If $\boldsymbol{s}_*$ solves problem* (4.19), *then $\boldsymbol{z}_* = \frac{1}{2}\big[\boldsymbol{s}_* + \mathbf{1}\big]$ solves problem* (4.18).

Armed with this theorem, we can therefore rest assured that we may indeed always (try to) model a given optimization problem in terms of a binary QUBO and then transform it into a bipolar QUBO which we can solve by running a Hopfield net or a quantum computing algorithm.

Next, we make good of our promise and show how to rewrite the familiar binary subset sum QUBO in (4.1) in terms of the newly introduced general form in (4.4).

### *4.2.2 Subset Sum as a Binary or a Bipolar QUBO*

We already know that an SSP with inputs $(\mathcal{X}, T) \in \mathbb{Z}^n \times \mathbb{Z}$ can be formulated in terms of the following binary QUBO

$$\boldsymbol{z}_* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} \big( \boldsymbol{z}^\intercal \boldsymbol{x} - T \big)^2 \tag{4.22}$$

where the constant vector $\boldsymbol{x} \in \mathbb{Z}^n$ represents set $\mathcal{X}$ and the variable vectors $\boldsymbol{z} \in \{0, 1\}^n$ represent subsets $\mathcal{X}' \subseteq \mathcal{X}$. An apparent issue with this formulation is that its objective function $f(\boldsymbol{z}) = (\boldsymbol{z}^\intercal \boldsymbol{x} - T)^2$ does not seem to be of the form prescribed by our general Definition 4.1. However, resolving this dichotomy is but a matter of simple algebraic manipulations.

To begin with, we observe that the objective function in (4.22) expands as follows

$$\bigl(\boldsymbol{z}^\intercal \boldsymbol{x} - T\bigr)^2 = \boldsymbol{z}^\intercal \boldsymbol{x} \cdot \boldsymbol{z}^\intercal \boldsymbol{x} - 2\,T\,\boldsymbol{z}^\intercal \boldsymbol{x} + T^2 \tag{4.23}$$

$$= \boldsymbol{z}^\intercal \boldsymbol{x} \cdot \boldsymbol{x}^\intercal \boldsymbol{z} - 2\,T\,\boldsymbol{x}^\intercal \boldsymbol{z} + T^2 \tag{4.24}$$

where the last step works because inner products of real valued vectors are symmetric.

Next, we resort to Theorem 3.2 which allows us to rewrite the product $\boldsymbol{z}^\intercal \boldsymbol{x} \cdot \boldsymbol{x}^\intercal \boldsymbol{z}$ of two inner products in terms of the quadratic form $\boldsymbol{z}^\intercal [\boldsymbol{x}\boldsymbol{x}^\intercal]\,\boldsymbol{z}$. This way, we obtain

$$\bigl(\boldsymbol{z}^\intercal \boldsymbol{x} - T\bigr)^2 = \boldsymbol{z}^\intercal \bigl[\boldsymbol{x}\boldsymbol{x}^\intercal\bigr]\,\boldsymbol{z} - 2\,T\,\boldsymbol{x}^\intercal \boldsymbol{z} + T^2 \tag{4.25}$$

$$= \boldsymbol{z}^\intercal \bigl[\boldsymbol{x}\boldsymbol{x}^\intercal\bigr]\,\boldsymbol{z} - \bigl[2\,T\boldsymbol{x}\bigr]^\intercal \boldsymbol{z} + T^2 \tag{4.26}$$

where the constant term $T^2$ is independent of the decision variable $\boldsymbol{z}$. All in all, we therefore find that our plain vanilla SSP can be cast as a binary QUBO of the form

$$\boldsymbol{z}_* = \underset{\boldsymbol{z}\in\{0,1\}^n}{\operatorname{argmin}}\ \boldsymbol{z}^\intercal\,\boldsymbol{Q}\,\boldsymbol{z} + \boldsymbol{q}^\intercal \boldsymbol{x} \tag{4.27}$$

where the parameters have to be set to

$$\boldsymbol{Q} = \boldsymbol{x}\boldsymbol{x}^\intercal \quad \text{and} \quad \boldsymbol{q} = -2\,T\boldsymbol{x}\,. \tag{4.28}$$

Theorem 4.1 finally tells us that we may state the problem in terms of an equivalent bipolar QUBO

$$\boldsymbol{s}_* = \underset{\boldsymbol{s}\in\{\pm1\}^n}{\operatorname{argmin}}\ -\tfrac{1}{2}\,\boldsymbol{s}^\intercal\,\boldsymbol{W}\boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s} \tag{4.29}$$

whose parameters amount to

$$\boldsymbol{W} = -\tfrac{1}{2}\,\boldsymbol{x}\boldsymbol{x}^\intercal \quad \text{and} \quad \boldsymbol{b} = \tfrac{1}{2}\bigl[\boldsymbol{x}\boldsymbol{x}^\intercal \mathbf{1} - 2\,T\boldsymbol{x}\bigr] \tag{4.30}$$

and we shall use these later on.

Next, however, we first point out further interesting and useful algebraic properties of the objective functions of binary and bipolar QUBOs.

### *4.2.3 Alternative Forms of Binary and Bipolar QUBOs*

Binary and bipolar QUBOs deal with vectors of binary- and bipolar numbers and we already know that these have remarkable properties. In particular, we recall that a binary number $z \in \{0, 1\}$ is idempotent such that $z^2 = z$ and that a bipolar number $s \in \{\pm 1\}$ is involutory such that $s^2 = +1$. This has practical implication especially for the quadratic form which occurs in the objective function of a respective QUBO.

To see what this may mean, we begin by pointing out a general observation for which we let $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ be a square matrix and $\boldsymbol{v} \in \mathbb{R}^n$ be a vector and consider the quadratic form $\boldsymbol{v}^\intercal \boldsymbol{M}\, \boldsymbol{v}$. For this setting, we observe the following: Every square matrix $\boldsymbol{M}$ can be written as

$$\boldsymbol{M} = \boldsymbol{O} + \boldsymbol{D} \tag{4.31}$$

where matrix $\boldsymbol{O}$ gathers the *off-diagonal elements* of $\boldsymbol{M}$ and matrix $\boldsymbol{D}$ gathers the *diagonal elements* of $\boldsymbol{M}$. The square form $\boldsymbol{v}^\intercal \boldsymbol{M}\, \boldsymbol{v}$ can therefore be written as

$$\boldsymbol{v}^\intercal \boldsymbol{M}\, \boldsymbol{v} = \boldsymbol{v}^\intercal \big[\boldsymbol{O} + \boldsymbol{D}\big]\, \boldsymbol{v} = \boldsymbol{v}^\intercal \boldsymbol{O}\, \boldsymbol{v} + \boldsymbol{v}^\intercal \boldsymbol{D}\, \boldsymbol{v}\ . \tag{4.32}$$

Hence, if we apply the decomposition in (4.31) to matrix $\boldsymbol{Q}$ in the objective function of a binary QUBO, we can rewrite this objective as

$$\boldsymbol{z}^\intercal \boldsymbol{Q}\, \boldsymbol{z} + \boldsymbol{q}^\intercal \boldsymbol{z} = \boldsymbol{z}^\intercal \boldsymbol{O}\, \boldsymbol{z} + \boldsymbol{z}^\intercal \boldsymbol{D}\, \boldsymbol{z} + \boldsymbol{q}^\intercal \boldsymbol{z}\ . \tag{4.33}$$

Noting that the diagonal elements of $\boldsymbol{O}$ and the off-diagonal elements of $\boldsymbol{D}$ are all zero, we can thus expand the binary QUBO objective as follows

$$\boldsymbol{z}^\intercal \boldsymbol{Q}\, \boldsymbol{z} + \boldsymbol{q}^\intercal \boldsymbol{z} = \sum_{j \neq k} \sum_k O_{jk} z_j z_k + \sum_j D_{jj} z_j z_j + \sum_j q_j z_j\ . \tag{4.34}$$

Using the idempotency of binary numbers $z_j$ which gives $z_j z_j = z_j^2 = z_j$, we can further rewrite the objective into a more compact form, namely

$$\begin{aligned}
\boldsymbol{z}^\intercal \boldsymbol{Q}\, \boldsymbol{z} + \boldsymbol{q}^\intercal \boldsymbol{z} &= \sum_{j \neq k} \sum_k O_{jk} z_j z_k + \sum_j D_{jj} z_j + \sum_j q_j z_j && (4.35)\\
&= \sum_{j \neq k} \sum_k O_{jk} z_j z_k + \sum_j \big(D_{jj} + q_j\big) z_j && (4.36)\\
&\equiv \sum_j \sum_k Q'_{jk} z_j z_k && (4.37)\\
&= \boldsymbol{z}^\intercal \boldsymbol{Q}' \boldsymbol{z} && (4.38)
\end{aligned}$$

where the element of matrix $\boldsymbol{Q}'$ are given by

$$Q'_{jk} = \begin{cases} Q_{jk} & \text{if } j \neq k \\ Q_{jj} + q_j & \text{if } j = k\,. \end{cases} \tag{4.39}$$

Finally, we note that the convenient vector-to-matrix operator diag : $\mathbb{R}^n \to \mathbb{R}^{n \times n}$ with

$$\text{diag}\big[\boldsymbol{q}\big] = \begin{bmatrix} q_1 & & \\ & \ddots & \\ & & q_n \end{bmatrix} \tag{4.40}$$

allows us to express this newly introduced matrix $\boldsymbol{Q}'$ in a more compact form, namely

$$\boldsymbol{Q}' = \boldsymbol{Q} + \text{diag}\big[\boldsymbol{q}\big]\,. \tag{4.41}$$

All in all, we therefore have the following general result which is often applied in the literature on binary QUBOs.

**Theorem 4.2** *Every binary QUBO of the general form*

$$\boldsymbol{z}_* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\text{argmin}}\; \boldsymbol{z}^\intercal \boldsymbol{Q}\, \boldsymbol{z} + \boldsymbol{q}^\intercal \boldsymbol{z} \tag{4.42}$$

*can equivalently be expressed in the more compact form*

$$\boldsymbol{z}_* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\text{argmin}}\; \boldsymbol{z}^\intercal \boldsymbol{Q}'\, \boldsymbol{z} \tag{4.43}$$

*whose parameter matrix amounts to* $\boldsymbol{Q}' = \boldsymbol{Q} + \text{diag}\big[\boldsymbol{q}\big]$.

By the same token, we can apply the decomposition in (4.31) to matrix $\boldsymbol{W}$ in the objective function of a bipolar QUBO and write it as

$$-\tfrac{1}{2}\,\boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s} = -\tfrac{1}{2}\,\boldsymbol{s}^\intercal \boldsymbol{O}\, \boldsymbol{s} - \tfrac{1}{2}\,\boldsymbol{s}^\intercal \boldsymbol{D}\, \boldsymbol{s} + \boldsymbol{q}^\intercal \boldsymbol{s}\,. \tag{4.44}$$

Since the off-diagonal elements of $\boldsymbol{D}$ are all zero and bipolar numbers are involutory such that $s_j s_j = s_j^2 = +1$, we next observe the following identity for the second term on the right

$$\boldsymbol{s}^\intercal \boldsymbol{D}\, \boldsymbol{s} = \sum_j D_{jj} s_j s_j = \sum_j D_{jj} = \text{tr}\big[\boldsymbol{D}\big] = \text{tr}\big[\boldsymbol{W}\big] \tag{4.45}$$

where $\text{tr}\big[\boldsymbol{W}\big]$ denotes the trace of matrix $\boldsymbol{W}$ which is a quantity independent of $\boldsymbol{s}$. We therefore have

$$-\tfrac{1}{2}\,\boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s} = -\tfrac{1}{2}\,\boldsymbol{s}^\intercal \boldsymbol{O}\, \boldsymbol{s} + \boldsymbol{q}^\intercal \boldsymbol{s} + \textit{const}\,. \tag{4.46}$$

If we then consider the matrix-to-matrix operator diag : $\mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}$ which acts as follows

$$\mathrm{diag}\big[\boldsymbol{W}\big] = \begin{bmatrix} W_{11} & & \\ & \ddots & \\ & & W_{nn} \end{bmatrix}, \tag{4.47}$$

we further realize that we can write the matrix $\boldsymbol{O}$ of off-diagonal elements of $\boldsymbol{W}$ as

$$\boldsymbol{O} = \boldsymbol{W} - \mathrm{diag}\big[\boldsymbol{W}\big]. \tag{4.48}$$

All this leads to another general result, this time with respect to the form of bipolar QUBOs. While this additional result is not as popular or as well known as the one we stated for binary QUBOs, it will be fundamental for our idea of using Hopfield nets as QUBO solvers. We therefore summarize it in a theorem as well.

**Theorem 4.3** *Every bipolar QUBO of the form*

$$\boldsymbol{s}_* = \underset{\boldsymbol{s}\in\{\pm 1\}^n}{\mathrm{argmin}} \; -\tfrac{1}{2}\,\boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s} \tag{4.49}$$

*is equivalent to a bipolar QUBO of the form*

$$\boldsymbol{s}_* = \underset{\boldsymbol{s}\in\{\pm 1\}^n}{\mathrm{argmin}} \; -\tfrac{1}{2}\,\boldsymbol{s}^\intercal \Big[\boldsymbol{W} - \mathrm{diag}\big[\boldsymbol{W}\big]\Big]\boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s}\,. \tag{4.50}$$

*Every $\boldsymbol{s}_*$ which solves the problem in* (4.50) *also solves the problem in* (4.49).

## 4.3 Peculiarities of QUBOs

Next, we discuss peculiar properties of solutions of QUBOs which we should know about when working with them in practice. These properties may be of particular interest to readers who known about quadratic optimization over continuous spaces such as $\mathbb{R}^n$ because quadratic optimization over discrete sets such as, say, $\{0, 1\}^n$ has different characteristics.

We will once again structure our discussion along the example of the subset sum problem which we recall can be formalized in terms of the following binary QUBO

$$\boldsymbol{z}_* = \underset{\boldsymbol{z}\in\{0,1\}^n}{\mathrm{argmin}} \big(\boldsymbol{z}^\intercal \boldsymbol{x} - T\big)^2 \tag{4.51}$$

But why again would solving this QUBO solve the SSP?

Well, since it is easy to see that its quadratic objective is necessarily lower bounded by 0, we are guaranteed that

$$\forall\, z \in \{0,1\}^n : 0 \leq \left(z^\intercal x - T\right)^2 . \tag{4.52}$$

We therefore know that any *global minimizer* $z_*$ which would achieve this lower bound would provide us with

$$\left(z_*^\intercal x - T\right)^2 = 0 \quad \Leftrightarrow \quad z_*^\intercal x = T , \tag{4.53}$$

and would thus indicate a subset $\mathcal{X}_* \subseteq \mathcal{X}$ whose elements $x_j$ sum precisely to $T$. In other words, *if* such a minimizer existed and *if* we had an algorithm that can find it, then we could solve the underlying subset sum problem.

Note that our last statement came with a double emphasis on "if". The first one pertained to the existence of a solution and the second one to the existence of a solver. Next, we unravel both these "ifs".

### 4.3.1 Optimal- and Perfect Solutions

There are several fundamental observations regarding the solutions of QUBOs. For instance, a QUBO as in (4.51) will always have *at least one optimal solution* but not necessarily a *perfect solution*.

To see what this means, we note that solutions to a binary QUBO restricted to the set $\{0,1\}^n$ where we always assume that $n \in \mathbb{N}$ is finite. The size of this set is $2^n$ and therefore finite, too. In theory, we could thus exhaustively evaluate a QUBO's objective $f(z)$ for all $z \in \{0,1\}^n$. Since there are only finitely many $z$, this evaluation procedure will eventually stop; once it does, we will have come across at least one $z_*$ such that $f(z_*) \leq f(z)$ for all $z$. That is, we will have found at least one *global minimizer* of the objective and thus at least one *optimal solution* to the QUBO.

We further note that domain and range of the objective function of a binary QUBO are embedded in $\mathbb{R}^n$ and $\mathbb{R}$, respectively. We may therefore think of its objective as a quadratic function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Being quadratic, such a function will have a *finite* lower bound $l$. (Note that this bound does not generally have to be 0 as in our example above.) For a QUBO with optimal solution $z_*$, we are therefore guaranteed that $l \leq f(z_*) \leq f(z)$ for all $z$. Now, whenever an optimal solution $z_*$ attains this lower bound, i.e. whenever $l = f(z_*)$, we say that $z_*$ is a *perfect solution*.

The crux with respect to QUBO solutions is thus is that *every perfect solution is optimal but not every optimal solution is perfect*. While this may sound strange, it is easily illustrated via simple examples.

The following is a trivial (vectorized) subset sum problem that cannot be solved

$$x = \left[8, 1, 7, 2, 1\right]^\intercal \quad \text{and} \quad T = 5 . \tag{4.54}$$

The corresponding QUBO objective $f(z) = (z^\intercal x - T)^2$ thus cannot have a perfect solution which attains the lower bound of 0. Nevertheless, it has a unique minimizer

$$z_* = \left[0, 1, 0, 1, 1\right]^\intercal \tag{4.55}$$

and thus an optimal solution because it is easily verified that $0 < 1 = f(z_*) < f(z)$ for all $z \in \{0, 1\}^5$.

The problem therefore is that even *if* we had an algorithm to reliably find such optimal but imperfect solutions $z_*$, it would not suffice to rule out the existence of a perfect solution. For this, we would have to test all $z$ which is generally infeasible as there are $2^n$ of them. The good news is that not all problems are as nasty as the one we just went through. In other words, there are numerous practical problems with QUBO formulations whose optimal solutions are all we need.

Note that we may also encounter the opposite extreme where a QUBO actually has several perfect solutions. For example, here is another subset sum problem

$$x = \left[8, 1, 7, 2, 1\right]^\intercal \quad \text{and} \quad T = 8\,. \tag{4.56}$$

whose corresponding QUBO objective $f(z) = (z^\intercal x - T)^2$ is perfectly solved by

$$z_{*1} = \left[1, 0, 0, 0, 0\right]^\intercal \tag{4.57}$$

$$z_{*2} = \left[0, 1, 1, 0, 0\right]^\intercal \tag{4.58}$$

$$z_{*3} = \left[0, 0, 1, 0, 1\right]^\intercal. \tag{4.59}$$

For people with a background in machine learning this may seem odd because quadratic functions are convex and therefore have a unique global minimum, right? Right, as long as the domain over which to minimize is an unconstrained *continuous* domain such as $\mathbb{R}^n$. The feasible set of a QUBO, however, is a *discrete* domain such as $\{0, 1\}^n$ and the above example just demonstrated that quadratic functions over those may have several global minima.

Dealing with QUBOs with several optimal solutions, a potential problem is then that even *if* we had an algorithm to reliably find an optimal solution, it might not allow us to find all of them. The good news is that we often do not have to worry about finding all optima. In other words, there are numerous practical problems with QUBO formulations where one optimal solution is all we need.

### 4.3.2 Local- and Global Minima and Saddle Points

The overall bad news is that the artifacts we just went through are manifestations of the fact that QUBO solving is generally an NP-hard problem. Search spaces of QUBOs are exponentially large so that exhaustive exploration is usually infeasible. Moreover, search spaces of QUBOs are discrete structures, namely Boolean lattices, so that continuous optimization techniques do not apply. Instead, we typically have to resort to randomized algorithms or local search procedures but may still face challenges known from other settings.

Let us consider practical examples to visually explore what this could mean. To be able to do so, we need to restrict ourselves to examples whose search space is small enough for us to plot it. The SSP in (4.54) is such an example and we recall its ingredients for convenience:

$$\mathcal{X} = \{8, 1, 7, 2, 1, \} \Leftrightarrow \boldsymbol{x} = \left[8, 1, 7, 2, 1\right]^\intercal \quad \text{and} \quad T = 5 \,. \tag{4.60}$$

Also for convenience, we once again recall the binary QUBO formulation of this tiny instance of an SSP, namely

$$\begin{aligned} z_* &= \underset{z \in \{0,1\}^5}{\operatorname{argmin}} \left(z^\intercal \boldsymbol{x} - T\right)^2 & (4.61)\\ &= \underset{z \in \{0,1\}^5}{\operatorname{argmin}} \; z^\intercal \left[\boldsymbol{x}\boldsymbol{x}^\intercal\right] z - \left[2\,T\boldsymbol{x}\right]^\intercal z + T^2 & (4.62)\end{aligned}$$

where the feasible set $\{0, 1\}^5$ consists of the $2^5 = 32$ corners of the binary hypercube in $\mathbb{R}^5$ which we learned to visualize in terms of a lattice graph in Chap. 2.

Finally, we recall that we may drop the constant term $T^2$ from the above QUBO model to obtain an equivalent binary optimization problem

$$z_* = \underset{z \in \{0,1\}^5}{\operatorname{argmin}} \; z^\intercal \left[\boldsymbol{x}\boldsymbol{x}^\intercal\right] z - \left[2\,T\boldsymbol{x}\right]^\intercal z \tag{4.63}$$

which we may furthermore translate into a biplor optimization problem

$$\boldsymbol{s}_* = \underset{\boldsymbol{s} \in \{\pm 1\}^5}{\operatorname{argmin}} \; \tfrac{1}{4}\,\boldsymbol{s}^\intercal \left[\boldsymbol{x}\boldsymbol{x}^\intercal\right] \boldsymbol{s} + \tfrac{1}{2}\left[\boldsymbol{x}\boldsymbol{x}^\intercal \mathbf{1} - 2\,T\boldsymbol{x}\right]^\intercal \boldsymbol{s} \,. \tag{4.64}$$

Given these, we emphasize the following: While the initial binary QUBO formulation of our tiny SSP comes with an objective function which is lower bounded by 0, the dropping of the constant $T^2$ and the translation into bipolar form introduces novel objective which will generally have negative lower bounds.

In what follows, we shall work with the bipolar QUBO representation of our problem because this is what we will do in later chapters, too.

Now, since the dimensionality of our current problem is small ($n = 5$), we can easily enumerate *all* bipolar vectors $\boldsymbol{s} \in \{\pm 1\}^5$ and then plug them into the objective function

$$f(\boldsymbol{s}) = \tfrac{1}{4}\,\boldsymbol{s}^\intercal \left[\boldsymbol{x}\boldsymbol{x}^\intercal\right] \boldsymbol{s} + \tfrac{1}{2}\left[\boldsymbol{x}\boldsymbol{x}^\intercal \mathbf{1} - 2\,T\boldsymbol{x}\right]^\intercal \boldsymbol{s} \tag{4.65}$$

of the QUBO in (4.64) to brute force our SSP. This also allows us to visualize $f(\boldsymbol{s})$ in terms of a lattice graph with colored vertices whose colors indicate the values of $f$ at the different $\boldsymbol{s} \in \{\pm 1\}^5$. An example can be seen in Fig. 4.1.

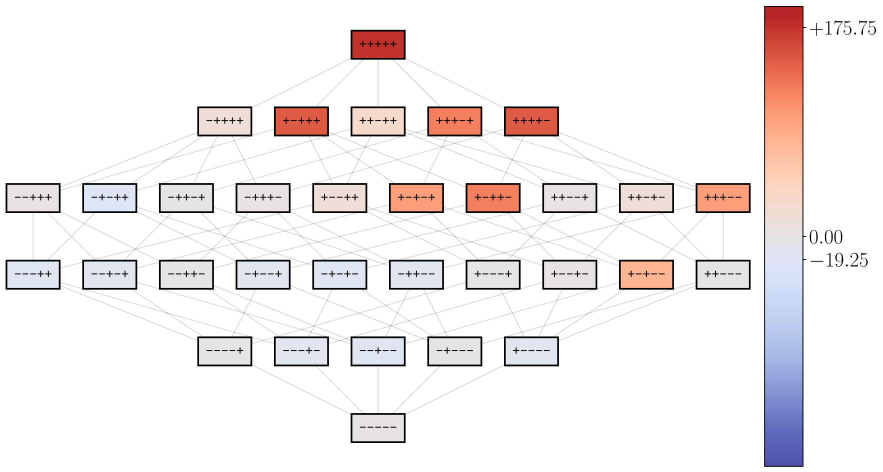


**Fig. 4.1** A lattice graph visualization of the bipolar QUBO objective $f(\boldsymbol{s})$ in (4.65) parameterized by $\boldsymbol{x}$ and $T$ in (4.60). Vertex labels are strings over $\{-,+\}$ which represent bipolar vectors $\boldsymbol{s} \in \{\pm 1\}^5$ and vertex colors represent values of $f$ at $\boldsymbol{s}$. Reddish colors indicate positive values; blueish colors indicate negative values. Function $f(\boldsymbol{s})$ attains its unique global maximum at $\boldsymbol{s} = +++++$ and its unique global minimum at $\boldsymbol{s} = -+-++$. The value of the former is about $+175.75$ and the value of the latter is about $-19.25$ (both rounded to two decimal places)

Each vertex of the graph in this figure represents a corner $\boldsymbol{s} \in \{\pm 1\}^5$ of the bipolar hypercube in $\mathbb{R}^5$ and is labeled by a corresponding string over the two symbols $-$ and $+$. Reddish vertex colors indicate positive values of $f(\boldsymbol{s})$ and blueish vertex colors indicate negative values of $f(\boldsymbol{s})$. While some of these colors or values clearly stand out, most of them are not very distinct. This phenomenon is not uncommon when working with QUBOs and can actually pose challenges when using quantum computers to solve them [8]. However, we shall regard them as minor as they can be overcome [9] and pose no difficulty when running Hopfield nets. Nevertheless, for our present discussion, it will be helpful to clearly see the different values of the objective function.

Figure 4.2 therefore shows another visualization of our objective function where vertex labels now represent the values of $f$ at $\boldsymbol{s}$. Looking at both Figs. 4.1 and 4.2, we therefore see that $f(\boldsymbol{s})$ has a unique global maximum at $\boldsymbol{s} = [+1, +1, +1, +1, +1]^\intercal$ and a unique global minimum at $\boldsymbol{s} = [-1, +1, -1, +1, +1]^\intercal$.

Hence, if we now consider each $\boldsymbol{s}$ as a bipolar indicator vector of $\mathcal{X}' \subseteq \mathcal{X}$ in the following sense

$$s_j = \begin{cases} +1 & \text{if } x_j \in \mathcal{X}' \\ -1 & \text{otherwise}\,, \end{cases} \tag{4.66}$$

we find that vector $\boldsymbol{s} = [-1, +1, -1, +1, +1]^\intercal$ with

$$f\big([-1, +1, -1, +1, +1]^\intercal\big) = -19.25 \tag{4.67}$$

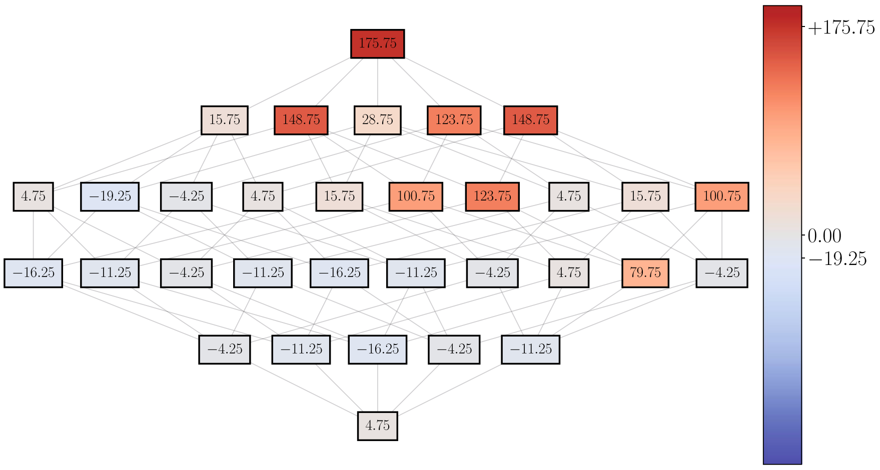


**Fig. 4.2** Another lattice graph visualization of the bipolar QUBO objective $f(\boldsymbol{s})$ in (4.65) with parameters $\boldsymbol{x}$ and $T$ as in (4.60). Here, each vertex $\boldsymbol{s} \in \{\pm 1\}^5$ is labeled with the value of $f(\boldsymbol{s})$

represents subset $\mathcal{X}' = \{1, 2, 1\}$ of superset $\mathcal{X} = \{8, 1, 7, 2, 1\}$ of our tiny SSP. From our discussion in the previous subsection we know that this is the optimal solution for the QUBO model of the SSP albeit not a perfect one because $1 + 2 + 1 = 4 \neq 5 = T$.

However, what is more interesting about our visualization of our current objective function $f(\boldsymbol{s})$, is to observe that it comes with several local minima. Consider, for instance, the vertex with objective value

$$f\big([-1, -1, +1, -1, -1]^\intercal\big) = -16.25\,. \tag{4.68}$$

This value is greater than the globally minimal value of $-19.25$ but smaller than the objective values of all the neighboring vertices. To see that this makes sense, we observe that vector $[-1, -1, +1, -1, -1]^\intercal$ represents $\{7\} \subseteq \mathcal{X}$ and that its neighbors (vectors with Hamming distance one) represent either the empty set $\emptyset \subseteq \mathcal{X}$ or a slightly larger set $\{7, x\} \subseteq \mathcal{X}$ with $x \in \mathcal{X} \setminus \{7\}$. In either case, the sum of the elements of these neighboring subsets is farther from the target value $T = 5$ than the sum of the elements of the original subset $\{7\}$.

Readers with a background in continuous optimization or machine learning will know that local minima of objective- or loss functions often pose considerable challenges. What we just saw is that they can also occur in the context of combinatorial optimization and we should brace ourselves for this whenever we work with QUBO models. Another well known issue regarding continuous objective- or loss functions is that they may have saddle points. These, too, may need to be dealt with when working with QUBO models.

As an example for the occurrence of saddle points in QUBO objective functions, we revisit the tiny SSP in (4.56). Its ingredients were

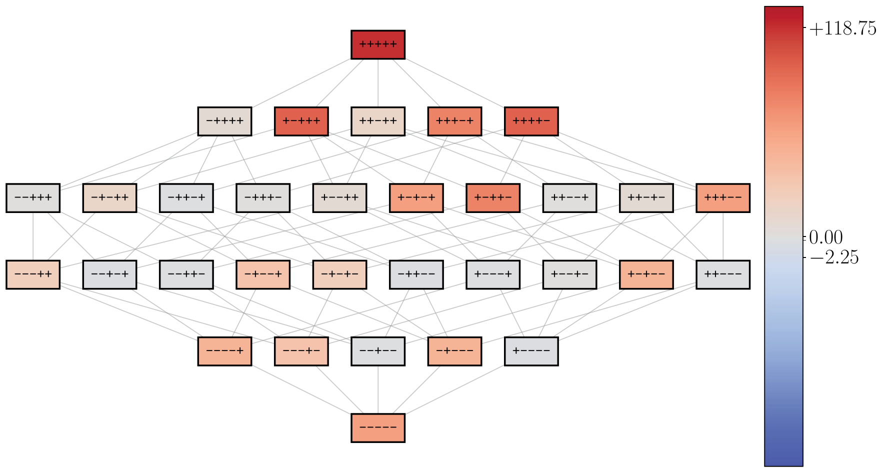


**Fig. 4.3** A lattice graph visualization of the bipolar QUBO objective $f(\mathbf{s})$ in (4.70) parameterized by $\mathbf{x}$ and $T$ in (4.69). Function $f(\mathbf{s})$ attains a unique global maximum at $\mathbf{s} = +++++$ and three global minima at $\mathbf{s} \in \{+----, -++---, --+-+\}$. The value of the former is about $+118.75$ and the values of the latter are about $-2.25$ (all rounded to two decimal places)

$$\mathcal{X} = \{8, 1, 7, 2, 1, \} \Leftrightarrow \mathbf{x} = \left[8, 1, 7, 2, 1\right]^\intercal \quad \text{and} \quad T = 8 \tag{4.69}$$

and, when formalized as a bipolar QUBO over decision variables $\mathbf{s} \in \{\pm 1\}^5$, the corresponding objective is once again given by

$$f(\mathbf{s}) = \tfrac{1}{4}\, \mathbf{s}^\intercal \left[\mathbf{x}\mathbf{x}^\intercal\right] \mathbf{s} + \tfrac{1}{2} \left[\mathbf{x}\mathbf{x}^\intercal \mathbf{1} - 2\, T \mathbf{x}\right]^\intercal \mathbf{s} \,. \tag{4.70}$$

Figures 4.3 and 4.4 show visualizations of this function and we direct our attention to the vertex representing vector $\mathbf{s} = [-1, +1, +1, -1, +1]^\intercal$ with

$$f\big([-1, +1, +1, -1, +1]^\intercal\big) = -1.25 \,. \tag{4.71}$$

Two of the four neighbors of this particular vertex have larger objective values of

$$f\big([-1, +1, +1, +1, +1]^\intercal\big) = \ \ 6.75 \tag{4.72}$$
$$f\big([-1, +1, -1, -1, +1]^\intercal\big) = 33.75 \tag{4.73}$$

and the other two neigbours have smaller objective values, namely

$$f\big([-1, -1, +1, -1, +1]^\intercal\big) = -2.25 \tag{4.74}$$
$$f\big([-1, +1, +1, -1, -1]^\intercal\big) = -2.25 \,. \tag{4.75}$$

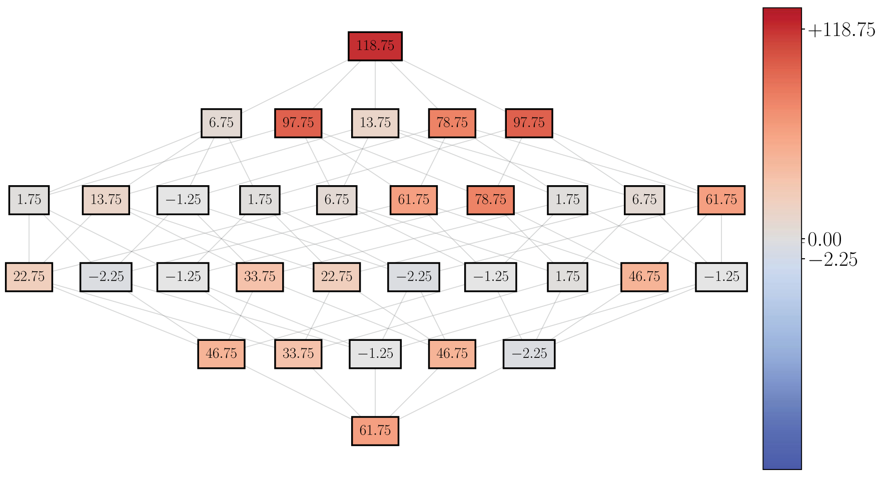


**Fig. 4.4** Another lattice graph visualization of the bipolar QUBO objective $f(\boldsymbol{s})$ in (4.70) with parameters $\boldsymbol{x}$ and $T$ as in (4.69). Here, each vertex $\boldsymbol{s} \in \{\pm 1\}^5$ is labeled with the value of $f(\boldsymbol{s})$

The particular vector $\boldsymbol{s} = [-1, +1, +1, -1, +1]^\intercal$ therefore is a relative minimum with respect to its neighboring vectors in (4.72) and (4.73) and a relative maximum with respect to its neighboring vectors in (4.74) and (4.75). In other words, it is a saddle point of a function over a discrete domain. This is a relevant observation because experience tells us that people new to the idea of quadratic unconstrained binary optimization are often surprised to learn that saddle points can occur in this context, too.

Finally, to wrap up our present discussion, we revisit another idea from Chap. 2 where we said that we may arrange the vertices of binary- or bipolar hypercubes or lattice graphs along the real number line. This idea provides us with another means of visualizing QUBO objectives.

For instance, in Fig. 4.5, we map the 32 bipolar vectors $\boldsymbol{s} \in \{\pm 1\}^5$ to the natural numbers $0, 1, \ldots, 31$ and plot the objective function $f(\boldsymbol{s})$ of the bipolar QUBO formalization of the SSP in (4.69) as a step function. Sometimes, visualizations like this can be helpful, for instance, in situations where bipolar vectors serve as representations of natural numbers. However, in our current setting where each $\boldsymbol{s}$ is a subset indicator vector, such visualizations run the risk of missing out on the adjacency relations among such vectors which are clearly visible in lattice graphs. Be this as it may, the plot in Fig. 4.5 further underlines the typically quite rugged nature of QUBO objective functions.

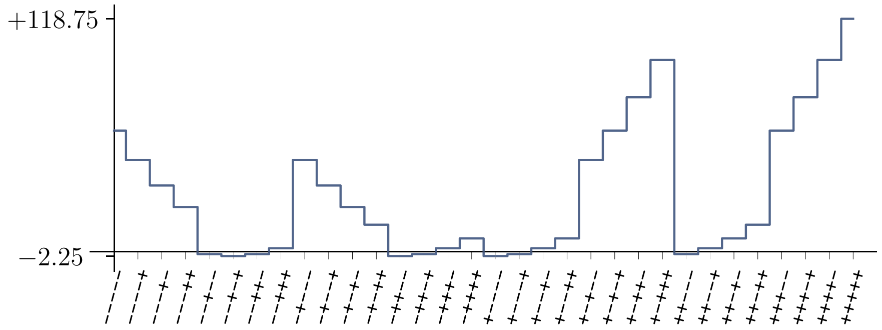


**Fig. 4.5** Yet another visualization of the bipolar QUBO objective $f(\boldsymbol{s})$ in (4.70) with parameters $\boldsymbol{x}$ and $T$ as in (4.69)

## 4.4 Code Examples

From a practical point of view, the arguably most important aspect we discussed in this chapter is the transformation of binary QUBOs into bipolar QUBOs summarized in Theorem 4.1. Again, this is because, in practice, it is usually easy to set up binary QUBO models and we will see examples for this soon. However, when running Hopfield nets or quantum algorithms for QUBO solving, it is often preferable to work with bipolar QUBOs. Next, we therefore look at how to code this transformation.

For convenience, we first recall that, if we are given the parameters $\boldsymbol{Q} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{q} \in \mathbb{R}^n$ of a binary QUBO, then

$$\boldsymbol{W} = -\tfrac{1}{2}\,\boldsymbol{Q} \tag{4.76}$$

$$\boldsymbol{b} = \tfrac{1}{2}\left[\boldsymbol{Q}\,\mathbf{1} + \boldsymbol{q}\right] \tag{4.77}$$

are the parameters of a corresponding bipolar QUBO. Looking at these equations, it should be easy to turn them into code.

Indeed, if we `import numpy as np`, then function `bin2bipQubo` in code snippet 4.1 immediately does the trick. It takes a 2D array `matQ` and a 1D array `vecQ` as inputs and returns corresponding arrays `matW` and `vecB`.

The only point worth noting is maybe our use of the `numpy` function `ones_like` which takes an array as input and produces an array of the same shape whose entries are all set to 1. Here, we can conveniently apply it to obtain an array which represents the vector of all ones $\mathbf{1} \in \mathbb{R}^n$.

In order to see this function `bin2bipQubo` in action, we once again revisit the tiny SSP in (4.60) and code its ingredients $\boldsymbol{x}$ and $T$ as

```
vecX = np.array([8, 1, 7, 2, 1])
trgt = 5
```

The parameters $\boldsymbol{Q} = \boldsymbol{x}\boldsymbol{x}^\intercal$ and $\boldsymbol{q} = -2\,T\boldsymbol{x}$ of the objective function of the respective binary QUBO in (4.63) are then very easy to come by. Since we already know

```
def bin2bipQubo(matQ, vecQ):
    '''
    convert binary QUBO to bipolar QUBO
    '''
    matW = -0.5 *  matQ
    vecB =  0.5 * (matQ @ np.ones_like(vecQ) + vecQ)

    return matW, vecB
```

**Code 4.1** A simple `numpy` implementation of a function that takes the parameters of a binary QUBO and returns the parameters of an equivalent bipolar QUBO.

how to code outer products, matrix $\boldsymbol{Q}$ and vector $\boldsymbol{q}$ can simply be computed and printed like this

```
matQ = np.outer(vecX, vecX)
vecQ = -2 * trgt * vecX
print (matQ, vecQ)
```

which results in

```
[[64  8 56 16  8]
 [ 8  1  7  2  1]
 [56  7 49 14  7]
 [16  2 14  4  2]
 [ 8  1  7  2  1]] [-80 -10 -70 -20 -10]
```

Finally, we simply have to execute the following to obtain and print representations of the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ of the corresponding bipolar QUBO in (4.64)

```
matW, vecB = bin2bipQubo(matQ, vecQ)
print (matW, vecB)
```

This results in

```
[[-32.   -4.  -28.   -8.   -4. ]
 [ -4.   -0.5  -3.5  -1.   -0.5]
 [-28.   -3.5 -24.5  -7.   -3.5]
 [ -8.   -1.   -7.   -2.   -1. ]
 [ -4.   -0.5  -3.5  -1.   -0.5]] [36.   4.5 31.5  9.   4.5]
```

and readers are very much encouraged to very that it makes sense (see Exercise 4.4).

## 4.5 Exercises

**4.1** Let $\boldsymbol{W}$ and $\boldsymbol{b}$ be the parameters of a bipolar QUBO according to Definition 4.2. Determine the the parameters $\boldsymbol{Q}$ and $\boldsymbol{q}$ of an affinely equivalent binary QUBO as in Definition 4.1.

**4.2** Implement `python/numpy` code that realizes the transformation you worked out in the previous exercise.

**4.3** The *Hadamard product* $\boldsymbol{A} \odot \boldsymbol{B} = \boldsymbol{C}$ of two same sized matrices $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{m \times n}$ produces another matrix $\boldsymbol{C} \in \mathbb{R}^{m \times n}$ with entries

$$a_{jk} \cdot b_{jk} = c_{jk} \; . \tag{4.78}$$

Recall the matrix-to-matrix operator diag[] from (4.47) and consider $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ and the identity matrix $\boldsymbol{I} \in \mathbb{R}^{n \times n}$. Convince yourself that the following holds true

$$\operatorname{diag}\big[\boldsymbol{M}\big] = \boldsymbol{M} \odot \boldsymbol{I} \; . \tag{4.79}$$

Recall the vector-to-matrix operator diag[] from (4.40) and consider $\boldsymbol{v} \in \mathbb{R}^{n}$, the all ones vector $\mathbf{1} \in \mathbb{R}^{n}$, and the identity matrix $\boldsymbol{I} \in \mathbb{R}^{n \times n}$. Convince yourself that the following holds true

$$\operatorname{diag}\big[\boldsymbol{v}\big] = \boldsymbol{v}\mathbf{1}^{\intercal} \odot \boldsymbol{I} \; . \tag{4.80}$$

**4.4** Verify all the objective function values we reported in Eqs. (4.71)–(4.75). That is, consider the tiny SSP in (4.69) and the bipolar QUBO objective in (4.70). Implement `python` / `numpy` code that computes parameter matrix $\boldsymbol{W} = -\frac{1}{2}\boldsymbol{x}\boldsymbol{x}^{\intercal}$ and parameter vector $\boldsymbol{b} = \frac{1}{2}[\boldsymbol{x}\boldsymbol{x}^{\intercal}\mathbf{1} - 2\,T\boldsymbol{x}]$ and evaluates $-\frac{1}{2}\boldsymbol{s}^{\intercal}\boldsymbol{W}\boldsymbol{s} + \boldsymbol{s}^{\intercal}\boldsymbol{b}$ for all vectors $\boldsymbol{s}$ mentioned in (4.71)–(4.75).

**4.5** Write `numpy` code that can brute force any bipolar QUBO. Given the QUBO parameters $\boldsymbol{W} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{b} \in \mathbb{R}^{n}$, your code should be able to iterate over all $\boldsymbol{s} \in \{\pm 1\}^{n}$ and determine for which of those the value of $-\frac{1}{2}\boldsymbol{s}^{\intercal}\boldsymbol{W}\boldsymbol{s} + \boldsymbol{s}^{\intercal}\boldsymbol{b}$ is minimal. Test your code on the tiny SSP in (4.69).

## References

1. Farhi, E., Goldstone, J., Gutmann, S., Sipser, M.: Quantum Computation by Adiabatic Evolution. arXiv:quant-ph/0001106 (2000)
2. Aharonov, D., et al.: Adiabatic Quantum Computation Is Equivalent to Standard Quantum Computation. In: Proc. Symp. on Foundations of Computer Science. IEEE (2004). https://doi.org/10.1109/FOCS.2004.8
3. Bian, Z., Chudak, F., Macready, W., Rose, G.: The Ising Model: Teaching an Old Problem New Tricks. Tech. rep., D-Wave Systems (2010)
4. Bauckhage, C., Sanchez, R., Sifa, R.: Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In: Proc. Int. Joint Conf. on Neural Networks. IEEE (2020). https://doi.org/10.1109/IJCNN48605.2020.9206916
5. Hadfield, S.: On the Representation of Boolean and Real Functions as Hamiltonians for Quantum Computing. ACM Trans. on Quantum Computing **2**(4) (2021). https://doi.org/10.1145/3478519
6. Ising, E.: Beitrag zur Theorie des Ferromagnetismus. Zeitschrift für Physik **31** (1925). https://doi.org/10.1007/BF02980577
7. Hopfield, J.: Neural Networks and Physical Systems with Collective Computational Abilities. PNAS **79**(8) (1982). https://doi.org/10.1073/pnas.79.8.2554
8. Gerlach, T., Mücke, S.: Investigating the Relation Between Problem Hardness and QUBO Properties. In: I. Miliou, N. Piatkowski, P. Papapetrou (eds.) Advances in Intelligent Data Analysis, *LNCS*, vol. 14642. Springer (2024). 10.1007/978-3-031-58553-1_14
9. Mücke, S., Gerlach, T., Piatkowski, N.: Optimum-preserving QUBO Parameter Compression. Quantum Macine Intelligence **7**(1) (2025). https://doi.org/10.1007/s42484-024-00219-3

# Chapter 5
# QUBO Models

## 5.1 Introduction

In Chap. 4, we had a closer look at QUBOs. We were mainly concerned with their general characteristics but also considered a few specific examples, namely QUBO models for plain vanilla subset sum problems. However, SSPs are really but the tip of the iceberg of problems that can be expressed in term of QUBOs.

In this chapter, we will therefore look at further QUBO models and how to derive them. Our application examples will consider problems in areas like combinatorial optimization, constraint satisfaction, data mining, and machine learning and are all in all supposed to illustrate that QUBOs provide us with a surprisingly versatile and widely applicable modeling framework [1–3].

However, just as in Chap. 4, we will still not yet discuss QUBO solving but leave this to the later chapters on Hopfield nets and quantum computing. The fact that we can do this underlines our point that there exist general, application independent frameworks for QUBO solving.

Since quantum computing is among these frameworks and since it is now steadily becoming practical, interest in QUBO models has recently grown noticeably. While they have a venerable history in combinatorial optimization and operations research, there are now increasing research efforts on obtaining QUBO formulations of other kinds of problems. Again, this is due to the crucial insight that, if a given problem can be formalized in terms of a QUBO, then it can be solved on (still quite limited present- and hopefully more powerful future) quantum computers [4–8].

Our overall goal with this chapter is thus to demonstrate how to "rethink" familiar, seemingly unrelated problems in terms of an overarching formalism that allows for quantum implementations. This will largely be an exercise in applied linear algebra since we will extensively work with matrix-vector formulations of QUBOs. Having said this, we must avoid the impression that QUBOs are the be-all and end-all of

C. Bauckhage and R. Sifa, *Quantum Computing from Hopfield Nets*, Cognitive Technologies, https://doi.org/10.1007/978-3-031-99402-9_5

modeling or problem solving. Throughout, we will therefore frequently emphasize simplifying modeling assumptions whenever we make them and point out potential limitations they may entail.

## 5.2 What Can QUBOs Do for Us?

Since QUBOs are optimization problems involving vectors over a Boolean domain

$$\mathbb{B} \in \big\{\{0, 1\}, \{\pm 1\}\big\} \, , \tag{5.1}$$

they apply to at least three general kinds of tasks.

First of all, they can model *Boolean satisfiability problems*. These are concerned with Boolean functions over $n$ Boolean variables and ask for an instantiation of those variables such that the function under consideration is satisfied, i.e. evaluates to `true`. Such $n$-SAT problems play an important role in circuit design and verification or in planning and scheduling among others and are notoriously difficult [9]. In theoretical computer science, they are often used in reduction proofs of the NP-completeness of other problems [10]. Expressing an $n$-SAT problem as a QUBO usually relies on Boolean logic in terms of binary- or bipolar vectors as discussed in Chap. 2. Alas, there are subtleties involved and the modeling process is surprisingly tedious [1]. Since we do not intend to open this can of worms, we will not discuss this use case any further.

Second and third of all, QUBOs naturally allow for modeling *subset selection problems* or *bi-partition problems* which frequently arise in the context of artificial intelligence, data mining, pattern recognition, or machine learning.

Letting $\mathcal{X}$ be some application dependent set of size $|\mathcal{X}| = n$, a *subset selection problem* asks for a subset $\mathcal{X}' \subset \mathcal{X}$ whose elements meet certain application dependent criteria. A *bi-partition problem*, on the other hand, asks for a partition $\mathcal{X} = \mathcal{X}_- \cup \mathcal{X}_+$ such that the elements of the two disjoint subsets $\mathcal{X}_-$ and $\mathcal{X}_+$ again meet certain application dependent criteria. Given these definitions, it is clear that subset selection can be thought of a special case of set bi-partition in that $\mathcal{X} = \mathcal{X}' \cup \big(\mathcal{X} \setminus \mathcal{X}'\big)$. Yet, we will continue to distinguish both settings for clarity.

The connection between QUBOs and subset selection- or bi-partition problems is as follows: If we consider, say, bipolar vectors $\boldsymbol{s} \in \{\pm 1\}^n$, we may always think of their entries as *indicator variables*. Dealing with a subset selection problem, these indicators would allow us to define a subset as

$$\mathcal{X}' = \big\{x_j \in \mathcal{X} \;\big|\; s_j = +1\big\} \tag{5.2}$$

and, for a bi-partition problem, we would have

$$\mathcal{X}_- = \left\{x_j \in \mathcal{X} \;\middle|\; s_j = -1\right\} \tag{5.3}$$

$$\mathcal{X}_+ = \left\{x_j \in \mathcal{X} \;\middle|\; s_j = +1\right\}. \tag{5.4}$$

Given a specific subset selection- or bi-partition problem, the "art" is then to model the properties of the sought after subset or partitions in terms of the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ of a bipolar QUBO whose solution $\boldsymbol{s}_*$ represents the solution to the task at hand.

Formulating QUBO models for subset selection- or bi-partition problems is really largely an art or a craft rather than a science. However, our following examples will show that there exist reoccurring principles which experienced modelers can harness to their advantage. For most these examples, we first formulate a binary QUBO and then either use or hint at the use of Theorem 4.1 to translate it into a bipolar QUBO. Finally, note that our examples are really but examples and do by no means constitute a complete list of what is possible with QUBO models.

### *5.2.1 Subset Sum Problems*

We already know how to set up QUBO models for plain vanilla SSPs where we are given inputs $(\mathcal{X}, T) \in \mathbb{Z}^n \times \mathbb{Z}$ and have to find a subset $\mathcal{X}' \subseteq \mathcal{X}$ whose elements sum to $T$. However, there also exist other flavors of SSPs and this is a good place to discuss them. To prepare ourselves for that discussion, be begin by recalling our insights from Chap. 4.

**Plain Vanilla Subset Sum**

Dealing with a plain vanilla SSP $(\mathcal{X}, T)$, we may formalize the problem in terms of the following binary QUBO

$$\boldsymbol{z}_* = \underset{\boldsymbol{z}\in\{0,1\}^n}{\operatorname{argmin}} \left(\boldsymbol{z}^\intercal \boldsymbol{x} - T\right)^2 \tag{5.5}$$

$$= \underset{\boldsymbol{z}\in\{0,1\}^n}{\operatorname{argmin}}\; \boldsymbol{z}^\intercal \left[\boldsymbol{x}\boldsymbol{x}^\intercal\right] \boldsymbol{z} - \left[2\, T \boldsymbol{x}\right]^\intercal \boldsymbol{z} + T^2 \tag{5.6}$$

where the constant vector $\boldsymbol{x} \in \mathbb{Z}^n$ represents set $\mathcal{X}$ and the variable vectors $\boldsymbol{z} \in \{0, 1\}^n$ represent subsets $\mathcal{X}' \subseteq \mathcal{X}$. That is, their entries $z_j \in \{0, 1\}$ are indicator variables which define

$$\mathcal{X}' = \left\{x_j \in \mathcal{X} \;\middle|\; z_j = 1\right\}. \tag{5.7}$$

We also recall that term $T^2$ is a constant independent of the decision variable $\boldsymbol{z}$. Since it thus does not impact the outcome of the argmin operation, we may just as well drop it so that the problem in (5.6) becomes

$$\boldsymbol{z}_* = \underset{\boldsymbol{z}\in\{0,1\}^n}{\operatorname{argmin}}\; \boldsymbol{z}^\intercal \left[\boldsymbol{x}\boldsymbol{x}^\intercal\right] \boldsymbol{z} - \left[2\, T \boldsymbol{x}\right]^\intercal \boldsymbol{z}\,. \tag{5.8}$$

We finally recall that, for every binary QUBO, there exists an equivalent bipolar QUBO. Indeed, if we express $z \in \{0, 1\}^n$ as

$$z = \tfrac{1}{2}[\boldsymbol{s} + \mathbf{1}] \tag{5.9}$$

where $\boldsymbol{s} \in \{\pm 1\}^n$ and $\mathbf{1} \in \mathbb{R}^n$ is the vector of all ones, plug this expression into (5.8), and drop resulting constant terms, we obtain the following

$$\boldsymbol{s}_* = \underset{\boldsymbol{s} \in \{\pm 1\}^n}{\operatorname{argmin}} \ \tfrac{1}{4}\, \boldsymbol{s}^\intercal \big[\boldsymbol{x}\boldsymbol{x}^\intercal\big]\, \boldsymbol{s} + \tfrac{1}{2}\, \big[\boldsymbol{x}\boldsymbol{x}^\intercal \mathbf{1} - 2\, T \boldsymbol{x}\big]^\intercal \boldsymbol{s}\,. \tag{5.10}$$

In short, we can therefore formalize any plain vanilla SSP as a bipolar QUBO over variables $\boldsymbol{s} \in \{\pm 1\}^n$ whose entries $s_j \in \{\pm 1\}$ are indicators which define

$$\mathcal{X}' = \big\{x_j \in \mathcal{X} \;\big|\; s_j = +1\big\}\,. \tag{5.11}$$

This way, we recognize the SSP as a specific instance of a subset selection problem as discussed above.

To wrap up this revision of the plain vanilla SSP and its representation in form of binary- and bipolar QUBOs, we point out that (5.8) and (5.10) are specific instances of the general models

$$\boldsymbol{z}_* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} \ \boldsymbol{z}^\intercal \boldsymbol{Q}\, \boldsymbol{z} + \boldsymbol{q}^\intercal \boldsymbol{z} \tag{5.12}$$

$$\boldsymbol{s}_* = \underset{\boldsymbol{s} \in \{\pm 1\}^n}{\operatorname{argmin}} \ -\tfrac{1}{2}\, \boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s}\ , \tag{5.13}$$

namely instances where the model parameters $\boldsymbol{Q}, \boldsymbol{W} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{q}, \boldsymbol{b} \in \mathbb{R}^n$ are set to

$$\boldsymbol{Q} = +\boldsymbol{x}\boldsymbol{x}^\intercal \tag{5.14}$$

$$\boldsymbol{q} = -2\, T \boldsymbol{x} \tag{5.15}$$

$$\boldsymbol{W} = -\tfrac{1}{2}\, \boldsymbol{x}\boldsymbol{x}^\intercal \tag{5.16}$$

$$\boldsymbol{b} = +\tfrac{1}{2}\, \big[\boldsymbol{x}\boldsymbol{x}^\intercal \mathbf{1} - 2\, T \boldsymbol{x}\big]\,. \tag{5.17}$$

Looking at these specific parameterizations, we finally note that they are interrelated because we have

$$\boldsymbol{W} = -\tfrac{1}{2}\, \boldsymbol{Q} \tag{5.18}$$

$$\boldsymbol{b} = +\tfrac{1}{2}\, \big[\boldsymbol{Q}\, \mathbf{1} + \boldsymbol{q}\big]\,. \tag{5.19}$$

This perfectly agrees with Theorem 4.1 in Chap. 4 which, as we recall, provides us with a general algebraic recipe for translating binary QUBOs into bipolar QUBOs.

## Minimum Size Subset Sum

In Chap. 1, we looked at a specific instance of an SSP whose size was $|\mathcal{X}| = 100$. Larger SSPs like this often have several solutions. Indeed, our specific SSP in (1.1) and (1.2) belongs into this category as it actually comes with, believe it or not, several tens of thousands of solutions. In practice, it is therefore often reasonable to ask: What is the smallest subset $\mathcal{X}'$ of $\mathcal{X}$ whose elements sum to $T$?

This question then leads to the minimum size SSP which we alluded to in Chap. 1 but did not yet formalize. Next, we make up for this and first cast the minimum size SSP as an optimization problem over sets and then rewrite it as a QUBO.

Written in terms of a minimization problem over subsets, the minimum size SSP can be formally expressed as follows

$$\begin{aligned} \mathbf{X}'_* = \underset{\mathcal{X}' \subseteq \mathcal{X}}{\operatorname{argmin}} & \; \big| \mathcal{X}' \big| \\ \text{s.t.} & \; \sum_{x \in \mathcal{X}'} x = T \,. \end{aligned} \tag{5.20}$$

Note what we are doing here: We are searching for a or the smallest subset $\mathcal{X}' \subseteq \mathcal{X}$ *such that* or *subject to* the *constraint* that its elements sum to $T$.

To rewrite this *constrained minimization problem* in terms of binary indicator vectors $z \in \{0, 1\}^n$, we first observe that we can compute the size of $\mathcal{X}'$ as follows

$$\big| \mathcal{X}' \big| = \mathbf{1}^{\intercal} z \,. \tag{5.21}$$

Using what we already know about the plain vanilla SSP, we can therefore formalize the minimum size SSP as a *constrained binary linear optimization* problem

$$\begin{aligned} z_* = \underset{z \in \{0,1\}^n}{\operatorname{argmin}} & \; \mathbf{1}^{\intercal} z \\ \text{s.t.} & \; \big( \boldsymbol{x}^{\intercal} z - T \big)^2 = 0 \,. \end{aligned} \tag{5.22}$$

But this is not a QUBO, i.e. not an *unconstrained quadratic* binary optimization problem. However, the following is

$$z_* = \underset{z \in \{0,1\}^n}{\operatorname{argmin}} \; \mathbf{1}^{\intercal} z + \lambda \big( \boldsymbol{x}^{\intercal} z - T \big)^2 \,. \tag{5.23}$$

Note what we did here: We got rid of the optimization constraint by moving it into the optimization objective. This required us to introduce a **Lagrange multiplier**

$$\lambda > 0 \,. \tag{5.24}$$

The *method of Lagrange multipliers* is a staple of mathematical optimization. There would be much to be said about it (see, for instance, [11, Appendix E]) but this would take us too far afield. Instead, we henceforth treat any Lagrange multipliers we encounter as additional, user defined problem parameters whose specification may require care but is usually nothing to worry about.

To see why we can afford to do this, we assume that $T \neq 0$ and observe that the minimization objective

$$f(z) = \mathbf{1}^\intercal z + \lambda\big(\boldsymbol{x}^\intercal z - T\big)^2 \tag{5.25}$$

in (5.23) consists of a sum of two non-negative terms. In order for this sum to be as small as possible, both its contributing terms must be as small as possible. The first term is minimal if $z = \mathbf{0}$ because $\mathbf{1}^\intercal\mathbf{0} = 0$ is its minimum value. However, in this case the second term evaluates to $\lambda(\boldsymbol{x}^\intercal\mathbf{0} - T)^2 = \lambda T^2 > 0$. On the other hand, the smallest possible value of the second term is also 0 and occurs whenever $\boldsymbol{x}^\intercal z = T$, i.e. whenever $z$ indicates a solution to the given subset sum problem. Yet, for such a solution we will have $\mathbf{1}^\intercal z > 0$.

In other words, we are dealing with a trade-off between two objectives, namely finding an SSP solution ($\boldsymbol{x}^\intercal z_* = T$) and minimizing its size ($\mathbf{1}^\intercal z_* \leq \mathbf{1}^\intercal z$ for all $z$).

The Lagrange multiplier $\lambda$ controls this trade-off. If $\lambda$ is large, it will severely penalize any $z$ for which $(\boldsymbol{x}^\intercal z - T)^2 > 0$, that is, it will penalize any $z$ which does not solve the given SSP. Any reasonable algorithm we might unleash on our minimization problem should thus have a preference for returning solutions that solve the SSP. At the same time, among all the $z$ which solve the SSP, that is, among all the $z$ for which $(\boldsymbol{x}^\intercal z - T)^2 = 0$, there will be some with a smaller value for $\mathbf{1}^\intercal z$ than others. Any reasonable algorithm should therefore automatically prefer solutions which not only solve the SSP but are also small.

In short, using what we already know about the plain vanilla SSP, we find that the following is indeed a valid binary QUBO model for the minimum size SSP

$$\begin{aligned} z_* &= \underset{z \in \{0,1\}^n}{\operatorname{argmin}} \; \mathbf{1}^\intercal z + \lambda\big[z^\intercal \boldsymbol{x}\boldsymbol{x}^\intercal z - 2\,T\,\boldsymbol{x}^\intercal z\big] && (5.26) \\ &= \underset{z \in \{0,1\}^n}{\operatorname{argmin}} \; z^\intercal\big[\lambda\,\boldsymbol{x}\boldsymbol{x}^\intercal\big]z + \big[\mathbf{1} - 2\,\lambda\,T\boldsymbol{x}\big]^\intercal z && (5.27) \end{aligned}$$

which we furthermore recognize to be of the standard form in Definition 4.1. Using Theorem 4.1, an equivalent bipolar QUBO model is easily obtained.

## Size Restricted Subset Sum

Another practically useful idea is to search for SSP solutions $\mathcal{X}'$ of size $|\mathcal{X}'| = K$ where $K$ is a user defined constant. Given what we already know, we can formalize this size restricted subset sum problem as follows

$$\begin{aligned} z_* = \underset{z \in \{0,1\}^n}{\operatorname{argmin}} \; & z^\intercal \boldsymbol{x}\boldsymbol{x}^\intercal z - 2\,T\,\boldsymbol{x}^\intercal z \\ \text{s.t.} \;\; & \mathbf{1}^\intercal z = K\,. \end{aligned} \tag{5.28}$$

Or, noting the equivalency $\mathbf{1}^\intercal z = K \Leftrightarrow (\mathbf{1}^\intercal z - K)^2 = 0$, we can equivalently ask for

$$\begin{aligned} z_* = \underset{z \in \{0,1\}^n}{\operatorname{argmin}} \; & z^\intercal \boldsymbol{x}\boldsymbol{x}^\intercal z - 2\,T\,\boldsymbol{x}^\intercal z \\ \text{s.t.} \; & \big(\mathbf{1}^\intercal z - K\big)^2 = 0 \end{aligned} \tag{5.29}$$

which involves an equal-to-zero constraint. This is preferable as it again allows us to resort to the method of Lagrange multipliers and thus to turn the original constrained problem into an unconstrained one.

In fact, given all we have seen so far, it is easily verified that the following is a standard binary QUBO model for the size restricted SSP

$$z_* = \underset{z \in \{0,1\}^n}{\operatorname{argmin}} \; z^\intercal \boldsymbol{x}\boldsymbol{x}^\intercal z - 2\,T\,\boldsymbol{x}^\intercal z + \lambda\big(\mathbf{1}^\intercal z - K\big)^2 \tag{5.30}$$

$$= \underset{z \in \{0,1\}^n}{\operatorname{argmin}} \; z^\intercal \boldsymbol{x}\boldsymbol{x}^\intercal z - 2\,T\,\boldsymbol{x}^\intercal z + \lambda\, z^\intercal \mathbf{1}\mathbf{1}^\intercal z - 2\,\lambda\,K\,\mathbf{1}^\intercal z \tag{5.31}$$

$$= \underset{z \in \{0,1\}^n}{\operatorname{argmin}} \; z^\intercal \big[\boldsymbol{x}\boldsymbol{x}^\intercal + \lambda\,\mathbf{1}\mathbf{1}^\intercal\big] z - 2\big[T\,\boldsymbol{x} + \lambda\,K\,\mathbf{1}\big]^\intercal z\,. \tag{5.32}$$

A corresponding bipolar QUBO model follows again immediately from Theorem 4.1.

**Remarks and Further Reading**

Of the three SSP flavors we just discussed, the minimum size SSP is harder than the other two. Consider this: For all three, worst case solving efforts are $O(n2^n)$ but plain vanilla- and size restricted SSPs can be verified in $O(n)$. It is easy to verify if a subset has size $K$ and sums to $T$. The latter is easy for minimum size SSPs, too, but how can we rest assured that a purported solution is minimal? Verifying this would itself require efforts of $O(2^n)$.

In fact, minimum size or shortest length constraints are what make many combinatorial problems NP-hard. Examples include *set cover* and *traveling sales person* for which there also exist QUBO models so that they could be solved on quantum computers. Alas, models for problems like these tend to be rather complex and often involve *auxiliary*- and/or *slack variables*. As we will not look into those, interested readers may consult the expositions in [1, 3, 12, 13].

### *5.2.2 K-Nearest Neighbors*

Searching $K$ nearest neighbors among high dimensional vectors is an important task in information retrieval and modern AI where vector databases play an increasingly prominent role. The general setting is as follows: Given a set $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ of $n$ data points $\boldsymbol{x}_j \in \mathbb{R}^m$ and a query point $\boldsymbol{y} \in \mathbb{R}^m$, find those $K \geq 1$ points in $\mathcal{X}$ whose distance to $\boldsymbol{y}$ is smallest. Figure 5.1 illustrates the idea for $m = 2$, $n = 16$, and $K = 5$.

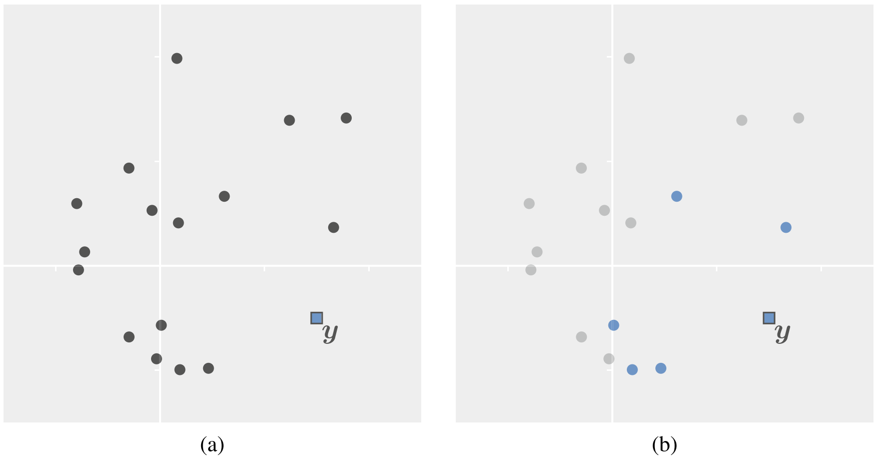


**Fig. 5.1** Didactic illustration of the problem of $K = 5$ nearest neighbor search. **a** Data points $\boldsymbol{x}_j \in \mathbb{R}^2$ and query point $\boldsymbol{y} \in \mathbb{R}^2$. **b** $K = 5$ nearest neighbors of $\boldsymbol{y}$

While this obviously constitutes a subset selection problem, it is rather benign. In contrast to many of its peers, it is not NP-hard but can actually be solved in a running time linear in $n$ (can you devise a corresponding algorithm?). While there is thus no practical advantage in casting $K$ nearest neighbor search as a QUBO, we will do it anyway just to show that we can and to get to know further useful principles.

Thinking in terms of subsets $\mathcal{X}' \subseteq \mathcal{X}$, the $K$-nearest neighbor problem can be formalized as a constrained minimization problem like this

$$\begin{aligned} \mathcal{X}'_* = \underset{\mathcal{X}' \subseteq \mathcal{X}}{\operatorname{argmin}} & \sum_{\boldsymbol{x}_j \in \mathcal{X}'} \left\| \boldsymbol{x}_j - \boldsymbol{y} \right\|^2 \\ \text{s.t.} & \left| \mathcal{X}' \right| = K \,. \end{aligned} \tag{5.33}$$

Note that we are working with squared Euclidean distances. In our current setting, this is possible because we are not interested in distance values themselves but in data points which minimize them. In other words, we are interested in relations among distances and for those we note the following equivalence

$$\left\| \boldsymbol{x}_j - \boldsymbol{y} \right\| \leq \left\| \boldsymbol{x}_k - \boldsymbol{y} \right\| \Leftrightarrow \left\| \boldsymbol{x}_j - \boldsymbol{y} \right\|^2 \leq \left\| \boldsymbol{x}_k - \boldsymbol{y} \right\|^2 . \tag{5.34}$$

Working with squared Euclidean distances often has algebraic advantages (we will see some of them soon) but always saves the hassle of having to compute square roots which is rather costly and should thus be avoided whenever possible.

If we next represent $\mathcal{X}'$ in terms of an indicator vector $\boldsymbol{z} \in \{0, 1\}^n$ and recall the equivalence $|\mathcal{X}'| = K \Leftrightarrow (\mathbf{1}^\intercal \boldsymbol{z} - K)^2 = 0$, we can equivalently express (5.33) as

$$\begin{aligned} \boldsymbol{z}_* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} & \sum_{j=1}^{n} z_j \cdot \left\| \boldsymbol{x}_j - \boldsymbol{y} \right\|^2 \\ \text{s.t.} & \left( \mathbf{1}^\intercal \boldsymbol{z} - K \right)^2 = 0 \,. \end{aligned} \tag{5.35}$$

A truly convenient idea is then to *pre-compute* a distance vector $\boldsymbol{d} \in \mathbb{R}^n$ whose entries are given by

$$d_j = \left\| \boldsymbol{x}_j - \boldsymbol{y} \right\|^2 . \tag{5.36}$$

Working with $\boldsymbol{d}$ then allows us to write (5.35) in a form we have seen before, namely

$$\begin{aligned} \boldsymbol{z}_* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} & \; \boldsymbol{d}^\intercal \boldsymbol{z} \\ \text{s.t.} & \left( \mathbf{1}^\intercal \boldsymbol{z} - K \right)^2 = 0 \,. \end{aligned} \tag{5.37}$$

Given what we already know about Lagrange multipliers and the irrelevance of constant terms, we therefore find the following binary QUBO model for the $K$-nearest neighbor problem

$$\boldsymbol{z}_* = \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} \; \boldsymbol{d}^\intercal \boldsymbol{z} + \lambda \left( \mathbf{1}^\intercal \boldsymbol{z} - K \right)^2 \tag{5.38}$$

$$= \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} \; \boldsymbol{d}^\intercal \boldsymbol{z} + \lambda \, \boldsymbol{z}^\intercal \mathbf{1} \mathbf{1}^\intercal \boldsymbol{z} - 2 \, \lambda \, K \, \mathbf{1}^\intercal \boldsymbol{z} \tag{5.39}$$

$$= \underset{\boldsymbol{z} \in \{0,1\}^n}{\operatorname{argmin}} \; \boldsymbol{z}^\intercal \left[ \lambda \, \mathbf{1} \mathbf{1}^\intercal \right] \boldsymbol{z} + \left[ \boldsymbol{d} - 2 \, \lambda \, K \, \mathbf{1} \right]^\intercal \boldsymbol{z} \,. \tag{5.40}$$

This obviously is in standard form (with $\boldsymbol{Q} = \lambda \, \mathbf{1}\mathbf{1}^\intercal$ and $\boldsymbol{q} = \boldsymbol{d} - 2 \, \lambda \, K \, \mathbf{1}$) so that a corresponding bipolar QUBO model follows immediately from Theorem 4.1.

**Remarks and Further Reading**

By pre-computing a distance vector $\boldsymbol{d}$, we turned the $K$-nearest neighbor problem in $\mathbb{R}^m$ into the problem of having to find the $K$ smallest elements of an ordered set of numbers. What we just did is therefore not restricted to (squared) Euclidean distance vectors. Indeed, we could generally work with $d_j = d(\boldsymbol{x}_j, \boldsymbol{y})$ where $d(\cdot\,,\cdot)$ is any valid distance measure. What we just did also extends to problems involving distance matrices. There exist many relevant instances of those but they are typically NP-hard. An example is the max-sum diversification problem for which QUBO models and solvers often lead to very good solutions [14]. We mentioned the max-sum diversification problem because it is closely connected to the $K$-clique problem in graph theory [15] which means that we we did here extends to that venerable field as well.

### *5.2.3 K-Means Clustering*

$K$-means clustering is a central concept in data science and unsupervised machine learning. Given a sample $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ of $n$ data points $\boldsymbol{x}_j \in \mathbb{R}^m$, it partitions the data into $K$ disjoint subsets or *clusters* $\mathcal{X}_k \subset \mathcal{X}$ of "similar" data points.

To formalize the notion of similar data points, $K$-means clustering models clusters in terms of prototypes and similarities in terms of Euclidean distances between prototypes and data points. In particular, it defines clusters and their prototypes as

$$\mathcal{X}_k = \Big\{ \boldsymbol{x}_j \in \mathcal{X} \;\Big|\; \big\| \boldsymbol{x}_j - \boldsymbol{\mu}_k \big\|^2 \leq \big\| \boldsymbol{x}_j - \boldsymbol{\mu}_l \big\|^2 \, \forall \, l \neq k \Big\} \tag{5.41}$$

$$\boldsymbol{\mu}_k = \frac{1}{|\mathcal{X}_k|} \sum_{\boldsymbol{x}_j \in \mathcal{X}_k} \boldsymbol{x}_j \tag{5.42}$$

where we recognize $\boldsymbol{\mu}_k$ as the arithmetic *mean* of the data in $\mathcal{X}_k$. Given these definitions, the problem of finding an optimal clustering is then commonly formalized as the problem of minimizing the *within cluster scatter*

$$S_W = \sum_{k=1}^{K} \sum_{\boldsymbol{x}_j \in \mathcal{X}_k} \big\| \boldsymbol{x}_j - \boldsymbol{\mu}_k \big\|^2 \tag{5.43}$$

with respect to the possible prototypes $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K \in \mathbb{R}^m$.

In the following, we consider the simplest version of this problem, namely $K = 2$ means clustering as in Fig. 5.2 where $\mathcal{X}$ is to be partitioned into only two disjoint clusters $\mathcal{X}_1$ and $\mathcal{X}_2$ of sizes $|\mathcal{X}_1| = n_1$ and $|\mathcal{X}_2| = n_2$ such that $n_1 + n_2 = n$. Our goal is to devise a QUBO model for this setting and next elaborate on corresponding ideas first reported in [16].

For convenience and without loss of generality, we assume that the given data points have been normalized to zero mean so that

$$\boldsymbol{\mu} = \frac{1}{n} \sum_{j=1}^{n} \boldsymbol{x}_j = \frac{1}{n} \sum_{k=1}^{2} \sum_{\boldsymbol{x}_j \in \mathcal{X}_k} \boldsymbol{x}_j = \frac{1}{n} \big[ n_1 \, \boldsymbol{\mu}_1 + n_2 \, \boldsymbol{\mu}_2 \big] = \boldsymbol{0} \tag{5.44}$$

where $\boldsymbol{\mu}$, $\boldsymbol{\mu}_1$, and $\boldsymbol{\mu}_2$ denote the overall sample mean and the means of clusters $\mathcal{X}_1$ and $\mathcal{X}_2$, respectively. Note that our assumption implies $n_1 \, \boldsymbol{\mu}_1 = -n_2 \, \boldsymbol{\mu}_2$ which is to say that the two means $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ of any two non-empty, disjoints subsets $\mathcal{X}_1$ and $\mathcal{X}_2$ of $\mathcal{X}$ will be of opposite sign.

Now, conventional approaches would solve $K = 2$ means clustering by minimizing the within cluster scatter in (5.43). Indeed, the well known algorithms of Lloyd [17], Hartigan [18], or MacQueen [19] all consider this objective. However, we will follow a different route and observe that minimizing the above within cluster scatter is equivalent to maximizing the following *between cluster scatter*

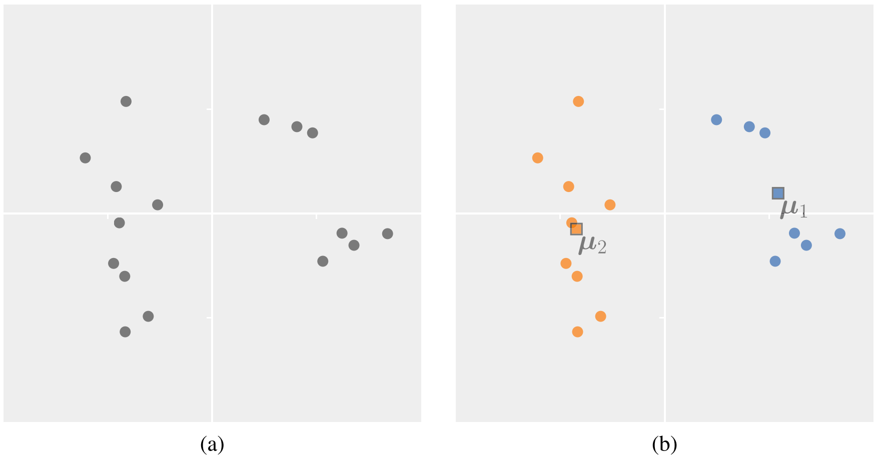


**Fig. 5.2** Didactic illustration of the problem of $K = 2$ means clustering. **a** Data points $\boldsymbol{x}_j \in \mathbb{R}^2$. **b** $K = 2$ means clustering result

$$S_B = \sum_{k_1=1}^{K} \sum_{k_2=1}^{K} n_{k_1} n_{k_2} \left\| \boldsymbol{\mu}_{k_1} - \boldsymbol{\mu}_{k_2} \right\|^2 . \tag{5.45}$$

This claim holds for $K > 1$ and follows from Fisher's analysis of variance [20, 21] which establishes that the *total scatter* of the data can be written as

$$S_T = \sum_{\boldsymbol{x}_j \in \mathcal{X}} \left\| \boldsymbol{x}_j - \boldsymbol{\mu} \right\|^2 = S_W + \frac{1}{2n} S_B \tag{5.46}$$

and implies that any decrease of $S_W$ in (5.43) entails an increase of $S_B$ in (5.45).

For our specific setting with $K = 2$, it is an easy exercise for the reader to verify that the maximization objective in (5.45) can be simplified to

$$S_B = 2\, n_1\, n_2 \left\| \boldsymbol{\mu}_1 - \boldsymbol{\mu}_2 \right\|^2 . \tag{5.47}$$

Interestingly, this simplification provides us with an intuition as to why general $K$-means clustering often tends to produce clusters of about equal sizes: In order for the between scatter in (5.47) to be large, both the distance $\|\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\|$ between the two cluster centers and the product $n_1 n_2$ of the two cluster sizes have to be large. But, since the sum $n_1 + n_2 = n$ is fixed, the product of the cluster sizes will be maximal if $n_1 = n_2 = \frac{n}{2}$ (see Exercise 5.1).

Given this observation, we next boldly assume that the two clusters that result from maximizing (5.47) are of about the same size $n_1 \approx n_2 \approx \frac{n}{2}$. This ansatz is not generally justified but allows us to approximate

$$2\, n_1\, n_2 \left\| \boldsymbol{\mu}_1 - \boldsymbol{\mu}_2 \right\|^2 \approx 2\, \tfrac{n^2}{4} \left\| \boldsymbol{\mu}_1 - \boldsymbol{\mu}_2 \right\|^2 \tag{5.48}$$

Working with this approximation, we then have

$$2\, \tfrac{n^2}{4} \left\| \boldsymbol{\mu}_1 - \boldsymbol{\mu}_2 \right\|^2 = 2 \left\| \tfrac{n}{2} \big[ \boldsymbol{\mu}_1 - \boldsymbol{\mu}_2 \big] \right\|^2 = 2 \left\| n_1\, \boldsymbol{\mu}_1 - n_2\, \boldsymbol{\mu}_2 \right\|^2 \tag{5.49}$$

which allows us to formalize the problem of $K = 2$ means clustering as the problem of having to solve

$$\boldsymbol{\mu}_{*1},\ \boldsymbol{\mu}_{*2} = \underset{\boldsymbol{\mu}_1,\, \boldsymbol{\mu}_2}{\operatorname{argmax}} \left\| n_1\, \boldsymbol{\mu}_1 - n_2\, \boldsymbol{\mu}_2 \right\|^2 . \tag{5.50}$$

To appreciate what this approximation of our clustering problem buys us, we next observe that the squared norm in the objective of (5.50) can be expressed in a form that does not explicitly depend on the two decision variables $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$. In order to see this, we first gather the given data point in a data matrix

$$\boldsymbol{X} = \begin{bmatrix} | & | & & | \\ \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_n \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{5.51}$$

and second represent the two sought after clusters $\mathcal{X}_1$ and $\mathcal{X}_2$ in terms of binary indicator vectors $\boldsymbol{z}_1, \boldsymbol{z}_2 \in \{0, 1\}^n$. This way, we can write

$$n_1\, \boldsymbol{\mu}_1 = \boldsymbol{X} \boldsymbol{z}_1 \tag{5.52}$$

$$n_2\, \boldsymbol{\mu}_2 = \boldsymbol{X} \boldsymbol{z}_2 \tag{5.53}$$

and consequently find that

$$\left\| n_1\, \boldsymbol{\mu}_1 - n_2\, \boldsymbol{\mu}_2 \right\|^2 = \left\| \boldsymbol{X} \big[ \boldsymbol{z}_1 - \boldsymbol{z}_2 \big] \right\|^2 = \left\| \boldsymbol{X} \boldsymbol{s} \right\|^2 \tag{5.54}$$

Note that the vector $\boldsymbol{s}$ we just introduced in (5.54) is guaranteed to be a bipolar vector because, in hard $K$ means clustering, every data point is assigned to one and only one cluster. The difference of the two binary cluster indicator vectors $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$ therefore amounts to

$$\boldsymbol{z}_1 - \boldsymbol{z}_2 = \boldsymbol{z}_1 - \big[ \mathbf{1} - \boldsymbol{z}_1 \big] = 2\, \boldsymbol{z}_1 - \mathbf{1} = \boldsymbol{s} \in \{\pm 1\}^n \tag{5.55}$$

This then immediately establishes that there exists a bipolar QUBO formulation of the problem of $K = 2$ means clustering of zero mean data. Since

$$\left\| \boldsymbol{X} \boldsymbol{s} \right\|^2 = \boldsymbol{s}^\mathsf{T} \boldsymbol{X}^\mathsf{T} \boldsymbol{X}\, \boldsymbol{s} \tag{5.56}$$

is convex in $\boldsymbol{s}$, the maximization problem in (5.50) is equivalent to the following minimization problem

$$\boldsymbol{s}_* = \underset{\boldsymbol{s} \in \{0,1\}^n}{\operatorname{argmin}} -\boldsymbol{s}^\intercal \boldsymbol{X}^\intercal \boldsymbol{X}\, \boldsymbol{s} \ . \tag{5.57}$$

This, in turn, can be recognized as a special case of the general form prescribed by Definition 4.1, namely the case where $\boldsymbol{W} = 2\, \boldsymbol{X}^\intercal \boldsymbol{X}$ and $\boldsymbol{b} = \boldsymbol{0}$.

Finally, we note that our zero mean assumption in (5.44) guarantees that any solution to (5.57) will be a vector $\boldsymbol{s}_*$ whose entries are not all equal and therefore induce a bi-partition of the data gathered in matrix $\boldsymbol{X}$.

**Remarks and Further Reading**

Our idea for how to set up a binary QUBO model for $K = 2$ means clustering was based on an approximative ansatz. Informed modeling choices such as this are not uncommon when working with QUBOs and we will see another example soon.

Bi-partition clustering with $n$ indicator variables for $n$ data points may appear trivial but our approach is versatile; it can be kernelized [22] and extends to relational clustering [16]. Also, if we were willing to work with $nK$ indicator variables, it would be straightforward to devise QUBO models for $K > 2$ means clustering [23]. Finally, we could consider more sophisticated approaches to vector quantization or feature selection to devise QUBOs for more general clustering tasks [24–26]. Interested readers are encouraged to consult the cited literature.

### 5.2.4 Linear Regression

Linear regression is yet another fundamental task in data science and data driven learning.The general multivariate setting is as follows: Given a training data sample $\big\{(\boldsymbol{x}_j, y_j)\big\}_{j=1}^n$ consisting of $n$ pairs of *regressors* $\boldsymbol{x}_j \in \mathbb{R}^m$ and *regressands* $y_j \in \mathbb{R}$, fit a multivariate linear model $f : \mathbb{R}^m \to \mathbb{R}$ which minimizes the *least squares loss*

$$L = \sum_{j=1}^{n} \Big( f(\boldsymbol{x}_j) - y_j \Big)^2 . \tag{5.58}$$

Figure 5.3 illustrates this task for an example where $m = 1$ and $n = 16$ and we next recall how it is typically solved.

To begin with, we recall or observe that a multivariate linear function $f$ of $\boldsymbol{x} \in \mathbb{R}^m$ is a function of the form $f(\boldsymbol{x}) = f(x_1, x_2, \ldots, x_m) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$ and that the task of fitting it to a given set of data is an exercise in estimating optimal parameters $w_0, w_1, \ldots, w_m \in \mathbb{R}$. We further observe that such a function is an inner product in disguise. Indeed, introducing these *feature*- and *weight vectors*

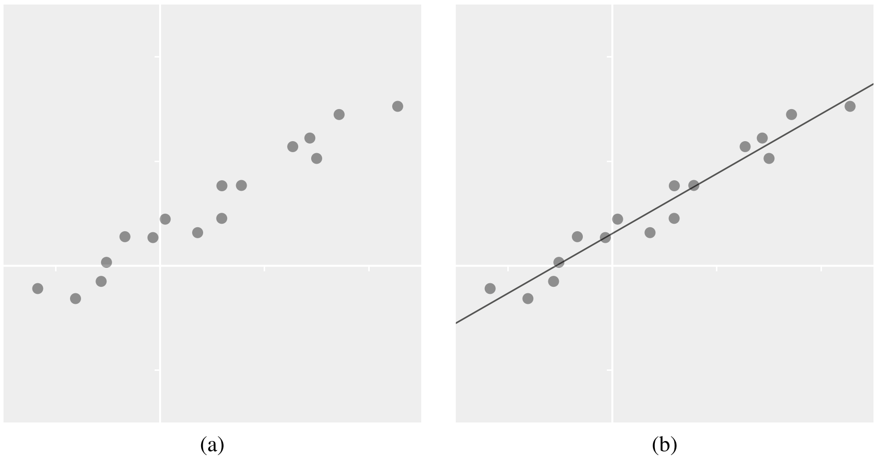


**Fig. 5.3** Didactic illustration of the problem of linear regression. **a** Training data $(x_j, y_j) \in \mathbb{R}^2$. **b** Linear model fitted to the data

$$\boldsymbol{\varphi}(\boldsymbol{x}) = \left[\, 1\,,\, x_1, \ldots, x_m \right]^\mathsf{T} \in \mathbb{R}^{m+1} \tag{5.59}$$

$$\boldsymbol{w} = \left[ w_0, w_1, \ldots, w_m \right]^\mathsf{T} \in \mathbb{R}^{m+1} \tag{5.60}$$

allows us to write a linear function $f(\boldsymbol{x})$ like this

$$f(\boldsymbol{x}) = \boldsymbol{\varphi}(\boldsymbol{x})^\mathsf{T} \boldsymbol{w}\ . \tag{5.61}$$

If we then let $\boldsymbol{\varphi}_j = \boldsymbol{\varphi}(\boldsymbol{x}_j)$ for each $\boldsymbol{x}_j$ in the given training data and furthermore introduce the following *feature matrix*- and *target vector* representations of our data

$$\boldsymbol{\Phi} = \begin{bmatrix} | & | & & | \\ \boldsymbol{\varphi}_1 & \boldsymbol{\varphi}_2 & \cdots & \boldsymbol{\varphi}_n \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{(m+1)\times n} \tag{5.62}$$

$$\boldsymbol{y} = \left[ y_1,\, y_2,\, \ldots,\, y_n \right]^\mathsf{T} \in \mathbb{R}^n\ , \tag{5.63}$$

we can rewrite the least squares loss in (5.58) as a matrix-vector expression, because

$$\sum_{j=1}^{n} \Big( f(\boldsymbol{x}_j) - y_j \Big)^2 = \sum_{j=1}^{n} \Big( \boldsymbol{\varphi}_j^\mathsf{T} \boldsymbol{w} - y_j \Big)^2 = \left\| \boldsymbol{\Phi}^\mathsf{T} \boldsymbol{w} - \boldsymbol{y} \right\|^2 . \tag{5.64}$$

This way, it becomes clear that linear regression is nothing but an unconstrained quadratic minimization problem, namely

$$\begin{aligned}
\boldsymbol{w}_* &= \underset{\boldsymbol{w}\in\mathbb{R}^{m+1}}{\operatorname{argmin}} \ \left\| \boldsymbol{\Phi}^\intercal \boldsymbol{w} - \boldsymbol{y} \right\|^2 && (5.65)\\
&= \underset{\boldsymbol{w}\in\mathbb{R}^{m+1}}{\operatorname{argmin}} \ \left[\boldsymbol{\Phi}^\intercal \boldsymbol{w} - \boldsymbol{y}\right]^\intercal \left[\boldsymbol{\Phi}^\intercal \boldsymbol{w} - \boldsymbol{y}\right] && (5.66)\\
&= \underset{\boldsymbol{w}\in\mathbb{R}^{m+1}}{\operatorname{argmin}} \ \boldsymbol{w}^\intercal \boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal \boldsymbol{w} - 2\,\boldsymbol{y}^\intercal \boldsymbol{\Phi}^\intercal \boldsymbol{w} + \boldsymbol{y}^\intercal \boldsymbol{y} && (5.67)\\
&= \underset{\boldsymbol{w}\in\mathbb{R}^{m+1}}{\operatorname{argmin}} \ \boldsymbol{w}^\intercal \boldsymbol{\Phi}\,\boldsymbol{\Phi}^\intercal \boldsymbol{w} - 2\,\boldsymbol{y}^\intercal \boldsymbol{\Phi}^\intercal \boldsymbol{w} && (5.68)
\end{aligned}$$

where the last step is possible because $\boldsymbol{y}^\intercal \boldsymbol{y}$ is a constant independent of the decision variable $\boldsymbol{w}$.

While the optimization problem in (5.68) already resembles a QUBO, it is of course not because its decision variable is a general real valued vector rather than a binary- or bipolar one. However, using an approach due to Date and Potok [27], we can *approximate* the above regression problem as a binary QUBO.

The compelling idea is to approximate each model parameter $w_j$ as a linear combination over a pre-defined choice of positive and negative powers of the number 2. For practical computations, it will be convenient to collect these powers of 2 in a vector such as, say

$$\boldsymbol{p} = \left[+2, -2, +1, -1, +\tfrac{1}{2}, -\tfrac{1}{2}, +\tfrac{1}{4}, -\tfrac{1}{4}, +\tfrac{1}{8}, -\tfrac{1}{8}, +\tfrac{1}{16}, -\tfrac{1}{16}\right]^\intercal \in \mathbb{R}^{12}\,. \qquad (5.69)$$

Why this particular choice? Because it turns out that the simple univariate linear model we fitted in Fig. 5.3b is given by $f(x) = 0.309 + 0.572\,x$ and the two parameters of this model can be quite accurately approximated as

$$\begin{aligned}
w_0 &= 0.309 \approx \tfrac{1}{4} + \tfrac{1}{16} = 0.3125 && (5.70)\\
w_1 &= 0.572 \approx \tfrac{1}{2} + \tfrac{1}{16} = 0.5625\,. && (5.71)
\end{aligned}$$

Of course, our vector $\boldsymbol{p}$ contains more numbers than just $+\frac{1}{2}$, $+\frac{1}{4}$, and $+\frac{1}{16}$ but this is exactly the point. In practice, we do not already know the optimal values of the regression parameters $w_j$ but have to determine them in a data driven manner. Hence, if we decide to do this only approximately in terms of linear combinations of finitely many powers of 2 but still want to have decent accuracy, we need fairly large choices of such powers from which to assemble appropriate model parameters.

To see what we mean by assembling appropriate approximative model parameters from a coefficient vector $\boldsymbol{p}$, we stick with our example and consider the two particular 12-dimensional binary vectors

$$\begin{aligned}
\boldsymbol{z}_0 &= \left[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0\right]^\intercal && (5.72)\\
\boldsymbol{z}_1 &= \left[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0\right]^\intercal && (5.73)
\end{aligned}$$

because these now allow us to write the approximations of the above coefficients as

$$w_0 \approx \boldsymbol{p}^\intercal \boldsymbol{z}_0 = 0.3125 \tag{5.74}$$
$$w_1 \approx \boldsymbol{p}^\intercal \boldsymbol{z}_1 = 0.5625 \; . \tag{5.75}$$

If we furthermore let $\mathbf{0} \in \mathbb{R}^{12}$ denote the vector of all zeros and also introduce the following matrix and vector

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{p}^\intercal & \mathbf{0}^\intercal \\ \mathbf{0}^\intercal & \boldsymbol{p}^\intercal \end{bmatrix} \in \mathbb{R}^{2\times 24} \quad \text{and} \quad \boldsymbol{z} = \begin{bmatrix} \boldsymbol{z}_0 \\ \boldsymbol{z}_1 \end{bmatrix} \in \{0, 1\}^{24} \; , \tag{5.76}$$

we can write the two equations in (5.74) and (5.75) as a single matrix-vector equation

$$\boldsymbol{w} \approx \boldsymbol{P}\boldsymbol{z} \; . \tag{5.77}$$

Note that this approach works not just with the two specific 12- and 2-dimensional vectors $\boldsymbol{p}$ and $\boldsymbol{w}$ from our running example. Instead, the underlying algebraic idea generalizes. That is, we can always approximate any real number $w_j \in \mathbb{R}$ as an inner product

$$w_j \approx \boldsymbol{p}^\intercal \boldsymbol{z}_j \tag{5.78}$$

between a powers-of-two vector $\boldsymbol{p}$ and a binary vector $\boldsymbol{z}$ that indicates which powers of 2 to select for the approximation. In order for such an approximation to be as accurate as possible, we may need to work with potentially high dimensional vectors

$$\boldsymbol{p} \in \mathbb{R}^M \tag{5.79}$$
$$\boldsymbol{z}_j \in \{0, 1\}^M \tag{5.80}$$

but we are generally free to choose the dimension $M$ and entries of vector $\boldsymbol{p}$ to our liking. By the same token, we can always approximate any real vector $\boldsymbol{w} \in \mathbb{R}^{m+1}$ as a matrix-vector product

$$\boldsymbol{w} \approx \boldsymbol{P}\boldsymbol{z} \tag{5.81}$$

where

$$\boldsymbol{z} \in \{0, 1\}^{(m+1)M} \tag{5.82}$$

is now a potentially very high dimensional binary vector and notably and crucially

$$\boldsymbol{P} = \boldsymbol{I} \otimes \boldsymbol{p}^\intercal \in \mathbb{R}^{(m+1)\times(m+1)M} \tag{5.83}$$

results from a Kronecker product of the identity matrix $\boldsymbol{I} \in \mathbb{R}^{(m+1)\times(m+1)}$ and the transpose of a given powers-of-two vector $\boldsymbol{p}$.

If we then plug the approximation in (5.81) into the quadratic minimization problem in (5.68), we obtain the following approximative binary QUBO model for multivariate linear regression

$$z_* = \underset{z \in \{0,1\}^{(m+1)M}}{\operatorname{argmin}} \; z^\intercal P^\intercal \Phi \, \Phi^\intercal P z - 2\, y^\intercal \Phi^\intercal P z \,. \tag{5.84}$$

If we had an algorithm that could find an optimal solution to this QUBO, we could use this solution to optimally approximate the parameters $\boldsymbol{w}_* = \boldsymbol{P} \boldsymbol{z}_*$ of the regression problem we are actually concerned with. Finally, since the above binary QUBO is in standard form (with $\boldsymbol{Q} = \boldsymbol{P}^\intercal \boldsymbol{\Phi}\, \boldsymbol{\Phi}^\intercal \boldsymbol{P}$ and $\boldsymbol{q} = -2\, [\boldsymbol{y}^\intercal \boldsymbol{\Phi}^\intercal \boldsymbol{P}]^\intercal$), a corresponding bipolar QUBO once again follows immediately from Theorem 4.1.

**Remarks and Further Reading**

What we just did can be further generalized. If we wanted to, we could consider different vectors $\boldsymbol{p}_j$ for different model parameters $w_j$. If we furthermore wanted to, we could also consider different dimensions $M_j$ for the different $\boldsymbol{p}_j$. Note, however, that either extension would no longer allow us to compute matrix $\boldsymbol{P}$ as a simple Kronecker product. Finally, we are not at all restricted to working with powers of 2. Any other base would work as well; in fact, we do not even have to consider powers of some number but could gather any reasonable (likely problem-specific) choice of numbers in $\boldsymbol{p}$ or in the different $\boldsymbol{p}_j$.

The fact that we can approximate linear regression problems in terms of QUBO models has far reaching implications. For instance, many machine learning methods such as perceptron learning, support vector machine training, dimensionality reduction via principal component analysis, or $K$-means clustering involve various forms of regression problems and Date et al. discuss QUBO models for some of those [28]. Again, this is of current interest because it indicates that a wide variety of machine learning models can be ported to quantum computers.

Finally, we note that we can interpret all the above in terms of *approximative subset sum problems* where we are given some $w_j$ and $\boldsymbol{p}$ and have to find a binary $\boldsymbol{z}$ such that $(\boldsymbol{p}^\intercal \boldsymbol{z} - w_j)^2$ becomes minimal. While this idea of *approximating* real numbers $w_j$ in terms inner products $\boldsymbol{p}^\intercal \boldsymbol{z}$ is very general, applies far beyond regression, and further hints at the versatility of QUBOs, its numerical accuracy is necessarily restricted by the resolution (quantity and quality) of the numbers gathered in $\boldsymbol{p}$.

### *5.2.5 K-Rooks*

Our final use case in this chapter is a symbolic *constraint satisfaction problem*. The $K$-rooks problem asks for a placement of $K$ rooks on a $K \times K$ chessboard such that they do not threaten each other (see Fig. 5.4). It thus constitutes a constrained subset selection problem where $K$ out of $n = K^2$ squares need to be identified.

Note that this is a rather didactic problem which is not at all difficult to solve. For instance, the following algorithm does the trick: Starting with an empty $K \times K$ board, randomly pick a square in the first column and place the first rook there. This leaves $K - 1$ unthreatened squares in the second column; randomly pick one of those and place the second rook there. This leaves $K - 2$ unthreatened squares in the third

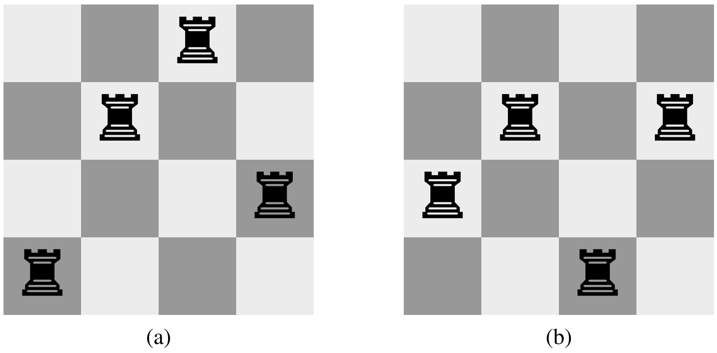


**Fig. 5.4** Illustration of the $K = 4$ rooks problem. **a** A a valid placement. **b** An invalid placement

column; randomly pick one of those and place the third rook there. Continue in this manner until you reach the $K$th column in which there is only 1 unthreatened square left where to place the $K$th rook.

Due to its simplicity, there is no practical advantage in casting the $K$-rooks problem as a QUBO, yet we will do it anyway just to show that we can and to learn about the important idea of combining several sub-QUBOs into a super-QUBO.

In order to cast the $K$-rooks problem as a QUBO, we need to encode it numerically. Her, an intuitive idea is to model any placement of rooks on a $K \times K$ chessboard in terms of a binary matrix $\mathbf{Z} \in \{0, 1\}^{K \times K}$ whose entries are given by

$$Z_{rc} = \begin{cases} 1 & \text{if a rook is placed on square } (r, c) \\ 0 & \text{otherwise .} \end{cases} \tag{5.85}$$

For examples, this representation allows us to model the $4 \times 4$ board configuration in Fig. 5.4a like this

$$\mathbf{Z} = \begin{bmatrix} 0\,0\,1\,0 \\ 0\,1\,0\,0 \\ 0\,0\,0\,1 \\ 1\,0\,0\,0 \end{bmatrix} \tag{5.86}$$

If we further let $\mathbf{1} \in \mathbb{R}^K$ denote the $K$-dimensional vector of all ones, it is easy to see that the following two QUBOs model the problems of finding configurations where there is one and only one rook per row and where there is one and only one rook per column, respectively

$$\mathbf{Z}_*^r = \underset{\mathbf{Z}\in\{0,1\}^{K\times K}}{\operatorname{argmin}} \left\| \mathbf{Z}\mathbf{1} - \mathbf{1} \right\|^2 \tag{5.87}$$

$$\mathbf{Z}_*^c = \underset{\mathbf{Z}\in\{0,1\}^{K\times K}}{\operatorname{argmin}} \left\| \mathbf{1}^\intercal \mathbf{Z} - \mathbf{1}^\intercal \right\|^2 = \underset{\mathbf{Z}\in\{0,1\}^{K\times K}}{\operatorname{argmin}} \left\| \mathbf{Z}^\intercal \mathbf{1} - \mathbf{1} \right\|^2 . \tag{5.88}$$

To be specific, the value of the objective function in (5.87) will be minimal for any binary matrix $\boldsymbol{Z}$ which contains only a single 1 per row and the value of the objective function in (5.88) will be minimal for any binary matrix $\boldsymbol{Z}$ which contains only a single 1 per column.

It is then also easy to see that we may combine the objectives of these two QUBOs to set up a QUBO which models the real $K$-rooks problem of finding a configuration where there is one and only one rook per row *and* one and only one rook per column. It is simply given by

$$\boldsymbol{Z}_* = \underset{\boldsymbol{Z} \in \{0,1\}^{K \times K}}{\operatorname{argmin}} \left\| \boldsymbol{Z}\mathbf{1} - \mathbf{1} \right\|^2 + \left\| \boldsymbol{Z}^\intercal \mathbf{1} - \mathbf{1} \right\|^2 \tag{5.89}$$

and we note that its overall objective simply sums the two individual objectives. In order for this sum to be minimal, both its terms have to be minimal. This will be the case for any binary matrix $\boldsymbol{Z}$ with a single 1 per row *and* a single 1 per column.

An issue with this intuitive formulation of the $K$-rooks problem is that the binary QUBO in (5.89) is not in standard form. While its objective function is quadratic, its decision variable is a binary matrix $\boldsymbol{Z}$ rather than a binary vector $\boldsymbol{z}$. In what follows, we therefore *vectorize* the problem in (5.89).

To begin with, we recall the idea of flattening higher order tensors and observe that any matrix $\boldsymbol{Z} \in \{0, 1\}^{K \times K}$ can be represented in term of the following vector

$$\boldsymbol{z} = \operatorname{vec}\big[\boldsymbol{Z}\big]^\intercal \in \{0, 1\}^{K^2} \tag{5.90}$$

where vec[] is an operator that takes a matrix and concatenates its rows to yield a vector.

Now, given $\boldsymbol{Z}$, $\boldsymbol{z}$ and $\mathbf{1}$ as above and the $K \times K$ identity matrix $\boldsymbol{I}$, we first of all consider the expression $\boldsymbol{Z}\mathbf{1}$ which occurs in the row objective of our $K$-rooks QUBO and claim the following identity (see Exercise 5.2):

$$\boldsymbol{Z}\mathbf{1} = \boldsymbol{C}_r \boldsymbol{z} \tag{5.91}$$

where

$$\boldsymbol{C}_r = \boldsymbol{I} \otimes \mathbf{1}^\intercal \in \{0, 1\}^{K \times K^2}. \tag{5.92}$$

In other words, we claim that the $K$-dimensional vector which results from computing the matrix-vector product $\boldsymbol{Z}\mathbf{1}$ can equivalently be computed by evaluating the matrix-vector product $\boldsymbol{C}_r \boldsymbol{z}$ where $\boldsymbol{C}_r$ is a Kronecker product matrix of the kind we already saw when discussing QUBOs for linear regression.

Second of all, we consider the expression $\boldsymbol{Z}^\intercal \mathbf{1}$ from the column objective of our $K$-rooks QUBO and claim the following identity (see Exercise 5.3):

$$\boldsymbol{Z}^\intercal \mathbf{1} = \boldsymbol{C}_c \boldsymbol{z} \tag{5.93}$$

where

$$\boldsymbol{C}_c = \mathbf{1}^\intercal \otimes \boldsymbol{I} \in \{0, 1\}^{K \times K^2}. \tag{5.94}$$

In other words, we claim that the $K$-dimensional vector resulting from $\boldsymbol{Z}^\intercal \mathbf{1}$ can also be computed as $\boldsymbol{C}_c \boldsymbol{z}$ where $\boldsymbol{C}_c$ is yet another Kronecker product matrix.

Given these claims, we can rewrite the row- and column objectives of the above $K$-rooks QUBO like this

$$\left\|\boldsymbol{Z}\mathbf{1} - \mathbf{1}\right\|^2 = \left\|\boldsymbol{C}_r \boldsymbol{z} - \mathbf{1}\right\|^2 = \boldsymbol{z}^\intercal \boldsymbol{C}_r^\intercal \boldsymbol{C}_r \boldsymbol{z} - 2\,\mathbf{1}^\intercal \boldsymbol{C}_r \boldsymbol{z} + \mathbf{1}^\intercal \mathbf{1} \tag{5.95}$$

$$\left\|\boldsymbol{Z}^\intercal\mathbf{1} - \mathbf{1}\right\|^2 = \left\|\boldsymbol{C}_c \boldsymbol{z} - \mathbf{1}\right\|^2 = \boldsymbol{z}^\intercal \boldsymbol{C}_c^\intercal \boldsymbol{C}_c \boldsymbol{z} - 2\,\mathbf{1}^\intercal \boldsymbol{C}_c \boldsymbol{z} + \mathbf{1}^\intercal \mathbf{1} \tag{5.96}$$

This, in turn, allows us to rewrite the whole QUBO as

$$\boldsymbol{z}_* = \underset{\boldsymbol{z} \in \{0,1\}^{K^2}}{\operatorname{argmin}}\ \boldsymbol{z}^\intercal \left[\boldsymbol{C}_r^\intercal \boldsymbol{C}_r + \boldsymbol{C}_c^\intercal \boldsymbol{C}_c\right] \boldsymbol{z} - 2\,\mathbf{1}^\intercal \left[\boldsymbol{C}_r + \boldsymbol{C}_c\right] \boldsymbol{z} \tag{5.97}$$

where we again dropped constant terms independent of $\boldsymbol{z}$.

If we had an algorithm which could solve this vectorized $K$-rooks QUBO for an optimal binary vector $\boldsymbol{z}_* \in \{0, 1\}^{K^2}$, we could subsequently compute

$$\boldsymbol{Z}_* = \operatorname{mat}\left[\boldsymbol{z}_*^\intercal\right] \in \{0, 1\}^{K \times K} \tag{5.98}$$

where the operator mat[] simply undoes the effect of vec[] in (5.90). This recovers a corresponding binary matrix representation of the solution from which it is easier to infer where to place rooks on the board.

Finally, we point out that optimization problem (5.97) constitutes a binary QUBO in standard form with $\boldsymbol{Q} = \boldsymbol{C}_r^\intercal \boldsymbol{C}_r + \boldsymbol{C}_c^\intercal \boldsymbol{C}_c$ and $\boldsymbol{q} = -2\,[\boldsymbol{C}_r + \boldsymbol{C}_c]^\intercal \mathbf{1}$. A corresponding bipolar QUBO therefore follows once again from applying Theorem 4.1.

## Remarks and Further Reading

Constraint satisfaction problems (CSPs) arise all over the place. For example, Boolean satisfiablility, maximum graph cut, or map coloring are all but specific instances of of this large family of problems. Since there are numerous CSPs of real world importance, there would be much to say about their general characteristics and the many solution strategies that have been developed over decades. Here, however, we simply remark the following: The recreational $K$-rooks problem exemplifies that many logic puzzles are CSPs. We saw that there exist polynomial time algorithms for the $K$-rooks problem, however, most logic puzzles and, in fact, most real world CSPs are NP-complete. A well known, easily understood example is Sudoku whose solutions are difficult to find but quickly verified. Sudoku, too, can be expressed as a QUBO which, at first sight, seems to require a large amount of bivalent variables and constraints [29, 30]. However, recent research shows that hybrid techniques which combine QUBO modeling with classical CSP heuristics and reward functions can significantly reduce the number of variables and constraints [31].

## 5.3 Code Examples

Most of the QUBO models, we discussed in this chapter were data dependent in that they built on instances of sets of numbers or data points. The $K$-rooks problem, on the other hand, only requires us to specify a single parameter $K \in \mathbb{N}$ when formulating a corresponding QUBO model. For simplicity and overhead reduction, our following code examples will therefore focus on the binary $K$-rooks QUBO.

As always, we need to `import numpy as np` to be able to implement vector $\mathbf{1} \in \mathbb{R}^K$, matrix $\boldsymbol{I} \in \mathbb{R}^{K \times K}$ and matrices $\boldsymbol{C}_r$ and $\boldsymbol{C}_c$ in (5.92) and (5.94) as `numpy` arrays. Given some choice for $K$, this is as easy as

```
K = 4
matI   = np.eye(K)
vec1   = np.ones(K)
matCr = np.kron(matI, vec1)
matCc = np.kron(vec1, matI)
```

The only thing we probably need to mention about this snippet is that `np.ones(K)` produces a 1D row array so that `vec1` actually models the transposed vector $\mathbf{1}^\intercal$. With this, our implementations `matCr` and `matCc` of $\boldsymbol{C}_r$ and $\boldsymbol{C}_c$ do indeed make sense.

Next, we can just as easily implement the two parameters $\boldsymbol{Q} = \boldsymbol{C}_r^\intercal \boldsymbol{C}_r + \boldsymbol{C}_c^\intercal \boldsymbol{C}_c$ and $\boldsymbol{q} = -2\,[\boldsymbol{C}_r + \boldsymbol{C}_c]^\intercal \mathbf{1}$ of the binary $K$-rooks QUBO. For better readability, we split this into several steps which also emphasize that we are dealing with a super-QUBO which is composed of two sub-QUBOs, namely

```
matQr = matCr.T @ matCr
vecQr = -2 * matCr.T @ vec1

matQc = matCc.T @ matCc
vecQc = -2 * matCc.T @ vec1

matQ  = matQr + matQc
vecQ  = vecQr + vecQc
```

Finally, we leave it to the reader to feed these array representations of the binary QUBO parameters $\boldsymbol{Q}$ and $\boldsymbol{q}$ to function `bin2bipQubo` from Chap. 4 so as to obtain representations of the corresponding bipolar QUBO parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. Printing and comparing all these arrays might be an instructive exercise.

## 5.4 Exercises

**5.1** Prove the following: If $n \geq 2$ positive numbers $x_1, \ldots, x_n$ have a constant sum of

$$S_n = \sum_{j=1}^{n} x_j \tag{5.99}$$

the value of their product

$$P_n = \prod_{j=1}^{n} x_j \tag{5.100}$$

is largest, if $x_j = \frac{S_n}{n}$ for all $1 \leq j \leq n$.

**5.2** Let $\boldsymbol{I}$ be the $K \times K$ identity matrix, $\mathbf{1}$ be the $K$-dimensional vector of all ones, $\boldsymbol{Z}$ be any $K \times K$ binary matrix, and $\boldsymbol{z} = \text{vec}[\boldsymbol{Z}^\intercal]$ be a corresponding $K^2$-dimensional binary vector. Prove the following identity

$$\boldsymbol{Z}\mathbf{1} = \left[\boldsymbol{I} \otimes \mathbf{1}^\intercal\right] \boldsymbol{z} \,. \tag{5.101}$$

**5.3** Consider $\boldsymbol{I}$, $\mathbf{1}$, $\boldsymbol{Z}$, and $\boldsymbol{z}$ as in the previous exercise and prove the following identity

$$\boldsymbol{Z}^\intercal\mathbf{1} = \left[\mathbf{1}^\intercal \otimes \boldsymbol{I}\right] \boldsymbol{z} \,. \tag{5.102}$$

**5.4** Our code examples demonstrated how to implement the matrices $\boldsymbol{C}_r = \boldsymbol{I} \otimes \mathbf{1}^\intercal$ and $\boldsymbol{C}_c = \mathbf{1}^\intercal \otimes \boldsymbol{I}$ in terms of `numpy` arrays. Run this code for case where, say, $K = 4$, print the resulting arrays, and ponder what you observe.

**5.5** Use the mixed product properties of Kronecker products to give simplified expressions for the following two matrices

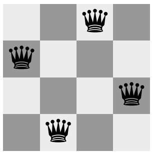
$$\left[\boldsymbol{I} \otimes \mathbf{1}^\intercal\right]^\intercal \left[\boldsymbol{I} \otimes \mathbf{1}^\intercal\right] \quad \text{and} \quad \left[\mathbf{1}^\intercal \otimes \boldsymbol{I}\right]^\intercal \left[\mathbf{1}^\intercal \otimes \boldsymbol{I}\right] . \tag{5.103}$$

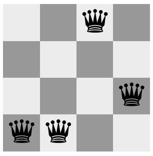
**5.6** Implement `numpy` code to brute force the binary $K$-rooks QUBO in (5.97) with $K = 4$. Reshape you resulting solutions $\boldsymbol{z}_* \in \{0, 1\}^{16}$ into $\boldsymbol{Z}_* \in \{0, 1\}^{4\times 4}$ to verify if they make sense.

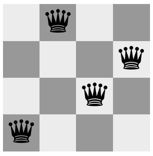
**5.7** Proceed as in the previous exercise but w.r.t. the corresponding bipolar QUBO.

**5.8** Consider the combinatorics of the $K$-rooks problem. How many configurations or chessboard states $\boldsymbol{z} \in \{0, 1\}^{K^2}$ are there? How many of those are configurations involving exactly $K$ occupied squares? How many of those represent solutions to the $K$-rooks problem? Give corresponding mathematical expressions and evaluate these expressions for $K \in \{4, 5, 6, 7, 8\}$. Ponder what you observe.

**5.9** The $K$-queens problems asks for a placement of $K$ queens on a $K \times K$ chessboard so that they do not threaten each other. This is like the $K$-rooks problem on steroids. Whereas the $K$-rooks problem with its mere row and column constraints is trivial, the $K$-queens problem is not because it there are additional diagonal constraints to be satisfied. The following illustrates this for the case where $K = 4$.


valid solution


invalid placement


invalid placement

How many diagonal constraints are there? Can you formalize them in terms of matrix-vector expressions? Can you devise a (binary) QUBO formulation of the $K$-queens problem?

**5.10** Implement a classical backtracking algorithm in `python/numpy` that solves the $K$-queens problems. Test (and verify) your code for growing numbers of $K \geq 4$. At which point do you lose patience and are no longer willing to wait for your code to terminate?

**5.11** In Exercise 1.6, you implemented `numpy` code to generate instances of subset sum problems. Use your code from back then to create vectors $\boldsymbol{x}$ and targets $T$ and then implement `numpy` code that produces the binary QUBO parameters $\boldsymbol{Q}$ and $\boldsymbol{q}$ of corresponding plain vanilla, minimum size, and size restricted SSPs. Once these are available, implement `numpy` code to brute force the respective QUBOs. Up until which problem size are you willing to wait for your code to terminate?

## References

1. Glover, F., Kochenberger, G., Hennig, R., Du, Y.: Quantum Bridge Analytics I: A Tutorial on Formulating and Using QUBO Models. Annals of Operations Research **314**(1) (2022). https://doi.org/10.1007/s10479-022-04634-2
2. Glover, F., Kochenberger, G., Ma, M., Du, Y.: Quantum Bridge Analytics II: QUBO-Plus, Network Optimization and Combinatorial Chaining for Asset Exchange. Annals of Operations Research **314**(1) (2022). https://doi.org/10.1007/s10479-022-04695-3
3. Lucas, A.: Ising Formulations of many NP Problems. Frontiers in Physics **2**(5) (2014). https://doi.org/10.3389/fphy.2014.00005
4. Farhi, E., Goldstone, J., Gutmann, S., Sipser, M.: Quantum Computation by Adiabatic Evolution. arXiv:quant-ph/0001106 (2000)
5. Aharonov, D., et al.: Adiabatic Quantum Computation Is Equivalent to Standard Quantum Computation. In: Proc. Symp. on Foundations of Computer Science. IEEE (2004). https://doi.org/10.1109/FOCS.2004.8
6. Bian, Z., Chudak, F., Macready, W., Rose, G.: The Ising Model: Teaching an Old Problem New Tricks. Tech. rep., D-Wave Systems (2010)

7. Bauckhage, C., Sanchez, R., Sifa, R.: Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In: Proc. Int. Joint Conf. on Neural Networks. IEEE (2020). https://doi.org/10.1109/IJCNN48605.2020.9206916
8. Hadfield, S.: On the Representation of Boolean and Real Functions as Hamiltonians for Quantum Computing. ACM Trans. on Quantum Computing **2**(4) (2021). https://doi.org/10.1145/3478519
9. Cook, S.: The Complexity of Theorem-proving Procedures. In: Proc. Symp. on Theory of Computing. ACM (1971). https://doi.org/10.1145/800157.805047
10. Karp, R.: Reducibility among Combinatorial Problems. In: R. Miller, J. Thatcher (eds.) Complexity of Computer Computations, The IBM Research Symposia Series. Springer (1972). 10.1007/978-1-4684-2001-2_9
11. Bishop, C.: Pattern Recognition and Machine Learning. Springer (2006)
12. Hopfield, J., Tank, D.: "Neural" Computation of Decisions in Optimization Problems. Biological Cybernetics **52** (1985). https://doi.org/10.1007/BF00339943
13. Bauckhage, C.: Hopfield Nets for Set Cover. Tech. rep., Machine Learning Rhine Ruhr (2022). Available on researchgate.net
14. Bauckhage, C., Sifa, R., Wrobel, S.: Adiabatic Quantum Computing for Max-Sum Diversification. In: Proc. Int. Conf. on Data Mining. SIAM (2020). https://doi.org/10.1137/1.9781611976236.39
15. Welke, P., Schulz, T., Bauckhage, C.: Computational Complexity of Max-Sum Diversification. Tech. rep., Machine Learning Rhine Ruhr (2021). Available on researchgate.net
16. Bauckhage, C., et al.: Ising Models for Binary Clustering via Adiabatic Quantum Computing. In: Proc. Int. Conf. on Energy Minimization Methods in Computer Vision and Pattern Recognition, *LNCS*, vol. 10746. Springer (2017). 10.1007/978-3-319-78199-0_1
17. Lloyd, S.: Least Squares Quantization in PCM. IEEE Trans. on Information Theory **28**(2) (1982). https://doi.org/10.1109/TIT.1982.1056489
18. Hartigan, J., Wong, M.: Algorithm AS 136: A $k$-Means Clustering Algorithm. J. of the Royal Statistical Society C **28**(1) (1979). https://doi.org/10.2307/2346830
19. MacQueen, J.: Some Methods for Classification and Analysis of Multivariate Observations. In: Proc. Berkeley Symp. on Mathematical Statistics and Probability (1967)
20. Fisher, R.: On the Probable Error of a Coefficient Correlation Deduced from a Small Sample. Metron **1** (1921)
21. Bauckhage, C.: $k$-Means and Fisher's Analysis of Variance. Tech. rep., Fraunhofer Center for Machine Learning (2018). Available on researchgate.net
22. Bauckhage, C., Ojeda, C., Sifa, R., Wrobel, S.: Adiabatic Quantum Computing for Kernel k=2 Means Clustering. In: Proc. KDML-LWDA (2018). ceur-ws.org/Vol-2191/paper3.pdf
23. Bauckhage, C.: k-Means Clustering via the Frank-Wolfe Algorithm. In: Proc. KDML-LWDA (2016). ceur-ws.org/Vol-1670/paper-65.pdf
24. Bauckhage, C., Piatkowske, N., Sifa, R., Hecker, D., Wrobel, S.: A QUBO Formulation of the k-Medoids Problem. In: Proc. KDML-LWDA (2019). ceur-ws.org/Vol-2454/paper_39.pdf
25. Bauckhage, C., Ramamurthy, R., Sifa, R.: Hopfield Networks for Vector Quantization. In: I. Farkas, P. Masulli, S. Wermter (eds.) Artificial Neural Networks and Machine Learning, *LNCS*, vol. 12397. Springer (2020). 10.1007/978-3-030-61616-8_16
26. Mücke, S., Heese, R., Müller, S., Wolter, M., Piatkowski, N.: Feature Selection on Quantum Computers. Quantum Macine Intelligence **5**(11) (2023). https://doi.org/10.1007/s42484-023-00099-z
27. Date, P., Potok, T.: Adiabatic Quantum Linear Regression. Scientific Reports **11** (2021). https://doi.org/10.1038/s41598-021-01445-6
28. Date, P., Arthur, D., Pusey-Nazzaro, L.: QUBO Formulations for Training Machine Learning Models. Scientific Reports **11** (2021). https://doi.org/10.1038/s41598-021-89461-4
29. Hopfield, J.: Searching for Memories, Sudoku, Implicit Check Bits, and the Iterative Use of Not-Always-Correct Rapid Neural Computation. Neural Computation **20**(5) (2008). https://doi.org/10.1162/neco.2007.09-06-345

30. Bauckhage, C., Beaumont, F., Müller, S.: Hopfield Nets for Sudoku. Tech. rep., Machine Learning Rhine Ruhr (2021). Available on researchgate.net
31. Mücke, S.: A Simple QUBO Formulation of Sudoku. In: Proc. Genetic and Evolutionary Computation Conf. (2024). https://doi.org/10.1145/3638530.3664106

# Part II
# Hopfield Nets

# Chapter 6
# Hopfield Nets

## 6.1 Introduction

*Hopfield nets* form a comparatively simple class of *recurrent neural networks*. They have a venerable history [1–3] and were originally introduced as simple mathematical models of pattern memorization, associative recall, and concept formation in biological brains. Their original claim to fame is therefore that they can act as *autoassociative memories*.

However, we will mainly be interested in another, arguably even more intriguing capability of Hopfield nets. Indeed, we will largely be concerned with the fact that they can also act as *problem solvers* and be used to search for (approximate) solutions to difficult combinatorial optimization problems [4]. For quite some time, this crucial capability has been largely overlooked but, now that working quantum computers have become available, is attracting quickly growing interest. This, in turn, is because there exists a surprisingly close (conceptual) connection between Hopfield nets and (adiabatic) quantum computing [5].

Speaking of this connection, we should mention that Hopfield nets are named after John Hopfield although he himself acknowledges that others had thought of them before [3]. However, he was the first to realize and investigate the fact that the computations they perform can be understood as an *energy minimization process*. Put more elaborately, he was the first to focus on their *dynamics* rather than on how to train them and it was his pioneering bridge building between neurocomputing and physics for which Hopfield was awarded one half of the 2024 Nobel prize in physics.

To further set the stage for our following discussion, we have two more disclaimers: First, we will follow in Hopfield's footsteps and ignore the question of how Hopfield nets could learn from data. That is, we will focus on applications of

C. Bauckhage and R. Sifa, *Quantum Computing from Hopfield Nets*,
Cognitive Technologies, https://doi.org/10.1007/978-3-031-99402-9_6

trained or appropriately pre-parameterized Hopfield nets and it will soon become clear why we can afford to do so. Second, even prior to the 2024 Nobel prize, there has been a resurgence of research on Hopfield nets, especially related to their use as associative memories. This is because the original Hopfield model was recently generalized, especially with respect to the notion of the *energy function* of a Hopfield net. Such generalized energy functions can exponentially increase the storage capacity of Hopfield nets with provable guarantees and corresponding model parameters can be learned from just a few training examples [6–10]. While all this attracted even more interest when it further became clear that the computations of a generalized Hopfield net are equivalent to the attention mechanisms employed in transformer networks [11], we will utterly ignore such *modern Hopfield nets* but instead only consider their *classical* variants.

In what follows, we will acquaint ourselves with the basic ideas and modeling assumptions behind Hopfield nets. That is, we will list defining structural properties of Hopfield nets, introduce the fundamental concept of the energy of a Hopfield net, and discuss mechanisms of how Hopfield nets may evolve over time. Our discussion will involve terminology commonly used in the context of artificial neural networks. While we will thus use terms such as *connection weights*, *bias terms*, or *activation functions* which we shall clarify in passing; yet, this jargon will not be essential as this book is not about neural networks. But let us provide some context anyway.

Above, we already said that a Hopfield net is a comparatively simple instance of a recurrent neural network. To contextualize this statement, we observe or recall that an **artificial neural network** is an ensemble of interconnected computational units called **artificial neurons** which constitute rather rough mathematical models of the functionality of the nerve cells in biological brains. Each such neuron, let's call it $j$, receives (numerical) inputs from (some of) its peers, computes a *weighted sum* of these inputs, subtracts a *bias term* or *threshold value*, passes the result through a nonlinear *activation function*, and forwards its output to other neurons. In short, an artificial neuron is nothing but a function $f_j : \mathbb{R}^{n_j} \to \mathbb{R}$ parameterized by a weight vector $\boldsymbol{w}_j \in \mathbb{R}^{n_j}$ and a bias value $b_j \in \mathbb{R}$ such that

$$f_j\big(\boldsymbol{x} \mid \boldsymbol{w}_j, b_j\big) = g_j\left(\sum_{k=1}^{n_j} w_{jk} x_k - b_j\right) = g_j\left(\boldsymbol{w}_j^\intercal \boldsymbol{x} - b_j\right) \tag{6.1}$$

Traditional choices of the activation function $g_j : \mathbb{R} \to \mathbb{R}$ are step- or threshold functions, the logistic function, or the hyperbolic tangent. Modern popular choices are the rectified linear unit (ReLU) and its many (nonlinear) variants.

In a **feed-forward net**, neurons are arranged in layers with a specific in- and output layer in which information enters and leaves the network. Layers sandwiched in between are called hidden layers and the more there are, the deeper is the network. Information flow in a feed-forward net may skip layers but is always unidirectional (see Fig. 6.1a). The fundamental task regarding such networks is to train them, i.e. to

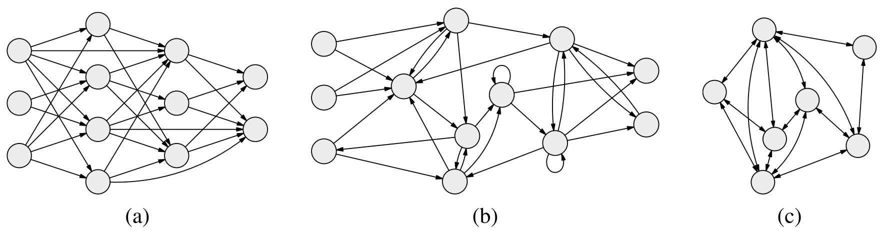


**Fig. 6.1** Simple schematic illustrations of different kinds of artificial neural network architectures. **a** A feed-forward net. **b** A recurrent net. **c** A Hopfield net

automatically adjust their parameters, such that they perform a desired mapping from inputs to outputs. Once trained satisfactory, a feed-forward net can be deployed; since it computations proceed in a layer-wise manner, they terminate once the output layer has been reached.

In a **recurrent net**, information flows back and forth between neurons so that there exist feedback loops. Generally, there is thus no real layer structure but there still may be designated in- and output neurons. Information flow between connected neurons may be uni- or bidirectional and neurons may even feed their output back to their input (see Fig. 6.1b). Again, the fundamental problem is how to train recurrent networks for deployment. However, their deployment differs from that of simple feed-forward nets. Typically, a recurrent net receives a continuous stream of inputs and, since information flows back and forth between its neurons, its computations in principle proceed forever. Examples of applications where such ever lasting computations are called for are text generation or the simulation of fluids or other dynamical systems.

With all this, we may tentatively say that a **Hopfield net** is a structurally restricted recurrent net without designated in- and output neurons. In principle, each neuron could act as an interface to the outside world. Also, if any two neurons are connected, information flow between them must be bidirectional. However, self-connections are not allowed so that neurons cannot feed their output back to their input (see Fig. 6.1c). Again, one could ask for how to train a Hopfield net. In contrast to most other kinds of neural networks where training usually employs variants of the backpropagation algorithm [12], classical Hopfield nets are typically trained using variants of Hebbian learning [13]. However, and again in contrast to most other kinds of neural networks, Hopfield nets can also perform interesting tasks when parameterized manually and we will soon see why. Being recurrent neural networks, their computations do, in principle, again proceed forever but typically reach useful results after only finitely many *asynchronous updates* of their neurons.

All in all there thus are two aspects which distinguish Hopfield nets from other neural networks, namely their *structure* and their *update mechanism*. Next, we discuss both these aspects in more detail.

## 6.2 Defining Structural Properties

A **Hopfield net** is a recurrent neural network of $n$ interconnected artificial neurons which we denote as

$$s_1, s_2, \ldots, s_n \; . \tag{6.2}$$

We opt for this notation because we will henceforth often identify neurons with their *activation status*. Indeed, at any point in time, each Hopfield neuron is supposed to be either `active` or `inactive` and we represent these states using $+1$ for the former and $-1$ for the latter. In short, the *micro state* of a neuron, i.e. of a single component of a Hopfield net, is a bipolar variable

$$s_j \in \{\pm 1\} \; . \tag{6.3}$$

This, in turn, implies that, at any point in time, the *macro state* of a whole Hopfield net can be represented as a bipolar vector

$$\boldsymbol{s} = [s_1, s_2, \ldots, s_n]^\intercal \in \{0, 1\}^n \tag{6.4}$$

and, since there are "only" $2^n$ such vectors, we immediately realize that a Hopfield net can only have finitely many distinct macro states.

Given the above `active` or `inactive` postulate, we realize that each Hopfield neuron must be a classical *bipolar threshold unit*. That is, if $w_{jk} \in \mathbb{R}$ denotes the (synaptic) weight of the connection between neurons $s_j$ and $s_k$ and if $b_j \in \mathbb{R}$ denotes the bias value of neuron $s_j$, then neuron $s_j$ computes its state like this

$$s_j = \begin{cases} +1 & \text{if } \sum\limits_{k=1}^{n} w_{jk}\, s_k - b_j \geq 0 \\ -1 & \text{otherwise} \; . \end{cases} \tag{6.5}$$

This expression can be written more compactly if we gather all the weights $w_{jk}$ in a vector $\boldsymbol{w}_j \in \mathbb{R}^n$ and make use of the macro state $\boldsymbol{s}$. With these, we have

$$s_j = \begin{cases} +1 & \text{if } \boldsymbol{w}_j^\intercal \boldsymbol{s} - b_j \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{6.6}$$

$$= \operatorname{sign}\big(\boldsymbol{w}_j^\intercal \boldsymbol{s} - b_j\big) \tag{6.7}$$

which is of the form in (6.1) where the activation function $g_j$ is chosen to be the following **bipolar sign function**

$$\operatorname{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ +1 & \text{if } x \geq 0 \; . \end{cases} \tag{6.8}$$

In other words, each neuron in a Hopfield net receives and weights inputs from its peers, sums these weighted inputs, and thresholds this sum against a bias value. These computations happen at discrete points in time and may activate or deactivate the neuron.

Since the neurons in a Hopfield net are all of the same type or, put differently, since they all employ the same activation function, the above is a rather simple mechanism as far as neural networks go. Hopfield nets are therefore not as universal as neural networks could be in general. However, they still provide a useful class of tools and their simplicity can also be seen as a favorable property.

For instance, since all Hopfield neurons are of the same type, we realize that an individual Hopfield net can be fully specified in terms of only two but admittedly large parameters, namely

$$\boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1^\mathsf{T} \\ \boldsymbol{w}_2^\mathsf{T} \\ \vdots \\ \boldsymbol{w}_n^\mathsf{T} \end{bmatrix} \in \mathbb{R}^{n \times n} \quad \text{and} \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \in \mathbb{R}^n \tag{6.9}$$

where matrix $\boldsymbol{W}$ gathers all the weights of the network's (synaptic) connections and vector $\boldsymbol{b}$ gathers all the biases of the network's neurons.

So far, there is nothing too spacial about the definitions we just gave since they would similarly occur in the context of other fully connected recurrent neural nets with sigmoidal activations. However, Hopfield nets are more specific or restricted recurrent neural networks, because in order for the pair $(\boldsymbol{W}, \boldsymbol{b})$ to specify a Hopfield net, the weight matrix $\boldsymbol{W}$ has to obey two important structural constraints.

We already said that the information flow between any two Hopfield neurons $s_j$ and $s_k$ is supposed to be bidirectional. That is, if the synaptic weight $w_{jk} \neq 0$, we must also have $w_{kj} \neq 0$. Next, we additionally declare that it is even more crucially assumed that both these weights are identical. In other words, in a Hopfield we must insist on

$$w_{jk} = w_{kj} \; . \tag{6.10}$$

We also already said that Hopfield neurons are not allowed to feed their output back to their input. In terms of synaptic weights, this means that we must furthermore insist on

$$w_{jj} = 0 \; . \tag{6.11}$$

Taken together, both these structural constraints imply that the weight matrix of a Hopfield net must by *symmetric* as well as *hollow*

$$\boldsymbol{W} = \boldsymbol{W}^\intercal \tag{6.12}$$

$$\operatorname{diag}\big[\boldsymbol{W}\big] = \boldsymbol{0} \tag{6.13}$$

where a **hollow matrix** is a square matrix with all zeros on its diagonal.

Now, since these two properties of the weight matrix $\boldsymbol{W}$ are truly crucial for our upcoming analysis of the general dynamics or evolution of the states of a Hopfield net, we better emphasize them by means of an abstract or symbolic example.

If there were no restrictions on the connection weights of a Hopfield net, its weight matrix would generally look like this

$$\boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{bmatrix}. \tag{6.14}$$

However, because of the structural constraints in (6.12) and (6.13), the weight matrix of a Hopfield net must look like this

$$\boldsymbol{W} = \begin{bmatrix} 0 & w_{12} & \cdots & w_{1n} \\ w_{12} & 0 & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1n} & w_{2n} & \cdots & 0 \end{bmatrix}. \tag{6.15}$$

Looking at the structure of the $n \times n$ weight matrix in (6.15) and not forgetting about the $n$-dimensional bias vector, we thus observe that a Hopfield net of $n$ neurons only comes with

$$\frac{n^2 - n}{2} + n \tag{6.16}$$

adjustable or *free parameters*. As neural networks go, this is not a lot. Yet, there still are important use cases for models of this kind.

Now, to wrap up our discussion up until this point, we have the following summary of facts.

**Definition 6.1** A **Hopfield net** is a fully connected recurrent neural network of $n$ neurons. At any point in time, the **micro state**

$$s_j \in \{\pm 1\} \tag{6.17}$$

of each of these neurons is a bipolar variable. Whenever a neuron is triggered to update its state, it computes

$$s_j = \operatorname{sign}\big(\boldsymbol{w}_j^\intercal \boldsymbol{s} - b_j\big) \tag{6.18}$$

where $\boldsymbol{w}_j \in \mathbb{R}^n$ is its weight vector, $b_j \in \mathbb{R}$ is its bias value, and the vector

$$\boldsymbol{s} \in \{\pm 1\}^n \tag{6.19}$$

characterizes the current **macro state** of the network.

An individual Hopfield net can therefore be uniquely specified in terms of a pair of parameters $(\boldsymbol{W}, \boldsymbol{b})$ where $\boldsymbol{W} \in \mathbb{R}^{n \times n}$ is the network's **weight matrix** and $\boldsymbol{b} \in \mathbb{R}^n$ is its **bias vector**. A fundamentally important structural restriction for the weight matrix of any Hopfield net is that it must be **symmetric** and **hollow** such that

$$\boldsymbol{W} = \boldsymbol{W}^\intercal \tag{6.20}$$

$$\operatorname{diag}\big[\boldsymbol{W}\big] = \boldsymbol{0} \,. \tag{6.21}$$

Of course this definition raises at least two follow-up questions. What could we possibly mean by a neuron being *triggered* to update its state? And why would we insist on distinguishing between *micro states* of individual neurons and *macro states* of whole networks?

Both questions will be answered once we look at *update mechanisms* and the temporal evolution or *dynamics* of Hopfield nets. However, to prepare ourselves for that discussion, we need yet another crucial concept, namely the notion of the *energy function* of a Hopfield net.

To make long story short, *we* define the *energy* of a Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ in a macro state $\boldsymbol{s}$ as

$$E(\boldsymbol{s}) = -\tfrac{1}{2} \sum_j \sum_k w_{jk}\, s_j\, s_k + \sum_j b_j\, s_j \,. \tag{6.22}$$

While we will leave a deeper dive into the physical significance of this expression to later a chapter, we note our emphasis on "we" in the above statement. This is because, in the scientific literature, the following definition is somewhat more common

$$E(\boldsymbol{s}) = -\sum_{j<k} w_{jk}\, s_j\, s_k + \sum_j b_j\, s_j \,. \tag{6.23}$$

However, since the weight matrix of a Hopfield net must be hollow, both expressions are equivalent and readers are encouraged to verify this for themselves. Having said this, we next claim that *our* definition is preferable because it allows for a compact matrix-vector formulation. Indeed, readers are again encouraged to verify that (6.22) can be equivalently expressed as follows.

**Definition 6.2** The **energy** of a Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ in macro state $\boldsymbol{s}$ is

$$E(\boldsymbol{s}) = -\tfrac{1}{2}\, \boldsymbol{s}^{\intercal} \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^{\intercal} \boldsymbol{s} \ . \tag{6.24}$$

Now this is striking, isn't it? What we recognize from (6.24) is that the energy function of a Hopfield net is but a specific instance of the objective function of a bipolar QUBO as in Definition 4.2!

We said "specific", because the weight matrix of a Hopfield net must be symmetric and hollow whereas there generally are no such constraints on the parameter matrix of a bipolar QUBO. However, we also recall Theorem 4.2.3 which says that, for every bipolar QUBO with parameter matrix $\boldsymbol{W}$, there exists an equivalent bipolar QUBO with hollowed out parameter matrix $\boldsymbol{W} - \text{diag}[\boldsymbol{W}]$.

All this then means that the problem of finding a global minimizer $\boldsymbol{s}_*$ of a bipolar QUBO with a symmetric parameter matrix is equivalent to the problem of finding a minimum energy state $\boldsymbol{s}_*$ of a corresponding Hopfield net, namely

$$\boldsymbol{s}_* = \text{argmin}_{\boldsymbol{s}}\ E(\boldsymbol{s}) \ . \tag{6.25}$$

This is once again striking! While we did not emphasize it in Chap. 5, all the application driven QUBO models we derived there actually came with symmetric parameter matrices. Indeed, every QUBO will necessarily have a symmetric parameter matrix (see Exercise 6.1). This thus explains why QUBOs can be solved by running a Hopfield nets; at least as long as we had algorithms for running Hopfield nets which would consistently evolve their macro states towards globally minimum energy states.

Unfortunately, as of this writing, we do not know of any algorithm which could be guaranteed to always steer a Hopfield net towards a state of globally minimum energy within polynomially many steps. If there was such an algorithm, then QUBO solving would no longer be difficult, or, more sensationally, we would have P = NP. There is no free lunch! Although we now established a connection between QUBO solving and minimizing Hopfield energies, we merely swapped one generally NP-hard problem for another. On the plus side, we can now rest assured that there exists a universal, problem independent toolbox for (approximate) QUBO solving because, as we will see next, there exists a family of algorithms which *on average* does a very good job in globally minimizing Hopfield energies.

## 6.3 Classical Update Mechanisms

Since a Hopfield net is a recurrent neural network, its macro state keeps evolving once the net has started running. But what does this mean? How does it run? How do Hopfield neurons know when to update their micro states? What kind of algorithms can control this process? And what has all this to do with energy minimization?

To be able to answer these questions, we need to extend our notation so as to incorporate the temporal aspect of running Hopfield nets. Hence, to denote a network's macro state at time $t \in \mathbb{N}$, we henceforth write

$$\boldsymbol{s}_t \equiv \boldsymbol{s}(t) \tag{6.26}$$

and the micro state of neuron $s_j$ at time $t$ will be slightly awkward but compactly written as

$$s_j^t \equiv s_j(t)\ . \tag{6.27}$$

Now, without further ado, given a Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ which has been initialized in an **initial state**

$$\boldsymbol{s}_0 \in \{\pm 1\}^n \tag{6.28}$$

there are *two* fundamental mechanisms for how the network's state may evolve. Either, all its neurons always update their states simultaneously. Or, its neurons update their states one at a time in "some" sequential manner. In a more fanciful terminology, the former kind of updates are called **synchronous updates** and the latter **asynchronous updates**.

From an algorithmic point of view, there is not much to be said about synchronous updates. They explicitly update the network's macro state in the following sense: Given the network's current state $\boldsymbol{s}_t$, we have to evaluate the activation functions of all its neurons to get the successor state $\boldsymbol{s}_{t+1}$. The following piece of pseudo code summarizes this idea.

**Algorithm 1** Synchronous updates of a Hopfield net

```
Input: Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ and initial state $\boldsymbol{s}_0$
  for $t = 0, 1, 2, \ldots$
      update the network's macro state
          $\boldsymbol{s}_{t+1} = \operatorname{sign}\big(\boldsymbol{W}\boldsymbol{s}_t - \boldsymbol{b}\big)$
```

Thee are basically only two things to note about this pseudo code. First, when we write $\operatorname{sign}(\boldsymbol{W}\boldsymbol{s}_t - \boldsymbol{b})$, we understand this as a function which takes an $n$-dimensional vector, computes the sign of all its components, and returns those in an $n$-dimensional vector. Second, it is of course impossible to perform infinitely many iterations. In practice, we would therefore have to terminate the loop at "some" point according to "some" criterion. But let us stick with purely theoretical considerations for now.

Asynchronous updates explicitly update micro states and thus implicitly update the macro state as well. The basic idea is as follows: In any iteration $t$, "somehow" select an update neuron $s_u$ and update its state from $s_u^t$ to $s_u^{t+1}$. While this leaves the states of the $n-1$ other neurons intact, it still impacts the network's overall activation pattern since changing just a single micro state automatically changes the current macro state. The following pseudo code summarizes the procedure.

**Algorithm 2** Asynchronous updates of a Hopfield net

```
Input: Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ and initial state $\boldsymbol{s}_0 = \big[s_1^0, s_2^0, \ldots, s_n^0\big]^\intercal$
  for $t = 0, 1, 2, \ldots$
      "somehow" pick an update neuron $s_u$
      update only this neuron's micro state
          $s_u^{t+1} = \operatorname{sign}\big(\boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u\big)$
```

An obvious question at this point is then what it means to "somehow" pick an update neuron?

Popular classical answers are to either select it randomly or to proceed in a round robin manner. For the former, the usual procedure is to draw the update index uniformly at random, i.e. to sample it from the uniform distribution over the index set of the neurons. Formally, we henceforth express this as $u \sim \mathcal{U}[\{1, 2, \ldots, n\}]$. For the latter, it is common to select $s_1$ in the initial iteration, $s_2$ in the next one and so on and so forth until $s_n$ has been selected. Once this happens, the merry-go-round restarts with $s_1$. Formally, this means $u = t \bmod n + 1$ which accounts for the fact that, mathematically, neuron indices are counted from 1 but time steps are counted from 0.

For now, we leave it at that because the next chapter will introduce a much cleverer idea for how to determine which neuron to update in any iteration.

Another rather obvious question at this point is whether or not it will make a difference if we run synchronous or asynchronous updates? Will the different update mechanisms cause the macro state of a Hopfield net to change forever or will they allow it to eventually settle to a certain configuration?

Now these are truly fundamental questions and to be able to see why, we need to take step back and reconsider what we are dealing with. Indeed, a running Hopfield net constitutes a **discrete-time non-linear dynamical system**. To keep things simple, we shall understand these as systems of coupled variables whose values evolve over time. In particular, we may understand a running Hopfield net as a system of variables whose evolution is governed by the following system of equations

$$\begin{aligned}
s_1(t+1) &= f_1\big(s_1(t), s_2(t), \ldots, s_n(t), t\big) \\
s_2(t+1) &= f_2\big(s_1(t), s_2(t), \ldots, s_n(t), t\big) \\
&\vdots \\
s_n(t+1) &= f_n\big(s_1(t), s_2(t), \ldots, s_n(t), t\big)
\end{aligned} \tag{6.29}$$

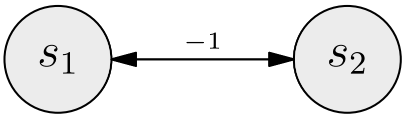


**Fig. 6.2** A super simple Hopfield net

where the $f_j$ are time-dependent non-linear functions given by

$$f_j\big(s_1(t), \ldots, s_n(t), t\big) = \begin{cases} \text{sign}\big(\boldsymbol{w}_j^\intercal \boldsymbol{s}(t) - b_j\big) & \text{if } s_j \text{ updates at time } t \\ s_j(t) & \text{otherwise} \end{cases} \tag{6.30}$$

and we note that, under synchronous updates, every neuron updates at time $t$ whereas, under asynchronous updates, onle one neuron updates at time $t$.

Now that we know that we dealing with non-linear dynamical systems, we must admit that we cannot study them in detail because non-linear dynamics are a topic on their own (highly reccomended references are [14, Chap. 13] and [15]). We therefore simply claim that, in the long run, non-linear dynamical systems may converge to a *stable state*, begin to *oscillate* between (two or more) states, or enter unpredictable chaotic regimes. Luckily, the latter are rarely an issue when dealing with Hopfield nets whose possible macro states $\{\pm 1\}^n$ form the corners of the bipolar hypercube in $\mathbb{R}^n$ or, equivalently, a Boolean lattice with $2^n$ vertices and thus a discrete state space.

Coming back to the above questions, we are therefore left to wonder if the different update mechanisms (synchronous and asynchronous) will cause the macro state of a Hopfield net to oscillate or to converge to a stable state? Will either of this always happen or only under certain conditions? How prone are the different update mechanisms to either of these possible outcomes?

To make a long story short, we can confidently state that the synchronous update mechanism in Algorithm 1 *may* run into cycles or begin to oscillate between states. To confirm this claim, we only need to find a single example where this happens.

Such an example is easy to come by. Consider, for instance, the super simple Hopfield net of only two neurons in Fig. 6.2. Its weight matrix is given by

$$\boldsymbol{W} = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \tag{6.31}$$

and, to keep things as simple as possible, we let its bias vector be

$$\boldsymbol{b} = \boldsymbol{0} \,. \tag{6.32}$$

To empirically examine the dynamics of this network under synchronous updates, we also need an initial macro state to start from and, to corroborate our claim, we opt for this specific choice

$$\boldsymbol{s}_0 = \begin{bmatrix} s_1^0 \\ s_2^0 \end{bmatrix} = \begin{bmatrix} +1 \\ +1 \end{bmatrix} \,. \tag{6.33}$$

Given all these preparations, we can now run the iteration $\boldsymbol{s}_{t+1} = \text{sign}(\boldsymbol{W}\boldsymbol{s}_t)$ and, if we do, observe that it produces the following (temporal) sequence of macro states

$$\boldsymbol{s}_1 = \text{sign}\big(\boldsymbol{W}\boldsymbol{s}_0\big) = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \tag{6.34}$$

$$\boldsymbol{s}_2 = \text{sign}\big(\boldsymbol{W}\boldsymbol{s}_1\big) = \begin{bmatrix} +1 \\ +1 \end{bmatrix} \tag{6.35}$$

$$\boldsymbol{s}_3 = \text{sign}\big(\boldsymbol{W}\boldsymbol{s}_2\big) = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \tag{6.36}$$

$$\vdots$$

which will obviously forever cycle between state $+\mathbf{1} \in \{\pm 1\}^2$ and state $-\mathbf{1} \in \{\pm 1\}^2$. Why obviously? Because the computation $\boldsymbol{s}_{t+1} = \text{sign}(\boldsymbol{W}\boldsymbol{s}_t)$ is deterministic with no randomness or noise involved. Hence, if at one point during the network's evolution, we have seen the transition $+\mathbf{1} \mapsto -\mathbf{1}$ and re-encounter state $+\mathbf{1}$ at a later point in time, we know that its successor must be $-\mathbf{1}$ again. In our example, it takes only two steps until we re-encounter the particular state $+\mathbf{1}$ so that we are dealing with a cycle of length two.

Interestingly, if we consider the exact same network and initial state as above but let it evolve under the asynchronous mechanism in Algorithm 2 with round robin selection of the updating neuron, things turn out to be different.

For this setting, we find

$$\boldsymbol{s}_1 = \begin{bmatrix} \text{sign}\big(\boldsymbol{w}_1^\intercal \boldsymbol{s}_0\big) \\ +1 \end{bmatrix} = \begin{bmatrix} -1 \\ +1 \end{bmatrix} \tag{6.37}$$

$$\boldsymbol{s}_2 = \begin{bmatrix} -1 \\ \text{sign}\big(\boldsymbol{w}_2^\intercal \boldsymbol{s}_1\big) \end{bmatrix} = \begin{bmatrix} -1 \\ +1 \end{bmatrix} \tag{6.38}$$

$$\boldsymbol{s}_3 = \begin{bmatrix} \text{sign}\big(\boldsymbol{w}_1^\intercal \boldsymbol{s}_2\big) \\ +1 \end{bmatrix} = \begin{bmatrix} -1 \\ +1 \end{bmatrix} \tag{6.39}$$

$$\vdots$$

and thus observe (very rapid) convergence to a **fixed point** or *stable state* within the *state space* of our super simple Hopfield net.

We will soon see that this behavior of the network under asynchronous updates is not at all a coincidence. But let us first more carefully examine what we can learn from our examples of the possible behavior of Hopfield nets.

In general, **stability theory** of non-linear dynamical systems is not a trivial matter and the notion of stable fixed points can be quite involved. However, for the dynamics of a Hopfield net, things a comparatively easy as we may consider its energy function $E : \{\pm 1\}^n \to \mathbb{R}$ as a discrete surrogate for what is known as a **Lyapunov function**. To make another very long story short, these allow for characterizing (the behavior of)

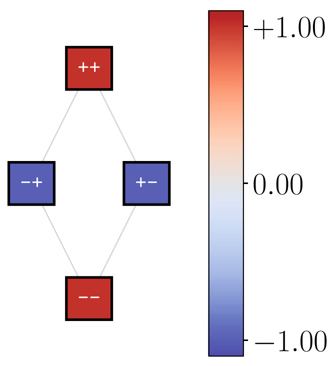


**Fig. 6.3** Energy landscape of the super simple Hopfield net

points in continuous state spaces; in particular, they will assume a (local) minimum at a stable fixed point. These, in turn, may be thought of as states of a dynamical system which, when reached, will never be left again. All this thus allows us to define a stable macro state of a Hopfield net as follows.

**Definition 6.3** A macro state $\boldsymbol{s}$ of a Hopfield net is **stable**, if the network's energy function has a (local) minimum at $\boldsymbol{s}$.

To further clarify the meaning behind this definition, we consider the visualization of the energy landscape of a our super simple Hopfield net shown in Fig. 6.3.

Our Hopfield net of $n = 2$ neurons can only have $2^n = 4$ distinct macro states which form a rather small Boolean lattice. Moreover, for our particular choice of the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ in (6.31) and (6.32), respectively, there are only two distinct values for the network's energy function. In both our above updating examples, we considered the initial macro state $\boldsymbol{s}_0 = \left[s_1^0, s_2^0\right]^\intercal = \left[+1, +1\right]^\intercal$ which corresponds to the top vertex in the lattice graph visualization in Fig. 6.3 and has a high energy value.

We just saw that a synchronous update of $\boldsymbol{s}_0 = [+1, +1]^\intercal$ leads to $\boldsymbol{s}_1 = [-1, -1]^\intercal$ which corresponds to the bottom vertex in the lattice graph visualization in Fig. 6.3 and also has a high energy value. Looking at the figure, we thus realize that synchronous updates can cause Hopfield macro states to *jump* to far away points in the state space but may not necessarily impact energy values.

We also just saw that an asynchronous update of neuron $s_1$ initially in micro state $s_1^0 = +1$ causes is to switch to micro state $s_1^1 = -1$ which, in turn, causes the initial macro state to transit from $\boldsymbol{s}_0 = [+1, +1]^\intercal$ to $\boldsymbol{s}_1 = [-1, +1]^\intercal$. Note that these correspond to neighboring vertices in the lattice graph visualization! Also note that the new macro state has a lower energy value than the initial one.

We furthermore just saw that a subsequent asynchronous update, i.e. an evaluation of the activation function of neuron $s_2$ currently in micro state $s_2^1 = +1$ does not cause it to switch but to keep its micro state. We thus have the new macro state $\boldsymbol{s}_2 = \boldsymbol{s}_1$

and therefore reached a fixed point in the iterations performed by Algorithm 2. We also note that $\boldsymbol{s}_2 = [-1, +1]^\intercal$ is a local minimizer of the network's energy function as both its neighboring vertices $[+1, +1]^\intercal$ and $[-1, -1]^\intercal$ have higher energy values. Transiting from the current state $\boldsymbol{s}_2$ to any of its neighbors would thus increase the energy of the network.

All in all, our example therefore suggests that: Asynchronous updates either cause arbitrary Hopfield macro states to *transit* to a neighboring state which seems to entail a reduction of energy or they cause stable Hopfield macro states of (locally) minimal energy to *stay intact*.

So far, this is only an empirical observation gleaned from a single trivial example. However, it does hold in general because we have the following important theorem.

**Theorem 6.1** *If the neurons of a Hopfield net update their micro states in an asynchronous manner, the macro state of the network will converge to a stable state. This convergence is guaranteed to happen within a finite number of update steps.*

***Proof*** From all we have seen so far, we know that an asynchronous update at time $t$ only affects the update neuron $s_u$. The micro state of any any other neuron $s_j \neq s_u$ is left intact so that

$$s_{j \neq u}^{t+1} = s_{j \neq u}^{t} \; . \tag{6.40}$$

With respect to the update neuron, two things may happen. Either, its micro state is left intact

$$s_u^{t+1} = s_u^t \tag{6.41}$$

or its micro state changes. Since $s_u$ is a bipolar threshold unit, we have $s_u \in \{\pm 1\}$ so that a change in state means a change in sign

$$s_u^{t+1} = -s_u^t \; . \tag{6.42}$$

Now consider the energy difference of the energies of a Hopfield net before and after an asynchronous update. In particular, consider

$$\Delta E = E_{t+1} - E_t \; . \tag{6.43}$$

For for the simple case in (6.41), we obviously have $\Delta E = 0$ but, for the much more interesting case in (6.42), we have a lot of work to do, so buckle up.

First of all, we recall that the energy of a Hopfield net in a macro state $\boldsymbol{s}$ is given by $E(\boldsymbol{s}) = -\frac{1}{2}\, \boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s}$ and observe that the quadratic form $\boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s}$ can be expanded like this

$$\sum_j \sum_k w_{jk}\, s_j\, s_k = \sum_{j \neq u} \sum_{k \neq u} w_{jk}\, s_j\, s_k \tag{6.44}$$

$$+ \sum_{j \neq u} w_{ju}\, s_j\, s_u \tag{6.45}$$

$$+ \sum_{k \neq u} w_{uk}\, s_u\, s_k \tag{6.46}$$

$$+ w_{uu}\, s_u\, s_u \ . \tag{6.47}$$

Now, since the weight matrix of a Hopfield net must by symmetric ($\boldsymbol{W} = \boldsymbol{W}^\intercal$) as well as hollow (diag$[\boldsymbol{W}] = \boldsymbol{0}$), we have $w_{jk} = w_{kj}$ as well as $w_{uu} = 0$. Using this, the above expansion simplifies to

$$\sum_j \sum_k w_{jk}\, s_j\, s_k = \sum_{j \neq u} \sum_{k \neq u} w_{jk}\, s_j\, s_k + 2 \sum_{k \neq u} w_{uk}\, s_u\, s_k \ . \tag{6.48}$$

This then allows us to express the energies $E_t$ and $E_{t+1}$ from before and after an asynchronous update as

$$E_t = -\tfrac{1}{2} \left( \sum_{j \neq u} \sum_{k \neq u} w_{jk}\, s_j^t\, s_k^t + 2 \sum_{k \neq u} w_{uk}\, s_u^t\, s_k^t \right) + \sum_j b_j\, s_j^t \tag{6.49}$$

and

$$E_{t+1} = -\tfrac{1}{2} \left( \sum_{j \neq u} \sum_{k \neq u} w_{jk}\, s_j^{t+1}\, s_k^{t+1} - 2 \sum_{k \neq u} w_{uk}\, s_u^{t+1}\, s_k^{t+1} \right) + \sum_j b_j\, s_j^{t+1} \ . \tag{6.50}$$

Plugging these lengthy expressions into (6.43) therefore provides us with

$$\Delta E = - \sum_{k \neq u} w_{uk}\, s_u^{t+1}\, s_k^{t+1} + \sum_j b_j\, s_j^{t+1} + \sum_{k \neq u} w_{uk}\, s_u^t\, s_k^t - \sum_j b_j\, s_j^t \tag{6.51}$$

which looks suspiciously simply but we we note that, thanks to (6.40), the two double sums cancel each other out.

If we next pull out the factors $s_u^{t+1}$ and $s_u^t$ of the two sums over $k$ and perform simple algebraic rearrangements, we obtain

$$\Delta E = -s_u^{t+1} \sum_{k \neq u} w_{uk}\, s_k^{t+1} + \sum_j b_j\, s_j^{t+1} + s_u^t \sum_{k \neq u} w_{uk}\, s_k^t - \sum_j b_j\, s_j^t \tag{6.52}$$

$$= s_u^t \sum_{k \neq u} w_{uk}\, s_k^t - s_u^{t+1} \sum_{k \neq u} w_{uk}\, s_k^{t+1} - \sum_j b_j \left( s_j^t - s_j^{t+1} \right) . \tag{6.53}$$

Next, we observe that (6.40) provides us with two crucial implications, namely $k \neq u \Rightarrow s_k^{t+1} = s_k^t$ and $j \neq u \Rightarrow s_j^{t+1} = s_j^t$ so that the above further simplifies to

$$\Delta E = \left(s_u^t - s_u^{t+1}\right) \sum_{k \neq u} w_{uk}\, s_k^t - b_u \left(s_u^t - s_u^{t+1}\right) \tag{6.54}$$

$$= \left(s_u^t - s_u^{t+1}\right) \left( \sum_{k \neq u} w_{uk}\, s_k^t - b_u \right) \tag{6.55}$$

$$= -2\, s_u^{t+1} \left( \sum_{k \neq u} w_{uk}\, s_k^t - b_u \right) \tag{6.56}$$

where the last step made use of (6.42).

At this point, rather late in the game, we note that our way of writing sums over $k \neq u$ is good bookkeeping but unnecessarily cumbersome since $w_{uu} = 0$. Using this, we may just as well write

$$\Delta E = -2\, s_u^{t+1} \left( \sum_{k} w_{uk}\, s_k^t - b_u \right) \tag{6.57}$$

$$= -2\, s_u^{t+1} \left( \boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u \right) . \tag{6.58}$$

We therefore find that the energy difference depends on the product of $s_u^{t+1}$ and $\boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u$ and we recall that the former is a function of the latter, namely

$$s_u^{t+1} = \operatorname{sign}\left( \boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u \right) . \tag{6.59}$$

But this means that the signs of $s_u^{t+1}$ and $\boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u$ are the same which implies that their product $s_u^{t+1}(\boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u)$ will always be non-negative. However, since the energy difference also involves an overall negative sign, we finally realize that

$$\Delta E = -2\, s_u^{t+1} \left( \boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u \right) \leq 0 . \tag{6.60}$$

Now this is a truly fundamental result! It shows that neither for the case in (6.41) nor for the case in (6.42) will an asynchronous update step ever increase the energy of a Hopfield net. Moreover, since there are “only” $2^n$ distinct macro states a Hopfield network can be in, asynchronous “energy minimization” will require only finitely many update steps until it reaches a macro state where the network’s energy function is at a (local) minimum. Such a state is stable; once it is reached further asynchronous updates will leave it intact. □

Note that the theorem we just proved states that asynchronous updates of the micro states of a Hopfield net necessarily lead to a macro state that is stable. However, it does not make any claim whatsoever about the quality of this stable state. Ideally, it would be a global minimizer of the network's energy function but it might just as well be only a local minimizer.

While this may sound sobering, it cannot be otherwise unless P = NP. Because if we had an asynchronous update procedure that was guaranteed to always find a global energy minimum of a Hopfield net regardless of its parameters and initial state, we would have a polynomial algorithm for the generally NP-hard problem of finding a or the minimum energy state of a Hopfield net.

On the plus side, we now know about a family of algorithms which will find at least local minima of the energy function of a Hopfield net, namely Algorithm 2 with different mechanisms for "somehow" selecting the update neuron $s_u$ in iteration $t$. As we did not have to specify the selection mechanism for our proof of Theorem 6.1, we are guaranteed that the fundamental result in (6.60) will hold for any selection mechanisms we may think of. As this will be of interest later on, we stress the result in (6.60) in terms of its own theorem.

**Theorem 6.2** *If a Hopfield net* $(\boldsymbol{W}, \boldsymbol{b})$ *is currently in macro state* $\boldsymbol{s}_t$ *and if some neuron* $s_u$ *updates its micro state from* $s_u^t$ *to* $s_u^{t+1} = \operatorname{sign}\bigl(\boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u\bigr)$*, then the network transits to a macro state* $\boldsymbol{s}_{t+1}$ *and its energy decreases by an amount of*

$$\Delta E = -2\, s_u^{t+1} \bigl(\boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u\bigr) \leq 0\,. \tag{6.61}$$

Finally, before we move on to discussing code examples, we should probably stress (one more time) that the guarantees provided by Theorems 6.1 and 6.2 only hold for Hopfield nets as in Definition 6.1. If we were to work with recurrent bipolar threshold networks whose weight matrices are neither symmetric nor hollow, both theorems would no longer hold.

## 6.4 Code Examples

Next, we discuss ideas for how to code functions for running Hopfield nets. Our code snippets will make heavy use of `numpy` functionalities and, at times, work with random numbers. For everything that is to come, we therefore require the following import and declaration

```
import numpy as np
rng = np.random.default_rng()
```

Recall that a major theme in the above text was that synchronous and asynchronous mechanisms for updating the state of a Hopfield net are agnostic about specifics of its parameters and initial state. In what follows, we will therefore only barely discuss how to set these up and leave interesting application examples to the next

```
def signum(x):
    return np.where(x >= 0, +1, -1)
```

**Code 6.1** A possible `numpy` implementations of the bipolar sign function in (6.62).

chapter. However, for readers who cannot wait until then but already want to run their own Hopfield nets, we emphasize again that we are concerned with Hopfield nets according to Definition 6.1. That is, all our code snippets which involve a weight matrix, a bias vector, and a state vector assume that these are of commensurable sizes and appropriate types, i.e. $\boldsymbol{W} \in \mathbb{R}^{n \times n}$, $\boldsymbol{b} \in \mathbb{R}^n$, and $\boldsymbol{s} \in \{\pm 1\}^n$. Moreover, and most crucially, they assume that the weight matrix is symmetric ($\boldsymbol{W} = \boldsymbol{W}^\intercal$) and hollow ($\text{diag}[\boldsymbol{W}] = \boldsymbol{0}$).

In short, we will again present code for considerate users as none of our functions performs type checking or exception handling. Also and for simplicity, our codes follow the imperative paradigm and do not make use of `python`'s object-oriented capabilities. That is, we will leave the idea of implementing a Hopfield network class to the reader as all the functions we discuss can easily be turned into methods. Having said all this, we are finally good to go.

The most important thing to worry about when implementing Hopfield network functionalities is providing the bipolar function sign : $\mathbb{R} \to \{\pm 1\}$ from (6.8) whose definition we repeat for convenience

$$\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ +1 & \text{if } x \geq 0 \,. \end{cases} \tag{6.62}$$

We emphasize this point because most mathematicians would likely define sign as a function sign : $\mathbb{R} \to \{-1, 0, +1\}$ with

$$\text{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ \;\;0 & \text{if } x = 0 \\ +1 & \text{if } x > 0 \end{cases} \tag{6.63}$$

which is what the `numpy` function `sign` computes. However, coding a `numpy` function for (6.62) is easy and there are numerous possible approaches we could consider. Here, we opt to work with `numpy`'s function `where` as shown in code snippet Code 6.1.

There is not much to be said about our custom function `signum` but we note that its parameter `x` may be a single number or any array of numbers. Due to `numpy`'s broadcasting capabilities, it will work properly in either case.

A useful helper function to have is `hnetEnergy` which computes the energy $E(\boldsymbol{s}) = -\frac{1}{2}\,\boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s}$ of a Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ in a macro state $\boldsymbol{s}$. Code snippet Code 6.2 presents a straightforward, plain vanilla `numpy` implementation.

```
def hnetEnergy(vecS, matW, vecB):
    return -0.5 * vecS @ matW @ vecS + vecB @ vecS
```

**Code 6.2** Plain vanilla `numpy` code for computing the energy of a Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ in state $\boldsymbol{s}$.

```
def hnetRunSync(vecS, matW, vecB, tmax=100, verbose=True):
    for t in range(tmax):

        if verbose:
            s = ''.join(['+' if x >= 0 else '-' for x in vecS])
            E = hnetEnergy(vecS, matW, vecB)
            print ('t = {:1d}, s = {},  E = {:+.1f}'.format(t, s, E))

        vecS = signum(matW @ vecS - vecB)

    return vecS
```

**Code 6.3** Simple `python`/`numpy` code for running a Hopfield net using synchronous updates.

A `numpy` function that realizes the synchronous Hopfield network updates in Algorithm 1 is just as easy to come by. Parameters `vecS`, `matW`, and `vecB` of function `hnetRunSync` in code snippet Code 6.3 are self explanatory. Its parameter `tmax` indicates the number of updates to be performed and its parameter `verbose` indicates whether or not the function should display information while running.

The body of `hnetRunSync` thus performs `tmax` iterations in each of which the current time $t$, the current macro state $\boldsymbol{s}_t$, and the corresponding energy $E(\boldsymbol{s}_t)$ are printed if desired. Most importantly though, each iteration also updates state $\boldsymbol{s}_t$ as prescribed by Algorithm 1.

To test function `hnetRunSync`, we may simply replicate our synchronous update experiment with the super simple Hopfield net in Fig. 6.2 using something like

```
# Hopfield net parameters
# and initial macro state
matW = np.array([[ 0,-1],
                 [-1, 0]])
vecB = np.array( [ 0, 0] )
vecS = np.array( [+1,+1] )

hnetRunSync(vecS, matW, vecB, tmax=4)
```

This produces the following output

```
t = 0, s = ++,  E = +1.0
t = 1, s = --,  E = +1.0
t = 2, s = ++,  E = +1.0
t = 3, s = --,  E = +1.0
```

and is therefore well in line with our above theoretical results and considerations.

Function `hnetRunAsynRND` in code snippet Code 6.4 presents an idea for how to realize the asynchronous update mechanism in Algorithm 2 using a uniformly random selection of the neuron $s_u$ which updates its micro state at time $t$. This code is noticeably similar to what we just discussed but note our use of the `python`

```
def hnetRunAsynRND(vecS, matW, vecB, tmax=100, verbose=True):
    for t, u in enumerate(rng.integers(0, len(vecS), tmax)):

        if verbose:
            s = ''.join(['+' if x >= 0 else '-' for x in vecS])
            E = hnetEnergy(vecS, matW, vecB)
            print ('t = {:1d}, s = {},  E = {:+.1f}'.format(t, s, E))

        vecS[u] = signum(matW[u] @ vecS - vecB[u])

    return vecS
```

**Code 6.4** Simple `python` / `numpy` code for running a Hopfield net using asynchronous updates with uniform random selection of the neuron $s_u$ that updates at time $t$.

function `enumerate` and the `integers` method of the random number generator `rng`. The way we invoke them produces an iterator of `tmax` elements which are random integers between 0 and $n - 1$ where $n$ denotes the number of neurons in the Hopfield net under consideration which, in turn, corresponds to `len (vecS)`.

To test this function `hnetRunSyncRND`, we may try to see if its is able to "replicate" our asynchronous update experiment with the super simple Hopfield net in Fig. 6.2. Note that we put replicate into quotation marks because, in the text above, we considered asynchronous updates with round robin selection of the updating neuron. Hence, if we use something like this

```
matW = np.array([[ 0,-1],
                 [-1, 0]])
vecB = np.array( [ 0, 0] )
vecS = np.array( [+1,+1] )

hnetRunAsynRND(vecS, matW, vecB, tmax=4)
```

we should expect to see similar behavior but nor necessarily the exact same behavior we have seen previously. Indeed, our first ever run of the above snippet resulted in

```
t = 0, s = ++,  E = +1.0
t = 1, s = +-,  E = -1.0
t = 2, s = +-,  E = -1.0
t = 3, s = +-,  E = -1.0
```

which differs from what we saw during our theoretical discussion (can you spot the differences?) but nevertheless confirms our theoretical analysis of the behavior and characteristics of asynchronous updates of a Hopfield net.

Finally, we suggest a piece of code that will simplify our lives in the next chapter. Function `hnetInitState` in code snippet Code 6.5 is a convenience function for getting an initial state $\boldsymbol{s}_0$ of a Hopfield net. Its first parameter `n` indicates the numbers of neurons and thus the dimensionality of the state vector of a Hopfield net. Its second parameter `mthd` indicates which method to use to compute $\boldsymbol{s}_0$ and its third parameter `mu` is required by one of these methods.

If `mthd` is set to `top`, the function returns the vector $+\mathbf{1} \in \{\pm 1\}^n$ which corresponds to the *top* vertex in our lattice graph visualizations of the energy landscape of a Hopfield net (see Fig. 6.3). Conversely, if `mthd` is set to `bot`, the function returns

```
def hnetInitState(n, mthd='rnd', mu=0.5):
    if mthd == 'top':
        return +np.ones(n)

    if mthd == 'bot':
        return -np.ones(n)

    return bin2bip(bernoulli(mu, n))
```

**Code 6.5** Convenience function for initializing a state $\boldsymbol{s}_0 \in \{\pm 1\}^n$ of a Hopfield net of $n$ neurons.

the vector $-\mathbf{1} \in \{\pm 1\}^n$ which corresponds to the *bottom* vertex in our lattice graph visualizations of the energy landscape of a Hopfield net (see Fig. 6.3).

The default behavior of `hnetInitState` is to return a random vector $\boldsymbol{s} \in \{\pm 1\}^n$. To accomplish this, we apply the functions `bin2bip` and `bernoulli` whose inner workings we discussed in Chap. 2. Here, we therefore simply note that parameter `mu` represents the success rate $0 \leq \mu \leq 1$ of a Bernoulli distribution and allows for control of the percentage of entries of $+1$ in $\boldsymbol{s}$. For instance, running

```
print (hnetInitState(20, mthd='rnd', mu=0.1))
print (hnetInitState(20, mthd='rnd', mu=0.9))
```

will produce an output like this

```
[-1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
[ 1  1  1  1  1  1  1 -1  1  1  1  1  1  1  1  1  1  1  1  1]
```

## 6.5 Exercises

**6.1** Consider the following squared Euclidean distance between two vectors $\boldsymbol{M}\boldsymbol{u}$ and $\boldsymbol{v}$ of commensurable dimensions and the squared difference between two scalars $\boldsymbol{u}^\intercal\boldsymbol{v}$ and $K$

$$\left\|\boldsymbol{M}\boldsymbol{u} - \boldsymbol{v}\right\|^2 = \left[\boldsymbol{M}\boldsymbol{u} - \boldsymbol{v}\right]^\intercal \left[\boldsymbol{M}\boldsymbol{u} - \boldsymbol{v}\right] \tag{6.64}$$

$$\left(\boldsymbol{u}^\intercal\boldsymbol{v} - K\right)^2 = \left(\boldsymbol{u}^\intercal\boldsymbol{v} - K\right) \cdot \left(\boldsymbol{u}^\intercal\boldsymbol{v} - K\right) . \tag{6.65}$$

Continue expanding the respective right hand sides to show that both expressions (implicitly) involve symmetric matrices.

**6.2** Ponder what you proved in the previous exercise and contextualize your result w.r.t. the various QUBO models we discussed in Chap. 5. At which point in their derivations did we come across squared Euclidean distances or squared differences? Could you think of an optimization problem with quadratic objectives or constraints for which there exists a QUBO model which does not involve symmetric matrices? Consider this question carefully and answer it as rigorously as possible.

**6.3** Implement a `numpy` function `hnetRunAsynRRB` that realizes the asynchronous mechanism in Algorithm 2 with *round robin* selection of the neuron $s_u$ that updates in iteration $t$.

**Note:** In the text, we said "neuron indices are counted from 1 but time steps are counted from 0". However, pay attention to the fact that, when working with `numpy` arrays to represent Hopfield net parameters and state vectors, the indices of array elements are counted from 0 rather than from 1.

**6.4** Let $\boldsymbol{W}$ be the weight matrix of some Hopfield net and let $\boldsymbol{s}_+$ be one of its macro states. Consider the negative counterpart of this state, namely $\boldsymbol{s}_- = -\boldsymbol{s}_+$ and prove the fallowing equality $\boldsymbol{s}_+^\intercal \boldsymbol{W} \boldsymbol{s}_+ = \boldsymbol{s}_-^\intercal \boldsymbol{W} \boldsymbol{s}_-$.

**6.5** Consider a Hopfield net of 4 neurons where

$$\boldsymbol{W} = \begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix} \quad \text{and} \quad \boldsymbol{b} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} . \tag{6.66}$$

What are the lowest and highest energy values of this network? For which macro states will these energy values be achieved?

**6.6** Let $\boldsymbol{x} \in \{\pm 1\}^n$ be some bipolar vector and consider a Hopfield net of $n$ neurons whose parameters are

$$\boldsymbol{W} = \boldsymbol{x}\boldsymbol{x}^\intercal - \boldsymbol{I} \quad \text{and} \quad \boldsymbol{b} = \boldsymbol{0} . \tag{6.67}$$

What are the lowest and highest energy values of this network? For which macro states will these energy values be achieved?

**6.7** Reconsider matrix $\boldsymbol{W}$ in (6.67). Why did we subtract the identity matrix $\boldsymbol{I}$ from the outer product matrix $\boldsymbol{x}\boldsymbol{x}^\intercal$?

**6.8** In the last but one exercise you should have found that a Hopfield net with weights $\boldsymbol{W} = \boldsymbol{x}\boldsymbol{x}^\intercal - \boldsymbol{I}$ and biases $\boldsymbol{b} = \boldsymbol{0}$ has two global energy minima. How would you have to (re)parameterize it such that it assumes a unique global energy minimum in macro state $\boldsymbol{s} = \boldsymbol{x}$?

**6.9** Assume a Hopfield net of $n$ neurons that runs under an asynchronous update mechanism (such as realized by `hnetRunAsynRND` or `hnetRunAsynRRB`). How would you have to choose its weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$ such that the network eventually settles in macro state $\boldsymbol{s} = -\boldsymbol{1} \in \{\pm 1\}^n$ regardless of whatever state it starts in?

**6.10** Let $k \in \mathbb{N}$ and design the weight matrix and bias vector of a Hopfield net of $n > k$ neurons such that, when it runs under asynchronous updates, it will eventually settle in a macro state where only $k$ neurons are active regardless of whatever state it starts in.

## References

1. Little, W.: The Existence of Persistent States in the Brain. Mathematical Bioscience **19**(1–2) (1974). https://doi.org/10.1016/0025-5564(74)90031-5
2. Amari, S.I.: Neural Theory of Association and Concept-formation. Biological Cybernetics **26** (1977). https://doi.org/10.1007/BF00365229
3. Hopfield, J.: Neural Networks and Physical Systems with Collective Computational Abilities. PNAS **79**(8) (1982). https://doi.org/10.1073/pnas.79.8.2554
4. Hopfield, J., Tank, D.: "Neural" Computation of Decisions in Optimization Problems. Biological Cybernetics **52** (1985). https://doi.org/10.1007/BF00339943
5. Bauckhage, C., Sanchez, R., Sifa, R.: Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In: Proc. Int. Joint Conf. on Neural Networks. IEEE (2020). https://doi.org/10.1109/IJCNN48605.2020.9206916
6. Krotov, D., Hopfield, J.: Dense Associative Memory for Pattern Recognition. In: Proc. NeurIPS (2016)
7. Krotov, D., Hopfield, J.: Dense Associative Memory Is Robust to Adversarial Inputs. Neural Computation **30**(12) (2018). https://doi.org/10.1162/neco_a_0114
8. Krotov, D.: A New Frontier for Hopfield Networks. Nature Reviews Physics **5** (2023). https://doi.org/10.1038/s42254-023-00595-y
9. Demircigil, M., et al.: On a Model of Associative Memory with Huge Storage Capacity. J. of Statistical Physics **168** (2017). https://doi.org/10.1007/s10955-017-1806-y
10. Hillar, C., Tran, N.: Robust Exponential Memory in Hopfield Networks. J. Mathematical Neuroscience **8**(1) (2018). https://doi.org/10.1186/s13408-017-0056-2
11. Ramsauer, H., et al.: Hopfield Networks Is All You Need. In: Proc. Int. Conf. on Learning Representations (2021)
12. Rumelhart, D., Hinton, G., Williams, R.: Learning Representations by Back-Propagating Errors. Nature **323**(6088) (1986). https://doi.org/10.1038/323533a0
13. Hebb, D.: The Organization of Behaviour. Wiley & Sons (1949)
14. Haykin, S.: Neural Networks and Learning Machines, 3rd edn. Pearson Education (2008)
15. Strogatz, S.: Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering, 2nd edn. Taylor & Francis (2015)

# Chapter 7
# Hopfield Nets in Action

## 7.1 Introduction

In Chap. 6, we defined the notion of a classical Hopfield net and its energy function and looked at synchronous- and asynchronous mechanisms for how it may evolve its state over time. With respect to the latter, we discussed the classical ideas of uniformly random- and round robin selection of which neuron to update in which iteration. However, we claimed that these ideas are a bit naïve and promised to study a better selection mechanism later on. In this chapter, we will make good on this promise. But what could this mean?

In Chap. 6, we also proved that, if all the neurons of a Hopfield net update their micro state asynchronously one at a time, then the macro state of the network will eventually settle in a stable state of (locally) minimum energy and that this will happen after only a finite number of updates. This is interesting from a practical point of view because we also saw that Hopfield energy functions and QUBO objective functions are closely related so that we may run Hopfield nets in order to solve QUBOs. Further recall that our proof of the energy minimizing behavior of an asynchronously evolving Hopfield net did not depend on how individual neurons are selected for update but holds for any selection mechanism that takes all neurons into account. Then how can some of these mechanisms be better than others?

Well, the issue lies in the terms "eventually" and "finite number". While it is true that uniformly random- and round robin selection of update neurons will only need a finite number of steps to eventually converge to a stable state of (locally) minimum energy, they select neurons in an *uniformed* manner that does not take into account any additional cues whose consideration could accelerate convergence.

In this chapter, we therefore study an *informed* selection mechanism that uses the *gradient* of the energy function to decide which neuron ought to update. We will see that this is an easily realized idea which works very well in practice. And, to corroborate this claim, we will look at several application examples which show how to use our ideas to quickly find (approximate) solutions to QUBOs.

C. Bauckhage and R. Sifa, *Quantum Computing from Hopfield Nets*,
Cognitive Technologies, https://doi.org/10.1007/978-3-031-99402-9_7

## 7.2 Informed Update Mechanisms

Since we just announced that gradients will play a crucial role in this chapter, we begin our discussion by recalling: The **gradient** of a continuous, differentiable, multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ evaluated at $\boldsymbol{x} \in \mathbb{R}^n$ is the vector of partial derivatives

$$\nabla f(\boldsymbol{x}) \equiv \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f(\boldsymbol{x})}{\partial x_1} \\ \frac{\partial f(\boldsymbol{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\boldsymbol{x})}{\partial x_n} \end{bmatrix} \tag{7.1}$$

and encodes the direction and magnitude of steepest ascent of $f$ at $\boldsymbol{x}$.

This characteristic of the gradient is famously used all throughout optimization and machine learning where we often need to find a minimizer of an objective function

$$\boldsymbol{x}_* = \operatorname{argmin}_{\boldsymbol{x} \in \mathbb{R}^n} f(\boldsymbol{x}) \; . \tag{7.2}$$

Since this is a minimization problem, it is common to work with the negative gradient $-\nabla f(\boldsymbol{x})$ which encodes the direction and magnitude of steepest descent of $f$ at $\boldsymbol{x}$. The most widely used and versatile method to minimize $f$ based on gradient information is **gradient descent** where we guess a (feasible) solution $\boldsymbol{x}_0$ and iteratively refine it using

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta_t \, \nabla f(\boldsymbol{x}_t) \tag{7.3}$$

where $\eta_t > 0 \in \mathbb{R}$ is called the *learning rate* at time $t$.

In practice, say, in the context of neural network training, things are usually much more involved than this but, luckily, for our purpose in this chapter, we do not have to worry about any of the potential intricacies (adaptive learning rates, line searches, early stopping, momentum terms, proximal schemes, ...). We will not even work with fully fledged gradient descent but only exploit the fact that $-\nabla f(\boldsymbol{x})$ points in the direction of steepest descent of $f$ at $\boldsymbol{x}$.

### *7.2.1 Gradients of Hopfield Energy Functions*

Recall that the energy of a Hopfield net $(\boldsymbol{W}, \boldsymbol{b}) \in \mathbb{R}^{n \times n} \times \mathbb{R}^n$ which has been initialized to a macro state $\boldsymbol{s} \in \{\pm 1\}^n$ is given by

$$E(\boldsymbol{s}) = -\tfrac{1}{2}\, \boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s} \; . \tag{7.4}$$

Now, *symbolically*, it is no problem whatsowever to derive the expression on the right with respect to vector $\boldsymbol{s}$. Indeed, resorting to the symbolic differentiation rules for multivariate calculus (which are, for instance, conveniently listed in [1]), we can almost effortlessly obtain an expression for $-\nabla E(\boldsymbol{s})$.

But why are we so hesitant and emphasize the word symbolically? Well, so far, we understood the energy function of a Hopfield net as a function $E : \{\pm 1\}^n \to \mathbb{R}$ which is not continuous and thus not differentiable.

However, its domain, i.e. the set of bipolar vectors $\{\pm 1\}^n$, is embedded in the Euclidean vector space $\mathbb{R}^n$. We may therefore more generally think of the energy as a function $E : \mathbb{R}^n \to \mathbb{R}$ which we *restrict* to $\{\pm 1\}^n$ whenever we want to evaluate the energy of a Hopfield net in state $\boldsymbol{s}$.

The point we are trying to make is that we could more generally think of an energy $E(\boldsymbol{x}) = -\frac{1}{2}\,\boldsymbol{x}^\intercal \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}^\intercal\boldsymbol{x}$ for any $\boldsymbol{x} \in \mathbb{R}^n$. The parameters of this function are still the parameters $(\boldsymbol{W}, \boldsymbol{b})$ of the Hopfield net under consideration but it is now a continuous, differentiable function. Given this *extended* function, we may still evaluate it at any $\boldsymbol{s} \in \{\pm 1\}^n$ to determine the energy $E(\boldsymbol{s})$ of state $\boldsymbol{s}$ of our Hopfield net. But what is more is that we can now also legally compute the negative gradient $-\nabla E(\boldsymbol{s})$ of the energy of that state.

While this may sound contrived, we will soon realize the curious fact that the theory of Hopfield nets implicitly assumes this generalized point of view as well. In fact, attentive readers may be puzzled by what we are going to state in the following theorem. Indeed, once we have proven it, there will be a lot to discuss.

**Theorem 7.1** *The negative gradient of the energy $E(\boldsymbol{s})$ of a macro state $\boldsymbol{s}$ of a Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ is given by*

$$-\nabla E(\boldsymbol{s}) = \boldsymbol{W}\boldsymbol{s} - \boldsymbol{b}\ . \tag{7.5}$$

***Proof*** Letting $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ be a square real valued matrix and $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$ be two real valued vectors, we recall the following basic facts: The derivatives of the quadratic form $\boldsymbol{v}^\intercal \boldsymbol{M}\boldsymbol{v}$ and the inner product $\boldsymbol{u}^\intercal\boldsymbol{v}$ both with respect to $\boldsymbol{v}$ are

$$\frac{\partial}{\partial \boldsymbol{v}}\,\boldsymbol{v}^\intercal \boldsymbol{M}\,\boldsymbol{v} = \left[\boldsymbol{M} + \boldsymbol{M}^\intercal\right]\boldsymbol{v} \tag{7.6}$$

$$\frac{\partial}{\partial \boldsymbol{v}}\,\boldsymbol{u}^\intercal\boldsymbol{v} = \boldsymbol{u}\ . \tag{7.7}$$

The negative derivative of the Hopfield energy $E(\boldsymbol{s})$ w.r.t. $\boldsymbol{s}$ is therefore given by

$$-\frac{\partial}{\partial \boldsymbol{s}}\,E(\boldsymbol{s}) = -\frac{\partial}{\partial \boldsymbol{s}}\left[-\frac{1}{2}\,\boldsymbol{s}^\intercal \boldsymbol{W}\boldsymbol{s} + \boldsymbol{b}^\intercal\boldsymbol{s}\right] \tag{7.8}$$

$$= \frac{\partial}{\partial \boldsymbol{s}}\left[\frac{1}{2}\,\boldsymbol{s}^\intercal \boldsymbol{W}\boldsymbol{s} - \boldsymbol{b}^\intercal\boldsymbol{s}\right] \tag{7.9}$$

$$= \frac{1}{2}\left[\boldsymbol{W} + \boldsymbol{W}^\intercal\right]\boldsymbol{s} - \boldsymbol{b} \tag{7.10}$$

$$= \boldsymbol{W}\boldsymbol{s} - \boldsymbol{b} \tag{7.11}$$

where the last step used the fact that the weight matrix of a Hopfield net must be symmetric $\boldsymbol{W} = \boldsymbol{W}^\intercal$ so that $\boldsymbol{W} + \boldsymbol{W}^\intercal = 2\boldsymbol{W}$. □

Probably the most striking observation about Theorem 7.1 is that we have seen the expression $\boldsymbol{W}\boldsymbol{s} - \boldsymbol{b}$ before!

Indeed, recall that a synchronous update of the macro state $\boldsymbol{s}_t$ of a Hopfield net is computed as

$$\boldsymbol{s}_{t+1} = \operatorname{sign}\bigl(\boldsymbol{W}\boldsymbol{s}_t - \boldsymbol{b}\bigr)\ . \tag{7.12}$$

Given our newfound knowledge, we can now equivalently write such an update as

$$\boldsymbol{s}_{t+1} = \operatorname{sign}\bigl(-\nabla E(\boldsymbol{s}_t)\bigr)\ . \tag{7.13}$$

In other words, we may now *interpret* a synchronous update of a Hopfield net as the network first computing the real vector $-\nabla E(\boldsymbol{s}_t) \in \mathbb{R}^n$ and second using the sign operator to project this vector onto the nearest corner $\boldsymbol{s}_{t+1} \in \{\pm 1\}^n$ of the $n$-dimensional bipolar hypercube.

Perhaps even more striking is to observe that the computations of individual neurons (implicitly) involve the energy gradient as well. To see this, we recall that, when triggered to update its micro state, a single neuron, say $s_u$, computes

$$s_u^{t+1} = \operatorname{sign}\bigl(\boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u\bigr) \tag{7.14}$$

where the scalar $\boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u$ is nothing but the $u$-th component of the vector $\boldsymbol{W}\boldsymbol{s}_t - \boldsymbol{b}$. Formally, we will express this as

$$\boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u = \bigl[\boldsymbol{W}\boldsymbol{s}_t - \boldsymbol{b}\bigr]_u \tag{7.15}$$

and therefore have the following interpretation of an asynchronous update of a micro state

$$\begin{aligned} s_u^{t+1} &= \operatorname{sign}\Bigl(\bigl[\boldsymbol{W}\boldsymbol{s}_t - \boldsymbol{b}\bigr]_u\Bigr) && (7.16)\\ &= \operatorname{sign}\Bigl(\bigl[-\nabla E(\boldsymbol{s}_t)\bigr]_u\Bigr) && (7.17)\\ &= \operatorname{sign}\Bigl(-\boldsymbol{e}_u^\intercal \nabla E(\boldsymbol{s}_t)\Bigr) && (7.18) \end{aligned}$$

where $\boldsymbol{e}_u$ denotes the $u$-th standard basis vector in $\mathbb{R}^n$.

In other words, we may now *interpret* an asynchronous update of a single neuron in a Hopfield net as the neuron first computing the real vector $-\nabla E(\boldsymbol{s}_t) \in \mathbb{R}^n$, second projecting out its $u$-th components, and third using the sign operator to map this real number to the nearest bipolar number $s_u^{t+1} \in \{\pm 1\}$.

What is curious about our gradient-based interpretations of synchronous- and asynchronous updates of a Hopfield net is that they are now blatantly obvious but rarely mentioned in the books or in scientific papers. This is regrettable because it impedes further insights. For instance, we could now go through a sequence of less tedious computations to show that asynchronous updates of a Hopfield net realize an energy minimization process. But there also are more interesting insights and applications.

### 7.2.2 Gradient Informed Asynchronous Updates

From Chap. 6 we already know that the decrease in energy due to an asynchronous update of neuron $s_u \in \{\pm 1\}$ which flips its state from $s_u^t$ to $s_u^{t+1} = -s_u^t$ is given by

$$\Delta E_u = -2\, s_u^{t+1} \big( \boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u \big) \,. \tag{7.19}$$

The expression in (7.19) can thus be understood as the answer to the question: *What happened after $s_u$ flipped its state*? However, in what follows, it will be preferable (logically and computationally) to ask: *What will happen when $s_u$ flips its state*? Using the identity $s_u^t = -s_u^{t+1}$, this can be answered as follows

$$\Delta E_u = 2\, s_u^t \big( \boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u \big) \,. \tag{7.20}$$

Given what we know now, we realize that this expression, too, implicitly involves a gradient. Indeed, we can rewrite it as

$$\Delta E_u = 2\, s_u^t \big[ -\nabla E(\boldsymbol{s}_t) \big]_u = -2\, s_u^t \big[ \nabla E(\boldsymbol{s}_t) \big]_u \,. \tag{7.21}$$

which we furthermore recognize as the $u$-th entry of the following $n$-dimensional vector

$$\Delta \boldsymbol{E} = -2\, \boldsymbol{s}_t \odot \nabla E(\boldsymbol{s}_t) \tag{7.22}$$

where $\odot$ denotes the *Hadamard product* or element-wise product of two vectors.

But what can we gain from this new insight? Is there any practical application of this vector of energy differences?

Well, the vector

$$\Delta \boldsymbol{E} = \big[ \Delta E_1, \Delta E_2, \ldots, \Delta E_n \big]^\intercal \tag{7.23}$$

can be used to realize an *informed* neuron selection mechanism for an asynchronous update process. This is because it contains all energy decreases that could result from an asynchronous update step. In other words, for every neuron $s_j$, it contains the decrease in energy $\Delta E_j$ we would observe if neuron $s_j$ was selected to update its state in iteration $t$ of an asynchronous updating process.

This is of practical interest because it first of all only takes low efforts or negligible overhead to compute $\Delta \boldsymbol{E}$ but the valuable information contained in $\Delta \boldsymbol{E}$ second of all allows us to determine which neuron(s) would maximally decrease the Hopfield energy when updated. Selecting only such neurons for update would realize a form of *steepest energy descent* rather than mere energy descent and would thus accelerate convergence to a stable state in the Hopfield energy landscape.

In short, what we are proposing is the following idea: In each iteration $t$ of an asynchronous Hopfield network update process, compute vector $\Delta \boldsymbol{E}$ as in (7.22) and determine the indices of its entries of minimum value

$$\mathcal{I} = \big\{ j \bigm| \Delta E_j = \min\{\Delta E_1, \Delta E_2, \ldots, \Delta E_n\}\big\} . \tag{7.24}$$

Note that we do indeed have to determine indices of *entries of minimum value*, because $\Delta E_j$ signifies a decrease in energy so that the smaller $\Delta E_j$, the more the energy will decrease. Also note that we do indeed have to determine an *index set* since the vector $\Delta \boldsymbol{E}$ may generally have several entries whose value is minimal.

Once the index set $\mathcal{I}$ has been computed, we propose to randomly select an update index $u \sim \mathcal{U}[\mathcal{I}]$ and to trigger neuron $s_u$ to update its state.

The following piece of pseudo code summarizes the overall procedure for steepest energy descent via asynchronous updates.

**Algorithm 1** Informed asynchronous updates of a Hopfield net realizing steepest energy descent

```
Require: Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ and initial state $\boldsymbol{s}_0 = \big[s_1^0, s_2^0, \ldots, s_n^0\big]^\intercal$
  for $t = 0, 1, 2, \ldots$ do
    compute
      $-\nabla E(\boldsymbol{s}_t) = \boldsymbol{W}\boldsymbol{s}_t - \boldsymbol{b}$
      $\Delta \boldsymbol{E} = -2\,\boldsymbol{s}_t \odot \nabla E(\boldsymbol{s}_t)$
    choose
      $u \sim \mathcal{U}\big[\{ j \mid \Delta E_j = \min\{\Delta \boldsymbol{E}\}\}\big]$
    update
      $s_u^{t+1} = \mathrm{sign}\Big(\big[-\nabla E(\boldsymbol{s}_t)\big]_u\Big)$
```

Before we move on to discuss practical applications of this idea, we note that what we proposed here can be seen as adhering to the paradigm of *informed machine learning* [2, 3]. This umbrella term refers to the idea of combining data-driven and knowledge-based techniques at various stages of the machine learning pipeline. Knowledge integration may, for instance, happen in modeling, in model training or in model application and is usually intended to reduce complexity, to circumvent the need for large amounts of data, or to accelerate computations. What we just went through is an example of informed computations during the application phase of a model. Algorithm 1 makes use of our background knowledge about gradients of Hopfield energies to reduce the number of iterations required to reach a stable state of (locally) minimum energy. Below, we illustrate that its informed neuron selection mechanism really leads to faster convergence than the classical mechanisms we discussed in the previous chapter.

## 7.3 Hopfield Nets as Problem Solvers

Next, we finally follow up on our announcements from Chaps. 4 and 5 and run Hopfield nets to solve constraint satisfaction problems and combinatorial optimization problems. In Chap. 5, we discussed several such problems in detail and demonstrated how to formalize them in terms of binary- or bipolar QUBOs. Here, we focus on two examples, namely the $K$-rooks problem and the plain vanilla subset sum problem (SSP). We consider the former because the behavior of a Hopfield net for solving it is easy to analyze and the latter because we returned to it again and again ever since we first encountered it in Chap. 1.

Since we already discusses the mathematics behind the QUBO models of both problems, we next only recall essential ideas where necessary and otherwise refer to Chap. 5. Our focus in the following will be on practical code examples and their behavior. Throughout, we will frequently reuse snippets from Chaps. 4, 5, and 6 and assume that readers are familiar with those.

### *7.3.1 Code Examples*

By now, it should be no surprise that all our upcoming `python/numpy` codes require the following import and declaration to work properly.

```
import numpy as np
rng = np.random.default_rng()
```

#### Gradient Informed Asynchronous Updates

To begin with, we need to think about how to implement the pseudo code for informed asynchronous updates of a Hopfield net in Algorithm 1. Given what we already saw in Chap. 6 when we discussed function `hnetRunAsynRND` in code snippet 6.4 which realizes asynchronous updates of a Hopfield with uniformly random selection of the updating neurons, this will be rather straightforward.

Indeed, function `hnetRunAsynGRD` in code snippet 7.1 provides a more or less immediate `numpy` implementation of Algorithm 1. Attentive readers will note that we do not include functionalities for information display as we did before but readers are of course welcome to extend the function correspondingly. Its parameters `vecS`, `matW`, and `vecB` represent an initial macro state $\boldsymbol{s}_0$, the weight matrix $\boldsymbol{W}$, and the

```
def hnetRunAsynGRD(vecS, matW, vecB, tmax=100):
    for t in range(tmax):
        grdE = matW @ vecS - vecB
        dltE = vecS * grdE
        updt = rng.choice(np.where(dltE == dltE.min())[0])

        vecS[updt] = signum(grdE[updt])

    return vecS
```

**Code 7.1** Simple `numpy` code for running a Hopfield net via asynchronous updates using a gradient informed mechanism for selecting the neuron $s_u$ that updates at time $t$.

bias vector $\boldsymbol{b}$ of a Hopfield net. Parameter `tmax` indicates the number of iterations to be performed.

In each iteration $t$ we first compute the negative gradient $-\nabla E(\boldsymbol{s}_t)$ in (7.1) and then the vector $\Delta \boldsymbol{E}$ from (7.22). Note that we drop the constant factor of 2 which does not impact the subsequent search for minimal entries and ever so slightly saves on computation time. To compute the index set $\mathcal{I}$ from (7.24), we work with the basic `numpy` functions `where` and `min` as shown. To randomly select an element from the resulting 1D array, we use `rng.choice`. The resulting integer `updt` represents the index $u$ of the neuron $s_u$ which is selected to reevaluate or update its current micro state $s_u^t$ and we finally implement Eq. (7.17) to realize this update which involves function `signum` from code snippet 6.1 in Chap. 6.

## The $K$-Rooks Problem

In Chap. 5, we discussed the $K$-rooks problem which asks for a placement of $K$ rooks on a $K \times K$ chessboard such that they do not threaten each other. We opted to identify the search space of this problem with the set of all $K \times K$ binary matrices $\boldsymbol{Z}$ with $Z_{rc} = 0$ indicating that the square in row $r$ and column $c$ is empty and $Z_{rc} = 1$ indicating that the corresponding square contains a rook.

To express the problem as a binary QUBO, we vectorized these matrices in a row-wise manner, for instance

$$\boldsymbol{Z} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \Leftrightarrow \quad \boldsymbol{z} = \begin{bmatrix}0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0\end{bmatrix}^\intercal \tag{7.25}$$

and modeled the row- and column sum-to-one constraints as

$$\left\| \left[\boldsymbol{I} \otimes \boldsymbol{1}^\intercal\right]\boldsymbol{z} - \boldsymbol{1} \right\|^2 \equiv \left\| \boldsymbol{C}_r \boldsymbol{z} - \boldsymbol{1} \right\|^2 = \boldsymbol{1} \tag{7.26}$$

$$\left\| \left[\boldsymbol{1} \otimes \boldsymbol{I}^\intercal\right]\boldsymbol{z} - \boldsymbol{1} \right\|^2 \equiv \left\| \boldsymbol{C}_c \boldsymbol{z} - \boldsymbol{1} \right\|^2 = \boldsymbol{1} \tag{7.27}$$

```
def hnetParametersRooks(k):
    matI   = np.eye(k)
    vec1   = np.ones(k)
    matCr  = np.kron(matI, vec1)
    matCc  = np.kron(vec1, matI)

    matQr, vecQr = matCr.T @ matCr, -2 * matCr.T @ vec1
    matQc, vecQc = matCc.T @ matCc, -2 * matCc.T @ vec1

    matQ, vecQ = matQr + matQc, vecQr + vecQc

    matW, vecB = bin2bipQubo(matQ, vecQ)

    # NOTE: weight matrices of Hopfield nets must be hollow
    np.fill_diagonal(matW, 0)

    return matW, vecB
```

**Code 7.2** Simple `numpy` code to compute the parameters of a Hopfield net for the $K$-rooks problem.

where $\mathbf{1}$ is the $K$-dimensional vector of all ones and $\boldsymbol{I}$ denotes the $K \times K$ identity matrix. Some tedious yet straightforward algebra then led to the following problem formulation

$$z_* = \operatorname{argmin}_{z \in \{0,1\}^{K^2}} z^\intercal \big[ \boldsymbol{C}_r^\intercal \boldsymbol{C}_r + \boldsymbol{C}_c^\intercal \boldsymbol{C}_c \big] z - 2\,\mathbf{1}^\intercal \big[ \boldsymbol{C}_r + \boldsymbol{C}_c \big] z \,. \tag{7.28}$$

To be able to solve this binary QUBO by means of running a Hopfield net, we first have to translate it to a bipolar QUBO which we recall is possible using Theorem 4.1. We further recall that the parameter matrix of a bipolar QUBO will always be symmetric but not necessarily hollow which is a requirement for the weight matrix of a Hopfield net. However, Theorem 4.3 guarantees that we can hollow out the parameter matrix of a bipolar QUBO without affecting its solution.

Given these prerequisites, we can now understand the rationale behind function `hnetParametersRooks` in code snippet 7.2 which we use to set up the parameters $(\boldsymbol{W}, \boldsymbol{b})$ of a Hopfield net that can solve the $K$-rooks problem. The first couple of lines of code will be familiar from Chap. 5 where we already discussed how to set up the parameters of the binary QUBO in (7.28). Here, we store them in arrays `matQ` and `vecQ` which we then feed into function `bin2bipQubo` from code snippet 4.1 in Chap. 4 to obtain arrays `matW` and `vecB` representing the parameters of the equivalent bipolar QUBO. To be able to use these as the weight- and bias parameters of a corresponding Hopfield net, we finally set all the diagonal elements of `matW` to zero which is easily done by invoking `numpy`'s function `fill_diagonal`.

Setting up the parameters of a Hopfield net for solving, say, the $K = 4$-rooks problem is thus as simple as

```
k = 4; matW, vecB = hnetParametersRooks(k)
```

Given the resulting arrays `matW` and `vecB`, we can now examine the behavior of function `hnetRunAsynGRD` from code snippet 7.1 which, of course, also require

**Table 7.1** Exemplary evolutions of macro states and energies of Hopfield nets running gradient informed asynchronous updates to solve the 4-rooks problem. (**a**) Initial state $\boldsymbol{s}_0 = -\mathbf{1}$. (**b**) Initial state $\boldsymbol{s}_0 = +\mathbf{1}$. (**c**) Random initial state $\boldsymbol{s}_0$ with entries $s_j^0 \sim 2 \cdot f_{\text{Ber}}(z \mid \mu = 0.5) - 1$

| $t$ | $\boldsymbol{s}_t$ | $E(\boldsymbol{s}_t)$ |
|---|---|---|
| 0 | −−−−−−−−−−−−−−−− | −8.0 |
| 1 | −−−−−−−−−+−−−−−− | −10.0 |
| 2 | −−−−−−−−−+−−−−−+ | −12.0 |
| 3 | −−−−+−−−−+−−−−−+ | −14.0 |
| 4 | −−+−+−−−−+−−−−−+ | −16.0 |
| 5 | −−+−+−−−−+−−−−−+ | −16.0 |
| 6 | −−+−+−−−−+−−−−−+ | −16.0 |
| 7 | −−+−+−−−−+−−−−−+ | −16.0 |
| 8 | −−+−+−−−−+−−−−−+ | −16.0 |
| 9 | −−+−+−−−−+−−−−−+ | −16.0 |
| 10 | −−+−+−−−−+−−−−−+ | −16.0 |
| 11 | −−+−+−−−−+−−−−−+ | −16.0 |
| 12 | −−+−+−−−−+−−−−−+ | −16.0 |
| 13 | −−+−+−−−−+−−−−−+ | −16.0 |
| 14 | −−+−+−−−−+−−−−−+ | −16.0 |
| 15 | −−+−+−−−−+−−−−−+ | −16.0 |
| 16 | −−+−+−−−−+−−−−−+ | −16.0 |
| 17 | −−+−+−−−−+−−−−−+ | −16.0 |
| 18 | −−+−+−−−−+−−−−−+ | −16.0 |
| 19 | −−+−+−−−−+−−−−−+ | −16.0 |
| 20 | −−+−+−−−−+−−−−−+ | −16.0 |

(a)

| $t$ | $\boldsymbol{s}_t$ | $E(\boldsymbol{s}_t)$ |
|---|---|---|
| 0 | ++++++++++++++++ | +56.0 |
| 1 | ++++++−+++++++++ | +46.0 |
| 2 | −+++++−+++++++++ | +36.0 |
| 3 | −+++++−++++−++++ | +26.0 |
| 4 | −+++++−++++−+−++ | +16.0 |
| 5 | −+−+++−++++−+−++ | +10.0 |
| 6 | −+−+++−+−++−+−++ | +4.0 |
| 7 | −+−+++−−−++−+−++ | −2.0 |
| 8 | −+−+++−−−++−+−−+ | −6.0 |
| 9 | −+−+++−−−−+−+−−+ | −10.0 |
| 10 | −+−++−−−−−+−+−−+ | −12.0 |
| 11 | −+−++−−−−−+−−−−+ | −14.0 |
| 12 | −+−−+−−−−−+−−−−+ | −16.0 |
| 13 | −+−−+−−−−−+−−−−+ | −16.0 |
| 14 | −+−−+−−−−−+−−−−+ | −16.0 |
| 15 | −+−−+−−−−−+−−−−+ | −16.0 |
| 16 | −+−−+−−−−−+−−−−+ | −16.0 |
| 17 | −+−−+−−−−−+−−−−+ | −16.0 |
| 18 | −+−−+−−−−−+−−−−+ | −16.0 |
| 19 | −+−−+−−−−−+−−−−+ | −16.0 |
| 20 | −+−−+−−−−−+−−−−+ | −16.0 |

(b)

| $t$ | $\boldsymbol{s}_t$ | $E(\boldsymbol{s}_t)$ |
|---|---|---|
| 0 | ++++++++−+−−−−+− | +12.0 |
| 1 | +++++−++−+−−−−+− | +4.0 |
| 2 | ++−++−++−+−−−−+− | −4.0 |
| 3 | ++−++−+−−+−−−−+− | −8.0 |
| 4 | −+−++−+−−+−−−−+− | −12.0 |
| 5 | −−−++−+−−+−−−−+− | −14.0 |
| 6 | −−−++−−−−+−−−−+− | −16.0 |
| 7 | −−−++−−−−+−−−−+− | −16.0 |
| 8 | −−−++−−−−+−−−−+− | −16.0 |
| 9 | −−−++−−−−+−−−−+− | −16.0 |
| 10 | −−−++−−−−+−−−−+− | −16.0 |
| 11 | −−−++−−−−+−−−−+− | −16.0 |
| 12 | −−−++−−−−+−−−−+− | −16.0 |
| 13 | −−−++−−−−+−−−−+− | −16.0 |
| 14 | −−−++−−−−+−−−−+− | −16.0 |
| 15 | −−−++−−−−+−−−−+− | −16.0 |
| 16 | −−−++−−−−+−−−−+− | −16.0 |
| 17 | −−−++−−−−+−−−−+− | −16.0 |
| 18 | −−−++−−−−+−−−−+− | −16.0 |
| 19 | −−−++−−−−+−−−−+− | −16.0 |
| 20 | −−−++−−−−+−−−−+− | −16.0 |

(c)

us to provide an initial macro state `vecS`. In the following snippet, we therefore iterate over the values of the `mthd` parameter of function `hnetInitState` from code snippet 6.5 in Chap. 6 to see what happens if we initialize $\boldsymbol{s}_0$ to $-\mathbf{1}$, to $+\mathbf{1}$, and to a random bipolar Bernoulli vector with entries $s_j \sim 2 \cdot f_{\text{Ber}}(z \mid \mu) - 1$ where $\mu = 0.5$.

```
for mthd in ['bot', 'top', 'rnd']:
    vecS = hnetInitState(k**2, mthd)
    vecS = hnetRunAsynGRD(vecS, matW, vecB, tmax=21)
```

Running a verbose version of `hnetRunAsynGRD` which prints information about states and energies, this will result in outputs such as shown in Table 7.1.

The evolution in Table 7.1a starts in the initial state $\boldsymbol{s}_0 = -\mathbf{1}$ which models the situation where no rook is places on the board; the energy $E(\boldsymbol{s}_0) = -8$ of this state is fairly low. After one asynchronous update using gradient information, one of the neurons of the network has switched its state from inactive to active and the resulting macro state $\boldsymbol{s}_1$ represent a situation where one rook has been placed in the board; the energy $E(\boldsymbol{s}_1) = -10$ of this state is lower than that of the initial state. After four iterations, four neurons have switched their states from inactive to active and the resulting macro state $\boldsymbol{s}_4$ represent a situation where four rooks have been placed on the board such that they do not threaten each other. Indeed, we have

$$2\,\boldsymbol{s}_4 - \mathbf{1} = \boldsymbol{z} = \left[0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1\right]^{\intercal} \Leftrightarrow \boldsymbol{Z} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.29}$$

and note that the energy value $E(\boldsymbol{s}_1) = -16$ is the smallest possible value that can occur in our current setting (can you confirm this for yourself?). In other words, after only four iterations of steepest energy descent, our Hopfield net has reached stable macro state $\boldsymbol{s}_*$ of globally minimal energy $E(\boldsymbol{s}_*)$ and Table 7.1a illustrates that any further iterations do leave this state intact.

The evolution in Table 7.1b starts in the initial state $\boldsymbol{s}_0 = +\mathbf{1}$ which models the situation where there is a rook in every square of the board and the corresponding energy $E(\boldsymbol{s}_0) = +54$ is at a maximum. Here, it takes 12 iterations of steepest energy descent until the network settles in a stable state. This makes sense because 12 rooks have to be removed from a completely filled $4 \times 4$ board to end up with a configuration where there are only 4 rooks left. We furthermore note that, in this experiment, too, our Hopfield net successfully finds another valid solution to the problem.

Finally, the evolution in Table 7.1c which starts from a random initial state marks a middle ground. Here, there are 10 initially active neurons and it takes six iterations of steepest energy descent for the network to deactivate 6 of them to end up in a stable state which encodes a valid solution. However, this is but one example of a run with a random initial state. Other random initial states may not be as benign and it may take activation and deactivation of more neurons to progress towards a solution.

To qualitatively compare the behavior of gradient informed asynchronous updates against the behavior of asynchronous updates with a uniformly random selection of the updating neuron at time $t$, we may run the following snippet in which we work with function `hnetRunAsynRND` from code snippet 6.4 in Chap. 6.

```
for mthd in ['bot', 'top', 'rnd']:
    vecS = hnetInitState(k**2, mthd)
    vecS = hnetRunAsynRND(vecS, matW, vecB, tmax=21)
```

Examples of the kind of outputs this produces are shown in Table 7.2. Looking at the evolutions of macro states and energies shown in the different panels, we note that, regardless of the initial state, there occur shorter or longer sequences of steps in which neither macro state nor energy of the Hopfield net change. This is because, in these experiments, neurons are randomly selected to reevaluate or update their micro states and we saw in Chap. 6 that there may be situations where $s_u^{t+1} = s_u^t$. When working with random selections of the updating neurons, these situation thus seem to be more likely than when working with gradient informed selections. As a consequence, we further observe that Hopfield nets running this update mechanism may get stuck at saddle points and thus take much longer to reach stable states of (locally) minimum energy.

Now, having seen and said all this, we must once again emphatically emphasize that the $K$-rooks problem is trivial. Although we did not visualize its state space of size $2^{K^2}$, it just would have been too cluttered and unclear, corresponding QUBO objective functions or Hopfield energies defined over this space only have comparatively few local minima so that Hopfield nets will almost always evolve to macro states that represent valid solutions [4]. Readers are strongly encouraged to further experiment with the above code snippets to see this for themselves. Most other prob-

**Table 7.2** Exemplary evolutions of macro states and energies of Hopfield nets running random asynchronous updates to solve the 4-rooks problem. (**a**) Initial state $\boldsymbol{s}_0 = -\mathbf{1}$. (**b**) Initial state $\boldsymbol{s}_0 = +\mathbf{1}$. (**c**) Random initial state $\boldsymbol{s}_0$ with entries $s_j^0 \sim 2 \cdot f_{\text{Ber}}(z \mid \mu = 0.5) - 1$

| $t$ | $\boldsymbol{s}_t$ | $E(\boldsymbol{s}_t)$ |
|---|---|---|
| 0 | −−−−−−−−−−−−−−−− | −8.0 |
| 1 | −−−−−−−−−−+−−−−− | −10.0 |
| 2 | −−−−−−−−−−++−−−− | −10.0 |
| 3 | −−−−−+−−−−++−−−− | −12.0 |
| 4 | −−−−−+−−−−++−−−+ | −12.0 |
| 5 | −−−−−+−−−−++−−−+ | −12.0 |
| 6 | −−+−−+−−−−++−−−+ | −12.0 |
| 7 | −−+−−+−−−−++−−−+ | −12.0 |
| 8 | −−+−−+−−−−++−−−+ | −12.0 |
| 9 | −−+−−+−−−−++−−−+ | −12.0 |
| 10 | −−+−−+−−−−+−−−−+ | −14.0 |
| 11 | −−+−−+−−+−+−−−−+ | −14.0 |
| 12 | −−+−−+−−+−+−−−−+ | −14.0 |
| 13 | −−+−−+−−+−+−−−−+ | −14.0 |
| 14 | −−+−−+−−+−+−−−−+ | −14.0 |
| 15 | −−+−−+−−+−+−−−−+ | −14.0 |
| 16 | −−+−−+−−+−+−−−−+ | −14.0 |
| 17 | −−+−−+−−+−+−−−−+ | −14.0 |
| 18 | −−+−−+−−+−+−−−−+ | −14.0 |
| 19 | −−+−−+−−+−+−−−−+ | −14.0 |
| 20 | −−+−−+−−+−+−−−−+ | −14.0 |

(a)

| $t$ | $\boldsymbol{s}_t$ | $E(\boldsymbol{s}_t)$ |
|---|---|---|
| 0 | ++++++++++++++++ | +56.0 |
| 1 | −+++++++++++++++ | +46.0 |
| 2 | −++++++++++−++++ | +36.0 |
| 3 | −++++++++−+−++++ | +28.0 |
| 4 | −++++++++−+−+−++ | +20.0 |
| 5 | −++++++++−+−+−++ | +20.0 |
| 6 | −++−+++++−+−+−++ | +14.0 |
| 7 | −++−+++++−+−+−++ | +14.0 |
| 8 | −++−++−++−+−+−++ | +4.0 |
| 9 | −++−++−++−+−+−++ | +4.0 |
| 10 | −+−−++−++−+−+−++ | +0.0 |
| 11 | −+−−+−−++−+−+−++ | −4.0 |
| 12 | −+−−+−−−+−+−+−++ | −6.0 |
| 13 | −+−−+−−−+−+−+−++ | −6.0 |
| 14 | −+−−+−−−+−+−+−++ | −6.0 |
| 15 | −+−−+−−−+−+−+−++ | −6.0 |
| 16 | −+−−+−−−+−+−+−−+ | −10.0 |
| 17 | −+−−+−−−+−+−+−−+ | −10.0 |
| 18 | −+−−+−−−+−+−+−−+ | −10.0 |
| 19 | −+−−+−−−+−+−+−−+ | −10.0 |
| 20 | −+−−+−−−+−+−+−−+ | −10.0 |

(b)

| $t$ | $\boldsymbol{s}_t$ | $E(\boldsymbol{s}_t)$ |
|---|---|---|
| 0 | +−−++++−−+++++−+ | +10.0 |
| 1 | +−−+++−−−+++++−+ | +6.0 |
| 2 | −−−+++−−−+++++−+ | +2.0 |
| 3 | −−−+++−−−+++++−+ | +2.0 |
| 4 | −−−++−−−−+++++−+ | −2.0 |
| 5 | −−−++−−−−+++++−+ | −2.0 |
| 6 | −−−++−−−−+++−+−+ | −6.0 |
| 7 | −−−++−−−−+++−+−+ | −6.0 |
| 8 | −−−++−−−−+−+−+−+ | −8.0 |
| 9 | −−−++−−−−+−+−+−+ | −8.0 |
| 10 | −−+++−−−−+−+−+−+ | −8.0 |
| 11 | −−+++−−−−+−+−+−− | −12.0 |
| 12 | −−+++−−−−+−+−+−− | −12.0 |
| 13 | −−+++−−−−+−+−+−− | −12.0 |
| 14 | −−+++−−−−+−+−+−− | −12.0 |
| 15 | −−+−+−−−−+−+−+−− | −14.0 |
| 16 | −−+−+−−−−+−+−+−− | −14.0 |
| 17 | −−+−+−−−−−−+−+−− | −16.0 |
| 18 | −−+−+−−−−−−+−+−− | −16.0 |
| 19 | −−+−+−−−−−−+−+−− | −16.0 |
| 20 | −−+−+−−−−−−+−+−− | −16.0 |

(c)

```python
def hnetParametersSSP(vecX, trgt):
    matQ, vecQ = np.outer(vecX, vecX), -2*trgt * vecX

    matW, vecB = bin2bipQubo(matQ, vecQ)

    # NOTE: weight matrices of Hopfield nets must be hollow
    np.fill_diagonal(matW, 0)

    return matW, vecB
```

**Code 7.3** Simple `numpy` code to compute Hopfield net parameters for the plain vanilla SSP.

lems we might want to solve by running Hopfield nets are not so benign [5–7] and we address this issue with our next example.

## The Plain Vanilla SSP

Recall that the plain vanilla subset sum problem with a set of $n$ integers $\mathcal{X}$ and a target value $T$ asks for a subset $\mathcal{X}' \subseteq \mathcal{X}$ such that the $x \in \mathcal{X}'$ precisely sum to $T$. In Chap. 5, we saw that we may represent the given set $\mathcal{X}$ in terms of a vector $\boldsymbol{x} \in \mathbb{Z}^n$ and any subset $\mathcal{X}'$ in terms of an indicator vector $\boldsymbol{z} \in \{0, 1\}^n$ to formalize this SSP as a binary QUBO, namely

$$\boldsymbol{z}_* = \operatorname{argmin}_{\boldsymbol{z} \in \{0,1\}^n} \boldsymbol{z}^\intercal \big[\boldsymbol{x}\boldsymbol{x}^\intercal\big] \boldsymbol{z} - \big[2\,T\boldsymbol{x}\big]^\intercal \boldsymbol{z}\ . \tag{7.30}$$

Next, we discuss `python` / `numpy` for solving this problem by means of running Hopfield nets. This first of all requires us to instantiate the parameters $(\boldsymbol{W}, \boldsymbol{b})$ of a corresponding Hopfield net and code snippet 7.3 shows how to accomplish this. It defines a function `hnetParametersSSP` with inputs `vecX` and `trgt` which represent the problem parameters $\boldsymbol{x}$ and $T$, respectively.

Within the body of this function, we first compute two arrays `matQ` and `vecQ` which represent the two parameters $\boldsymbol{Q} = \boldsymbol{x}\boldsymbol{x}^\intercal$ and $\boldsymbol{q} = 2\,T\boldsymbol{x}$ of the binary QUBO in (7.30). Given these, we once again resort to function `bin2bipQubo` from code snippet 4.1 to obtain arrays `matW` and `vecB` representing the parameters of the equivalent bipolar QUBO. Finally, to be able to use these as the weight matrix and bias vector of a corresponding Hopfield net, we once again zero out the diagonal of `matW` just as we did above.

Now, in order to practically work with a moderately challenging SSP, we return the the problem instance we specified at the very beginning of Chap. 1. Implemented in terms a `python` integer and a `numpy` integer array, the ingredients of this particular SSP read

```
trgt = 8364
vecX = np.array([265, 453, 311, 876, 158, 344,  65, 411, 366, 314,
                 200, 816, 305, 716, 892, 787,  45, 258, 967, 669,
                 525, 638, 351, 438, 839, 438, 732, 675, 429, 278,
                 175, 192, 408, 243, 733, 176, 111, 570, 332, 766,
                 925, 680, 305, 805, 167, 162, 442, 404,  14, 603,
                 279, 636, 927, 757, 780, 892, 178, 148,  11, 766,
                 272, 520, 317, 573,  89, 776, 389, 444, 429, 984,
                 296,  68, 415, 257, 205, 409, 664, 472, 888,  25,
                 641, 367, 791, 687, 344, 320, 936, 520,  36, 494,
                 719, 362,  78, 437, 841, 163, 869, 945, 918, 840])
```

and setting up the parameters of a respective Hopfield net for solving it is as easy as

```
matW, vecB = hnetParametersSSP(vecX, trgt)
```

Since we are dealing with an SSP of problem size $|\mathcal{X}| = 100$, the macro states of our Hopfield net are vectors $\boldsymbol{s} \in \{\pm 1\}^{100}$ and the size of its state space is $2^{100}$. As modern QUBOs go, this has to be considered a rather moderately large state space but it is of course way too big to allow for exhaustive exploration. Moreover, given combinatorics like these, it is more than likely that the corresponding Hopfield energy function will have numerous local minima. This, in turn, will make it more than likely that our Hopfield net will get stuck in one of those regardless of what initial state it starts evolving from.

As simple heuristic for addressing both these problems is to run our Hopfield net several times, each time starting from a randomly initialized state. At the end of each run, we may then verify if it found a valid solution and could record those if we were so inclined. The following code snippet realizes this idea; yet, for simplicity, it does not record but only prints solutions.

```
tmax, numruns = 100, 100

for run in range(numruns):
    vecS = hnetInitState(len(vecX), mthd='rnd', mu=0.01)
    vecS = hnetRunAsynGRD(vecS, matW, vecB, tmax)

    mask = np.where(vecS>0, True, False)
    ssum = np.sum(vecX[mask])

    if ssum == trgt:
        print ('run:', run)
        print ('sol:', vecX[mask])
```

Looking at this snippet, we observe that it performs 100 runs of our asynchronous update algorithm with gradient informed selection of the updating neuron. For each of these runs, the number of iterations performed by the algorithm is set to 100, too. We further observe that the initial state of a run is a random bipolar Bernoulli vector where the success rate is deliberately chosen to be small, namely $\mu = 0.01$, so that only about 1% of the neuron of the network are initially active. Once the Hopfield net has terminated a run, we compute the sum of those elements of array `vecX` for which the corresponding entry of array `vecS` has a value of $+1$. If this sum equals the target value, we let the snippet print this solution.

With this particular parameterization, the above snippet executes in a fraction of a second on a standard desktop PC and much, much faster running times are possible when working tailor-made GPU implementations [8]. But what kind of results does it produce? Well, here is an exemplary output.

```
run: 18
sol: [ 65 967 925 805 927 892 984 936 945 918]
run: 53
sol: [ 45 967 925 162 927 892 984 296 367 936 945 918]
run: 89
sol: [967 925 305 805 927  11 984 641 936 945 918]
```

Looking at this result, it is clear that our SSP has more than one solution which in turn implies that the energy function of our Hopfield net has several global minima. But there is more.

If we rerun the above snippet with a larger choice of the Bernoulli parameter $\mu$, say with $\mu = 0.1$, it will produce outputs like these.

```
run: 3
sol: [200 258 967 351 675 278 192 408 111 603 984 296 415 841 945 840]
run: 14
sol: [311 876 314 892 638 570 305 892 776 984  68 320  36 437 945]
run: 19
sol: [265 453 314 716 967 442 429 984 888 791 687 320 163 945]
run: 21
sol: [314 200 967 669 429 408 925 442 603  11  89 389 444 984  25 520 945]
run: 29
sol: [876 258 967 732 192 404 389 984 320 936 520 841 945]
run: 30
sol: [200 967 351 925 927 892 984 205 936  36  78 945 918]
run: 31
sol: [892 967 839 438 732 603 892  11  89 776 444 841 840]
run: 63
sol: [453 967 675 192 570 925  14 927 776 984 936 945]
run: 67
sol: [453 816 892 967 438 305 162 404 279 927 892 272 573 984]
run: 71
sol: [200 816 967 175 925 148 272 389 984 888 936 719 945]
run: 81
sol: [314 258 967 438 176 332 680  14 927 984 687 344 936 362 945]
run: 84
sol: [344 892 967 175 925 404 927 984  25 936 945 840]
run: 93
sol: [876 158 344 967 675 927 317  89 776 984  68 320 945 918]
run: 99
sol: [816  45 967 675 332 925 279 573  89 429 984 664 641 945]
```

Looking at this result, we observe that the average size of the subsets found are large than in our first experiment and that more of these larger and unique subsets have been found in (only) 100 runs. If we now were to go on increasing $\mu$ and the number of runs, we would find even more solutions. Readers should really try this out! In fact, in an experiment with one million runs, we have found more than 45,000 solutions to our exemplary SSP. While this sounds like a lot, we note the following: For an SSP of problem size $|\mathcal{X}| = 100$, there are

$$2^{100} = 1,267,650,600,228,229,401,496,703,205,376 \tag{7.31}$$

potential solutions. Given a search space of this magnitude, namely of size $O(10^{30})$, a solution space of size $O(10^4)$ pales in comparison. We are thus searching for the proverbial needle in a haystack and it is interesting to see that Hopfield nets with steepest energy descent updates are very well able to perform searches like these.

So, how many solutions are there in total? We will never know! The combinatorics of our problem defy exhaustive searches and randomized approaches such as ours simply cannot guarantee any conclusive answers.

## 7.3.2 *Remarks on Local Search*

Our functions `hnetRunAsynGRD` and `hnetRunAsynRND` run Hopfield nets in an asynchronous manner. The former employs an informed strategy and thus works

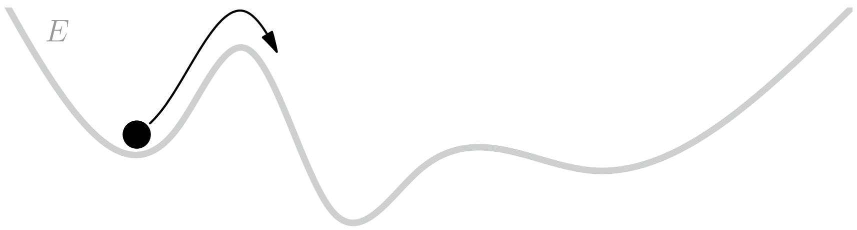


**Fig. 7.1** Caricature of the energy function and macro state of a Hopfield net. In the situation shown, the state of the network is trapped in a local minimum of the energy function. Asynchronous updates of individual neurons will only take effect if they decrease the energy and even then would only locally move the macro state. Under asynchronous local updates, the current state is therefore stable and cannot escape from its position. This would instead require (several) steps in a direction of increasing energy after which the network could proceed towards the global minimum of its energy function

smarter than the latter but both trigger neurons to update their micro states one at at time. In Chap. 6, we proved that both procedures are therefore guaranteed to converge to a stable macro state of the network for which its energy function will attain a (local) minimum. But let us be more specific about asynchronous updates.

Assume that, at time $t$, neuron $s_u$ is triggered to update its micro state and that it has been a while since it was last selected to do so. If the network's macro state did not change in the meantime, then the deterministic computation $s_u^{t+1} = \text{sign}(\boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u)$ at the current time $t$ produces the same result as at the time it was last triggered. But this means that neuron $s_u$ will not change its current state so that we will have $s_u^{t+1} = s_u^t$. However, if the network's macro state did change, then triggering the computation $s_u^{t+1} = \text{sign}(\boldsymbol{w}_u^\intercal \boldsymbol{s}_t - b_u)$ may or may not cause the neuron to flip its activation; whatever happens depends on how the macro state has changed and is generally difficult to predict.

In any case we note that, whenever an asynchronous update causes a neuron to flip its state such that $s_u^{t+1} = -s_u^t$, the network as a whole will transit from a macro state $\boldsymbol{s}_t$ to another macro state $\boldsymbol{s}_{t+1}$ which differs in one and only one component. In other words, if an asynchronous update flips a neuron, the whole network transits from its current macro state to a neighboring macro state or, put differently still, makes a *local* move in its state space rather than a long range jump.

It is for these reasons that asynchronous update algorithms for problem solving are also called **local search** procedures. So what it there to remark about those?

Well, people with a background in optimization or machine learning will know that function minimization via local search easily runs the risk of getting trapped in local minima. To fathom the direness of such an occurrence, way may consider the sketch in Fig. 7.1.

The grey curve depicted in this figure symbolizes an energy function and the black ball symbolizes the state of a dynamical system searching for an energy minimum. With respect to Hopfield nets, this picture is of course but a caricature; macro states of Hopfield nets aren't balls rolling around in energy landscapes and state spaces of

Hopfield nets are neither one-dimensional nor continuous. Be that as it may, what the figure shows is a state space in which there are three points where the energy assumes a locally minimal value with one of these values being furthermore globally minimal. We therefore say that the energy function has two local minima and a global minimum and we recall that the goal of an energy minimization process would be to identify or find the global minimum. However, the figure also shows that the current state of the system coincides with a point of mere locally minimal energy or, put differently, that the system is currently at a local minimum of the energy function.

Now, if a system with a continuous state space such as in the figure uses local gradients to search for global minima of an objective, it cannot escape from a local minimum because there the gradient will be zero and therefore not allow the system to improve its state. For a system with a discrete state space such as a Hopfield net, the gradient at a local minimum may not equal zero but will typically not be strong enough to allow for improvements. Also, even if both kinds of systems had the capability to randomly step away from a local minimum to explore other points in its vicinity, their general philosophy of "going downhill" would cause them to return to the local minim shortly after. Minimization by means of local updates due to local information such as local derivatives or differences may thus indeed get trapped in sub-optimal states.

But is this really bad? Well, reconsider, for instance, our investigation of steepest energy descent Hopfield nets for SSP solving. Recall that we performed two experiments with 100 runs in which our network found 3 and 14 solutions, respectively. While we celebrated these performances as a success, we could have also considered them a failure because 97 and 86 out of 100 runs did not produce a solution. In general, say, when dealing with much larger and much more difficult problems, results of local searches can be even more disappointing. So, yes, in the grand scheme of things convergence to local minima is usually a bad outcome.

On the other hand, if a system or procedure for minimizing an objective had a more global view, it could "know" that "going downhill" is not always the best option. For instance, Fig. 7.1 also indicates that if the system in this example was capable of stepping into a direction of increasing energy, it could first climb hills and later descend into deeper valleys. Of course there are numerous ideas for how to realize such capabilities. Often, these take inspiration from biology such as in the case of genetic algorithms or ant algorithms. However, especially in the context of Hopfield nets, there also exist physics informed approaches and we will study those in the next chapter.

## 7.4 Exercises

**7.1** In graph theory, the *hop count distance* $d(v_j, v_k)$ between any two vertices $v_j$ and $v_k$ of a graph corresponds to the minimum number of steps or edge traversals required to get from $v_j$ to $v_k$. What is the maximum hop count distance between any two vertices in the Boolean lattice graph representing the state space of a Hopfield net of $n$ neurons? In other words, what is the hop count distance between two vertices in a Boolean lattice graph that are as far away from each other as possible?

**7.2** Consider a Hopfield net of $n$ neurons that is in some initial state $\boldsymbol{s}_0$ and runs the steepest energy descent procedure in Algorithm 1. Under these conditions, what is the maximum number of steps or iterations for network to reach a stable state $\boldsymbol{s}$?

**7.3** Consider a Hopfield net of $n$ neurons and give big $O$ expressions for the running time complexity of computing (a) the vector $-\nabla E(\boldsymbol{s})$ in (7.5), (b) the vector $\Delta \boldsymbol{E}$ in (7.22), and (c) the set $\mathcal{I}$ in (7.24).

**7.4** Consider a Hopfield net of $n$ neurons and give a big $O$ expression for the running time which the steepest energy descent procedure in Algorithm 1 requires to converge to a stable state.

**7.5** Asynchronously run a Hopfield net to solve the minimum size SSP in (5.27). Be bold and create your own instance of fairly large SSP for experimentation. Experiment with different kinds of neuron selection mechanisms (round robin, uniformly random, gradient informed) and ponder what you observe.

**7.6** Proceed as in the previous exercise but consider the size restricted SSP in (5.32).

**7.7** Create your own instance of linear regression problems and run Hopfield nets to solve the corresponding QUBO model in (5.84).

## References

1. Petersen, K., Pedersen, M.: The Matrix Cookbook. Technical University of Denmark (2012)
2. von Rueden, L., et al.: Informed Machine Learning – A Taxonomy and Survey of Integrating Prior Knowledge into Learning Systems. IEEE Trans. on Knowledge and Data Engineering **35**(1) (2023). https://doi.org/10.1109/TKDE.2021.3079836
3. Schulz, D., Bauckhage, C. (eds.): Informed Machine Learning. Springer (2025)
4. Bauckhage, C., Sanchez, R., Sifa, R.: Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In: Proc. Int. Joint Conf. on Neural Networks. IEEE (2020). https://doi.org/10.1109/IJCNN48605.2020.9206916
5. Bauckhage, C., et al.: Ising Models for Binary Clustering via Adiabatic Quantum Computing. In: Proc. Int. Conf. on Energy Minimization Methods in Computer Vision and Pattern Recognition, *LNCS*, vol. 10746. Springer (2017). https://doi.org/10.1007/978-3-319-78199-0_1
6. Bauckhage, C., Ramamurthy, R., Sifa, R.: Hopfield Networks for Vector Quantization. In: I. Farkas, P. Masulli, S. Wermter (eds.) Artificial Neural Networks and Machine Learning, *LNCS*, vol. 12397. Springer (2020). https://doi.org/10.1007/978-3-030-61616-8_16
7. Bauckhage, C., Sifa, R., Wrobel, S.: Adiabatic Quantum Computing for Max-Sum Diversification. In: Proc. Int. Conf. on Data Mining. SIAM (2020). https://doi.org/10.1137/1.9781611976236.39
8. Biesner, D., et al.: Solving Subset Sum Problems Using Quantum Inspired Optimization Algorithms with Applications in Auditing and Financial Data Analysis. In: Proc. Int. Conf. Machine Learning and Applications. IEEE (2022). https://doi.org/10.1109/ICMLA55696.2022.00150

# Chapter 8
# Hopfield Nets and Statistical Mechanics

## 8.1 Introduction

In our introduction to Chap. 6, we briefly explained why one half of the 2024 Nobel prize in physics was awarded to John Hopfield. In this chapter, we now address the second half of said prize.

It was awarded to Geoffrey Hinton who is most famous for being one of the popularizers of the error backpropagation algorithm for neural network training [1]. Yet, the Nobel committee cited his role in extending Hopfield nets to what he and his coworkers called Boltzmann machines [2–4] as well as his contributions to the development of restricted Boltzmann machines [5]. Now, those with a background in physics will know that Ludwig Boltzmann was an Austrian physicist who pioneered the field of statistical mechanics or, more general, statistical physics. The fact that his name features prominently in the names of the machines Hinton worked on therefore suggests that these involve tools from statistical physics. Indeed they do, and Hinton was awarded the Nobel prize for introducing these tools to machine learning.

In this chapter, we will apply some of those tools to Hopfield nets. The theme will be to understand a Hopfield net as an ensemble of interacting microscopic entities and to apply statistical techniques to reason about their collective behavior or, equivalently, about the ensemble's macroscopic state.

To segue into this study, we first address the open question of why we may talk about the *energy* of a Hopfield net and dicuss *Ising models* of *spin systems* [6], their energies, and relations to Hopfield nets. Given this foundation, we then look at *stochastic Hopfield nets* or *Boltzmann machines* with asynchronous *stochastic updates*. These still realize local searches as studied in the previous chapters but allow for extensions towards more global perspectives. Two prominent techniques in this regard are *stochastic simulated annealing* and *mean field annealing* and we finally study those to learn about completely different ways of running Hopfield nets.

C. Bauckhage and R. Sifa, *Quantum Computing from Hopfield Nets*,
Cognitive Technologies, https://doi.org/10.1007/978-3-031-99402-9_8

Overall, this chapter is a transitional chapter. Its main purpose is to build on what we already know to introduce jargon and ideas from physics which will feature prominently later on.

## 8.2 Ising Models

In the previous chapters, we saw that the energy

$$E(\boldsymbol{s}) = -\tfrac{1}{2}\,\boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s} \tag{8.1}$$

of a Hopfield net $(\boldsymbol{W}, \boldsymbol{b}) \in \mathbb{R}^{n \times n} \times \mathbb{R}^n$ in a macro state $\boldsymbol{s} \in \{\pm 1\}^n$ plays a crucial role in theory and practice of Hopfield nets. While we did not yet explain why the expression in (8.1) is called an *energy*, in Chap. 4, we already alluded to its connection to *Ising models* of *spin systems* studied in physics. Next, we will discuss this connection in detail. Since we postponed this discussion until now, it is clear that it will not be essential for our understanding of Hopfield nets. However, it will allow us to expand our point of view and to begin to get used the jargon of physics. This, in turn, will help us to prepare ourselves for the remainders of this chapter and this book.

In its simplest form, an **Ising model** is a $d$-way array of magnetically coupled dipole moments which arise due to the **spin** of elementary particles. Regardless of the kind of particle under consideration, spins will be found in either one of two states which are commonly called *up* ($\uparrow$) and *down* ($\downarrow$) and typically represented in terms of the bipolar numbers $+1$ and $-1$, respectively.

Theoretically, the $d$ in $d$-way array can be any natural number greater then zero. Yet, when modeling physical phenomena in our three-dimensional world, we usually work with $d \in \{1, 2, 3\}$ and all our upcoming examples will consider 2-way arrays. Why? Simply because these are arguably more interesting than 1-way arrays and easier to visualize than 3-way arrays.

Given what we said so far, we may also conceptualize a 2-way Ising model as an undirected graph whose vertices reside on a grid of $R$ rows and $C$ columns. Such a **grid graph** has $n = R \cdot C$ vertices which we may enumerate as $s_1, s_2, \ldots, s_n$. Moreover, each vertex represents a spin variable which takes values in $\{\pm 1\}$ according to some physical process. Just as in the case of Hopfield nets, we may therefore think of an Ising model as an ensemble of interacting bipolar micro states $s_j$ whose macro state can be represented as a bipolar vector $\boldsymbol{s} = [s_1, s_2, \ldots, s_n]^\intercal \in \{\pm 1\}^n$. The following figure illustrates all these ideas.

Looking at Fig. 8.1 it is clear that Ising models and Hopfield nets are closely related concepts. However, whereas the neurons of a Hopfield net can be arbitrarily interconnected, couplings between spins in an Ising model are more restricted. Except for the elements on the boundaries, each $s_j$ has an *immediate* neighbor to the top, to the right, to the bottom, and to the left.

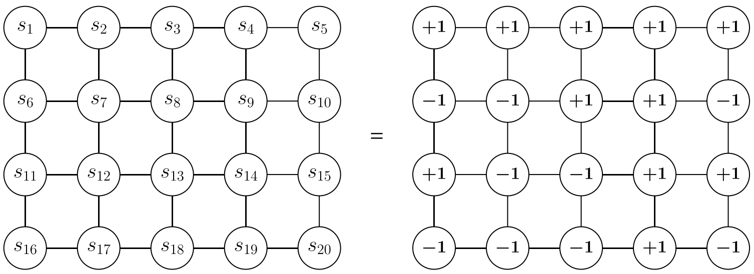

$s_1$ $s_2$ $s_3$ $s_4$ $s_5$
$s_6$ $s_7$ $s_8$ $s_9$ $s_{10}$
$s_{11}$ $s_{12}$ $s_{13}$ $s_{14}$ $s_{15}$
$s_{16}$ $s_{17}$ $s_{18}$ $s_{19}$ $s_{20}$
=
+1 +1 +1 +1 +1
−1 −1 +1 +1 −1
+1 −1 −1 +1 +1
−1 −1 −1 +1 −1

**Fig. 8.1** Visualization of a 2-way or rectangular Ising model with $n = 20$ spin variables $s_j \in \{\pm 1\}$

Now, the physical phenomenon an Ising model intends to describe is as follows: If a certain spin is in a certain state, then it is *energetically favorable* for its immediate neighbors to be in this state, too. The overall energy of a spin system will thus be minimal if all its micro states are identically $+1$ or $-1$ such that its macro state is either $\boldsymbol{s} = +\mathbf{1}$ or $\boldsymbol{s} = -\mathbf{1}$. For every other macro state, the system's energy will be higher and it will be maximal if no two immediate neighbors have the same spin. This behavior of the system's energy is captured by the following expression

$$E(\boldsymbol{s}) = -\tfrac{1}{2} \sum_j \sum_k w_{jk}\, s_j\, s_k = -\tfrac{1}{2}\, \boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} \tag{8.2}$$

where $\boldsymbol{W}$ is a square matrix over $\{0, w\}$ for some constant $w > 0$ whose entries are given by

$$w_{jk} = \begin{cases} w & \text{if } s_j \text{ and } s_k \text{ are coupled} \\ 0 & \text{otherwise .} \end{cases} \tag{8.3}$$

Looking at this definition of the coupling matrix $\boldsymbol{W}$, we remark that we will generally have $\boldsymbol{W} = w\boldsymbol{A}$ where $\boldsymbol{A}$ is the adjacency matrix of a $d$-dimensional grid graph; our example in Fig. 8.1 is but a specific instance of the case with $d = 2$. This particular structure of the coupling matrix creates another close connection to Hopfield nets: Being a scaled adjacency matrix of an undirected graph without self-loops, $\boldsymbol{W}$ is necessarily symmetric $\boldsymbol{W} = \boldsymbol{W}^\intercal$ and hollow $\operatorname{diag}[\boldsymbol{W}] = \mathbf{0}$.

So far, the energy in Eq. (8.2) only represents the energy of an isolated spin system as it only sums contributions due to internal interactions. However, in the real world, individual spins in a spin system also experience external influences caused by magnetic fields. The equation for the energy of an Ising model thus needs to be extended to account for effects of surrounding fields. Due to the simplicity of Ising models, a corresponding generalization of the energy simply reads

$$E(\boldsymbol{s}) = -\tfrac{1}{2} \sum_j \sum_k w_{jk}\, s_j\, s_k + \sum_j b_j\, s_j = -\tfrac{1}{2}\, \boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s}\ . \tag{8.4}$$

This expression is now structurally equivalent to what we defined as the energy of a Hopfield net. And, since this expression followed from basic physical considerations about energetic properties of certain physical systems, we finally have justification to talk about energies of Hopfield nets.

To wrap up this first dive into physics, we emphasize that Ising models are really but models. They provide a simplified description of very complex physical systems. For instance, they assume interactions among spins to be local and to only happen among immediate neighbors. They do not account for long range interactions and also assume all local interactions to be of the same strength. Their coupling matrices $\boldsymbol{W} = w\boldsymbol{A}$ thus have fewer degrees of freedom than weight matrices of Hopfield nets. In this sense, Hopfield nets are (slight) generalizations of Ising models which is interesting because many of the mathematical tools that have been developed to analyze the characteristics and behaviors of Ising models also apply to Hopfield nets. Finally, we remark that, despite their simplicity or many simplifying assumptions, Ising models nevertheless allow for the study of a wide range of complex phenomena arising in systems of interacting components. A beautiful account of what is possible in this regard can be found in [7, Chap. 31].

## 8.3 Boltzmann Machines

Next, we will assume a different point of view on Hopfield nets and consider tools from the venerable field of statistical physics to analyze their behavior. To this end, we will continue drawing physical analogies to, say, chunks of ferromagnetic material whose macroscopic magnetic moment arises from the interactions of numerous microscopic magnetic dipoles. Since it would take us way too far afield if we derived the corresponding physical models from scratch, we will consider them axiomatic and simply declare that:

At *thermodynamic temperature* $T$, the *probability* of observing a Hopfield net in a specific macro state $\boldsymbol{s}$ is given by the **Boltzmann distribution**

$$p(\boldsymbol{s}) = \frac{1}{Z} \cdot e^{-\frac{1}{k_B T} E(\boldsymbol{s})} \tag{8.5}$$

where $k_B$ is the *Boltzmann constant* and physicists refer to the normalization constant

$$Z = \sum_{\boldsymbol{s}'} e^{-\frac{1}{k_B T} E(\boldsymbol{s}')} \tag{8.6}$$

as the **partition function**.

To compute the latter for a Hopfield net of $n$ neurons, we would have to sum over all its $2^n$ possible macro states and we know tha this is practically infeasible for large $n$. However, for the results we derive below, this is no reason for concern since we only work with expressions where any occurrences of $Z$ will cancel out.

When expressed in Joule per Kelvin, the value of **Boltzmann constant** is given by $k_B = 1.380649 \times 10^{-23}$. However, as computer scientists, we need not worry about its physical significance and may just as well drop it from our considerations.

**Notation**

For simplicity, we henceforth work with

$$k_B \equiv 1\ . \tag{8.7}$$

The notion of the **thermodynamic temperature** $T$ can be understood as follows: *Thermal fluctuations* of a system are random deviations from the system's *average state at equilibrium*. At high temperatures $T$, such thermal fluctuations are larger and more likely. In the context of Hopfield nets, this means that, at high temperatures, neurons may likely and spontaneously switch their micro states so that the macro state of the network is less deterministic than at low temperatures.

In order to reduce notational clutter, we henceforth work with the reciprocal of the thermodynamic temperature, namely the **thermodynamic beta**

$$\beta \equiv \frac{1}{T}\ . \tag{8.8}$$

Together with $k_B = 1$, the Boltzmann distribution in Eq. (8.5) will then read

$$p(\boldsymbol{s}) = \frac{1}{Z} \cdot e^{-\beta\, E(\boldsymbol{s})} \tag{8.9}$$

and we point out that this joint probability of $n$ random variables can generally be factored as

$$p(\boldsymbol{s}) = p(s_1, \ldots, s_j, \ldots, s_n) = p(s_j \mid s_{k \neq j}) \cdot p(s_{k \neq j}) \tag{8.10}$$

Given these preparatory remarks, we now consider a Hopfield net that has been running for a while and thus has reached a certain macro state $\boldsymbol{s}$. With respect to this state, we are interested in the conditional probability $p(s_j = +1 \mid s_{k \neq j})$.

To determine this probability, we observe the following: If the micro states of all $s_{k \neq j}$ are held fixed and only $s_j$ is allowed to update, there are two possible resulting macro states, namely

$$\boldsymbol{s}_+ = \left[s_1, \ldots, s_j = +1, \ldots, s_n\right]^\intercal \tag{8.11}$$

$$\boldsymbol{s}_- = \left[s_1, \ldots, s_j = -1, \ldots, s_n\right]^\intercal\ . \tag{8.12}$$

Given these macro states, we can compute the *log-odds* for $s_j$ having a value of $+1$. These odds are given by

$$\ln \frac{p(\boldsymbol{s}_+)}{p(\boldsymbol{s}_-)} = \ln \frac{1}{Z} - \beta\, E\big(\boldsymbol{s}_+\big) - \ln \frac{1}{Z} + \beta\, E\big(\boldsymbol{s}_-\big) \tag{8.13}$$

$$= \beta \Big( E\big(\boldsymbol{s}_-\big) - E\big(\boldsymbol{s}_+\big) \Big) \tag{8.14}$$

$$\equiv \beta\, \Delta E\ . \tag{8.15}$$

Using the factorization in Eq. (8.10), we can also consider an alternative, slightly more involved but altogether equivalent expression for these odds, namely

$$\ln \frac{p(\boldsymbol{s}_+)}{p(\boldsymbol{s}_-)} = \ln \frac{p(s_j = +1 \mid s_{k \neq j}) \cdot p(s_{k \neq j})}{p(s_j = -1 \mid s_{k \neq j}) \cdot p(s_{k \neq j})} \tag{8.16}$$

$$= \ln \frac{p(s_j = +1 \mid s_{k \neq j})}{p(s_j = -1 \mid s_{k \neq j})} \tag{8.17}$$

$$= \ln\, p(s_j = +1 \mid s_{k \neq j}) - \ln\, p(s_j = -1 \mid s_{k \neq j}) \tag{8.18}$$

$$= \ln\, p(s_j = +1 \mid s_{k \neq j}) - \ln \Big(1 - p(s_j = +1 \mid s_{k \neq j})\Big) \tag{8.19}$$

and we encourage the reader to verify that the last step does indeed make sense.

Now that we have two expression for the same quantity, we can hope to gain deeper insights from equating them. Indeed, if we equate (8.15) and (8.19), we find

$$\ln\, p(s_j = +1 \mid s_{k \neq j}) - \ln \Big(1 - p(s_j = +1 \mid s_{k \neq j})\Big) = \beta\, \Delta E \tag{8.20}$$

and an exponentiation of both sides of this equation provides us with

$$\frac{p(s_j = +1 \mid s_{k \neq j})}{1 - p(s_j = +1 \mid s_{k \neq j})} = e^{\beta\, \Delta E}\ . \tag{8.21}$$

In order to solve this expression for the sought after probability $p(s_j = +1 \mid s_{k \neq j})$, we note that (8.21) is equivalent to

$$p(s_j = +1 \mid s_{k \neq j}) = e^{\beta\, \Delta E} - e^{\beta\, \Delta E}\, p(s_j = +1 \mid s_{k \neq j}) \tag{8.22}$$

which, in turn, is equivalent to

$$p(s_j = +1 \mid s_{k \neq j}) + e^{\beta\, \Delta E}\, p(s_j = +1 \mid s_{k \neq j}) = e^{\beta\, \Delta E} \tag{8.23}$$

so that we find

$$p(s_j = +1 \mid s_{k \neq j}) \cdot \big(1 + e^{\beta\, \Delta E}\big) = e^{\beta\, \Delta E} \tag{8.24}$$

and therefore

$$p(s_j = +1 \mid s_{k \neq j}) = \frac{e^{\beta\,\Delta E}}{1 + e^{\beta\,\Delta E}}\ . \tag{8.25}$$

To further consolidate this result, we multiply the probability in (8.25) with a rather intricate version of the number 1, namely

$$1 = \frac{e^{-\beta\,\Delta E}}{e^{-\beta\,\Delta E}}\ . \tag{8.26}$$

This finally leads to

$$\frac{e^{\beta\,\Delta E}}{1 + e^{\beta\,\Delta E}} \cdot \frac{e^{-\beta\,\Delta E}}{e^{-\beta\,\Delta E}} = \frac{1}{e^{-\beta\,\Delta E} + 1} = \frac{1}{1 + e^{-\beta\,\Delta E}}\ . \tag{8.27}$$

All in all, we have therefore established the following quite remarkable and truly crucial result.

**Theorem 8.1** *At any given thermodynamic temperature $T = \frac{1}{\beta}$, the probability*

$$p(s_j = +1 \mid s_{k \neq j}) = \frac{1}{1 + e^{-\beta\,\Delta E}} \tag{8.28}$$

*of a Hopfield neuron $s_j$ being active can be computed from a* ***logistic function*** *of the scaled energy difference $\beta\,\Delta E$ between its inactive and active state.*

Next, let us reconsider the energy difference $\Delta E = E\big(\boldsymbol{s}_-\big) - E\big(\boldsymbol{s}_+\big)$ based on what we know from Chap. 6. From what we saw there, we know that we can expand the expression for the energy of a Hopfield net in state $\boldsymbol{s}$ like this

$$E(\boldsymbol{s}) = -\tfrac{1}{2} \sum_{k \neq j} \sum_{l \neq j} w_{kl}\, s_k\, s_l - \sum_{k \neq j} w_{jk}\, s_j\, s_k + \sum_{k \neq j} b_k\, s_k + b_j\, s_j \tag{8.29}$$

For the two specific states $\boldsymbol{s}_+$ and $\boldsymbol{s}_-$ with $s_j = +1$ and $s_j = -1$, respectively, we therefore have

$$E(\boldsymbol{s}_+) = -\tfrac{1}{2} \sum_{k \neq j} \sum_{l \neq j} w_{kl}\, s_k\, s_l - \sum_{k \neq j} w_{jk}\, s_k + b_j \tag{8.30}$$

$$E(\boldsymbol{s}_-) = -\tfrac{1}{2} \sum_{k \neq j} \sum_{l \neq j} w_{kl}\, s_k\, s_l + \sum_{k \neq j} w_{jk}\, s_k - b_j\ . \tag{8.31}$$

But this means that the energy difference between the states where $s_j$ is inactive and where $s_j$ is active can be expressed as

$$\Delta E = E(\boldsymbol{s}_-) - E(\boldsymbol{s}_+) = 2 \left( \sum_{k \neq j} w_{jk}\, s_k - b_j \right) . \tag{8.32}$$

Looking at this expression, we once again note that our way of summing over $k \neq j$ is good bookkeeping but unnecessarily cumbersome since $w_{jj} = 0$. Using this, we may therefore just as well write

$$\Delta E = 2 \left( \sum_{k} w_{jk}\, s_k - b_j \right) = 2 \left( \boldsymbol{w}_j^\intercal \boldsymbol{s} - b_j \right) . \tag{8.33}$$

It is therefore surprisingly easy to practically compute the probability in (8.28). Given a thermodynamic temperature $T = 1/\beta$, the weights $\boldsymbol{W}$, the biases $\boldsymbol{b}$, and the current macro state $\boldsymbol{s}$ of a Hopfield net, the right hand side of (8.28) can be readily evaluated.

Based on this observation, we now introduce the crucial idea of asynchronous *stochastic* updates of a neuron $s_j$ of a Hopfield net. In this framework, we first compute the probability $p(s_j = +1 \mid s_{k \neq j})$ which we suspiciously abbreviate as $\mu_j$. That is, we compute

$$\mu_j = \frac{1}{1 + e^{-\frac{2}{T}(\boldsymbol{w}_j^\intercal \boldsymbol{s} - b_j)}} \tag{8.34}$$

for which we are guaranteed that it is a number between 0 and 1. We may therefore treat it as the success parameter of a Bernoulli distribution over a binary random variable and sample this distribution

$$z_j \sim f_{\text{Ber}}(z \mid \mu_j) \tag{8.35}$$

Finally, we can turn this sampled binary number into a bipolar number

$$s_j = 2 \cdot z_j - 1 \tag{8.36}$$

and thus obtain a micro state update of the neuron under consideration.

A Hopfield net whose neurons update like this is a called a *stochastic Hopfield net* or a **Boltzmann machine** [2–4] and there would be much more to study. However, making a long story short, we simply mention the following essential results.

In contrast to the asynchronous updates of a deterministic Hopfield net, the above asynchronous updates of a stochastic Hopfield net may cause its energy to increase. However, this may be beneficial as it can allow the network to escape from local minima in the energy landscape.

At (very) high thermodynamic temperatures $T \gg 0$, we have $\mu_j \approx 1/2$ regardless of the value of the energy difference $\Delta E$. At high temperatures, stochastic updates of a neuron will therefore equally likely increase or decrease the network's energy.

At lower temperatures, on the other hand, the value of $\Delta E$ will matter. Recall that we understand it as the difference between the energies of states $\boldsymbol{s}_-$ and $\boldsymbol{s}_+$ and thus as the decrease or increase in energy due to switching micro state $s_j$ from $+1$ to $-1$. If such a switch would decrease the overall energy, then $\Delta E \leq 0$. However, if deactivating $s_j$ is energetically favorable, then the probability $\mu_j = p(s_j = +1 \mid s_{k \neq j})$ should be small. And indeed, for cases where $\Delta E \leq 0$, the exponent of $e$ in (8.34) becomes positive so that $\mu_j$ becomes small. For lower and lower temperatures, it is thus less and less likely that stochastic updates increase the network's energy.

In fact, one can show that, in the limit $T \to 0$, stochastic Hopfield nets become deterministic Hopfield nets whose neurons update by evaluating $\text{sign}(\boldsymbol{w}_j^\intercal \boldsymbol{s} - b_j)$.

Moreover, one can show that, after many asynchronous stochastic updates, the probability for a stochastic Hopfield net to be in a specific macro state $\boldsymbol{s}$ follows the Boltzmann distribution in (8.5) and that this holds irrespective of the value of $T$ and of the macro state the network started in. The corresponding likelihood

$$\ln p(\boldsymbol{s}) \propto -\tfrac{1}{T}\, E(\boldsymbol{s}) \tag{8.37}$$

is therefore always proportional to the energy of the current macro state.

This is interesting, because, for two different states $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ with $E(\boldsymbol{s}_1) \leq E(\boldsymbol{s}_2)$, this implies $\ln p(\boldsymbol{s}_1) \geq \ln p(\boldsymbol{s}_2)$ which is to say that low energy states are more likely.

Note that the likelihood in (8.37) also depends on the thermodynamic temperature $T$. If we thus consider a state of low energy $E(\boldsymbol{s}) < 0$ and two temperatures $T_1 < T_2$, we find $\ln p(\boldsymbol{s} \mid T_1) > \ln p(\boldsymbol{s} \mid T_2)$. For low temperatures, it therefore appears even more likely to find the network in a low energy state. Recall, however, that, at low temperatures, stochastic updates mainly tend to decrease energies. For a stochastic Hopfield net running at low temperatures, there is thus a certain risk that it will get trapped in local energy minima so that low temperatures per se are not necessarily advantageous.

All these observations have led researchers to consider even more sophisticated ideas for how to run Hopfield nets. A very prominent idea in this regard is to work with a dynamic called **simulated annealing** and we will discuss it next.

## 8.4 Simulated Annealing

The term *simulated annealing* alludes to the process of annealing in metallurgy which is a technique that involves repeated heating and controlled cooling of a material to increase the size of its constituent crystals and reduce potential defects. While there are several independent sources which adapted this mechanism to computational physics or chemistry, the term simulated annealing itself was famously coined by Kirkpatrick, Gelatt, and Vecchi who rigorously pioneered the use of statistical physics methods in general optimization and problem solving [8].

**Algorithm 1** Stochastic simulated annealing of a Hopfield net

```
Input: Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ and a random initial state $\boldsymbol{s} = \big[s_1, s_2, \ldots, s_n\big]^\intercal$

for $t = 0$ to $t_{\max}$
  $T = \Big(1 - \frac{t}{t_{\max}}\Big) T_{\text{high}} + \frac{t}{t_{\max}} T_{\text{low}}$
  for $r = 1$ to $r_{\max}$
    for $j = 1$ to $n$
      $\mu_j = \Big(1 + e^{-\frac{2}{T}\left(\boldsymbol{w}_j^\intercal \boldsymbol{s} - b_j\right)}\Big)^{-1}$
      $s_j \sim 2 \cdot f_{\text{Ber}}(z \mid \mu_j) - 1$
```

In the context of Boltzmann machines, simulated annealing was first considered shortly after they had been introduced [9, 10]. Here, the annealing processes are typically realized as follows.

Consider a random initial macro state $\boldsymbol{s}$ and a fairly high predefined temperature $T_{\text{high}}$. At this temperature, consider a fairly large number $r_{\max}$ of updating rounds (e.g. at least twice as large as the number of neurons of the network under consideration). In each round, asynchronously (re)compute the states $s_j$ of all neurons according to the stochastic update mechanism in (8.34)–(8.36). Then, (slightly) lower the temperature and repeat until a predefined temperature $T_{\text{low}}$ is reached.

The exact strategy for how to cool the thermodynamic temperature $T$ from $T_{\text{high}}$ down to $T_{\text{low}}$ is known as an **annealing schedule** and there exists a wide spectrum of possibilities. Many of these consider a sequence of time steps $0 \leq t \leq t_{\max}$ and define $T$ as a function of $t$. A comparatively simple example is this *linear* cooling schedule

$$T(t) = \Big(1 - \frac{t}{t_{\max}}\Big) T_{\text{high}} + \frac{t}{t_{\max}} T_{\text{low}} \tag{8.38}$$

but *logarithmic*, *exponential*, or *informed* schedules are common choices, too.

As a whole, the above procedure is called **stochastic simulated annealing** and the pseudo code in Algorithm 1 summarizes its essentials. Note that this pseudo code assumes a linear cooling schedule and a round robin selection of the updating neuron. These are, however, not the only possibilities; other (appropriate) mechanisms would work as well.

An interpretation of the behavior of this algorithm as a way of running a Hopfield net is as follows: At the initially high temperatures, the neurons of the network flip more or less randomly which allows the network to *explore* its whole state space. If this exploration runs for a long enough time, the network will likely approach a macro state in the vicinity of a (global) energy minimum. Gradually reducing the temperature will cause the network to explore less and less but to begin to *descend* down the energy landscape. This way, it may converge to a Boltzmann distribution where the network's energy fluctuates around the global minimum.

## 8.5 Mean Field Annealing

Let us take a step back and reconsider what we are doing when we use asynchronous stochastic updates or stochastic simulated annealing to run a Hopfield net. Since both these approaches involve the following computational steps

$$\mu_j = \left(1 + e^{-\frac{2}{T}\left(\boldsymbol{w}_j^\intercal \boldsymbol{s} - b_j\right)}\right)^{-1} \tag{8.39}$$

$$s_j \sim 2 \cdot f_{\text{Ber}}(z \mid \mu_j) - 1 \ , \tag{8.40}$$

we realize that both these approaches treat the states of the neurons of a Hopfield net as bipolar *random variables*.

In other words, both these approaches implicitly see a Hopfield net as a dynamic ensemble of random microscopic entities whose macroscopic state is thus a random entity as well. A core idea of statistical physics is to abstract away from the often intractable exact details of the behavior of such random ensembles but to consider their average behavior instead. Next, we will briefly study how such a statistical point of view allows for the design of yet another algorithm for running Hopfield nets.

To begin with, we recall that Theorem 8.1 establishes the conditional probability

$$p(s_j = +1 \mid s_{k \neq j}) = \frac{1}{1 + e^{-\beta\,\Delta E}} \tag{8.41}$$

for a Hopfield neuron $s_j$ to be in state $+1$ given the current states of all the other neurons in the network. But what about the conditional *expected value*

$$\mathbb{E}\big[s_j \bigm| s_{k \neq j}\big] \tag{8.42}$$

of the state of neuron $s_j$?

Well, since $s_j$ is a bipolar random variable and can therefore only take the two values $+1$ and $-1$, its expectation value is easily computed

$$\begin{aligned}
\mathbb{E}\big[s_j \bigm| s_{k \neq j}\big] &= (+1) \cdot p(s_j = +1 \mid s_{k \neq j}) + (-1) \cdot p(s_j = -1 \mid s_{k \neq j}) && (8.43)\\
&= p(s_j = +1 \mid s_{k \neq j}) - \big(1 - p(s_j = +1 \mid s_{k \neq j})\big) && (8.44)\\
&= 2 \cdot p(s_j = +1 \mid s_{k \neq j}) - 1 && (8.45)\\
&= \frac{2}{1 + e^{-\beta\,\Delta E}} - 1 \ . && (8.46)
\end{aligned}$$

So far so good, but what if we were to consider

$$\mathbb{E}\big[s_j \bigm| \mathbb{E}[s_{k \neq j}]\big] \equiv \langle s_j \rangle \tag{8.47}$$

where the expectation of $s_j$ is not conditioned on some current values of the $s_{k \neq j}$ but on their expected values? Well, in this so called *mean field approximation* of the states of the neurons of a Hopfield net, we would have

$$\langle s_j \rangle = \frac{2}{1 + e^{-\beta \langle \Delta E \rangle}} - 1 \tag{8.48}$$

where $\langle \Delta E \rangle$ denotes the expected difference in the energy of a Hopfield net if neuron $s_j$ would switch from inactive to active. In order to compute it, we simply have to compute the expectation of the right hand side of (8.33), namely

$$\langle \Delta E \rangle = 2 \left\langle \sum_k w_{jk} \, s_k - b_j \right\rangle = 2 \left( \sum_k w_{jk} \, \langle s_k \rangle - b_j \right) = 2 \left( \boldsymbol{w}_j^{\mathsf{T}} \langle \boldsymbol{s} \rangle - b_j \right) . \tag{8.49}$$

Using this expression, we can therefore write

$$\langle s_j \rangle = 2 \cdot \left( 1 + e^{-2 \beta \left( \boldsymbol{w}_j^{\mathsf{T}} \langle \boldsymbol{s} \rangle - b_j \right)} \right)^{-1} - 1 \tag{8.50}$$

and emphasize the following: At thermodynamic temperature $T = 1/\beta$, the mean or expected value $\langle s_j \rangle$ of the micro state of a neuron in a Hopfield net depends on the mean or expected value $\langle \boldsymbol{s} \rangle$ of the network's macro state. The following theorem concisely summarizes all of this.

**Theorem 8.2** *At any given thermodynamic temperature* $T = \frac{1}{\beta}$*, the expected value* $-1 \leq \langle s_j \rangle \leq +1$ *of the micro state of a Hopfield neuron* $s_j$ *is given by*

$$\langle s_j \rangle = \frac{2}{1 + e^{-2 \beta \left( \boldsymbol{w}_j^{\mathsf{T}} \langle \boldsymbol{s} \rangle - b_j \right)}} - 1 \tag{8.51}$$

*where* $\langle \boldsymbol{s} \rangle$ *denotes the expected value of the network's macro state.*

But can this statistical view where we look at averages of random variables rather than at specific instantiations of random variables be put to work? Yes it can! It allows for running Hopfield nets via **mean field annealing** [11, 12] which can be seen as a deterministic approximation to stochastic simulated annealing. The method is not at all restricted to Hopfield nets but, for those, it usually proceeds as follows.

Consider an initial random average macro state $\langle \boldsymbol{s} \rangle$ and a fairly high temperature $T_{\text{high}}$. At this temperature, consider a fairly large number $r_{\max}$ of asynchronous updating rounds. In each round, randomly pick a neuron and (re)compute its expected activation $\langle s_j \rangle$ according to (8.50). Then, lower the temperature and repeat until a predefined temperature $T_{\text{low}}$ is reached. Once this has happened, compute a state vector $\boldsymbol{s}$ whose entries are given by

$$s_j = \begin{cases} +1 & \text{if } \langle s_j \rangle \geq 0 \\ -1 & \text{otherwise} . \end{cases} \tag{8.52}$$

In terms of pseudo code, the whole procedure can be summarized as in Algorithm 2.

**Algorithm 2** Mean field annealing of a Hopfield net

```
Input: Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$

initialize $\langle \boldsymbol{s} \rangle$ with
    $\langle s_j \rangle \sim \mathcal{N}\big(x \mid \mu = 0, \sigma^2 = 1\big)$
for $t = 0$ to $t_{\max}$
    $T = \Big(1 - \frac{t}{t_{\max}}\Big) T_{\text{high}} + \frac{t}{t_{\max}} T_{\text{low}}$
    for $r = 1$ to $r_{\max}$
        $j \sim \mathcal{U}\big[\{1, 2, \ldots, n\}\big]$
        $\langle s_j \rangle = 2 \cdot \Big(1 + e^{-\frac{2}{T}\big(\boldsymbol{w}_j^\intercal \langle \boldsymbol{s} \rangle - b_j\big)}\Big)^{-1} - 1$
compute $\boldsymbol{s}$ with
    $s_j = \begin{cases} +1 & \text{if } \langle s_j \rangle \geq 0 \\ -1 & \text{otherwise} \end{cases}$
```

Looking at this pseudo code, we have several specific and general remarks. On the specific side, we note that we initialize the average macro state $\langle \boldsymbol{s} \rangle$ by sampling its entries $\langle s_j \rangle$ from a Gaussian distribution with zero mean and unit variance. This is a simple idea which usually works well in practice but other random initializations are possible, too. We also note that we are once again working with a simple linear annealing schedule but could as well consider other schemes.

On the general side, we observe that the algorithm involves elements of chance namely when sampling (the index of) the neuron whose expected activation should be (re)computed. This (re)computation, however, does not involve randomness. It is in this sense, that mean field annealing is a deterministic approximation to stochastic simulated annealing. In mean field annealing, entries $\langle s_j \rangle$ of $\langle \boldsymbol{s} \rangle$ are determined in a deterministic manner so that neither $\langle s_j \rangle$ nor $\langle \boldsymbol{s} \rangle$ are random variables. In stochastic simulated annealing, on the other hand, entries $s_j$ of $\boldsymbol{s}$ are randomly drawn from a bipolar Bernoulli distribution so that $s_j$ and $\boldsymbol{s}$ are both random variables.

However, at the initially high temperatures, the entries of $\langle \boldsymbol{s} \rangle$ will have higher variances than at later stages with lower temperatures. Mean field annealing therefore also realizes an exploration and a descent phase. Contrary to simulated annealing, both phases are less volatile or noisy which means that mean field annealing behaves less erratic and more robustly than stochastic simulated annealing. In practice, it usually works well since it tends to converge faster and to lower energy states than its stochastic counterpart.

To provide at least anecdotal evidence for these claims, we recall that challenging constraint satisfaction problems such as Sudoku puzzles can be formalized as QUBOs so that we may run Hopfield nets in an attempt to solve them [13–15]. However, it turns out that conventional local search mechanisms tend to fail miserably on all but the simplest instances [13] whereas simulated- and mean field annealing are capable of solving even hard Sudokus [14, 15]. Granted, these, too, may need several runs with different random initializations to find the correct solution. However, while both techniques perform surprisingly well in general, personal experience shows that mean field annealing usually needs fewer runs to succeed.

## 8.6 Exercises

**8.1** Empirically verify the expression in (8.2) as an appropriate model of the internal energy of Ising model. To this end, implement `python` / `numpy` code that initializes a 2-way Ising model of size $R \times C$ in the following states

$$\boldsymbol{s}_{+} = \left[+1, +1, +1, +1, +1, \ldots\right]^{\intercal} \tag{8.53}$$

$$\boldsymbol{s}_{-} = \left[-1, -1, -1, -1, -1, \ldots\right]^{\intercal} \tag{8.54}$$

$$\boldsymbol{s}_{\pm} = \left[+1, -1, +1, -1, +1, \ldots\right]^{\intercal} \tag{8.55}$$

$$\boldsymbol{s}_{\mp} = \left[-1, +1, -1, +1, -1, \ldots\right]^{\intercal} \tag{8.56}$$

and then computes the corresponding energies using $\boldsymbol{W} = \boldsymbol{A}$ where $\boldsymbol{A}$ is the adjacency matrix of the graph representation of the model.

**Hint:** Likely the main challenge of this task is to code the model's coupling matrix. However, it becomes less of an issue when working with `python`'s graph processing module `networkx`. It provides the functions `grid_graph` and `adjacency_matrix` which set up grid graphs and compute graph adjacency matrices. Carefully read the manuals to make sure you use them correctly.

**8.2** Theoretically verify that the expression in (8.2) is an appropriate model of the internal energy of an $R \times C$ Ising model. That is, prove that it is minimal if the system is maximally ordered (all spins are in the same state) and maximal if the system is maximally unordered (every two neighboring spins are in different states).

**Hint:** The proof follows from answering these questions: How many couplings $w_{jk}$ are there in an $R \times C$ Ising model? What is the value of the product $s_j s_k$ of two neighboring spins $s_j$ and $s_k$ if they are in the same state or if they are in different states?

**8.3** Give an expression for the expected energy $\mathbb{E}\big[E(\boldsymbol{s})\big]$ of a Hopfield net where $\boldsymbol{s}$ is distributed according to the Boltzmann distribution

$$p(\boldsymbol{s}) = \tfrac{1}{Z} \, e^{-\beta \, E(\boldsymbol{s})} \tag{8.57}$$

with

$$Z = \sum_{\boldsymbol{s}'} e^{-\beta \, E(\boldsymbol{s}')} \, . \tag{8.58}$$

**8.4** For $Z$ as above, compute the expression

$$\tfrac{\partial}{\partial \beta} \ln Z \tag{8.59}$$

and compare the result you obtain to what you computed in the previous exercise.

**8.5** For $p(\boldsymbol{s})$ as above, compute the expression

$$\tfrac{\partial}{\partial \boldsymbol{s}} \, p(\boldsymbol{s}) \, . \tag{8.60}$$

For which value(s) of $0 \leq p(\boldsymbol{s}) \leq$ will the magnitude of this gradient be minimal? For which value(s) will it be maximal?

**8.6** Implement a `python/numpy` function `hnetRunSSA` that realizes the stochastic simulated annealing procedure in Algorithm 1. Input parameters of this function should be the usual Hopfield net parameters `vecS`, `matW`, and `vecB` as well as two real numbers `Th` and `Tl` indicating the initial high- and final low temperature of the annealing process. Finally, there should two integers `tmax` and `rmax` representing the numbers of time steps and rounds per time step to be considered for the annealing process.

**Note:** The pseudo code in Algorithm 1 involves a simple linear cooling schedule

$$T = \left(1 - \tfrac{t}{t_{\max}}\right) T_{\text{high}} + \tfrac{t}{t_{\max}} \, T_{\text{low}} \tag{8.61}$$

which crucially assumes (integer) time steps such that $0 \leq t \leq t_{\max}$. However, if we work with `python` constructs such as `for t in range(tmax): ...`, then $t$ will be bounded like this $0 \leq t < t_{\max}$. Be aware of this issue and resolve it appropriately.

**8.7** Use your function `hnetRunSSA` to solve the 4-rooks problem. Experience shows that this should work well if `vecS` is randomly initialized and the annealing parameters are chosen to be `Th = 10.0`, `Tl = 0.5`, `tmax = 25`, and `rmax = 100`. However, also experiment with other parameter choices to gain experience with the behavior of simulated annealing for (simple) combinatorial problem solving.

**8.8** Implement a `python/numpy` function `hnetRunMFA` that realizes the mean field annealing procedure in Algorithm 2. It should take input parameters `matW`, `vecB`, `Th`, `Tl`, `tmax` and `rmax` and consider (integer) time steps where $0 \leq t \leq t_{\max}$.

**8.9** Use your function `hnetRunMFA` to solve the 4-rooks problem. Experience shows that good parameters are once again `Th=10.0`, `Tl=0.5`, `tmax=25`, and `rmax=100`. However, also experiment with other choices to explore the behavior of mean field annealing.

**8.10** Design and execute systematic experiments in which you compare the performances (running time, time to convergence, solution quality, ...) of your functions `hnetRunSSA` and `hnetRunMFA` on a plain vanilla subset sum problem.

## References

1. Rumelhart, D., Hinton, G., Williams, R.: Learning Representations by Back-Propagating Errors. Nature **323**(6088) (1986). https://doi.org/10.1038/323533a0
2. Fahlman, S., Hinton, G., Sejnowski, T.: Massively Parallel Architectures for AI: NETL, Thistle, and Boltzmann Machines. In: Proc. AAAI Conf. on Artificial Intelligence (1983)
3. Ackley, D., Hinton, G., Sejnowski, T.: A Learning Algorithm for Boltzmann Machines. Cognitive Science **9**(1) (1985). https://doi.org/10.1016/S0364-0213(85)80012-4
4. Hinton, G., Sejnowski, T.: Learning and Relearning in Boltzmann Machines. In: D. Rummelhart, J. McClelland (eds.) Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations. MIT Press (1986). https://doi.org/10.5555/104279.104291
5. Hinton, G.: Training Products of Experts by Minimizing Contrastive Divergence. Neural Computation **14**(8) (2002). https://doi.org/10.1162/089976602760128018
6. Ising, E.: Beitrag zur Theorie des Ferromagnetismus. Zeitschrift für Physik **31** (1925). https://doi.org/10.1007/BF02980577
7. MacKay, D.: Information Theory, Inference, and Learning Algorithms. Cambridge University Press (2003)
8. Kirkpatrick, S., Gelatt, C., Vecchi, M.: Optimization by Simulated Annealing. Science **220**(4598) (1983). https://doi.org/10.1126/science.220.4598.671
9. Feldman, J.: Energy and the Bahvior of Connectionist Models. Tech. Rep. 155, Computer Science Department at the University of Rochester (1985)
10. Touretzky, D., Hinton, G.: Symbols among the Neurons: Details of a Connectionist Inference Architecture. In: Proc. Int. Joint Conf. on Artificial Intelligence (1985). https://doi.org/10.5555/1625135.1625179
11. Peterson, C., Anderson, J.: A Mean Field Theory Learning Algorithm for Neural Networks. Complex Systems **1**(5) (1987). https://doi.org/10.1007/978-94-011-5014-9_20
12. Bilbro, G., et al.: Optimization by Mean Field Annealing. In: Proc. NIPS (1988)
13. Hopfield, J.: Searching for Memories, Sudoku, Implicit Check Bits, and the Iterative Use of Not-Always-Correct Rapid Neural Computation. Neural Computation **20**(5) (2008). https://doi.org/10.1162/neco.2007.09-06-345
14. Bauckhage, C., Beaumont, F., Müller, S.: Hopfield Nets for Sudoku. Tech. rep., Machine Learning Rhine Ruhr (2021). Available on researchgate.net
15. Mücke, S.: A Simple QUBO Formulation of Sudoku. In: Proc. Genetic and Evolutionary Computation Conf. (2024). https://doi.org/10.1145/3638530.3664106

# Chapter 9
# Hopfield Nets and Quantum Mechanics

## 9.1 Introduction

In this chapter, we will view Hopfield nets from a perspective that is likely alien to most computer scientists and machine learners. Physicists, on the other hand, might consider it to be quite natural.

To make a long story short, we will consider completely different representations of the states and energy functions of Hopfield nets and connect them to the kind of tensor product spaces we studied in Chap. 3. When working with digital computers, these representations are of little *practical* value as they do not provide us with a generally feasible alternative for Hopfield energy minimization. However, their *theoretical* significance must not be underestimated. Indeed, what we are going to do is to have a first look at mathematical objects or computational "data structures" which are fundamental to quantum mechanics and quantum computing. Having said this, we will not yet study any actual principles of quantum mechanics but build on what we already know about Hopfield nets to familiarize ourselves with the incredibly important notions of the *Hamiltonian* of a physical system, its *eigenstates*, and *eigenvalues*.

As it will turn out, these objects or data structures contain all the information we could ever be interested in when using Hopfield nets as problem solvers. The price we have to pay for this almost unbelievable gain is that they are exponentially large in the respective problem sizes. However, while digital computer will therefore not be able to handle them in general, quantum computers can handle them easily.

All in all, the ideas we will discuss may seem crazy and the punch lines of our discussion will drop out of the blue sky. Alas, it is what it is; our ideas are based on *inspired* observations and do not logically follow from anything we have discussed so far. To make them more tangible, we will therefore develop them along an exemplary problem and corresponding Hopfield net whose state space is small enough to be explored exhaustively. This will allow us to support our abstract theoretical claims through practical computations and thus to build a solid foundation for the final chapters of this book.

## 9.2 An Exemplary Hopfield Net

To be able to see our upcoming ideas in action, we will develop them along a simple practical example. We therefore recall the $K$-nearest neighbor problem from Chap. 5 which we reduced to the problem of finding the $K$ smallest entries of a given (distance) vector $\boldsymbol{d} \in \mathbb{R}^n$.

Here, we shall reduce complexity even further; letting $K = 1$, we will consider

$$d_* = \underset{d_j}{\operatorname{argmin}} \left[d_1, d_2, \ldots, d_n\right]^{\mathsf{T}} . \tag{9.1}$$

which is a trivial subset selection problem as it asks for a single entry $d_j$ of $\boldsymbol{d}$ such that $d_j \leq d_k$ for all $k \neq j$. In Eq. (5.40), we already formalized this problem as a binary QUBO. Using Theorem 4.1 and the fact that $K = 1$, the corresponding bipolar QUBO reads

$$\boldsymbol{s}_* = \underset{\boldsymbol{s} \in \{\pm 1\}^n}{\operatorname{argmin}} \ \tfrac{1}{4}\, \boldsymbol{s}^{\mathsf{T}} \left[\lambda\, \mathbf{1}\mathbf{1}^{\mathsf{T}}\right] \boldsymbol{s} + \tfrac{1}{2} \left[\lambda\, \mathbf{1}\mathbf{1}^{\mathsf{T}}\mathbf{1} + \boldsymbol{d} - 2\,\lambda\,\mathbf{1}\right]^{\mathsf{T}} \boldsymbol{s} \tag{9.2}$$

where $\mathbf{1} \in \mathbb{R}^n$ and $\lambda > 0$ are a vector of all ones and a Lagrange multiplier.

Using the close connection between bipolar QUBOs and Hopfield nets and the fact that $\mathbf{1}^{\mathsf{T}}\mathbf{1} = n$, we therefore find that

$$\boldsymbol{W} = -\tfrac{\lambda}{2} \left[\mathbf{1}\mathbf{1}^{\mathsf{T}} - \boldsymbol{I}\right] \tag{9.3}$$

$$\boldsymbol{b} = \ \tfrac{1}{2} \left[\boldsymbol{d} + (n-2)\,\lambda\,\mathbf{1}\right] \tag{9.4}$$

are the parameters of a Hopfield net of $n$ neurons $s_1, s_2, \ldots, s_n$ which we could run to solve our problem. Note that we sneakily hollowed out the weight matrix by subtracting the $n \times n$ identity $\boldsymbol{I}$ from the the matrix of all ones $\mathbf{1}\mathbf{1}^{\mathsf{T}}$.

Finally, to really make our practical example specific and computable, we will henceforth work with this deliberately low dimensional vector

$$\boldsymbol{d} = \left[2, 1, 3\right]^{\mathsf{T}} \tag{9.5}$$

for which $\lambda = 2.0$ is a suitable choice of the problem's Lagrange parameter.

All in all, we are therefore considering a trivial problem. Seen from the points of view of (9.1) and (9.2), it is solved by $d_* = 1$ and $\boldsymbol{s}_* = [-1, +1, -1]^{\mathsf{T}}$, respectively. But let us move on and view it from a totally different perspective.

To prepare ourselves for this perspective, we recall that general goal of running a Hopfield net for problem solving, is to find a global minimizer of its energy function

$$s_* = \underset{s \in \{\pm 1\}^n}{\operatorname{argmin}} \; E(s) \; . \tag{9.6}$$

Since the size of the state space of a Hopfield net of $n$ neurons is $2^n$, we so far always argued that it is generally impossible to determine $s_*$ via exhaustive search. However, our above toy problem is small enough to actually allow for this strategy. We can iterate over the mere $2^3 = 8$ macro states $s \in \{\pm 1\}^3$ of the corresponding network and compute their energies $E(s)$ to identify a minimal one. But we can do even more. Due to the small state space size, we can practically enumerate all states using the integers $x \in \{0, 1, \dots, 7\}$ and represent each of these numbers in terms of a *one-hot vector* $\boldsymbol{x} \in \{0, 1\}^8$. So, let us do just that.

Given our exemplary problem specification, we may

```
import numpy as np
```

and execute the following code to prepare the corresponding Hopfield net for practical computations and experimentation.

```
vecD, lmbd = np.array([2,1,3]), 2.0

n = len(vecD)
matI = np.eye(n)
vec1 = np.ones(n)

vecB =  0.5 * (vecD + (n-2) * lmbd * vec1)
matW = -0.5 * lmbd * (np.outer(vec1, vec1) - matI)
```

The following `numpy` code then realizes the idea of enumerating all macro states of this network and of computing one-hot representations and the respective energies. It involves functions `bin2bip` and `hnetEnergy` from Chaps. 2 and 6 and introduces another helper function `num2bits` which computes the $n$-bit representation of an integer $0 \leq x < 2^n$.

```
def num2bits(x, n):
    '''
    return n bit representation of 0 <= x < 2**n as an array
    '''
    return np.array([(x >> j) & 1 for j in range(n)])[::-1]


for x in range(2**n):
    vecX = np.zeros(2**n); vecX[x] = 1
    vecS = bin2bip(num2bits(x, n))
    enrg = hnetEnergy(vecS, matW, vecB)
    print (x, vecX, vecS, enrg)
```

Looking at this code, we observe that state enumeration in terms of numbers $x$ happens implicitly by looping over `range(2**n)`. The corresponding one-hot vectors $\boldsymbol{x}$ are created by initializing an array of $2^n$ zeros and setting its $x$-th entry to one. Corresponding network states $s$ are created by mapping $x$ to an array of binary numbers which is then mapped to an array of bipolar numbers. Once a state is available, it is straightforward to compute its energy $E(s)$. Finally, all these objects get printed out.

**Table 9.1** Representations of the states of a 3 neuron Hopfield net and their respective energies

| Number $x$ | One-hot vector $\boldsymbol{x}$ | State vector $\boldsymbol{s}$ | Energy $E(\boldsymbol{s})$ |
|---|---|---|---|
| 0 | $[1, 0, 0, 0, 0, 0, 0, 0]^\intercal$ | $[-1, -1, -1]^\intercal$ | −3.0 |
| 1 | $[0, 1, 0, 0, 0, 0, 0, 0]^\intercal$ | $[-1, -1, +1]^\intercal$ | −2.0 |
| 2 | $[0, 0, 1, 0, 0, 0, 0, 0]^\intercal$ | $[-1, +1, -1]^\intercal$ | −4.0 |
| 3 | $[0, 0, 0, 1, 0, 0, 0, 0]^\intercal$ | $[-1, +1, +1]^\intercal$ | +1.0 |
| 4 | $[0, 0, 0, 0, 1, 0, 0, 0]^\intercal$ | $[+1, -1, -1]^\intercal$ | −3.0 |
| 5 | $[0, 0, 0, 0, 0, 1, 0, 0]^\intercal$ | $[+1, -1, +1]^\intercal$ | +2.0 |
| 6 | $[0, 0, 0, 0, 0, 0, 1, 0]^\intercal$ | $[+1, +1, -1]^\intercal$ | 0.0 |
| 7 | $[0, 0, 0, 0, 0, 0, 0, 1]^\intercal$ | $[+1, +1, +1]^\intercal$ | +9.0 |

We also note that our code does not include any functionality for finding an energy minimum because its outputs are all we need for what will follow. These outputs are summarized in Table 9.1 so that we can clearly see and scrutinize them.

## 9.3 Energies as Eigenvalues

Looking at Table 9.1, we again recognize the exponential nature of Hopfield energy minimization. It lists $2^n$ one-hot vectors, $2^n$ state vectors, and $2^n$ energies which we were only able to compute because our specific setting involves a small enough $n$.

Let us dwell on this some more. While our above `numpy` snippet consists of only a few lines of code and will *theoretically* work for any $n$, its output is exponentially large in $n$. The use of our snippet is therefore *practically* restricted to small choices of $n$. In other words, there is a discrepancy between the length of the description for how to compute a table such as Table 9.1 and the length of said table. Put differently, while our code is compact, the tables it produces are generally not. This discrepancy also applies to the compactness of symbolic mathematical expressions we may use to reason about the exponentially many and, in parts, exponentially large mathematical entities we are dealing with. This then brings us to the topic of this section.

In what follows, we will work with binary one-hot vectors and real valued energies as in Table 9.1 and refer to them as $\boldsymbol{x}_j$ and $E_j$ where $0 \leq j < 2^n$.

Assuming these, we now introduce an *inspired* mathematical object, namely the following matrix

$$\hat{\boldsymbol{H}} = \sum_j E_j \, \boldsymbol{x}_j \boldsymbol{x}_j^\intercal \,. \tag{9.7}$$

Looking at this matrix, we note that it is symmetric and of size $2^n \times 2^n$ because it is a weighted sum of outer products of vectors of size $2^n$. We also realize that it is a diagonal matrix because binary one-hot vectors are pairwise orthonormal such that

$$\boldsymbol{x}_j^\intercal \boldsymbol{x}_k = \delta_{jk} = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases} \tag{9.8}$$

which, in turn, implies the so called *completeness relation*

$$\sum_j \boldsymbol{x}_j \boldsymbol{x}_j^\intercal = \boldsymbol{I} \; . \tag{9.9}$$

Finally, since each outer product summand $\boldsymbol{x}_j \boldsymbol{x}_j^\intercal$ in (9.7) is multiplied by an energy $E_j$, we realize that these energy values will occur along the diagonal of matrix $\hat{\boldsymbol{H}}$.

Indeed, if we practically compute the expression in (9.7) using the specific one-hot vectors and energies listed in Table 9.1, we obtain the following result

$$\hat{\boldsymbol{H}} = \begin{bmatrix} -3 &&&&&&& \\ & -2 &&&&&& \\ && -4 &&&&& \\ &&& +1 &&&& \\ &&&& -3 &&& \\ &&&&& +2 && \\ &&&&&& 0 & \\ &&&&&&& +9 \end{bmatrix} . \tag{9.10}$$

But why would such a matrix be of interest? Why would we consider it in the context of Hopfield energy minimization? What could we use it for or what could we learn from it?

Well, when working with digital computers, matrices like these are of admittedly limited *practical* use. While they contain the energies of all macro states of an associated Hopfield net, their exponential sizes make it generally impossible to compute them. However, their *theoretical* value must not be underestimated. To see why, we consider yet another *inspired* observation.

Because of the downright trivial structure of matrix $\hat{\boldsymbol{H}}$, its **eigenvectors** $\boldsymbol{u}$ and **eigenvalues** $\eta$ are easily found. Recalling that eigenvectors and eigenvalues of a matrix such as $\hat{\boldsymbol{H}}$ are defined through the relation

$$\hat{\boldsymbol{H}} \boldsymbol{u} = \eta \, \boldsymbol{u} \tag{9.11}$$

we realize that, in the case of $\hat{\boldsymbol{H}}$, they are simply given by the $\boldsymbol{x}_j$ and $E_j$ themselves. This is because we have

$$\hat{\boldsymbol{H}} \boldsymbol{x}_k = \left[ \sum_j E_j \, \boldsymbol{x}_j \boldsymbol{x}_j^\intercal \right] \boldsymbol{x}_k = \sum_j E_j \, \boldsymbol{x}_j \left[ \boldsymbol{x}_j^\intercal \boldsymbol{x}_k \right] = \sum_j E_j \, \boldsymbol{x}_j \, \delta_{jk} = E_k \boldsymbol{x}_k \; . \tag{9.12}$$

Now, this is a profound result! It tells us that the energies of a Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ correspond to the eigenvalues of the associated matrix $\hat{\boldsymbol{H}}$. If we further recall that the set of all eigenvalues of a matrix its called its **spectrum**, the above result tells us that the energy function of a Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ and the spectrum of its associated matrix $\hat{\boldsymbol{H}}$ are essentially the same entity.

Moreover, looking back at Table 9.1, we see that, for every macro state $\boldsymbol{s}_j$ of a Hopfield net, there is a corresponding one-hot vector $\boldsymbol{x}_j$. In a sense (which we will make precise below), $\boldsymbol{s}_j$ and $\boldsymbol{x}_j$ are thus but different representations of the same entity. We can therefore say that the macro states of a Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ correspond to the eigenvectors of its associated matrix $\hat{\boldsymbol{H}}$.

All of this is then to say that we may equivalently solve the energy minimization problem in (9.6) by finding an eigenvector / eigenvalue pair $(\boldsymbol{x}_*, E_*)$ of $\hat{\boldsymbol{H}}$ such that $E_*$ is a minimal eigenvalue (in general there could be several of those).

These relations between the states and energies of a Hopfield net and the eigenvectors and eigenvalues of its associated matrix $\hat{\boldsymbol{H}}$ are not accidental because it turns out that $\hat{\boldsymbol{H}}$ as defined in (9.7) is the *Hamiltonian* of its associated Hopfield net. Hamiltonians play a crucial role in all of physics but especially in quantum mechanics and we will see more of them later on.

However, physicists would scoff at the way we defined $\hat{\boldsymbol{H}}$ in (9.7) because our (perfectly fine) definition assumes all the $2^n$ energies

$$E_j = -\tfrac{1}{2}\, \boldsymbol{s}_j^{\intercal} \boldsymbol{W} \boldsymbol{s}_j + \boldsymbol{b}^{\intercal} \boldsymbol{s}_j \tag{9.13}$$

of an $n$ neuron Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ to be known. But if they were known already, we would not have to set up the Hamiltonian to determine them. Indeed, in physics, things are usually the other way around: We are given the Hamiltonian of a system and use it to determine the system's energy. There therefore has to be a different way of setting up the Hamiltonian of a system and we next look at how this looks like in the case of Hopfield nets.

## 9.4 Hamiltonian Operators

To begin with, we consider a more general definition of the Hamiltonian of a physical system. For the time being, we will keep things simple, and focus on systems with discrete state spaces. Also, to make sense of the following statements, we recall that, if a matrix acts on a vector to produce another vector, it is called an *operator*.

**Definition 9.1** The **Hamiltonian** $\hat{\boldsymbol{H}}$ of a physical system is an operator whose spectrum contains all possible results of measuring (or computing) the system's energy.

If the the states of the system are (representable as) $N$-dimensional vectors, then $\hat{\boldsymbol{H}}$ is an $N \times N$ matrix. In general, this matrix can be complex valued but its eigenvalues will always be real.

Looking at this definition, we note that it already anticipates our upcoming study of quantum computing where complex valued matrices will play a crucial role. For a Hopfield net, however, $\hat{\boldsymbol{H}}$ is always a real valued matrix. We also note that the definition assumes that states of a system are $N$-dimensional vectors. In general, these may be vectors over $\mathbb{C}$ but, for a Hopfield net, they are always vectors over $\mathbb{R}$. Finally, we note that we may generally be dealing with infinite dimensional vectors where $N \to \infty$. Such vectors are said to be vectors in a *Hilbert space* and we will study this notion later on. However, for a Hopfield net of $n$ neurons, we will always have a finite dimension $N = 2^n$.

So, how could the Hamiltonian of a given Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ be defined without having to compute its exponentially many energies first? In other words, could it be defined in terms of only the polynomially many parameters contained in $\boldsymbol{W}$ and $\boldsymbol{b}$?

What follows will be even more *inspired* observations. To make them tangible, we first focus on our exemplary Hopfield net of only three neurons $s_1$, $s_2$, and $s_3$ and generalize this example afterwards.

To begin with, we associate each of our 3 neurons $s_k$ with a $2^3 \times 2^3$ matrix $\boldsymbol{Z}_k$. Each of these 3 matrices will be defined in terms of

$$\boldsymbol{I} = \begin{bmatrix} +1 & 0 \\ 0 & +1 \end{bmatrix} \quad \text{and} \quad \boldsymbol{Z} = \begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix} \tag{9.14}$$

where $\boldsymbol{I}$ is the $2 \times 2$ identity and $\boldsymbol{Z}$ is a famous quantum mechanical operator, namely the *third Pauli spin matrix*. Given $\boldsymbol{I}$ and $\boldsymbol{Z}$, we now simply declare

$$\boldsymbol{Z}_1 = \boldsymbol{Z} \otimes \boldsymbol{I} \otimes \boldsymbol{I} = \begin{bmatrix} +1 &&&&&&& \\ & +1 &&&&&& \\ && +1 &&&&& \\ &&& +1 &&&& \\ &&&& -1 &&& \\ &&&&& -1 && \\ &&&&&& -1 & \\ &&&&&&& -1 \end{bmatrix} \tag{9.15}$$

**Table 9.2** Macro- and corresponding micro states of a 3 neuron Hopfield net and entries of the negative diagonals of the tensor product matrices associated with each neuron

| Macro state $\boldsymbol{s}_j$ | Micro states $s_k$ | | | Negative diagonals of matrices $\boldsymbol{Z}_k$ | | |
|---|---|---|---|---|---|---|
| | $s_1$ | $s_2$ | $s_3$ | $-\operatorname{diag}[\boldsymbol{Z}_1]$ | $-\operatorname{diag}[\boldsymbol{Z}_2]$ | $-\operatorname{diag}[\boldsymbol{Z}_3]$ |
| $\boldsymbol{s}_0$ | $-1$ | $-1$ | $-1$ | $-1$ | $-1$ | $-1$ |
| $\boldsymbol{s}_1$ | $-1$ | $-1$ | $+1$ | $-1$ | $-1$ | $+1$ |
| $\boldsymbol{s}_2$ | $-1$ | $+1$ | $-1$ | $-1$ | $+1$ | $-1$ |
| $\boldsymbol{s}_3$ | $-1$ | $+1$ | $+1$ | $-1$ | $+1$ | $+1$ |
| $\boldsymbol{s}_4$ | $+1$ | $-1$ | $-1$ | $+1$ | $-1$ | $-1$ |
| $\boldsymbol{s}_5$ | $+1$ | $-1$ | $+1$ | $+1$ | $-1$ | $+1$ |
| $\boldsymbol{s}_6$ | $+1$ | $+1$ | $-1$ | $+1$ | $+1$ | $-1$ |
| $\boldsymbol{s}_7$ | $+1$ | $+1$ | $+1$ | $+1$ | $+1$ | $+1$ |

$$\boldsymbol{Z}_2 = \boldsymbol{I} \otimes \boldsymbol{Z} \otimes \boldsymbol{I} = \begin{bmatrix} +1 &&&&&&& \\ & +1 &&&&&& \\ && -1 &&&&& \\ &&& -1 &&&& \\ &&&& +1 &&& \\ &&&&& +1 && \\ &&&&&& -1 & \\ &&&&&&& -1 \end{bmatrix} \tag{9.16}$$

$$\boldsymbol{Z}_3 = \boldsymbol{I} \otimes \boldsymbol{I} \otimes \boldsymbol{Z} = \begin{bmatrix} +1 &&&&&&& \\ & -1 &&&&&& \\ && +1 &&&&& \\ &&& -1 &&&& \\ &&&& +1 &&& \\ &&&&& -1 && \\ &&&&&& +1 & \\ &&&&&&& -1 \end{bmatrix} \tag{9.17}$$

and observe that each $\boldsymbol{Z}_k$ is necessarily a diagonal matrix because it is a Kronecker product of diagonal matrices.

Now, these are truly inspired definitions but Table 9.2 shows that there really is a connection between the micro states of the 3 neurons of our simple Hopfield net and the diagonals of their 3 associated matrices. Using the enumeration we introduced above, the table lists all the $0 \leq j < 2^3$ possible macro states of our network. For each $\boldsymbol{s}_j = [s_1, s_2, s_3]^\intercal \in \{\pm 1\}^3$, it shows the values of the $s_k$ together with the $j$-th entries (counted from 0) of the *negative* diagonals of the $\boldsymbol{Z}_k$ and we recognize clear correspondences between the latter two.

But why the negative diagonals? Well, the way we defined the $\boldsymbol{Z}_k$ follows common conventions in quantum computing. We could have defined them differently but, in the end, the sign discrepancy does not really matter as long as we are aware of it.

What is arguably most striking about the content of Table 9.2 is the following: *The k-th entry macro state $\boldsymbol{s}_j$ of our Hopfield net equals the j-th entry of the negative diagonal of matrix $\boldsymbol{Z}_k$.* That is, we have

$$\big[\boldsymbol{s}_j\big]_k = -\Big[\operatorname{diag}\big[\boldsymbol{Z}_k\big]\Big]_j . \tag{9.18}$$

This is a profound observation because it allows us to express the energy $E_j$ of macro state $\boldsymbol{s}_j$ of our Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ as follows

$$\begin{aligned}
E_j &= -\tfrac{1}{2}\, \boldsymbol{s}_j^\intercal \boldsymbol{W} \boldsymbol{s}_j + \boldsymbol{b}^\intercal \boldsymbol{s}_j && (9.19)\\
&= -\tfrac{1}{2} \sum_k \sum_l w_{kl} \big[\boldsymbol{s}_j\big]_k \big[\boldsymbol{s}_j\big]_l + \sum_k b_k \big[\boldsymbol{s}_j\big]_k && (9.20)\\
&= -\tfrac{1}{2} \sum_k \sum_l w_{kl} \Big[\operatorname{diag}\big[\boldsymbol{Z}_k\big]\Big]_j \Big[\operatorname{diag}\big[\boldsymbol{Z}_l\big]\Big]_j - \sum_k b_k \Big[\operatorname{diag}\big[\boldsymbol{Z}_k\big]\Big]_j . && (9.21)
\end{aligned}$$

But we can go even further and rewrite the last expression in a much more compact manner. To this end, we simply note that entry $j$ of $\operatorname{diag}\big[\boldsymbol{Z}_k\big]$ correspond to entry $jj$ of $\boldsymbol{Z}_k$ or, formally

$$\Big[\operatorname{diag}\big[\boldsymbol{Z}_k\big]\Big]_j = \Big[\boldsymbol{Z}_k\Big]_{jj} . \tag{9.22}$$

Using this, we have

$$E_j = -\tfrac{1}{2} \sum_k \sum_l w_{kl} \big[\boldsymbol{Z}_k\big]_{jj} \big[\boldsymbol{Z}_l\big]_{jj} - \sum_k b_k \big[\boldsymbol{Z}_k\big]_{jj} . \tag{9.23}$$

If we next recall that the $\boldsymbol{Z}_k$ are *diagonal* matrices, we can further use the following identity

$$\big[\boldsymbol{Z}_k\big]_{jj} \big[\boldsymbol{Z}_l\big]_{jj} = \big[\boldsymbol{Z}_k \boldsymbol{Z}_l\big]_{jj} \tag{9.24}$$

to write the energy of macro state $\boldsymbol{s}_j$ even more compactly

$$E_j = -\tfrac{1}{2} \sum_k \sum_l w_{kl} \big[\boldsymbol{Z}_k \boldsymbol{Z}_l\big]_{jj} - \sum_k b_k \big[\boldsymbol{Z}_k\big]_{jj} . \tag{9.25}$$

So far, all this is interesting but does not really answer our question about the Hamiltonian of our Hopfield net. We therefore furthermore observe that we may gather all energies of all macro states in a vector $\boldsymbol{E} \in \mathbb{R}^{2^3}$ such that

$$\boldsymbol{E} = \begin{bmatrix} E_0 \\ E_1 \\ \vdots \\ E_7 \end{bmatrix} = -\tfrac{1}{2} \sum_k \sum_l w_{kl} \begin{bmatrix} [\boldsymbol{Z}_k \boldsymbol{Z}_l]_{00} \\ [\boldsymbol{Z}_k \boldsymbol{Z}_l]_{11} \\ \vdots \\ [\boldsymbol{Z}_k \boldsymbol{Z}_l]_{77} \end{bmatrix} - \sum_k b_k \begin{bmatrix} [\boldsymbol{Z}_k]_{00} \\ [\boldsymbol{Z}_k]_{11} \\ \vdots \\ [\boldsymbol{Z}_k]_{77} \end{bmatrix} . \tag{9.26}$$

With this, we are ready for the punchline of our present discussion and now simply define

$$\hat{\boldsymbol{H}} = \mathrm{diag}\big[\boldsymbol{E}\big] . \tag{9.27}$$

If we then finally once again recall that the matrices $\boldsymbol{Z}_k$ are diagonal so that their products $\boldsymbol{Z}_k \boldsymbol{Z}_l$ are diagonal as well, we realize that the diagonal matrix which we just defined can also be computed as

$$\hat{\boldsymbol{H}} = -\tfrac{1}{2} \sum_k \sum_l w_{kl} \, \boldsymbol{Z}_k \boldsymbol{Z}_l - \sum_k b_k \, \boldsymbol{Z}_k . \tag{9.28}$$

Indeed, if we practically evaluate the expression on the right hand side of (9.28) for our simple exemplary Hopfield net, we obtain

$$\hat{\boldsymbol{H}} = \begin{bmatrix} -3 &&&&&&& \\ & -2 &&&&&& \\ && -4 &&&&& \\ &&& +1 &&&& \\ &&&& -3 &&& \\ &&&&& +2 && \\ &&&&&& 0 & \\ &&&&&&& +9 \end{bmatrix} \tag{9.29}$$

and thus recover our earlier practical result in (9.10).

All that is then left to do is to generalize our considerations from the special case of a Hopfield net of 3 neurons $s_1, s_2, s_3$ to the general case of a Hopfield net of $n$ neurons $s_1, s_2, \ldots, s_n$. However, since the mathematical expression which we arrived at in (9.28) is generic, this generalization is easily accomplished.

Note that we deliberately only wrote summation indices in (9.28) and refrained from specifying their ranges. For our specific example of a Hopfield net of $n = 3$ neurons both summation indices $k$ and $l$ range from 1 to 3. In general, they would range from 1 to $n$ and the Hopfield net parameters $w_{kl}$ and $b_k$ would be the entries of an $n \times n$ matrix $\boldsymbol{W}$ and an $n$ vector $\boldsymbol{b}$. However, so far, we only defined 3 matrices $\boldsymbol{Z}_k$ to be associated with the 3 neurons of our exemplary network and the only question therefore is how to generalize these to a general choice of $n$.

Well, the answer simply follows from generalizing the pattern which can be easily recognized in equations (9.15)–(9.17). Since this generalization is straightforward and important, we state it as a "defining theorem".

**Theorem 9.1** *Given a Hopfield net of $n$ neurons $s_1, s_2, \ldots, s_n$, each neuron $s_k$ can be uniquely associated with a $2^n \times 2^n$ matrix*

$$\boldsymbol{Z}_k = \underbrace{\boldsymbol{I} \otimes \boldsymbol{I} \otimes \cdots \otimes \boldsymbol{I}}_{k-1\ terms} \otimes \boldsymbol{Z} \otimes \underbrace{\boldsymbol{I} \otimes \boldsymbol{I} \otimes \cdots \otimes \boldsymbol{I}}_{n-k\ terms} \tag{9.30}$$

*where $\boldsymbol{I}$ is the $2 \times 2$ identity matrix and $\boldsymbol{Z}$ is the third Pauli spin matrix.*

Give this generalization of the $\boldsymbol{Z}_k$, we can generalize and summarize our results so far in another "defining theorem" where we now include summation ranges.

**Theorem 9.2** *The Hamiltonian operator associated with a Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ of $n$ neurons $s_1, s_2, \ldots, s_n$ is the following $2^n \times 2^n$ matrix*

$$\hat{\boldsymbol{H}} = -\frac{1}{2} \sum_{k=1}^{n} \sum_{l=1}^{n} w_{kl}\, \boldsymbol{Z}_k\, \boldsymbol{Z}_l - \sum_{k=1}^{n} b_k\, \boldsymbol{Z}_k\ . \tag{9.30}$$

*It admits a spectral decomposition*

$$\hat{\boldsymbol{H}} = \sum_{j=0}^{2^n-1} E_j\, \boldsymbol{x}_j \boldsymbol{x}_j^\intercal \tag{9.31}$$

*in terms of pairwise orthonormal* ***eigenstates*** $\boldsymbol{x}_j \in \{0, 1\}^{2^n}$ *and corresponding* ***eigenenergies*** $E_j \in \mathbb{R}$.

With this theorem we have arranged everything in a more appropriate order. Equation (9.30) defines the Hamiltonian of a Hopfield net in terms of the network's parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ and our earlier definition in terms of eigenstates and eigenenergies now follows as a consequence in Eq. (9.31).

Note that the right hand sides of (9.31) and (9.30) are but different ways of expressing the same entity. However, the computations they prescribe are anything but identical. On a digital computer, both will generally be infeasible but the former are even more infeasible than the latter: While the right hand side of (9.31) involves $2^n$ summations of exponentially large matrices, the right hand side of (9.30) only requires $n^2 + n$ summations of exponentially large matrices. The latter is obviously preferable if only we could practically compute with matrices of size $2^n \times 2^n$.

On a standard desktop computer this will feasible for $n \lesssim 26$. For larger $n$, supercomputers are called for; but even these will reach their limits once $n \gtrsim 100$. Quantum computers, on the other hand, could handle any $n$ in theory. In practice, however, they still suffer from shortcomings of current technology so that present day quantum devices also reach their limits at $n \approx 100$.

Assuming we could handle exponentially large matrices, Theorem 9.2 is quite profound. Above, we already said that we may (try to) solve a Hopfield energy minimization problem by finding an eigenstate/eigenenergy pair $(\boldsymbol{x}_*, E_*)$ of the network's Hamiltonian such that $E_*$ is minimal. Formally, we may express this task as follows

$$\begin{aligned} \boldsymbol{x}_*, E_* &= \underset{j}{\operatorname{argmin}} \; E_j \\ &\text{s.t.} \quad \hat{\boldsymbol{H}} \boldsymbol{x}_j = E_j \, \boldsymbol{x}_j \end{aligned} \tag{9.33}$$

and, now that we are armed with the definition in (9.30), can solve it without having to know the eigenstates and eigenenergies in advance. Solving eigenvector/eigenvalue problems like this is a standard task in linear algebra and the hard sciences and there thus exist numerous numerical algorithms for this purpose [1, 2]. Of course, these methods are not generally feasible for exponentially large matrices, but there also exist related quantum computing algorithms which allow for computing the sought after eigenstate $\boldsymbol{x}_*$.

In short, we can therefore hope to solve Hopfield energy minimization problems without having to run Hopfield nets. However, being able to find a *bottom* eigenstate $\boldsymbol{x}_*$ of $\hat{\boldsymbol{H}}$ will only indirectly solve our problem if we are actually interested in the corresponding minimum energy macro state $\boldsymbol{s}_*$ of the associated Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$. This then begs the question if there is a systematic connection between the two?

## 9.5 Tensor Product States

Next, we consider yet another *inspired* idea. However, it builds on material we studied in Chaps. 2 and 3 so that it may not seem as outlandish as our other inspired ideas so far.

Consider this: If we can solve a Hopfield energy minimization problem without having to run a Hopfield net, we do not have to think of its neurons as computational units anymore but may really only think of them as variables whose values signify (in)activity. So far, we represented these as

$$\texttt{inactive} \quad \Leftrightarrow \quad -1 \tag{9.34}$$

$$\texttt{active} \quad \Leftrightarrow \quad +1 \tag{9.35}$$

and thus assumed the micro states $s_k$ of a Hopfield net to be numbers which take values in the Boolean domain $\mathbb{B} = \{\pm 1\}$. However, we already know that there are many Boolean domains and, now that we do no longer need to think of neurons as computational units, we may just as well use any of these Boolean domains to represent their states. For instance, here is an *inspired* choice

$$\texttt{inactive} \quad \Leftrightarrow \quad \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{9.36}$$

$$\texttt{active} \quad \Leftrightarrow \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{9.37}$$

where we think of micro states of a Hopfield net as 2-dimensional, orthonormal, binary vectors $z_k$ which take values in the Boolean domain $\mathbb{B} = \big\{[1, 0]^\intercal, [0, 1]^\intercal\big\}$.

But if micro states are vectors, then what are the corresponding macro states? Well, there would be many possible modeling choices but, in light of our above discussion, the following turns out to be a surprisingly reasonable *inspired* idea.

**Definition 9.2** The macro state of a system of $n$ binary vector valued micro states $z_1, z_2, \ldots, z_n$ is the following tensor product vector

$$\boldsymbol{x} = z_1 \otimes z_2 \otimes \cdots \otimes z_n \ . \tag{9.38}$$

To motivate why this is a reasonable definition of the macro state of a system of vector valued micro states, we return to our exemplary simple Hopfield net with 3 conventional micro states $s_k \in \{\pm 1\}$ and recall that its possible conventional macro states $\boldsymbol{s}$ are listed in Table 9.1.

Consider, for instance, the particular macro state

$$\boldsymbol{s} = \big[-1, +1, -1\big]^\intercal \in \{\pm 1\}^3 \ . \tag{9.39}$$

Working with our new micro state representations $z_k \in \big\{[1, 0]^\intercal, [0, 1]^\intercal\big\}$, this particular macro state will be represented as

$$\boldsymbol{x} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \big[0, 0, 1, 0, 0, 0, 0, 0\big]^\intercal \in \{0, 1\}^{2^3} \tag{9.40}$$

and we observe that this vector $\boldsymbol{x}$ is also listed in Table 9.1, namely as the one-hot representation of vector $\boldsymbol{s}$.

Indeed, it is an easy exercise for the reader to verify that the one-hot representation of every $\boldsymbol{s} = [s_1, s_2, s_3]^\intercal$ in Table 9.1 can be computed as $\boldsymbol{x} = z_1 \otimes z_2 \otimes z_3$ where

$$z_k = \begin{cases} [1, 0]^\intercal & \text{if } s_k = -1 \\ [0, 1]^\intercal & \text{if } s_k = +1 \ . \end{cases} \tag{9.41}$$

Another easy exercise left to the reader is to verify that this connection between vectors $\boldsymbol{s}$ and $\boldsymbol{x}$ holds for any $n$ and not just for the special case of $n = 3$ considered in Table 9.1.

We also emphasize that, if we are dealing with $n$ vectors $z_k \in \{[1, 0]^\intercal, [0, 1]^\intercal\}$, then there will be $2^n$ possible instantiations of this set of vectors and each corresponding tensor product will be a $2^n$-dimensional vector. In short, there are $2^n$ vectors

$$\boldsymbol{x} = z_1 \otimes z_2 \otimes \cdots \otimes z_n \in \{0, 1\}^{2^n} . \tag{9.42}$$

We therefore find that the number of possible vectors $\boldsymbol{x}$ corresponds to the number of possible macro states of a Hopfield net of $n$ neurons and that their dimensionality corresponds to the dimensionality of the eigenvectors of the associated Hamiltonian. In fact, looking the spectral decomposition in (9.31), we realize that the set of possible vectors $\boldsymbol{x}$ coincides with the set of eigenvectors of the Hamiltonian associated with a Hopfield net. All in all, we may summarize all these insights in terms of the following "theorem".

**Theorem 9.3** *A Hopfield net* $(\boldsymbol{W}, \boldsymbol{b})$ *with macro states* $\boldsymbol{s}$ *and energy function* $E(\boldsymbol{s})$ *and its associated Hamiltonian* $\hat{\boldsymbol{H}}$ *with eigenstates* $\boldsymbol{x}$ *and eigenenergies* $E$ *are but two sides of the same coin.*

Finally, we return to our question from the end of the last section and look at how to map one-hot vectors $\boldsymbol{x} = z_1 \otimes z_2 \otimes \cdots \otimes z_n$ to corresponding bipolar vectors $\boldsymbol{s} = [s_1, s_2, \ldots, s_n]^\intercal$. While we saw above that the procedure for mapping $\boldsymbol{s}$ to $\boldsymbol{x}$ is easily written down in mathematical notation, there is no such simple notation for the direction we interested in right now. We may therefore immediately give the procedure in terms of the following `numpy` code which reuses the functions `bin2bip` and `num2bits` we already discussed at the beginning of this chapter.

```
def onehot2bip(vecX):
    '''
    turn a one-hot vector vecX in {0,1}**(2**n)
    into a bipolar vector vecS in {-1,+1}**n
    '''
    j = np.argmax(vecX)
    n = int(np.log2(len(vecX)))
    return bin2bip(num2bits(j,n))
```

## 9.6 Fun with Tensor Product States

To wrap this chapter up, it might be a fun idea to practically work tensor product state representations.

First, however, we better reconsider some of the material from Chap. 2. We already know that the set $\mathbb{B} = \{\pm 1\}$ forms a Boolean domain, because we can define a negation $\texttt{NOT} : \mathbb{B} \to \mathbb{B}$ as $\texttt{NOT}(s_k) = -s_k$ which turns its inputs into their logical opposites. But what about the set

$$\mathbb{B} = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\} ? \tag{9.43}$$

Above, we claimed it was a Boolean domain. To verify this claim, we therefore need to find a function $\texttt{NOT} : \mathbb{B} \rightarrow \mathbb{B}$ that turns each $z_k \in \mathbb{B}$ into its logical opposite.

Such a function is easy to come by We simply consider $\texttt{NOT}(z_k) = \boldsymbol{X} z_k$ where

$$\boldsymbol{X} = \begin{bmatrix} 0 & +1 \\ +1 & 0 \end{bmatrix} \tag{9.44}$$

is known as the *first Pauli spin matrix*. Using it, we have

$$\begin{bmatrix} 0 & +1 \\ +1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{9.45}$$

$$\begin{bmatrix} 0 & +1 \\ +1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{9.46}$$

and therefore verified that $\mathbb{B}$ in (9.43) is indeed a Boolean domain. But applications of $\boldsymbol{X}$ go far beyond this simple use case.

Say we were given a macro state $\boldsymbol{s} = [s_1, s_2, s_3]^\intercal$ of a Hopfield net of 3 neurons and wanted to flip its second micro state $s_2$. In terms of a linear algebraic operation, we can accomplish this via the following matrix-vector multiplication

$$\begin{bmatrix} +1 & & \\ & -1 & \\ & & +1 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} s_1 \\ -s_2 \\ s_3 \end{bmatrix} . \tag{9.47}$$

But can we also realize such an operation when working with the corresponding tensor state representation $\boldsymbol{x} = z_1 \otimes z_2 \otimes z_3$? Can we just as easily flip its second factor $z_2$?

Yes, we can! Recalling what we learned about *mixed product properties* of tensor products in Chap. 3, we simply consider the following matrix-vector product

$$\big[\boldsymbol{I} \otimes \boldsymbol{X} \otimes \boldsymbol{I}\big]\big[z_1 \otimes z_2 \otimes z_3\big] = \boldsymbol{I} z_1 \otimes \boldsymbol{X} z_2 \otimes \boldsymbol{I} z_3 \tag{9.48}$$

and realize that this indeed flips the second factor while keeping the others intact.

Note the curious structure of the matrix with which we acted on $\boldsymbol{x}$. In accordance with what we did with the third Pauli spin matrix in (9.15)–(9.17), we may call it

$$\boldsymbol{X}_2 = \boldsymbol{I} \otimes \boldsymbol{X} \otimes \boldsymbol{I} \tag{9.49}$$

and realize that its subscript "2" indicates which factor of $\boldsymbol{x} = z_1 \otimes z_2 \otimes z_3$ it flips.

But can we also flip more than one factor simultaneously? Say, the second and third one? Yes, we can. We would simply have to work with

$$\boldsymbol{X}_{23} = \boldsymbol{X}_2\boldsymbol{X}_3 = \big[\boldsymbol{I} \otimes \boldsymbol{X} \otimes \boldsymbol{I}\big]\big[\boldsymbol{I} \otimes \boldsymbol{I} \otimes \boldsymbol{X}\big] = \boldsymbol{I} \otimes \boldsymbol{X} \otimes \boldsymbol{X} \,. \tag{9.50}$$

Finally, here is a first look at something we can do with vectors $\boldsymbol{z}_k$ in the Boolean domain in (9.43) which we simply cannot similarly do with their corresponding numbers $s_k$ in the Boolean domain $\{\pm 1\}$. Consider the $2 \times 2$ *Hadamard matrix*

$$\boldsymbol{H} = \tfrac{1}{\sqrt{2}}\big[\boldsymbol{X} + \boldsymbol{Z}\big] = \tfrac{1}{\sqrt{2}}\begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} . \tag{9.51}$$

If we let it act on, say, $\boldsymbol{z}_k = [1, 0]^\intercal$, we obtain

$$\boldsymbol{H}\boldsymbol{z}_k = \tfrac{1}{\sqrt{2}}\begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \tfrac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \tfrac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 0 \end{bmatrix} + \tfrac{1}{\sqrt{2}}\begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{9.52}$$

and therefore get a *non-zero* vector of unit length that is a linear combination of the two unit vectors $[1, 0]^\intercal$ and $[0, 1]^\intercal$. Try as we might, we could not realize such a non-zero *superposition* of states when working $s_k \in \{\pm 1\}$.

All in all, there therefore seems to be something special about the Boolean domain in (9.43) and about tensor products of its elements. In the remaining chapters of this book, we will investigate this further.

## 9.7 Exercises

**9.1** Implement `python/numpy` code to compute arrays which represent the matrices $\boldsymbol{I}$, $\boldsymbol{Z}$, $\boldsymbol{Z}_1$, $\boldsymbol{Z}_2$, and $\boldsymbol{Z}_3$ as defined in (9.14)–(9.17).

**9.2** Consider the arrays `matW` and `vecB` as computed in our introductory `numpy` snippet which implement the parameters of our tiny Hopfield net. Use these and your results from the previous exercise to write `python/numpy` code that computes (and prints) matrix $\hat{\boldsymbol{H}}$ in (9.28).

**9.3** The `numpy` module `linalg` provides two functions `eig` and `eigh` for solving eigenvalue/eigenvector indexeigenvector problems. When working with matrices over $\mathbb{R}$, `eig` applies universally whereas `eigh` only works for symmetric matrices but matrix $\hat{\boldsymbol{H}}$ in (9.28) is real valued and symmetric. Use both functions `eig` and `eigh` to compute its eigenvalues and eigenvectors and inspect your results. If you are so inclined, perform running time measurements and ponder their outcomes.

**Note:** If you have never worked with `eig` and/or `eigh` before, read their manuals very carefully to make sure you invoke them correctly.

**9.4** Consider $N \times N$ matrices $\boldsymbol{Z}_k$ and $\boldsymbol{Z}_l$ as in Theorem 9.1. If we use naïve matrix multiplication, then computing the product $\boldsymbol{Z}_k\boldsymbol{Z}_l$ is an $O(N^3)$ operation. However, since $\boldsymbol{Z}_k$ and $\boldsymbol{Z}_l$ are diagonal, $\boldsymbol{Z}_k\boldsymbol{Z}_l$ can be seamlessly computed in $O(N^2)$ operations. Explain why and how this is possible.

**9.5** Let $\boldsymbol{x}_0 = [1, 0]^\intercal$ and $\boldsymbol{x}_1 = [0, 1]^\intercal$ and consider the following matrices

$$\boldsymbol{P}_0 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \quad \text{and} \quad \boldsymbol{P}_1 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} . \tag{9.53}$$

Observe that $\boldsymbol{P}_0 + \boldsymbol{P}_1 = \boldsymbol{I}$ yields the $2 \times 2$ identity matrix. Compute the expressions $\boldsymbol{P}_0\boldsymbol{x}_0$, $\boldsymbol{P}_0\boldsymbol{x}_1$, $\boldsymbol{P}_0\big[\boldsymbol{x}_0 + \boldsymbol{x}_1\big]$, $\boldsymbol{P}_1\boldsymbol{x}_0$, $\boldsymbol{P}_1\boldsymbol{x}_1$, and $\boldsymbol{P}_1\big[\boldsymbol{x}_0 + \boldsymbol{x}_1\big]$. Ponder what you observe. Why would people call $\boldsymbol{P}_0$ and $\boldsymbol{P}_1$ *projection matrices* or *projectors* ?

**9.6** Write `python/numpy` code that realizes a function `CNOT` : $\{0, 1\}^2 \rightarrow \{0, 1\}^2$ which takes two inputs $x_1, x_2 \in \{0, 1\}$ and produces two outputs $y_1, y_2 \in \{0, 1\}$ with

$$y_1 = x_1 \tag{9.54}$$

$$y_2 = \begin{cases} x_2 & \text{if } x_1 = 0 \\ \texttt{NOT}(x_2) & \text{if } x_1 = 1 \ . \end{cases} \tag{9.55}$$

Run your function on all (four) possible inputs and have a look at its output $y_2$. Which Boolean function $y_2 = f(x_1, x_2)$ do you recognize?

**9.7** Let $\boldsymbol{x}_0$, $\boldsymbol{x}_1$, $\boldsymbol{P}_0$, and $\boldsymbol{P}_0$ be as in Exercise 9.5. Also let $\boldsymbol{I}$ and $\boldsymbol{X}$ be the $2 \times 2$ identity and first Pauli spin matrix, respectively. Write `numpy` code that computes the matrix

$$\boldsymbol{C}_{\texttt{NOT}} = \boldsymbol{P}_0 \otimes \boldsymbol{I} + \boldsymbol{P}_1 \otimes \boldsymbol{X} \tag{9.56}$$

and then evaluates the four expressions $\boldsymbol{C}_{\texttt{NOT}}[\boldsymbol{x}_0 \otimes \boldsymbol{x}_0]$, $\boldsymbol{C}_{\texttt{NOT}}[\boldsymbol{x}_0 \otimes \boldsymbol{x}_1]$, $\boldsymbol{C}_{\texttt{NOT}}[\boldsymbol{x}_1 \otimes \boldsymbol{x}_0]$, and $\boldsymbol{C}_{\texttt{NOT}}[\boldsymbol{x}_1 \otimes \boldsymbol{x}_1]$. Carefully think about the significance of the results you obtain. Maybe feed them in function `onehot2bip` to get a representation that is easier to interpret. Would you say that function `CNOT` from the previous exercise and matrix $\boldsymbol{C}_{\texttt{NOT}}$ from this exercise *essentially* do the same thing?

**9.8** Let $\boldsymbol{P}_0$, $\boldsymbol{P}_1$, $\boldsymbol{I}$, and $\boldsymbol{X}$ as in the previous exercise and prove the following identity

$$\boldsymbol{P}_0 \otimes \boldsymbol{I} + \boldsymbol{P}_1 \otimes \boldsymbol{X} = \boldsymbol{I} \otimes \boldsymbol{I} + \boldsymbol{P}_1 \otimes \big[\boldsymbol{X} - \boldsymbol{I}\big] . \tag{9.57}$$

**9.9** Note that $\boldsymbol{P}_0^2 = \boldsymbol{P}_0$, $\boldsymbol{P}_1^2 = \boldsymbol{P}_1$, $\boldsymbol{I}^2 = \boldsymbol{I}$, and $\boldsymbol{X}^2 = \boldsymbol{I}$. Work with the algebra of Kronecker products from Chap. 3 to symbolically compute $\boldsymbol{C}_{\texttt{NOT}}^2$, simplify the resulting expression to the extent possible, and ponder what you find.

**9.10** Note that $\boldsymbol{P}_0^\intercal = \boldsymbol{P}_0$, $\boldsymbol{P}_1^\intercal = \boldsymbol{P}_1$, $\boldsymbol{I}^\intercal = \boldsymbol{I}$, and $\boldsymbol{X}^\intercal = \boldsymbol{X}$. Symbolically compute $\boldsymbol{C}_{\texttt{NOT}}\boldsymbol{C}_{\texttt{NOT}}^\intercal$, simplify the resulting expression to the extent possible, and ponder what you find.

## References

1. Strang, G.: Linear Algebra and Its Applications, 4th edn. Cenage Learning (2005)
2. Golub, G., van Loan, C.: Matrix Computations, 4th edn. Johns Hopkins University Press (2012)

# Part III
# Quantum Computing

# Chapter 10
# Quantum Mechanics in a Nutshell

## 10.1 Introduction

Given all our preparations so far, we can finally begin to transit from digital information processing to quantum information processing. But we are not quite there yet. To be able to see how the *principles of quantum mechanics* can be utilized for computing, we need to ever so briefly familiarize ourselves with those principles and with the mathematics used to describe them. This may be easier said than done because quantum mechanics is all about the physics of the (sub)atomic world where things do not behave as in our everyday world. Most of what we are going to discuss next will thus be complex (no pun intended), unintuitive, and hard to accept.

Indeed, the challenges of learning about quantum mechanics are common lore. Usually, the main difficulty is to accept the many alien behaviors of quantum systems for what they are, namely well established physical phenomena for which there exist no counterparts or analog behaviors in the macroscopic world we are familiar with. Try as we might, it is typically impossible to explain quantum mechanical phenomena in terms of concepts or ideas we are used to or learned about in school. The best strategy for learners may therefore be to go with the flow and to be open minded and curious about quantum mechanics, its fundamental laws, and the strange effects these laws imply.

On the other hand, we are arguably well prepared for what comes next because we already know about abstract state spaces, tensor products, and Hamiltonian operators and their eigenvectors and eigenvalues. For us, there are thus "only" three difficulties left when studying the principles of quantum mechanics, two have to do with the nature of quantum systems and one with how people reason about them. First of all, mathematical descriptions of quantum states generally involve *complex numbers*. Second of all, there is an (unintuitive) aspect of *probability* in the interpretation of quantum states. And, third of all, while the mathematics of quantum mechanics is just complex valued linear algebra, it is usually written in the *Dirac notation* which offers great symbolic flexibility but needs getting used to.

C. Bauckhage and R. Sifa, *Quantum Computing from Hopfield Nets*,
Cognitive Technologies, https://doi.org/10.1007/978-3-031-99402-9_10

In this chapter, we will rise to the challenge and summarize the central mathematical concepts of quantum mechanics. First, we give a brief and incomplete account of the historical development of the field. This will allow us to introduce some of the (axiomatic) equations with which we will be working later. Then, we briefly study the notion of *Hilbert spaces* which are the arena in which mathematical quantum mechanics takes place in. This will involve abstract algebra which we only use for rigor; from a bigger picture point of view, this abstract algebra it not essential for the study of specific quantum systems but it provides us with a wider perspective on what we are dealing with. Next, we introduce the Dirac notation and painstakingly compare it to the linear algebraic notation most computer scientists should be familiar with. This will be tedious but beneficial to those who are serious about learning quantum computing. Finally, we will use all of this to discuss (some of) the mathematics of quantum mechanics.

Once again, all of this may appear very abstract and hard to accept when seen for the first time. At first sight, quantum mechanics seems to defy common sense or even to suggest that the nature of the (sub)atomic world is absurd. But it is what it is and there is not much we can do about it: The mathematical formulation of quantum mechanics as we know it today fully agrees with numerous physical experiments and we will have to spend some time on it to later be able to appreciate the fundamental ideas behind quantum computing.

However, prior to our upcoming study, we shall ever so briefly recall the notions of complex numbers, complex vectors, and complex matrices as this will tremendously simplify our discussion.

### *10.1.1 Complex Numbers*

In what follows, *complex numbers* will enter the stage for the first time. We therefore recall that a **complex number** $c \in \mathbb{C}$ can be written as

$$c = a + i\, b \tag{10.1}$$

where $a, b \in \mathbb{R}$ are real numbers and $i$ denotes the **imaginary unit** whose defining property is

$$i^2 = -1\ . \tag{10.2}$$

For any complex number $c$ expressed as in (10.1), its **complex conjugate** $c^* \in \mathbb{C}$ is given by

$$c^* = a - i\, b\ . \tag{10.3}$$

With respect to the algebra of complex numbers, we note that $(c^*)^* = c$ and that $c = c^*$ implies that $c \in \mathbb{R}$. Moreover, if any two complex numbers are expressed as in (10.1), then their sum and product amount to

$$\big(a_1 + i\,b_1\big) + \big(a_2 + i\,b_2\big) = \big(a_1 + a_2\big) + i\,\big(b_1 + b_2\big) \tag{10.4}$$

and

$$\begin{aligned}\big(a_1 + i\,b_1\big)\cdot\big(a_2 + i\,b_2\big) &= a_1a_2 + i\,a_1b_2 + i\,b_1a_2 + i^2b_1b_2 &(10.5)\\ &= \big(a_1a_2 - b_1b_2\big) + i\,\big(a_1b_2 + a_2b_1\big)\,. &(10.6)\end{aligned}$$

Hence, the product of any complex number $c$ and its complex conjugate $c^*$ always evaluates to

$$c\cdot c^* = \big(a + i\,b\big)\cdot\big(a - i\,b\big) = a^2 + b^2 \tag{10.7}$$

and the square root of this expression

$$|c| = \sqrt{c\cdot c^*} = \sqrt{a^2 + b^2} \geq 0 \tag{10.8}$$

is known as the **modulus**, **magnitude** or **absolute value** of $c$.

Alternatively, any complex number $c = a + i\,b$ can be also written in **polar form**

$$c = r\cdot\big(\cos\varphi + i\,\sin\varphi\big) = r\cdot e^{+i\,\varphi} \tag{10.9}$$

with

$$r = \sqrt{a^2 + b^2} \tag{10.10}$$

$$\varphi = \operatorname{arctan2}\frac{b}{a} \tag{10.11}$$

where $0 \leq \varphi < 2\,\pi$ is called the **argument** of $c$ and arctan2 is the 2-argument arctan function which can handle pathological cases where $a = 0$.

When a complex number $c$ is expressed as in (10.9), its complex conjugate $c^*$ can be expressed as

$$c^* = r\cdot\big(\cos\varphi - i\,\sin\varphi\big) = r\cdot e^{-i\,\varphi} \tag{10.12}$$

and we note that

$$c\cdot c^* = r\cdot e^{+i\,\varphi}\cdot r\cdot e^{-i\,\varphi} = r^2\cdot e^{+i\,\varphi - i\,\varphi} = r^2\,. \tag{10.13}$$

Finally, we point out the following important equivalence for any complex number of modulus one, namely

$$|c| = 1 \quad\Leftrightarrow\quad c = \big(\cos\varphi + i\,\sin\varphi\big) = e^{i\,\varphi} \tag{10.14}$$

which means that any $c$ with $|c| = 1$ can be identified with a point on the unit circle in the complex plane.

## *10.1.2 Complex Vectors and Matrices*

Up until now, we have only been working with real valued vectors but, from now on, we will frequently need to consider complex valued vectors $\boldsymbol{v} \in \mathbb{C}^n$.

There is nothing too special about those; we may simply think of them as collections of complex numbers gathered in a column array

$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \tag{10.15}$$

What is special about them are their transposes. Extending the transposition known from real valued vectors to their complex counterparts, we have

$$\boldsymbol{v}^\dagger = \left[v_1^*, \ldots, v_n^*\right] \tag{10.16}$$

and refer to $\boldsymbol{v}^\dagger$ as the **conjugate transpose** of $\boldsymbol{v}$. Put in rather simple terms, the *dagger operator* turns a column vector into a row vector *and* conjugates all its entries.

By the same token, we will henceforth often have to work with complex valued matrices $\boldsymbol{M} \in \mathbb{C}^{m \times n}$.

Again, there is nothing too special about those but we have to once more generalize their transposition. Indeed, the entries of the **conjugate transpose** $\boldsymbol{M}^\dagger$ of a complex matrix $\boldsymbol{M}$ are given by

$$\left[\boldsymbol{M}^\dagger\right]_{jk} = \left[\boldsymbol{M}\right]_{kj}^* \tag{10.17}$$

which means that the dagger operator once again performs transposition *and* complex conjugation simultaneously.

While we are at it, we may just as well emphasize that the matrices or *operators* which occur in quantum mechanics are always square and that there are two very special kinds of those.

**Definition 10.1** A matrix $\boldsymbol{H} \in \mathbb{C}^{n \times n}$ is **Hermitian**, if

$$\boldsymbol{H} = \boldsymbol{H}^\dagger \,. \tag{10.18}$$

**Definition 10.2** A matrix $\boldsymbol{U} \in \mathbb{C}^{n \times n}$ is **unitary**, if

$$\boldsymbol{U}\,\boldsymbol{U}^\dagger = \boldsymbol{U}^\dagger \boldsymbol{U} = \boldsymbol{I}\,. \tag{10.19}$$

Looking at these definitions, we observe: Complex Hermitian matrices generalize the notion of real symmetric matrices where $\boldsymbol{S} = \boldsymbol{S}^\intercal$. Complex unitary matrices generalize the notion of real orthogonal matrices where $\boldsymbol{O}\,\boldsymbol{O}^\intercal = \boldsymbol{O}^\intercal \boldsymbol{O} = \boldsymbol{I}$.

We also observe that, if $\boldsymbol{U}$ is unitary, then $\boldsymbol{U}^{-1} = \boldsymbol{U}^\dagger$ which is to say that unitary matrices are non-singular and invertible.

## 10.2 A Short (and Incomplete) History of Quantum Mechanics

Before we begin a deeper dive into the mathematics of quantum mechanics, we shall give a short account of milestones in the development of the field. Why? Because it is good to know what the fuss is all about and also because it allows us to introduce some of the equations we will be using later on.

**1900** Max Planck (Nobel prize 1918) explains a hitherto inexplicable phenomenon in the context of *black body radiation* by suggesting that the *energy* of light waves is not absorbed or re-emitted in a continuous manner but rather in discrete amounts or *quanta*. Mathematically, he expresses this as

$$E = h\,\nu \tag{10.20}$$

where $\nu$ denotes the *frequency* of a light wave and

$$h = 6.62607004 \times 10^{-34}\text{Js} \tag{10.21}$$

is nowadays known as *Planck's constant*. Planck's revolutionary idea worked indisputably well in practice but fundamentally conflicted with all past physical theory. Ever since the days of Newton and Leibniz physicists were use to thinking of natural phenomena in terms of continuous processes and the notion of nature operating on discrete scales had been inconceivable.

**1905** Albert Einstein (Nobel prize 1921) is able to explain the *photoelectric effect* by introducing the *photon* theory of light. Letting $c$ denote the speed of light and $\lambda$ its *wave length*, he finds the following expression for the (magnitude of the) *momentum* $p$ of a photon

$$p = \frac{E}{c} = \frac{h\,\nu}{c} = \frac{h}{\lambda} \tag{10.22}$$

and thus establishes a connection to Planck's work. At first, his idea of light as a particle was widely rejected but became universally accepted once Robert Millikan (Nobel prize 1923) confirmed it experimentally in 1919.

**1924** Louis de Broglie (Nobel prize 1929) has the outrageous idea that, if light waves are also particles, then maybe particles such as electrons are also matter waves. For wavelength and momentum of a particle of mass $m$ and velocity $v$, he finds

$$\lambda = \frac{h}{m\,v} \quad \text{and} \quad p = \frac{h}{\lambda} \; . \tag{10.23}$$

This hypothetical *wave-particle duality* was later confirmed in 1927 when Clinton Davisson (Nobel prize 1937) and Lester Germer performed diffraction experiments with slow moving electrons.

**1925** Erwin Schrödinger (Nobel prize 1933) reasons that, if particles are also waves, then there ought to be an equation for matter waves. His work leads to the famous *Schrödinger equation*

$$i\,\hbar\,\tfrac{\partial}{\partial t}\Psi\big(\boldsymbol{x},t\big) = \hat{H}(t)\,\Psi\big(\boldsymbol{x},t\big) \tag{10.24}$$

which governs the behavior of a non-relativistic quantum particle at position $\boldsymbol{x}$ and time $t$ and which we will work with later on. Based on his result, Schrödinger develops the framework of *wave mechanics* (WM).

Initially, the physical significance of the *wave function* $\Psi(\boldsymbol{x},t)$ was subject to debate but Max Born (Nobel prize 1954) quickly realizes that, when *normalizing the wave function* such that

$$\big|\Psi\big(\boldsymbol{x},t\big)\big|^2 = \int \Psi\big(\boldsymbol{x},t\big)\Psi^*\big(\boldsymbol{x},t\big)d\boldsymbol{x} = 1 \; , \tag{10.25}$$

the squared magnitude $|\Psi(\boldsymbol{x},t)|^2$ can be understood as the *probability* for finding the particle at position $\boldsymbol{x}$ at time $t$. This insight is now known as the *Born rule*.

At the same time, Werner Heisenberg (Nobel prize 1932) fiddles with Fourier series in order to understand the spectrum of the hydrogen atom.

**1926** Assisted by Born and Pascual Jordan, Heisenberg further develops his work into *matrix mechanics* (MM).

Here are some fun facts: Heisenberg and Schrödinger had a bitter rivalry over whose framework was "better". Trying to settle the argument, Schrödinger himself proved that WM $\subset$ MM. John von Neumann ended their quarrel in 1932 by proving both frameworks to be equivalent; this is made explicit in the *Dirac notation* of quantum mechanics. Also, believe it or not, most physicists at the time favored Schrödinger's framework because, back then, matrices were hardly ever used in science whereas equations describing the dynamics of waves were established tools of the trade.

**1927** Heisenberg shows that wave-particle duality leads to the famous *uncertainty principle*. It establishes that position and momentum of a particle cannot both be measured arbitrarily precisely. The more precisely one is determined, the less precisely the other can be known. This insight conflicted with the atom model proposed by Nils Bohr (Nobel prize 1922) which, at the time, was the best classically inspired model; it thus became clear that classical mechanics could not account for the physics of the (sub)atomic world.

**1927** Wolfgang Pauli (Nobel prize 1945) formulates the *Pauli equation*, an extended Schrödinger equation for spin $1/2$ particles which accounts for interactions of a particle with external electromagnetic fields.

**1928** Paul Dirac (Nobel prize 1933) introduces the *Dirac equation*, a relativistic extension of the Schrödinger equation.

**1932** John von Neumann (who, for unknown reasons, never received a Nobel prize) rigorously formulates quantum mechanics as the theory of *linear operators* on *Hilbert spaces*. He also introduces the formalism of *density operators* and the *von Neumann equation*.

**1948** Richard Feynman (Nobel prize 1965) develops the *path integral formulation* of quantum mechanics which, too, is equivalent to MM and WM.

## 10.3 The Mathematics of Quantum Mechanics

At this point, we are well prepared to risk a deeper dive into the mathematics of quantum mechanics. Next, we therefore clarify the notion of Hilbert spaces, introduce the Dirac notation, and finally look into (only a few of) the details of wave- and matrix mechanics.

### *10.3.1 Hilbert Spaces*

Above, we said that von Neumann developed a mathematical formulation of quantum mechanics in terms of *linear operators* acting on *Hilbert space vectors*. This framework has become a standard in theoretical physics and is usually the one considered in texts on quantum computing. But what are Hilbert spaces?

Well, to be able to answer this question, we first need to recall the general concepts of *vector spaces* and *inner product spaces*. The following will be dense and abstract but lays a rigorous foundation for many of our upcoming claims and derivations. Readers familiar with abstract linear algebra may safely skip a couple of paragraphs. Readers not familiar with abstract linear algebra should not worry too much about the upcoming details; they are good to know but, from a bigger picture point of view, not essential for developing a first understanding of the principles of quantum mechanics and quantum computing.

A **vector space** $\big(V, \boxplus, \boxdot\big)$ over a *field* $\mathbb{F}$ of scalars $a, b, \ldots$ consist of the following components

$$V \;\Leftrightarrow\; \text{a set of vectors } \boldsymbol{x}, \boldsymbol{y}, \ldots \tag{10.26}$$

$$\boxplus \;\Leftrightarrow\; \text{an operation } V \times V \rightarrow V \tag{10.27}$$

$$\boxdot \;\Leftrightarrow\; \text{an operation } \mathbb{F} \times V \rightarrow V \tag{10.28}$$

and its element have to satisfy several **axioms** with respect to **vector addition** $\boxplus$ and **scalar multiplication** $\boxdot$.

The set $V$ has to be closed under vector addiction, that is if $\boldsymbol{x}, \boldsymbol{y} \in V$ then $\boldsymbol{x} \boxplus \boldsymbol{y} \in V$. Vector addition has to be *commutative*, that is $\boldsymbol{x} \boxplus \boldsymbol{y} = \boldsymbol{y} \boxplus \boldsymbol{x}$. Vector addition has to be *associative*, that is $\boldsymbol{x} \boxplus [\boldsymbol{y} \boxplus \boldsymbol{z}] = [\boldsymbol{x} \boxplus \boldsymbol{y}] \boxplus \boldsymbol{z}$. There has to exist an element in $V$ that is *neutral* with respect to vector addition, that is $\exists\ \boldsymbol{0} \in V : \forall\ \boldsymbol{x} \in V : \boldsymbol{x} \boxplus \boldsymbol{0} = \boldsymbol{x}$. Finally, for each element of $V$ there has to exist an *inverse* with respect to vector addition, that is $\forall\ \boldsymbol{x} \in V : \exists\ [\boxminus \boldsymbol{x}] \in V : \boldsymbol{x} \boxplus [\boxminus \boldsymbol{x}] = \boldsymbol{0}$.

The set $V$ has to be closed under scalar multiplication, that is if $a \in \mathbb{F}$ and $\boldsymbol{x} \in V$ then $a \boxdot \boldsymbol{x} \in V$. Scalar multiplication has to be *compatible with field multiplication*, that is $a \boxdot [b \boxdot \boldsymbol{x}] = (a \cdot b) \boxdot \boldsymbol{x}$. Scalar multiplication has to *distribute over vector addition*, that is $a \boxdot [\boldsymbol{x} \boxplus \boldsymbol{y}] = a \boxdot \boldsymbol{x} \boxplus a \boxdot \boldsymbol{y}$. Scalar multiplication has to *distribute over field addition*, that is $(a + b) \boxdot \boldsymbol{x} = a \boxdot \boldsymbol{x} \boxplus b \boxdot \boldsymbol{x}$. There has to exist an element in $\mathbb{F}$ that is *neutral* with respect to scalar multiplication, that is $\exists\ 1 \in \mathbb{F} : \forall\ \boldsymbol{x} \in V : 1 \boxdot \boldsymbol{x} = \boldsymbol{x}$.

Note that the operations $+$ and $\boxplus$ as well as $\cdot$ and $\boxdot$ perform very different things. While $+$ adds two scalars, $\boxplus$ adds two vectors and while $\cdot$ multiplies two scalars, $\boxdot$ multiplies a scalar and a vector. However, these differences are rarely made explicit and instead of expressions such as

$$a \boxdot \big[\boldsymbol{x} \boxplus \boldsymbol{y}\big] = a \boxdot \boldsymbol{x} \boxplus a \boxdot \boldsymbol{y} \tag{10.29}$$

most people usually write

$$a \cdot \big[\boldsymbol{x} + \boldsymbol{y}\big] = a \cdot \boldsymbol{x} + a \cdot \boldsymbol{y}\ . \tag{10.30}$$

We, too, will henceforth follow this convention and therefore refer to a vector space as $(V, +, \cdot)$ or, even simpler, just as $V$.

Also note that the notion of a vector space is almost all-encompassing and that specific instances of vector spaces are a dime a dozen. Here are but a few examples

$\mathbb{R}^n$ : the vector space of real valued $n$ vectors
$\mathbb{C}^n$ : the vector space of complex valued $n$ vectors
$\mathbb{R}^{m \times n}$ : the vector space of real valued $m \times n$ matrices
$\mathbb{C}^{m \times n}$ : the vector space of complex valued $m \times n$ matrices

$F[a, b]$ : the vector space of functions $f : [a, b] \subset \mathbb{R} \to \mathbb{R}$
$C[a, b]$ : the vector space of functions $f : [a, b] \subset \mathbb{R} \to \mathbb{C}$
$\vdots$

An important extension of the notion of a vector space over $\mathbb{F}$ is the idea of an **inner product space** $\big(V, +, \cdot, \langle\cdot, \cdot\rangle\big)$ over $\mathbb{F}$. This is a vector space equipped with an additional operation

$$\langle \cdot\,, \cdot\rangle : V \times V \to \mathbb{F} \tag{10.31}$$

called an **inner product**. For this operation, too, several axioms have to be obeyed. Yet, before we list them, we need to step back a little.

While most readers will be familiar with inner products of real valued vectors, the underlying field $\mathbb{F}$ of an inner product space can- and in quantum mechanics will be the field $\mathbb{C}$ of complex numbers. This has implications for the axioms behind the inner product which, in most areas of computer science, is often only considered with respect to the field $\mathbb{R}$. However, since $\mathbb{R} \subset \mathbb{C}$, the following "simply" generalizes what most of us likely know already.

In an inner product space $V$ over $\mathbb{F}$, the inner product maps every pair of vectors $(\boldsymbol{x}, \boldsymbol{y}) \in V \times V$ to a scalar $\langle \boldsymbol{x}, \boldsymbol{y}\rangle \in \mathbb{F}$. It must be *conjugate symmetric* such that

$$\langle \boldsymbol{x}, \boldsymbol{y}\rangle = \langle \boldsymbol{y}, \boldsymbol{x}\rangle^* \tag{10.32}$$

and *sesquilinear* which means that, for $\boldsymbol{x}, \boldsymbol{y} \in V$ and $c \in \mathbb{F}$, it has to obey

$$\langle \boldsymbol{x}, \boldsymbol{y} + \boldsymbol{z}\rangle = \langle \boldsymbol{x}, \boldsymbol{z}\rangle + \langle \boldsymbol{y}, \boldsymbol{z}\rangle \tag{10.33}$$

$$\langle \boldsymbol{x} + \boldsymbol{y}, \boldsymbol{z}\rangle = \langle \boldsymbol{x}, \boldsymbol{y}\rangle + \langle \boldsymbol{x}, \boldsymbol{z}\rangle \tag{10.34}$$

$$\langle c\,\boldsymbol{x}, \boldsymbol{y}\rangle = c^*\langle \boldsymbol{x}, \boldsymbol{y}\rangle \tag{10.35}$$

$$\langle \boldsymbol{x}, c\,\boldsymbol{y}\rangle = c\,\langle \boldsymbol{x}, \boldsymbol{y}\rangle\ . \tag{10.36}$$

Finally, it has to be *positive definit*e such that

$$\langle \boldsymbol{x}, \boldsymbol{x}\rangle \geq 0 \tag{10.37}$$

$$\langle \boldsymbol{x}, \boldsymbol{x}\rangle = 0 \quad \Leftrightarrow \quad \boldsymbol{x} = \boldsymbol{0} \tag{10.38}$$

and we emphasize that conjugate symmetry implies $\langle \boldsymbol{x}, \boldsymbol{x}\rangle \in \mathbb{R}$ regardless of whether we have $\mathbb{F} = \mathbb{R}$ or $\mathbb{F} = \mathbb{C}$.

Inner product spaces are a dime a dozen, too. Here are but three specific examples for how to define appropriate inner products.

For vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ in the vector space $\mathbb{C}^n$ of complex valued $n$ vectors, we may define the inner product as

$$\langle \boldsymbol{x}, \boldsymbol{y}\rangle = \sum_{j=1}^{n} x_j^* y_j = \boldsymbol{x}^\dagger \boldsymbol{y}\ . \tag{10.39}$$

For matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ in the vector space $\mathbb{C}^{m \times n}$ of complex valued $m \times n$ matrices, we may define the inner product as

$$\langle \boldsymbol{X}, \boldsymbol{Y} \rangle = \sum_{j=1}^{m} \sum_{k=1}^{n} X^*_{jk} Y_{jk} = \mathrm{tr}\big[\boldsymbol{X}^\dagger \boldsymbol{Y}\big] . \tag{10.40}$$

And, the vector space $L^2$ of *square integrable functions* , that is the vector space of functions $f : \mathbb{R}^n \to \mathbb{C}$ for which we have

$$\int \big| f(\boldsymbol{x}) \big|^2 d\boldsymbol{x} < \infty , \tag{10.41}$$

can be equipped with the following inner product

$$\langle f, g \rangle = \int f^*(\boldsymbol{x})\, g(\boldsymbol{x})\, d\boldsymbol{x} . \tag{10.42}$$

Note what we did here. We gave an example of a vector space formed by a specific set of functions and turned it into an inner product space. Indeed, it a straightforward exercise for the reader to verify that the set $L^2 = \big\{ f : \mathbb{R}^n \to \mathbb{C} \mid f \text{ obeys } (10.41) \big\}$ meets all the axioms of a vector space. What is interesting about $L^2$ is that its elements are continuous functions over the infinitely large domain $\mathbb{R}^n$. We may therefore say that $L^2$ is an *infinite dimensional vector space*.

Finally, we declare that a vector space $V$ is said to be a **complete vector space** if every Cauchy sequence in $V$ converges in $V$.

This, too, is an admittedly abstract concept which is best explained by means of a counter example. Consider therefore the set of polynomials

$$P(x) = \sum_{j=0}^{\infty} a_j\, x^j . \tag{10.43}$$

This set forms an infinite dimensional vector space over the field $\mathbb{R}$ but it is not a complete space because, for $J \to \infty$, the following series of polynomials

$$P_J(x) = \sum_{j=0}^{J} \frac{1}{j!}\, x^j \tag{10.44}$$

converges to the exponential function $e^x$ which is not a polynomial.

The vector space of square integrable functions, on the other hand, is a complete vector space because the functions it contains are severely restricted in that $f(\boldsymbol{x})$ can not grow beyond all bounds for $\boldsymbol{x} \to \pm\infty$. But this is to say that $(L^2, +, \cdot, \langle \cdot, \cdot \rangle)$ is an example of a *complete inner product space*.

Why did we have to go through all this hassle? Because we can now finally provide the following definition.

**Definition 10.3** A **Hilbert space** $\mathbb{H}$ over a field $\mathbb{F}$ is a complete inner product space of possibly infinite dimensionality.

This notion plays a crucial role in quantum mechanics because, in order to be physical, solutions to Schrödinger equations must be objects in Hilbert spaces. Indeed, looking at the Born rule in (10.25), we realize that it demands that quantum mechanical wave functions are a special kind of square integrable functions.

Now here is the crazy thing: There also are discrete finite dimensional quantum systems. Especially in quantum computing, we are dealing with systems whose states correspond to finite dimensional vectors over $\mathbb{C}$. The respective vector spaces are still called Hilbert spaces but, to understand those, we would not have had to go through the above. However, all our work will pay off as it allows for very general statements about quantum mechanics which can be specialized to quantum computing.

### *10.3.2 The Dirac Notation*

Above, we just saw that continuous *functions are vectors* in infinite dimensional spaces. We usually denote a whole function by $f$ and its infinitely many entries by $f(\boldsymbol{x})$ which we read as “the value of $f$ at $\boldsymbol{x}$”. This is noteworthy because, in the preceding chapters of this book, we have been working with the more familiar concepts of discrete, finite dimensional vectors or Euclidean vectors. We usually denote a whole such vector as $\boldsymbol{v}$ and its finitely many entries as $v_j$ which we read as “the value of $\boldsymbol{v}$ in dimension $j$”.

In quantum mechanics, both these kinds of objects, continuous functions $f$ and discrete vectors $\boldsymbol{v}$, may represent the states of quantum systems. An example of a continuous quantum state would be the *position state* of a particle and an example of a discrete quantum state be the *spin state* of an electron. While these are different kinds of quantum states, the principles of quantum mechanics apply to both of them. In mathematical formulas, physicists therefore treat them equally. They simply as well as boldly write $|\psi\rangle$ to signify continuous and discrete states alike.

That is, physicists apply a notation due to Dirac where everything inside of “$|$” and “$\rangle$” denotes a state vector. For instance, we may write $|x\rangle$ to express a point in space or we may write $|\uparrow\rangle$ and $|\downarrow\rangle$ to refer to the possible spin states of an electron. In any case, we may use abstract symbols like these in mathematical equations.

Due to its symbolic flexibility, the Dirac notation is the de-facto notation in quantum mechanics and quantum computing but it definitely needs getting used to. Next, we define its key features and concepts and then try to warm up to its use.

**Definition 10.4** A **ket vector** $|\psi\rangle$ represents a state in a Hilbert space $\mathbb{H}$ over $\mathbb{C}$. Discrete and continuous ket vectors are understood as

$$|\psi\rangle = \begin{bmatrix} \vdots \\ \psi_{j-1} \\ \psi_j \\ \psi_{j+1} \\ \vdots \end{bmatrix} \quad \text{and} \quad |\psi\rangle = \begin{bmatrix} \vdots \\ \psi(\boldsymbol{x} - d\boldsymbol{x}) \\ \psi(\boldsymbol{x}) \\ \psi(\boldsymbol{x} + d\boldsymbol{x}) \\ \vdots \end{bmatrix}, \tag{10.45}$$

respectively.

**Definition 10.5** A **bra vector** $\langle\psi|$ is the conjugate transpose of a ket vector. Discrete and continuous bra vectors are understood as

$$\langle\psi| = \left[\cdots, \psi_{j-1}^*, \psi_j^*, \psi_{j+1}^*, \cdots\right] \tag{10.46}$$

and

$$\langle\psi| = \left[\cdots, \psi^*(\boldsymbol{x} - d\boldsymbol{x}), \psi^*(\boldsymbol{x}), \psi^*(\boldsymbol{x} + d\boldsymbol{x}), \cdots\right], \tag{10.47}$$

respectively.

But why would $\langle\psi|$ be called a bra vector and $|\psi\rangle$ be called a ket vector? Because of the following!

**Definition 10.6** The *inner product* or **bra-ket** of a bra- and a ket vector yields a scalar. For the discrete case, it is given by

$$\langle\psi\,|\,\phi\rangle = \sum_j \psi_j^* \phi_j \tag{10.48}$$

and, for the continuous case, it amounts to

$$\langle\psi\,|\,\phi\rangle = \int \psi^*(\boldsymbol{x})\,\phi(\boldsymbol{x})\,d\boldsymbol{x}\ . \tag{10.49}$$

Another useful concept is the outer product of a ket- and a bra vector. In previous chapters, we have already worked with out products of finite dimensional vectors and saw that they result in matrices. However, since state vectors in quantum mechanics may be infinite dimensional, we will henceforth avoid talking about *matrices* but switch to the more general notion of *operators* which may act on finite- and infinite state vectors.

**Definition 10.7** The *outer product* or **ket-bra** of a ket- and a bra vector yields an operator $|\psi\rangle\langle\phi|$ whose action on another ket vector $|\xi\rangle$ is as follows

$$|\xi\rangle \mapsto |\psi\rangle\langle\phi\,|\,\xi\rangle\,. \tag{10.50}$$

In the following, we look at a few examples of how the Dirac notation compares to more conventional notation. For no other reason than simplicity, we only consider expressions which involve Euclidean vectors and corresponding operators.

For starters, here are a few stand alone expressions in conventional- and in Dirac notation:

$$\begin{aligned}
\boldsymbol{v} &\quad\Leftrightarrow\quad |v\rangle && (10.51)\\
\boldsymbol{v}^\dagger &\quad\Leftrightarrow\quad \langle v| && (10.52)\\
\boldsymbol{u}\boldsymbol{v}^\dagger &\quad\Leftrightarrow\quad |u\rangle\langle v| && (10.53)\\
\boldsymbol{u}^\dagger\boldsymbol{v} &\quad\Leftrightarrow\quad \langle u \mid v\rangle && (10.54)\\
\boldsymbol{u}^\dagger\big[\boldsymbol{v}\boldsymbol{w}^\dagger\big]\boldsymbol{x} &\quad\Leftrightarrow\quad \langle u \mid v\rangle\langle w \mid x\rangle && (10.55)\\
\boldsymbol{u}^\dagger\boldsymbol{A}\,\boldsymbol{v} &\quad\Leftrightarrow\quad \langle u|\,A\,|v\rangle\ . && (10.56)
\end{aligned}$$

Next, here are a few equations where we let $\boldsymbol{x} \in \mathbb{C}^n$ be some complex vector and $\boldsymbol{e}_j \in \mathbb{R}^n$ denote a standard basis vector:

$$\begin{aligned}
\big[\boldsymbol{A}\boldsymbol{x}\big]^\dagger &= \boldsymbol{x}^\dagger\boldsymbol{A}^\dagger & \Leftrightarrow\quad \big[A|x\rangle\big]^\dagger &= \langle x|A^\dagger && (10.57)\\
\boldsymbol{x} &= \sum_j x_j\,\boldsymbol{e}_j & \Leftrightarrow\quad |x\rangle &= \sum_j x_j\,|j\rangle && (10.58)\\
\boldsymbol{x}^\dagger\boldsymbol{e}_k &= \sum_j x_j^*\,\boldsymbol{e}_j^\dagger\boldsymbol{e}_k & \Leftrightarrow\quad \langle x \mid k\rangle &= \sum_j x_j^*\,\langle j \mid k\rangle && (10.59)\\
\boldsymbol{x} &= \sum_j \boldsymbol{e}_j\,x_j & \Leftrightarrow\quad |x\rangle &= \sum_j |j\rangle\,x_j && (10.60)\\
&= \sum_j \boldsymbol{e}_j\big[\boldsymbol{e}_j^\dagger\,\boldsymbol{x}\big] & &= \sum_j |j\rangle\langle j \mid x\rangle && (10.61)\\
&= \left[\sum_j \boldsymbol{e}_j\boldsymbol{e}_j^\dagger\right]\boldsymbol{x} & &= \left[\sum_j |j\rangle\langle j|\right]|x\rangle\ . && (10.62)
\end{aligned}$$

### *10.3.3 Axioms and Some Consequences*

Now, to make a long story short, the next paragraph states the **axioms of quantum mechanics** in plain English. Yes, all the following statements are axiomatic and, according to current physical theory, not up for debate. To this day, humankind has performed many thousands of quantum mechanical experiments all of which can be explained within the following framework.

The state of any quantum mechanical system can be represented as a unit vector in an appropriate Hilbert space $\mathbb{H}$ over $\mathbb{C}$. Any observable physical property (spin, energy, position, momentum, ...) of a quantum system corresponds to a Hermitian operator on said space. A measurement performed on a given quantum state results in one of the eigenvalues of the respective operator and *collapses* the state to the corresponding eigenstate. The outcome of a quantum measurement is *probabilistic* rather than deterministic. However, once a measurement has caused collapse to an eigenstate, any further measurement will return the same value and leave the current state intact. Finally, if a quantum system consist of two subsystems with individual states in two Hilbert spaces $\mathbb{H}_1$ and $\mathbb{H}_2$, then the state of the joint system corresponds to a vector in the tensor product space $\mathbb{H}_1 \otimes \mathbb{H}_2$.

But why? Nobody knows! It might be futile to ask *why* this framework can describe nature on subatomic scales, it just can. It might be futile to ask *why* observables are operators and measurements are eigenvalues of operators, they just are. Asking these questions is akin to asking *why* Newton's 2nd law $\boldsymbol{F} = m\,\boldsymbol{a}$ describes the mechanics of our macroscopic world, it just does. Humanity does not know why nature follows certain laws, she simply does.

One axiom is still missing from our list, namely the one which describes how quantum states evolve when left alone and unmeasured. Let us therefore consider *isolated* or *closed* quantum systems which do not interact with the outside world.

In the *Schrödinger picture* of quantum mechanics, the evolution of such a system is governed by the Schrödinger equation in (10.24) where $\hbar = h/2\pi$ is the *reduced Planck constant* and $\hat{H}(t)$ is a Hermitian operator called the system's *Hamiltonian*.

Yes, this name is deliberate and refers to Hamiltonians such as we have already seen in Chap. 9. While the ones we saw there were of simple diagonal structure, the ones occurring in Schrödinger equations can have a more intricate structure but they will always be Hermitian and their eigenvalues will always represent possible outcomes of a measurement of the energy of the system at hand. Note therefore that different kinds of quantum systems will have different Hamiltonians and that one of the interesting or beautiful aspects of quantum mechanics is that the Schrödinger equation works irrespective of the specific choice of $\hat{H}(t)$.

Speaking about choices of the Hamiltonian, in its general form, the Schrödinger equation in (10.24) is way too general for our purposes in this chapter. In what follows, we therefore focus on the important special case of a *time independent Hamiltonian*

$$\forall\, t \;:\; \hat{H}(t) = \hat{H}\;. \tag{10.63}$$

We would not have to do this, but if we did not, the complexity of the upcoming equations would explode and we had to work with mathematical tools far beyond the scope of this book.

Written in Dirac notation, the Schrödinger equation for a system with constant Hamiltonian reads

$$i\,\hbar\,\frac{d}{dt}\,\big|\,\psi(t)\big\rangle = \hat{H}\,\big|\,\psi(t)\big\rangle \tag{10.64}$$

or, equivalently

$$\frac{d}{dt}\,\big|\,\psi(t)\big\rangle = -\frac{i}{\hbar}\,\hat{H}\,\big|\,\psi(t)\big\rangle\,. \tag{10.65}$$

If we next assume that the evolution of the quantum state starts at time $t = 0$, the solution to this fairly "simple" ordinary differential equation is given by

$$\big|\,\psi(t)\big\rangle = \int_0^t -\frac{i}{\hbar}\hat{H}\,\big|\,\psi(\tau)\big\rangle d\tau = e^{-i\,t\,\hat{H}/\,\hbar}\,\big|\,\psi(0)\big\rangle = U(t)\,\big|\,\psi(0)\big\rangle\,. \tag{10.66}$$

Looking at this solution, we need to be aware of two things. First of all, we recall or bluntly state that, if $A$ is any Hilbert space operator, then $A^0 = I$ and its exponential is defined as follows

$$e^A = \sum_{k=0}^{\infty} \frac{A^k}{k!}\,. \tag{10.67}$$

Second of all, it turns out that the operator which we introduced in (10.66), namely

$$U(t) = e^{-i\,t\,\hat{H}/\,\hbar} \tag{10.68}$$

is a *unitary operator*. We will rigorously prove this claim in a later chapter but can already verify it by showing that the dynamics encoded in the Schrödinger equation have no effect on the norm of quantum mechanical state vectors.

To prepare ourselves for this verification, we note that the squared norm of a ket $|\psi(t)\rangle$ is given by $\langle\psi(t)\mid\psi(t)\rangle$. The expression $\frac{d}{dt}\langle\psi(t)\mid\psi(t)\rangle$ therefore quantifies how this norm changes at time $t$. Next, we analyze an expression which involves this temporal derivative. In that analysis, we will use the following fact

$$i\,\hbar\,\frac{d}{dt}\,\big|\,\psi(t)\big\rangle = \hat{H}\,\big|\,\psi(t)\big\rangle \quad\Leftrightarrow\quad -i\,\hbar\,\frac{d}{dt}\big\langle\psi(t)\,\big| = \big\langle\psi(t)\,\big|\,\hat{H}^\dagger\,. \tag{10.69}$$

The expression we are going to analyze is $i\,\hbar\,\frac{d}{dt}\langle\psi(t)\mid\psi(t)\rangle$ and, using the chain rule, we find

$$i\,\hbar\,\frac{d}{dt}\big\langle\psi(t)\,\big|\,\psi(t)\big\rangle = i\,\hbar\left[\left[\frac{d}{dt}\big\langle\psi(t)\,\big|\right]\big|\,\psi(t)\big\rangle + \big\langle\psi(t)\,\big|\left[\frac{d}{dt}\,\big|\,\psi(t)\big\rangle\right]\right] \tag{10.70}$$

$$= \left[i\,\hbar\,\frac{d}{dt}\big\langle\psi(t)\,\big|\right]\big|\,\psi(t)\big\rangle + \big\langle\psi(t)\,\big|\left[i\,\hbar\,\frac{d}{dt}\,\big|\,\psi(t)\big\rangle\right] \tag{10.71}$$

Plugging in the right hand sides of the two equations in (10.69) then quickly yields

$$i\,\hbar\,\frac{d}{dt}\big\langle\psi(t)\,\big|\,\psi(t)\big\rangle = -\big\langle\psi(t)\,\big|\,\hat{H}^\dagger\,\big|\,\psi(t)\big\rangle + \big\langle\psi(t)\,\big|\,\hat{H}\,\big|\,\psi(t)\big\rangle \tag{10.72}$$

$$= -\big\langle\psi(t)\,\big|\,\hat{H}\,\big|\,\psi(t)\big\rangle + \big\langle\psi(t)\,\big|\,\hat{H}\,\big|\,\psi(t)\big\rangle \tag{10.73}$$

$$= 0 \tag{10.74}$$

where we used that $\hat{H}$ is Hermitian with $\hat{H}^\dagger = \hat{H}$.

In other words, we find that the norm of a quantum state which evolves under the Schrödinger equation does not change over time. But if this is so and if furthermore

$$\big|\,\psi(t)\big\rangle = U(t)\,\big|\,\psi(0)\big\rangle \tag{10.75}$$

according to (10.66), then $U(t)$ must be unitary because, for the norms of ket $|\psi(t)\rangle$ and ket $|\psi(0)\rangle$ to be the same, we must have

$$\big\langle\psi(t)\,\big|\,\psi(t)\big\rangle = \big\langle\psi(0)\,\big|\,U^\dagger(t)\,U(t)\,\big|\,\psi(0)\big\rangle = \big\langle\psi(0)\,\big|\,I\,\big|\,\psi(0)\big\rangle = \big\langle\psi(0)\,\big|\,\psi(0)\big\rangle\,. \tag{10.76}$$

This insight is backed up by the following general theorem whose formal proof is easy and therefore left as an exercise.

**Theorem 10.1** *Unitary operators preserve inner products and norms.*

Next, we return to the axiomatic statements that observable properties of quantum systems correspond to Hermitian operators, that measurements of a property yield an eigenvalue of the associated operator, and that measurements are non-deterministic but result in certain eigenvalues according to certain probabilities.

Since quantum mechanical measurements are probabilistic, it is common to ask for the *expected value* of a time-independent *observable* $A$ for a state $|\psi(t)\rangle$ at some time $t$. In the Schrödinger picture, this expected value is given by

$$\langle A\rangle = \big\langle\psi(t)\,\big|A\big|\,\psi(t)\big\rangle \tag{10.77}$$

$$= \big\langle\psi(0)\,\big|U^\dagger(t)\,A\,U(t)\big|\,\psi(0)\big\rangle \tag{10.78}$$

$$\equiv \big\langle\psi(0)\,\big|A(t)\big|\,\psi(0)\big\rangle \tag{10.79}$$

which is also the expected value of a time-dependent observable $A(t)$ at time $t$ in the *Heisenberg picture* of quantum mechanics. This picture considers state vectors to be time-independent and observables to evolve according to the *Heisenberg equation of motion*. Since we do not want to open yet another can of worms, we will not state this equation but simply note that the above hints at the equivalence of the Schrödinger picture and the Heisenberg picture.

But why is $\langle A \rangle$ as defined above an expected value? Well, to answer this question in full, we would have to reason about properties of continuous observables (position, momentum, ...) and of discrete observables (energy, spin, ...). However, since the former would require mathematical techniques far beyond our scope, we just consider the discrete case.

To begin with, we recall [1, 2] that every discrete Hermitian operator $A = A^\dagger$ admits a spectral decomposition

$$A = \sum_j a_j \,\big|\, a_j \big\rangle\big\langle a_j \,\big| \tag{10.80}$$

where the $a_j$ and $|a_j\rangle$ are its eigenvalues and eigenstates. Regarding the eigenvalues, we have the following theorem whose proof we leave as an exercise.

**Theorem 10.2** *The eigenvalues of Hermitian operators are real.*

Regarding the eigenstates, we observe that they are guaranteed to be orthonormal

$$\big\langle a_j \,\big|\, a_k \big\rangle = \delta_{jk} \tag{10.81}$$

and therefore also guaranteed to meet the completeness relation

$$\sum_j \big|\, a_j \big\rangle\big\langle a_j \,\big| = I \ . \tag{10.82}$$

With respect to a given observable $A$, any quantum state $|\psi\rangle$ can therefore be expanded as

$$\big|\, \psi \big\rangle = I \,\big|\, \psi \big\rangle = \sum_j \big|\, a_j \big\rangle\big\langle a_j \,\big|\, \psi \big\rangle \equiv \sum_j \big|\, a_j \big\rangle \, \psi_j \tag{10.83}$$

where $\psi_j = \langle a_j \mid \psi \rangle$ is the length of the projection of state $|\psi\rangle$ onto eigenstate $|a_j\rangle$ otherwise known as the *overlap* of $|\psi\rangle$ and $|a_j\rangle$.

Next, we note that the Born rule demands that quantum states must obey the following normalization

$$1 = \big\langle \psi \,\big|\, \psi \big\rangle = \sum_k \sum_j \psi_k^* \big\langle a_k \,\big|\, a_j \big\rangle \, \psi_j \ . \tag{10.84}$$

Using (10.81), this becomes

$$1 = \sum_k \sum_j \psi_k^* \, \psi_j \, \delta_{kj} = \sum_j \psi_j^* \, \psi_j = \sum_j |\psi_j|^2 \tag{10.85}$$

where each $|\psi_j|^2 \geq 0$. Now, since the $|\psi_j|^2$ are greater than or equal to zero and sum to one, we may understand them as probabilities. Since we furthermore have

$$\left| \psi_j \right|^2 = \left| \langle a_j \mid \psi \rangle \right|^2 \tag{10.86}$$

we also realize that these probabilities are proportional to the overlap of $|\psi\rangle$ and $|a_j\rangle$.

We already said that it is axiomatically postulated that a measurement of a property $A$ of a quantum state $|\psi\rangle$ causes the state to collapse to an eigenstate $|a_j\rangle$ and yields the corresponding eigenvalue $a_j$. What we did not yet say is that the probability for observing state collapse $|\psi\rangle \mapsto |a_j\rangle$ and measurement result $a_j$ is axiomatically postulated to be

$$p\Big( \left| \psi \right\rangle \mapsto \left| a_j \right\rangle \Big) = p\big(a_j\big) = \left| \langle a_j \mid \psi \rangle \right|^2 . \tag{10.87}$$

Finally, all of the above therefore allows us to rewrite the definition in (10.77) as

$$\begin{aligned}
\langle A \rangle &= \langle \psi \mid A \mid \psi \rangle && (10.88)\\
&= \sum_j a_j \, \langle \psi \mid a_j \rangle \langle a_j \mid \psi \rangle && (10.89)\\
&= \sum_j a_j \, \langle a_j \mid \psi \rangle^* \langle a_j \mid \psi \rangle && (10.90)\\
&= \sum_j a_j \, \left| \langle a_j \mid \psi \rangle \right|^2 && (10.91)\\
&= \sum_j a_j \, p\big(a_j\big) && (10.92)
\end{aligned}$$

and thus to obtain an expression which we now clearly recognize as an expected value.

## 10.4 Outlook and Further Reading

The content of this chapter was dense and abstract. Most introductory quantum mechanics texts begin by describing quantum mechanical experiments (the legendary double slit experiment, light polarization experiments, spin measurements, …) to expose *quantum weirdness* and develop the mathematical framework from there. We, however, deliberately left out the first step and dove right into mathematics.

Indeed, our philosophy for all that is to follow will be to abstract away from physical realizations of quantum systems and instead to focus on their mathematical essence. Our goal is to hit home that quantum mechanics and therefore quantum computing is just complex linear algebra in very high dimensional spaces and that quantum computing thus needs a different kind of thinking than we are used to from digital computing.

Nevertheless, it will be helpful to acquaint ourselves with at least some aspects of quantum weirdness and we are going to do just that in the next chapter. We can actually do this even if we ignore real world physical phenomena. Our strategy will be to axiomatically take the Schrödinger equation for granted and to consider several hypothetical scenarios with comparatively simple Hamiltonians. Solving the Schrödinger equation for these scenarios will reveal that it dictates solutions which do indeed defy common everyday intuition.

Finally, we are well aware that readers who have not yet been exposed to quantum mechanics may be left puzzled after reading this chapter. Those who are in need of or are interested in a longer, gentler introduction to the topic might want to read the quite recent book by Susskind and Friedman [3]. It covers the theoretical minimum of quantum mechanics in a humorous manner without sacrificing rigor which happens way too often in quantum texts for wider audiences.

Also, our above account of the principles of quantum mechanics was rigorous to the extent that is possible in a book whose main topic is information processing rather than physics. We necessarily had to gloss over much of the depth and breadth of the theory of quantum mechanics. Those interested in more extensive mathematical treatments may read the outstanding classic books by Shankar [4] and Sakurai [5] the latter of which was thoroughly revised just recently.

## 10.5 Exercises

**10.1** Use the inner product axioms in Eqs. (10.32)–(10.38) to prove $\langle \boldsymbol{x}, \boldsymbol{0} \rangle = 0$ for all $\boldsymbol{x}$.

**10.2** Implement a `numpy` function that computes the inner product of any two vectors $|x\rangle, |y\rangle \in \mathbb{C}^n$, namely

$$\langle x \mid y \rangle = \sum_{j=1}^{n} x_j^* \, y_j \ .$$

**Note:** This is not difficult if you invoke the `numpy` function `vdot`. The crucial point is that `vdot` behaves differently than the likely more familiar function `dot`.

Given your implementation, consider the following complex vector, complex number, and real number

$$|x\rangle = e^{i\frac{\pi}{4}} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad c = 2 + i \qquad r = 2$$

and compute the values of the following expressions

$$\langle x \,|\, x\rangle,\ \ \langle c\,x \,|\, x\rangle,\ \ c^*\langle x \,|\, x\rangle,\ \ \langle x \,|\, c\,x\rangle,\ \ c\,\langle x \,|\, x\rangle,\ \ \langle r\,x \,|\, x\rangle,\ \ \langle x \,|\, r\,x\rangle\,. \tag{10.93}$$

Take a moment to think about what you observe. Is there anything noteworthy?

**10.3** Prove Theorem 10.1. That is, consider two ket vectors $|\psi\rangle$ and $|\phi\rangle = A\,|\psi\rangle$ where $A$ is a unitary operator and show that

$$\langle \psi \mid \psi\rangle = \langle \phi \mid \phi\rangle \tag{10.94}$$

which immediately implies that $\| |\psi\rangle \| = \| |\phi\rangle \|$.

**10.4** Prove Theorem 10.2. Proceed as follows: Let $(\lambda, |u\rangle)$ be any eigenvalue / eigenvector pair of some (possibly complex valued) Hermitian operator $A$ and prove that the expressions $\langle u \mid u\rangle$ and $\langle u \mid A \,|\, u\rangle$ are both necessarily real. Then reason why this implies that $\lambda$ must be real.

**10.5** Prove that all the eigenvalues $\lambda$ of a unitary operator are of norm $|\lambda| = 1$. Reason why this implies that the norm of the determinant of a unitary operator is one.

**10.6** An operator $A$ is **normal** if it commutes with it conjugate transpose. That is, $A$ is normal, if

$$A\,A^\dagger = A^\dagger A\,. \tag{10.95}$$

Prove that Hermitian operators as well as unitary operators are always normal.

**10.7** It is obvious that the sum of any two Hermitian operators is always Hermitian, right? But what about the sum of two unitary operators?

**10.8** Let $A$ and $B$ be two unitary operators. Is their product $AB$ unitary, too? Never or always or sometimes under certain circumstances?

**10.9** Let $A$ and $B$ be two Hermitian operators. Is their product $AB$ Hermitian, too? Never or always or sometimes under certain circumstances?

**10.10** Let $A$ and $B$ be two unitary operators and show that the operator $A \otimes B$ is unitary.

**10.11** Here are two outstandingly important vectors

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{and} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}\,. \tag{10.96}$$

Given these, consider the following two vectors

$$|\alpha\rangle = \alpha_0 \, |0\rangle + \alpha_1 \, |1\rangle \tag{10.97}$$

$$|\beta\rangle = \beta_0 \, |0\rangle + \beta_1 \, |1\rangle \tag{10.98}$$

and assume their entries are normalized like this

$$|\alpha_0|^2 + |\alpha_1|^2 = 1 \tag{10.99}$$

$$|\beta_0|^2 + |\beta_1|^2 = 1 \; . \tag{10.100}$$

Finally, consider the following vector

$$|\gamma\rangle = |\alpha\rangle \otimes |\beta\rangle = \sum_{j=0}^{1} \sum_{k=0}^{1} \alpha_j \beta_k \, |j\rangle \otimes |k\rangle \equiv \sum_{j=0}^{1} \sum_{k=0}^{1} \gamma_{jk} \, |j\rangle \otimes |k\rangle \tag{10.101}$$

and prove that, under the normalizations in (10.99) and (10.100), the following is true

$$\sum_{j=0}^{1} \sum_{k=0}^{1} |\gamma_{jk}|^2 = 1 \; . \tag{10.102}$$

## References

1. Strang, G.: Linear Algebra and Its Applications, 4th edn. Cenage Learning (2005)
2. Golub, G., van Loan, C.: Matrix Computations, 4th edn. Johns Hopkins University Press (2012)
3. Susskind, L., Friedman, A.: Quantum Mechanics: The Theoretical Minimum. Penguin (2015)
4. Shankar, R.: Principles of Quantum Mechanics, 2nd edn. Springer (1994)
5. Sakurai, J., Napolitano, J.: Modern Quantum Mechanics, 3rd edn. Cambridge University Press (2020)

# Chapter 11
# Exploring Quantum Weirdness

## 11.1 Introduction

In the previous chapter, we introduced the mathematics behind quantum mechanics and saw that quantum systems are represented in terms of abstract states in complex valued vector spaces. We also saw that state changes are due to quantum mechanical operators and looked at a few of their general properties. Our discussion was woefully abstract and did not consider any specific example of a quantum system, its possible behaviors, and observable properties. In this chapter, we will make up for this and solve the Schrödinger equation for very simple systems and see what we can learn from that.

We therefore recall that the Schrödinger equation is a differential equation which describes how a quantum system with a known Hamiltonian evolves over time. If we assume a time-independent Hamiltonian, then the original Schrödinger equation reads

$$i\,\hbar\,\frac{\partial}{\partial t}\Psi\big(\boldsymbol{x},t\big) = \hat{H}\,\Psi\big(\boldsymbol{x},t\big) \tag{11.1}$$

and its solutions $\Psi(\boldsymbol{x},t)$ are called wave functions. They characterize the behavior of a quantum particle at location $\boldsymbol{x} \in \mathbb{R}^3$ and at time $t \in \mathbb{R}$ in the following sense: If a wave function is normalized such that

$$\big|\Psi\big(\boldsymbol{x},t\big)\big|^2 = \int \Psi\big(\boldsymbol{x},t\big)\Psi^*\big(\boldsymbol{x},t\big)d\boldsymbol{x} = 1\;, \tag{11.2}$$

then the squared magnitude $|\Psi(\boldsymbol{x},t)|^2$ represents the probability of finding the particle at position $\boldsymbol{x}$ at time $t$. In this chapter, we will explore what this may mean and what kind of *weird effects* it may entail.

However, we shall stick with an abstract point of view but this time to simplify matters. Indeed, here is a bold claim: Theoretical physics is not so much about nature itself but rather about *plausible* mathematical *models* or manageable abstractions of natural processes. In this spirit, we will next familiarize ourselves with *quantum weirdness* by studying three almost trivial, highly abstracted yet instructive settings.

To be specific, we will study the behavior of a 1D quantum particle which moves in a 1D world but is subject to different 1D energy fields which impact its behavior. While such a particle and such environments are hypothetical (the real world is three-dimensional), they allow us to work with simple Schrödinger equations whose solutions are still epistemologically interesting.

The three setting we shall consider are (1) a particle trapped within an infinite potential well, (2) a particle in a finite potential box within an infinite potential well, and (3) a particle facing a potential barrier within an infinite potential well. Yes, there is quite some jargon here and we promise to clarify it later on. Right now, the important point is that there are three scenarios of different characteristics. Hence, when solving the scenario specific Schrödinger equations, we will have to consider scenario specific Hamiltonians.

We will therefore discuss the notion of the Hamiltonian of a moving particle which provides with an opportunity to have another look at quantum operators on continuous and therefore infinite dimensional state spaces. These may again appear puzzling to those who have never seen them before but once again allow for a wider perspective on concepts most of us will be familiar with.

Given the most general, least detailed form of the Hamiltonian of a moving particle, we can already begin solving the Schrödinger equation which will be an exercise in differential equation solving [1, 2]. Working with the *separation of variables* ansatz, we will find that there are *two aspects* to the wave function of a quantum particle, namely a scenario independent temporal one and a scenario dependent spatial one. Especially our solutions to the latter one will reveal that quantum systems behave differently than classical systems. Most notably perhaps, we will find that their *energies are quantized* and that they are capable of *quantum tunneling* through energy barriers. We will also find that there exist various solutions to a scenario specific Schrödinger equation. Since any linear combination of different solution to a differential equations provides yet another solution, this implies that quantum systems can exist in a *superposition* of different states.

Let us conclude these introductory remarks with a couple of disclaimers. First of all, throughout this chapter we will be working with classical notation rather than with the Dirac notation as this makes it easier to recognize the gist behind the ideas we shall investigate. Second of all, there will be a lot of physical jargon and mathematical ideas which are not fundamentally essential to those whose main interest is in learning about quantum computing. However, we will encounter important concepts such as the *ground state* of a Hamiltonian which will reoccur in the next chapter. Third of all, it will turn out that the Schrödinger equation can hardly ever be solved symbolically, not even in simple settings such as those we will consider. This is not uncommon in the natural sciences where people often have to work with approximate numerical solutions because nature is too complex to accommodate our human desire for (reductionist) simplicity. In the code examples section, we will therefore look at how to numerically solve Schrödinger equations. This will reconnect our discussion to the discrete settings we studied in earlier chapters and lay another foundation for things still to come.

## 11.2 The Schrödinger Equation for a 1D Particle

As announced above, we next study three simplified scenarios involving simplified quantum systems. In each of our settings, we consider a quantum particle that moves through 1D space $x \in \mathbb{R}$ and time $t \in \mathbb{R}$. Assuming that the environment it lives in does not change with time, the Schrödinger equation which governs its behavior will thus simply read

$$i\,\hbar\,\frac{\partial}{\partial t}\Psi\big(x,t\big) = \hat{H}\,\Psi\big(x,t\big) \tag{11.3}$$

and we obviously first need to think about how the *time-independent* Hamiltonian operator $\hat{H}$ that occurs in this expression will look like.

When in motion, classical and quantum objects alike posses *kinetic energy* and *potential energy*. The former is due to their motion and the latter to their position in an environment which exerts forces caused by, say, gravitational- or electromagnetic fields. In quantum mechanics, these energies once again correspond to Hamiltonian operators. The general form of the Hamiltonian in (11.3) is thus

$$\hat{H} = \hat{T} + \hat{V} \tag{11.4}$$

where

$$\hat{V} = \hat{V}\big(x\big) \tag{11.5}$$

$$\hat{T} = \frac{\hat{p}^2}{2\,m} \tag{11.6}$$

are the **potential energy operator** and the **kinetic energy operator**, respectively. The latter involves the mass $m$ of a particle and is itself defined in terms of yet another operator, namely in terms of the **momentum operator**

$$\hat{p} = -i\,\hbar\frac{\partial}{\partial x}\;. \tag{11.7}$$

At this point, there already is a lot to unpack before we can continue. In what sense are $\hat{V}$, $\hat{T}$, and $\hat{p}$ operators? That is, why or how can we pretend to be able to add something like $\hat{V}(x)$ which looks like a function to an expression involving $\frac{\partial}{\partial x}$ which looks like an operation? Indeed, so far, we said that operators are possibly infinite dimensional matrices acting on objects in possibly infinite dimensional vector spaces. And this still holds true because we already know that we may think of continuous functions such $\Psi(x,t)$ as infinite dimensional vectors. However, to make sense of the above energy operators, we better interpret them in a more traditional sense, namely as generalized mathematical functions.

Recall therefore that we may think of a function as a procedure that takes some input (numbers, vectors, …) and produces some output (numbers, vectors, …). Operators generalize this concept in that they are procedures that take functions as inputs and produces functions as output. For instance, we are all familiar with the differentiation operator $\frac{d}{dx}$ which acts on a function $f(x)$ to produce its derivative $f'(x)$, aren't we? We also are familiar with the idea of doing algebra with operators. Just consider this example

$$\tfrac{d}{dx}\big(f(x)+g(x)\big) = \tfrac{d}{dx}f(x) + \tfrac{d}{dx}g(x) = f'(x) + g'(x)\ . \tag{11.8}$$

It is in this sense that $\hat{H} = \hat{T} + \hat{V}$ operates on the wave function $\Psi(x,t)$. If we plug (11.7) into (11.6) and then plug (11.6) and (11.5) into (11.4), the right hand side of the Schrödinger equation in (11.3) becomes

$$\left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + \hat{V}(x)\right]\Psi\big(x,t\big) = \left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\right]\Psi\big(x,t\big) + \left[\hat{V}(x)\right]\Psi\big(x,t\big) \tag{11.9}$$

which we may understand as executing two operations on $\Psi(x,t)$ whose results are then added. The first operation is to take a second partial derivative of the wave function and to multiply it with some constant and the second operation is to multiply another function with the wave function.

With this out of the way, we can proceed and ask how possible solutions $\Psi(x,t)$ to the Schrödinger equation in (11.3) will look like.

### *11.2.1 Separating Space and Time*

Differential equation solving is a craft rather than a science and generally requires experience. All that follows will indeed be based on experience gathered over hundreds of years of mathematical and physical research. In other words, we will take the approaches we apply next for granted and gloss over where they come from and why they work.

Note once again that we assume the Hamiltonian $\hat{H}$ in (11.3) to be independent of time which will simply things considerably. Of course, Schrödinger equations can also be set up and solved in the more general case of a time-dependent Hamiltonian but our assumption makes it easy to *guess* a general structure of the sought after wave function. In particular, we can apply the **separation of variables** *ansatz* and *suppose* that the wave function factors as follows

$$\Psi\big(x,t\big) = \psi(x)\cdot\phi(t)\ . \tag{11.10}$$

Given this ansatz, we emphasize that $\psi(x)$ is a function only of space $x$ and $\phi(t)$ is a function only of time $t$. Plugging their product into (11.3), the Schrödinger equation becomes

$$i\,\hbar\,\frac{\partial}{\partial t}\,\psi(x)\,\phi(t) = -\frac{\hbar^2}{2\,m}\,\frac{\partial^2}{\partial x^2}\,\psi(x)\,\phi(t) + \hat{V}\big(x\big)\,\psi(x)\,\phi(t) \tag{11.11}$$

which we can rewrite or rearrange as follows

$$i\,\hbar\,\psi(x)\,\frac{d}{dt}\,\phi(t) = -\frac{\hbar^2}{2\,m}\,\phi(t)\,\frac{d^2}{dx^2}\,\psi(x) + \hat{V}(x)\,\psi(x)\,\phi(t)\ . \tag{11.12}$$

Looking at this expression, we note that our separation ansatz has led to partial derivatives becoming standard derivatives which will indeed simplify things. Moreover, if we next move all terms involving $\phi(t)$ to one side and all terms involving $\psi(x)$ to the other side of the equation, we obtain

$$i\,\hbar\,\frac{1}{\phi(t)}\,\frac{d}{dt}\,\phi(t) = \frac{1}{\psi(x)}\left[-\frac{\hbar^2}{2\,m}\,\frac{d^2}{dx^2}\,\psi(x) + \hat{V}(x)\,\psi(x)\right]\ . \tag{11.13}$$

Looking at this equation, we emphatically emphasize the following: Its left hand side is a function solely of time $t$ and its right hand side is a function solely of space $x$. In order for two such functions to equal each other, they must be constant. Hence, in order for (11.13) to hold, we must have

$$\frac{i\,\hbar}{\phi(t)}\,\frac{d\phi(t)}{dt} = \gamma \tag{11.14}$$

$$\frac{1}{\psi(x)}\left[-\frac{\hbar^2}{2\,m}\,\frac{d^2\psi(x)}{dx^2} + \hat{V}(x)\,\psi(x)\right] = \gamma\ . \tag{11.15}$$

where the constant $\gamma$ is sometimes called the *separation constant*.

At this point, we have already done quite some heavy lifting but did not even specify any details of any of our envisaged scenarios. In what follows, we will continue with this strategy. We will first derive a solution for $\phi(t)$ which can be done without knowing about problem specific Hamiltonians. Only afterwards, once we are going to solve for $\psi(x)$, will we need to be more specific.

### 11.2.2 Solving for $\phi(t)$

Note that (11.14) does not involve a Hamiltonian and therefore describes an aspect of the sought after solution to our Schrödinger equation that is independent of the specifics of a quantum system. A straightforward rearrangement of (11.14) yields

$$\frac{d}{dt}\,\phi(t) = -\frac{i\,\gamma}{\hbar}\,\phi(t) \tag{11.16}$$

which we recognize as a simple ordinary differential equation whose solution is

$$\phi(t) = e^{-\frac{i\,\gamma}{\hbar}\,t}\ . \tag{11.17}$$

This is already quite a profound result because it tells us that, irrespective of what the spatial function $\psi(x)$ will turn out to look like, the temporal function $\phi(t)$ will always be a *complex exponential* that causes the wave function

$$\Psi\big(x,t\big) = \psi(x)\,e^{-\frac{i\,\gamma}{\hbar}\,t} \tag{11.18}$$

to *oscillate*. But let us go further.

Using *Euler's identity* $e^{ix} = \cos x + i\,\sin x$ and the fact that $\hbar = {}^{h}/_{2\pi}$, we can write

$$\phi(t) = \cos\!\Big(\tfrac{2\pi\,\gamma}{h}\,t\Big) - i\,\sin\!\Big(\tfrac{2\pi\,\gamma}{h}\,t\Big) = \cos\!\big(\omega\,t\big) - i\,\sin\!\big(\omega\,t\big) \tag{11.19}$$

where the last step introduces the *angular frequency*

$$\omega = \frac{2\pi\,\gamma}{h}\ .$$

Introducing even more physics terminology, we note that the angular frequency $\omega$ is related to the *oscillation frequency* $\nu$ via

$$\omega = 2\pi\,\nu \tag{11.20}$$

so that we obtain the following expression for the separation constant

$$\gamma = h\,\nu\ . \tag{11.21}$$

Now, that is another noteworthy insight because we have seen the right hand side of this equation before! Indeed, Planck (10.20) told us that

$$E = h\,\nu\ . \tag{11.22}$$

Without spending much efforts, we have thus established that *the temporal component of the wave function of a 1D quantum particle with (total) energy $E$ is given by*

$$\phi(t) = e^{-i\,E\,t/\hbar}\ . \tag{11.23}$$

Since we already took some time talking about operators acting on continuous functions, there is one more point to this result which we should mention. What we just found is a function $\phi(t)$ such that

$$\frac{d}{dt}\,\phi(t) = -\frac{i\,E}{\hbar}\,\phi(t)\ . \tag{11.24}$$

That is, we have found an *eigenfunction* of the operator $\frac{d}{dt}$ whose corresponding *eigenvalue* is $-\frac{i\,E}{\hbar}$. We emphasize this because experience tells us that most readers will be familiar with the idea of taking derivatives of exponential functions but not with the idea of exponential functions being eigenfunctions of the derivative operator. What we just saw therefore connects something most of us will know about to the kind of thinking used in quantum mechanics.

### 11.2.3 What About $\psi(x)$?

Next, we will have to solve equation (11.15) for $\psi(x)$. Using $\gamma = h\,\nu = E$, we can rewrite that equation as

$$\left[-\frac{\hbar^2}{2\,m}\,\frac{d^2}{dx^2} + \hat{V}(x)\right]\psi(x) = E\,\psi(x) \tag{11.25}$$

or, shorter as

$$\hat{H}\,\psi(x) = E\,\psi(x) \tag{11.26}$$

which is known as the **time-independent Schrödinger** equation.

There are two noteworthy aspects about this equation. First of all, we recognize it as another instance of an eigenvalue/eigenfunction problem. In other words, the above expression states that $\psi(x)$ has to be an eigenfunction of the Hamiltonian operator $\hat{H}$ and that the corresponding eigenvalue is $E$. While we will not directly use this insight in our upcoming mathematical discussion, it will be central to our code examples later on.

Second of all, contrary to the eigenvalue/eigenfunction problem for $\phi(t)$ in (11.24), the eigenvalue/eigenfunction problem for $\psi(x)$ in (11.26) now contains a Hamiltonian. Solving it will thus require us to finally become more specific about the three scenarios we kept mentioning since the introduction. We therefore recall that we promised to study the behavior of (1) a particle trapped within an infinite potential well, (2) a particle in a finite potential box within an infinite potential well, and (3) a particle facing a potential barrier within an infinite potential well. In what follows, we will consecutively work through these scenarios starting with the first and simplest one.

## 11.3 A Particle in an Infinite Potential Well

In this scenario, we consider a 1D quantum particle which is trapped in an infinitely deep energy well such as illustrated in Fig. 11.1. The impenetrable walls of the well are located at $x \in \left\{-\frac{w}{2}, +\frac{w}{2}\right\}$. Inside of the well, no forces whatsoever act on the particle; however, at both walls, infinitely large forces will always repel it. The (time-independent) potential energy of this system is therefore just

$$\hat{V}(x) = \begin{cases} 0 & \text{if } -\frac{w}{2} \leq x \leq +\frac{w}{2} \\ \infty & \text{otherwise}\ . \end{cases} \tag{11.27}$$

This is another hypothetical setting, in the real world there are no such things as infinitely large energies. However, it will again simplify our equation solving efforts. Since the particle can never leave the well, we will only have to worry about what can happen inside the well. And, even better, inside of the well there only is a zero potential energy field so that our particle can move freely. However, spoiler alert, it will turn that "moving freely" means something else in the quantum realm than it does in our everyday environments.

To begin with, we recall the time-independent Schrödinger equation from (11.25) which we can write as

$$-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}\,\psi(x) + \hat{V}(x)\,\psi(x) = E\,\psi(x)\ . \tag{11.28}$$

Now, since we only have to worry about the situation inside of the infinite potential well where we additionally have $\hat{V}(x) = 0$, this expression quickly simplifies. Indeed, if we also use $\hbar = \frac{h}{2\pi}$, we have the following

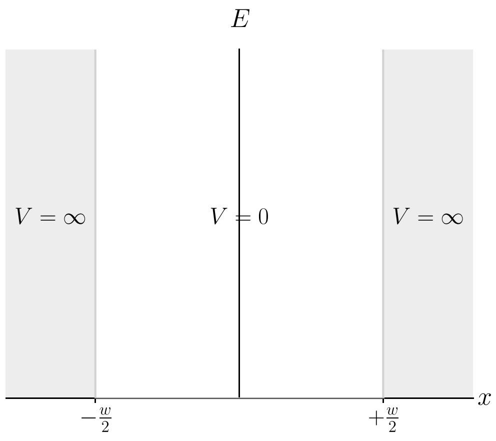


**Fig. 11.1** An infinite potential well

$$\frac{d^2}{dx^2}\,\psi(x) = -\frac{8\,\pi^2\,m\,E}{h^2}\,\psi(x)\;. \tag{11.29}$$

To solve this differential equation, we first further simplify matters and consider something not uncommon in physics, namely a *coordinate transformation*.

**Notation**

In all that follows, we will be working with transformed coordinates

$$x \leftarrow x + \frac{w}{2}$$

in which the walls of the potential $\hat{V}(x)$ reside at 0 and $w$, respectively.

Next, we interpret the fact that the particle can never leave the well as follows: The quantum mechanical probability $|\psi(x)|^2$ of observing it at at any location $x < 0$ or $x > w$ will be zero. In other words, outside of the well, we must have $\psi(x) = 0$. Assuming continuity of $\psi(x)$ at the walls of the well, we thus deduce the following *boundary conditions* for our problem

$$\psi(0) = 0 \tag{11.30}$$

$$\psi(w) = 0\;. \tag{11.31}$$

We further note that any solution to the comparatively simple second order differential equation in (11.29) will be of the general form

$$\psi(x) = \alpha\cos\left(\sqrt{\frac{8\,\pi^2\,m\,E}{h^2}}\,x\right) + \beta\sin\left(\sqrt{\frac{8\,\pi^2\,m\,E}{h^2}}\,x\right) \tag{11.32}$$

for some $\alpha, \beta \in \mathbb{R}$. Plugging $x = 0$ into (11.32) and using $\psi(0) = 0$, we simply obtain

$$0 = \alpha\cos(0) + \beta\sin(0) \tag{11.33}$$

which implies that $\alpha = 0$. However, plugging $x = w$ into (11.32) and using $\psi(w) = 0$ together with $\alpha = 0$, we obtain

$$0 = \beta\sin\left(\sqrt{\frac{8\,\pi^2\,m\,E}{h^2}}\,w\right)\;. \tag{11.34}$$

Ignoring the trivial solution ($\beta = 0$), we can therefore conclude that we must have

$$\sqrt{\frac{8\,\pi^2\,m\,E}{h^2}}\,w = 0, \pi, 2\pi, 3\pi, \ldots \tag{11.35}$$

Looking at this intermediate result, we note that, with one exception, each quantity in this expression is a (problem specific) constant; $\pi$ is $\pi$, $h$ is Planck's constant, $w$ is the width of the well, and $m$ is the mass of our (non-relativistic) particle.

The exception is the particle's energy $E$ which is a problem variable. Introducing natural numbers $n \in \mathbb{N}_+$, we may thus rewrite equation (11.35) as

$$\sqrt{\frac{8\,\pi^2\,m\,E_n}{h^2}}\,w = n\,\pi \ . \tag{11.36}$$

But this means that, simply by trying to solve the Schrödinger equation, we just stumbled across our first *quantum weirdness* result. This is because we just found that *the energy of our particle can only assume certain discrete values*, namely

$$E_n = \frac{n^2\,h^2}{8\,m\,w^2} = \frac{n^2\,\pi^2\,\hbar^2}{2\,m\,w^2} \ . \tag{11.37}$$

In other words, possible energies of our quantum particle are *quantized* and do not vary continuously.

We must also note the following: The case where $n = 0$ is a mathematically valid solution to our problem but physically impossible because it would mean that the probability of observing the particle within the well is zero. We therefore have to insist on $n \in \mathbb{N}_+$, that is on $n \geq 1$.

Finally, the value of the coefficient $\beta$ in (11.34) can be determined via *normalizing* the time-independent wave function according to the Born rule

$$\int_0^w \left|\psi(x)\right|^2 dx = 1 \ . \tag{11.38}$$

Plugging in all our intermediate results so far, provides us with

$$\int_0^w \beta^2 \sin^2\left(\frac{n\,\pi}{w}\,x\right)\,dx = 1$$

which, after some tedious yet straightforward calculus, results in

$$\beta = \sqrt{\frac{2}{w}} \tag{11.39}$$

and readers are strongly encouraged to verify this for themselves.

All in all, we thus established that the *the spatial component of the wave function of a 1D quantum particle trapped in a 1D potential well is given by*

$$\psi_n(x) = \sqrt{\frac{2}{w}}\,\sin\left(\frac{n\,\pi}{w}\,x\right) \tag{11.40}$$

for $0 \leq x \leq w$ and by $\psi_n(x) = 0$ elsewhere.

Together with our result for the temporal component of the wave function, we therefore have the following scenario specific theorem.

**Theorem 11.1** *If a 1D quantum particle of mass m moves in an infinite 1D potential well $\hat{V}(x)$ of width w, then within the well ($0 \leq x \leq w$), there are infinitely many solutions to the Schrödinger equation and they are given by*

$$\Psi_n\big(x,t\big) = \sqrt{\frac{2}{w}}\,\sin\left(\frac{n\,\pi}{w}\,x\right)\,e^{-i\,E_n\,t/\hbar} \tag{11.41}$$

$$E_n = \frac{n^2\,\pi^2\,\hbar^2}{2\,m\,w^2} \tag{11.42}$$

$$n \geq 1\;. \tag{11.43}$$

Let us emphasize the following: We have now found the wave function of our hypothetical 1D particle in a hypothetical 1D infinite potential well to be

$$\Psi_n\big(x,t\big) = \psi_n(x)\cdot\phi_n(t) \tag{11.44}$$

where $\phi_n(t)$ and $\psi_n(x)$ are solutions to eigenvalue/eigenfunction problems, namely

$$i\,\hbar\,\frac{d}{dt}\,\phi_n(t) = E_n\,\phi_n(t) \tag{11.45}$$

$$\hat{H}\,\psi_n(x) = E_n\,\psi_n(x) \tag{11.46}$$

with $\hat{H}$ being a problem specific Hamiltonian. Since the solutions $\phi_n(t)$ to (11.45) are complex exponentials, they cause $\Psi_n(x,t)$ to be an oscillating complex valued function. The solutions $\psi_n(x)$ to (11.46), on the other hand, are real valued functions. But how do they look like?

Figure 11.2 illustrates that the eigenstates $\psi_n(x)$ of our specific Hamiltonian $\hat{H}$ are really but simple sine waves. The figure shows the solutions for $1 \leq n \leq 5$ and we plotted the in a manner common in the physics literature. That is, we plotted function $\psi_n(x)$ at height $E_n$ above the $x$-axis. This allows us to see that higher energies go along with stronger oscillating eigenstates of $\hat{H}$. Indeed, the $n$-th solution $\psi_n(x)$ has $n-1$ nodes and we also see that solutions are alternatingly symmetric and anti-symmetric. The way we plotted these solutions further indicates why eigenstate $\psi_1(x)$ with eigenenergy $E_1$ is called the **ground state** of our quantum system. It is simply the solution of lowest energy and thus closest to the ground of the energy landscape.

But what does it mean for the state of a particle to be a wave function? Does it mean the particle is smeared out or distributed over space?

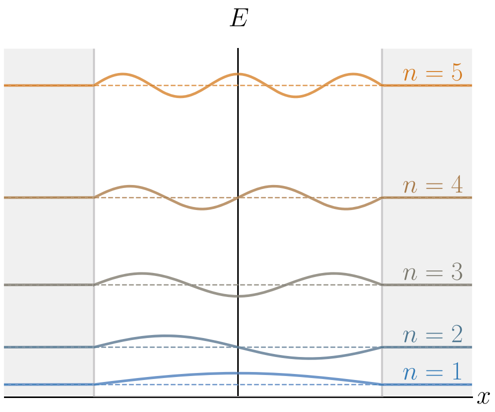


**Fig. 11.2** Spatial eigenstates of a 1D quantum particle trapped in an infinite potential well

No! The *physical aspect* of the wave function is its squared norm

$$\left|\Psi_n\big(x,t\big)\right|^2 = \int \Psi_n^*\big(x,t\big)\,\Psi_n\big(x,t\big)\,dx \tag{11.47}$$

which indicates the *probability* of finding the particle at location $x$ at time $t$. Given our solutions for $\Psi_n(x,t)$, we observe the following

$$\begin{aligned}\left|\Psi_n\big(x,t\big)\right|^2 &= \int \psi_n^*(x)\,\phi_n^*(t)\,\phi_n(t)\,\psi_n(x)\,dx && (11.48)\\ &= \int \psi_n^*(x)\,e^{+i\,E_n\,t/\hbar}\,e^{-i\,E_n\,t/\hbar}\,\psi_n(x)\,dx && (11.49)\\ &= \int \psi_n^*(x)\,\psi_n(x)\,dx && (11.50)\\ &= \left|\psi_n(x)\right|^2. && (11.51)\end{aligned}$$

With respect to the *basis states* $\Psi_n(x,t)$, the probability of finding the particle at location $x$ at time $t$ thus simply amounts to the squared norm $|\psi_n(x)|^2$ of their spatial components. Figure 11.3 shows the first five of these **probability amplitudes**.

Looking at this figure, we see more *quantum weirdness*. Except for the ground state, all other physically meaningful states of the particle seemingly forbid it from being at certain locations within the well. In other words, there are locations $x$ within the well where $|\psi_n(x)|^2 = 0$. This contradicts classical intuition: If quantum particles were billiard balls, this could not happen; yet, this behavior has been verified in numerous corresponding experiments. We therefore have that *((sub)atomic) quantum particles do not behave like macroscopic objects*.

But why don't we observe quantum effects or behaviors like this in our macroscopic world?

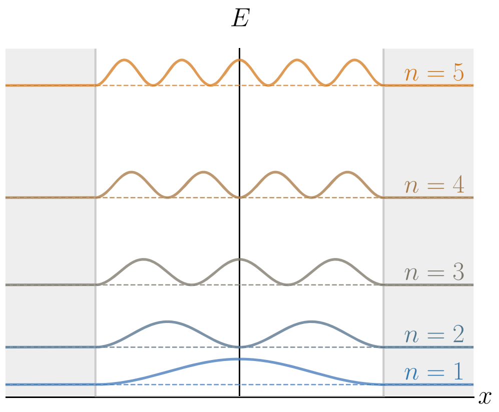


**Fig. 11.3** Probability amplitudes of the eigenstates of a 1D quantum particle trapped in an infinite potential well

Well, despite its utter simplicity, our current setting can provide an intuition as to why this is. Note, for instance, that the ground state energy of a particle in an infinite potential well is

$$E_1 = \frac{h^2}{8\,m\,w^2}$$

and recall that Planck's constant $h \in O\big(10^{-34}\big)$ is minuscule. For macroscopic objects, their mass $m$ and spatial range $w$ will be much much larger than for quantum particles. For macroscopic objects, the ground state energy is thus basically indistinguishable from zero. This is why macroscopic objects can seem to be at rest, i.e. can seem to have a specific position.

Also note that the difference between any two adjacent quantum energy levels amounts to

$$E_{n+1} - E_n = \frac{(n+1)^2\,h^2}{8\,m\,w^2} - \frac{n^2\,h^2}{8\,m\,w^2} = \frac{h^2\,(2\,n+1)}{8\,m\,w^2}\;. \tag{11.52}$$

For macroscopic objects with $m,\ w \gg 1$, it therefore tends to zero. In other words, for large objects, energies do not appear to be quantized but seem to vary continuously.

Here is another interesting question that connects what we just went through to our discussion in Chap. 10: At which location $x$ can we *expect* a particle in the ground state to be found?

Recalling that *expected values* of a quantum observable $O$ are written as $\langle O \rangle$ and noting that the observable we just asked for is the *position operator* $\hat{x}$ with possible measurement results $x$, we have that the expected position of a ground state particle trapped in an infinite potential well is given by

$$\langle \hat{x} \rangle = \int_0^w x \left| \Psi_1\big(x,t\big) \right|^2 dx = \int_0^w x \, \psi_1^*(x) \, \psi_1(x) \, dx$$
$$= \frac{2}{w} \int_0^w x \, \sin^2\left(\tfrac{\pi}{w} x\right) \, dx$$

for which tedious yet altogether straightforward calculus results in

$$\langle \hat{x} \rangle = \frac{w}{2} \, . \tag{11.53}$$

And what about the expected energy $\langle \hat{H} \rangle$ of a particle in some basis state $\Psi_n(x,t)$? Well, to answer this we need to evaluate

$$\langle \hat{H} \rangle = \int \Psi_n^*\big(x,t\big) \, \hat{H} \, \Psi_n\big(x,t\big) \, dx \tag{11.54}$$

which is easy, because we can use $\hat{H} \, \Psi_n(x,t) = E_n \Psi_n(x,t)$. Plugging this in, we find

$$\langle \hat{H} \rangle = E_n \int \Psi_n^*\big(x,t\big) \, \Psi_n\big(x,t\big) \, dx = E_n \tag{11.55}$$

and, in Exercise 11.2, you will have the opportunity to prove that we can be certain about this. In other words, if the particle is in state $\Psi_n(x,t)$, we can be certain to measure its energy as $E_n$.

Given all our work so far, we finally must emphasize the following: Our scenario of a single 1D particle in a 1D world can be dangerously misleading since we might be tempted to interpret its wave functions as physical waves in space. However, wave functions are not defined over the psychical space a quantum system lives in but over its **configuration** or **state space**.

Consider, for example, a system of *two* 1D particles trapped in a potential well. Here, the spatial component of the system's wave function would be a function $\psi(x_a, x_b)$. However, $(x_a, x_b)$ does not correspond to any single location in the 1D world the particles live in; rather, $(x_a, x_b)$ refers to two locations and thus to a *configuration* of the two particle system. Based on our discussion in Chap. 10, we already know that the joint spatial state of the two particles would thus have to be described as $\psi(x_a, x_b) = \psi_a(x_a) \otimes \psi_b(x_b)$. While it would merit study, we will not follow up on this extended setting but leave its study to the physics books.

## 11.4 A Particle within a Finite Potential Box

Our next hypothetical quantum system consists of a 1D particle located in a *finite* energy box of width $b$ within an infinite potential well of width $w$. This setting is illustrated in Fig. 11.4 and its mathematical treatment is almost identical to the one in

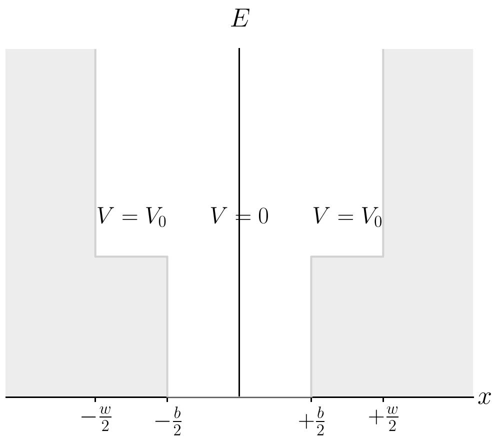


**Fig. 11.4** A finite potential box within an infinite potential well

our previous scenario. The only yet important difference is that we are now dealing with a potential

$$\hat{V}(x) = \begin{cases} 0 & \text{if } 0 \leq |x| \leq \frac{b}{2} \\ V_0 & \text{if } \frac{b}{2} < |x| \leq \frac{w}{2} \\ \infty & \text{otherwise} \end{cases} \tag{11.56}$$

where $V_0$ is some constant.

From our discussion above, we already know that the "real" problem in settings like this is to solve the time independent Schrödinger equation $\hat{H}\psi(x) = E\psi(x)$ and, to be able to this in our current scenario, we need an appropriate ansatz for $\psi(x)$.

To this end, we can again restrict our attention the region within the infinite well, consider the coordinate transformation $x \leftarrow x + \frac{w}{2}$, and note that $\hat{V}(x)$ is piece-wise constant within the transformed region $0 \leq x \leq w$.

A reasonable ansatz for $\psi(x)$ would therefore be the following piece-wise model

$$\psi(x) = \begin{cases} \psi_l(x) & \text{if } x < \frac{w}{2} - \frac{b}{2} \\ \psi_c(x) & \text{if } \frac{w}{2} - \frac{b}{2} \leq x \leq \frac{w}{2} + \frac{b}{2} \\ \psi_r(x) & \text{if } \frac{w}{2} + \frac{b}{2} < x \leq w \end{cases} \tag{11.57}$$

where $\psi_l(x) = A^{\,k_1 x}$, $\psi_c(x) = B\,\sin(k_2 x + D)$, and $\psi_r(x) = C\,e^{-k_1 x}$ and $k_1$ and $k_2$ are appropriate constant whose specific form we actually need not discuss. Why would this be an appropriate model? Because $\psi(x)$ must be square integrable ($\psi(x) \to 0$ as $x \to \pm\infty$) and the second derivative $\frac{d^2}{dx^2}\psi(x)$ has to be a multiple of $\psi(x)$ and these criteria are met. Why would we not have to discuss the constants $k_1$ and $k_2$? Because it would be gratuitous work. Why? Because, using this ansatz, we would need to determine the parameters $A$, $B$, $C$, and $D$. To this end, we would need to impose *continuity conditions* on our piece-wise functions which would lead to a

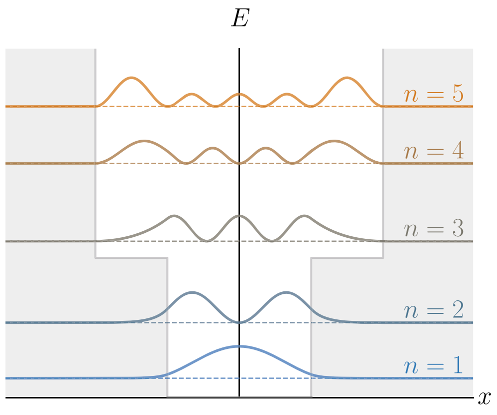


**Fig. 11.5** Probability amplitudes of the eigenstates of a 1D quantum particle within a box within in an infinite potential well

system of equations that cannot be solved for $D$. Try as we might, we would always end up with a *transcendental equation* for which there is no closed form solution.

In short, even for simple settings such as our 1D particle in a box, *it is common that solutions to the Schrödinger equation have to be determined numerically*.

Something like this is anything but uncommon in the sciences and numerical solvers for differential equations or eigenvalue/eigenvector problems are important tools of the trade. For time-independent Schrödinger equations, these are easily realized [3, 4] and we will discuss how to implement a simple solver further down.

For now, we may therefore simply look at the results we get when using such tools. Figure 11.5 shows the probability amplitudes of the first five basis states $\Psi_n(x, t)$ of our particle in a finite potential box within an infinite potential well. We see that energy levels are once again quantized and that higher energies again go along with more high frequent oscillations.

We also recognize yet another aspect of *quantum weirdness*. Especially for higher energies, there are non-zero probabilities for finding the particle *outside* of the box. Even if its energy $E_n < V_0$, there actually is a certain chance for this to happen as we note that the basis states $\Psi_1$ and $\Psi_2$ extend into the walls of the box. This contradicts classical intuition: If particles were billiard balls, this could not happen. Yet, this behavior can indeed be seen in numerous quantum mechanical experiments. We therefore have to stress again that *((sub)atomic) quantum particles do not behave like macroscopic objects*.

## 11.5 A Particle Facing a Finite Potential Barrier

Our third and final hypothetical scenario kind of reverses our second one. It is illustrated in Fig. 11.6 where we again recognize an infinite potential well but now with a *finite potential barrier* of width $b$ in its center.

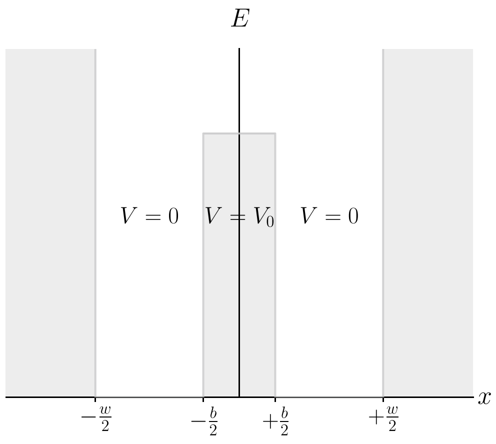


**Fig. 11.6** A finite potential barrier within an infinite potential well

With respect to solving the Schrödinger equation for a particle in this scenario, things are again almost identical to our previous scenario. The only difference is again in the potential which now reads

$$\hat{V}(x) = \begin{cases} V_0 & \text{if } 0 \leq |x| \leq \frac{b}{2} \\ 0 & \text{if } \frac{b}{2} < |x| \leq \frac{w}{2} \\ \infty & \text{otherwise} \end{cases} \tag{11.58}$$

where $V_0$ is once again some constant.

To make a long story short, this setting, too, can not be solved analytically but requires numerical techniques. We may therefore immediately look at Fig. 11.7 which shows the probability amplitudes of the first five basis states $\Psi_n(x, t)$ for this scenario.

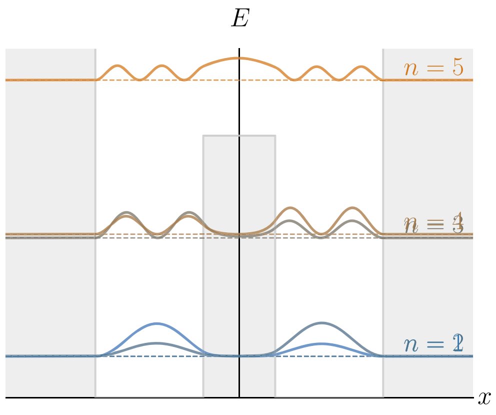


**Fig. 11.7** Probability amplitudes of the eigenstates of a 1D quantum particle within in an infinite potential well with a finite potential barrier

We once again see quantized energies some of which are, however, much closer together than in our previous settings. Again, higher energies entail more frequent oscillation and again there are "forbidden" spots where there is a zero probability of observing the particle.

What is most notable, however, is that the particle appears to be able to *quantum tunnel* through the energy barrier, even if its energy $E_n < V_0$. This is yet another aspect of *quantum weirdness* that contradicts intuition: If particles were billiard balls, this could not happen. And yet, quantum tunneling is an established phenomenon. On the downside, it causes headaches for semiconductor manufacturers because, the smaller their chip designs become, the more likely electrons will pass through insulating regions and thus to render chips unusable. On the upside, tunneling can be exploited for quantum information processing, in particular, in *adiabatic quantum computing* which we will study in the next chapter.

## 11.6 Quantum Superposition

There are numerous other phenomena of *quantum weirdness* which we cannot cover all. However, our work so far has prepared us to look at one more, very crucial aspect which is of utmost importance to quantum computing and thus often referred to as an aspect of *quantum supremacy*.

Note that the Schrödinger equation, say, in (11.3) is a linear differential equation since the operators $\frac{\partial}{\partial t}$ and $\hat{H}$ which act on the wave function $\Psi(x, t)$ are both linear.

But this means that, if $\Psi_1(x, t), \Psi_2(x, t), \ldots$ are distinct solution to the Schrödinger equation, then

$$\Psi(x, t) = \sum_j \alpha_j \, \Psi_j(x, t) \tag{11.59}$$

will be yet another solution. Such a solution will be physically meaningful, if the coefficients $\alpha_j \in \mathbb{C}$ obey the Born rule and are normalized as

$$\sum_j |\alpha_j|^2 = 1 \ . \tag{11.60}$$

We therefore have the following important result.

**Theorem 11.2** *In general, the wave function of a quantum mechanical system can exist in a* ***superposition*** *of (up to infinitely) many wave functions*

$$\Psi\big(x,t\big) = \sum_{j=1}^{\infty} \alpha_j \, \Psi_j\big(x,t\big) \tag{11.61}$$

*where the $\alpha_j \in \mathbb{C}$ obey the Born rule*

$$\sum_{j=1}^{\infty} |\alpha_j|^2 = 1 \ . \tag{11.62}$$

Quantum superposition, too, is a confirmed, fundamental quantum phenomenon for which there is no appropriate classical analogue. It is the reason why quantum computers can perform exponentially many computations in parallel and can occur in any kind of quantum systems, not just in the simple 1D particle systems we are studying here. But let us stick with these for simplicity.

We actually already worked with quantum superposition in Chap. 10. Can you spot where? There, however, our discussion was abstract, top-down, and detached from any practical example. Now that we know how quantum states or wave functions can actually look like, we can revisit superposition in a bottom-up manner.

To begin with, let us therefore ask for the physical meaning of the complex coefficients $\alpha_j$ in (11.61). Well, the Born rule tells us that we must have

$$\int \Psi^*\big(x,t\big)\,\Psi\big(x,t\big)\,dx = \int \Big(\sum_j \alpha_j^* \,\Psi_j^*\Big)\Big(\sum_k \alpha_k \Psi_k\Big)\,dx \tag{11.63}$$

$$= \int \left(\alpha_1^* \,\psi_1^* \,e^{+\frac{i E_1 t}{\hbar}} + \alpha_2^* \,\psi_2^* \,e^{+\frac{i E_2 t}{\hbar}} + \cdots\right) \tag{11.64}$$

$$\cdot \left(\alpha_1 \,\psi_1 \,e^{-\frac{i E_1 t}{\hbar}} + \alpha_2 \,\psi_2 \,e^{-\frac{i E_2 t}{\hbar}} + \cdots\right) dx = 1 \ . \tag{11.65}$$

Looking at this integral we note that it involves two kinds of terms. First of all there are terms where $j = k$ such that

$$\alpha_j^* \,\alpha_j \,e^{\frac{i\,(E_j - E_j)\,t}{\hbar}} \int \psi_j^* \,\psi_j \,dx = \alpha_j^* \,\alpha_j \int \psi_j^* \,\psi_j \,dx = \alpha_j^* \,\alpha_j = \big|\alpha_j\big|^2 \ . \tag{11.66}$$

Second of all, there are terms where $j \neq k$ and which read

$$\alpha_j^* \, \alpha_k \, e^{\frac{i\,(E_j - E_k)\,t}{\hbar}} \int \psi_j^* \, \psi_k \, dx \; . \tag{11.67}$$

If we plug in, say, the solutions $\psi_j(x) = \sqrt{2/w}\,\sin(j\,\pi/w\,x)$ for a 1D particle in an infinite well, then straightforward calculus reveals that

$$\int \psi_j^* \, \psi_k \, dx = \delta_{jk} = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases} \tag{11.68}$$

which means that *eigenstates of the Hamiltonian $\hat{H}$ are orthonormal.* Note that this holds in general not just for this simple setting! Also note that this finally explains why we have been calling the wave functions $\Psi_n(x,t) = \psi(x) \cdot \phi(t)$ *basis states.*

All in all, we thus confirmed the Born rule and established that we do indeed have

$$\int \Psi^* \, \Psi \, dx = \sum_j \alpha_j^* \, \alpha_j \int \psi_j^* \, \psi_j \, dx = \sum_j \left|\alpha_j\right|^2 = 1 \; . \tag{11.69}$$

But what is the physical meaning of the coefficients $\alpha_j$? To answer this question, we compute the energy expectation

$$\langle \hat{H} \rangle = \int \Psi^* \, \hat{H} \, \Psi \, dx = \int \Big(\sum_j \alpha_j^* \, \Psi_j^*\Big) \, \hat{H} \, \Big(\sum_k \alpha_k \Psi_k\Big) \, dx \; . \tag{11.70}$$

Using the linearity oh $\hat{H}$, the time-independent Schrödinger equation, and our above results, this expectation can be written as

$$\langle \hat{H} \rangle = \int \Big(\sum_j \alpha_j^* \, \Psi_j^*\Big) \Big(\sum_k \alpha_k \, \hat{H} \, \Psi_k\Big) \, dx \tag{11.71}$$

$$= \int \Big(\sum_j \alpha_j^* \, \Psi_j^*\Big) \Big(\sum_k \alpha_k \, E_k \Psi_k\Big) \, dx \tag{11.72}$$

$$= \sum_j \left|\alpha_j\right|^2 E_j \; . \tag{11.73}$$

The **amplitudes** $|\alpha_j|^2$ therefore correspond to the *probability* that a measurement of the energy of the system will produce- or result in a value of $E_j$.

Observe that this result for $\langle \hat{H} \rangle$ implies the following: A linear superposition state

$$\Psi\big(x,t\big) = \sum_j \alpha_j \, \Psi_j\big(x,t\big) = \sum_j \alpha_j \, \psi_j(x) \, \phi_j(t) \tag{11.74}$$

characterizes a quantum particle with *uncertain* energy.

This is in contrast to our result in (11.55) where we computed the expected energy of a basis state $\Psi_n(x,t)$ and found that we can be *certain* to measure it as $E_n$.

In short, general wave functions or superposition states such as in (11.61) have an uncertain energy. They are called *non-stationary states* and have some observable properties that will change over time. For instance, their probability amplitudes and thus their position expectation values oscillate. This is in contrast to basis wave functions or basis states which are *stationary states* and whose observable properties will not change over time.

## 11.7 Numerically Solving Schrödinger Equations

Above, we were concerned with solving simple or simplified Schrödinger equations

$$i\,\hbar\,\frac{\partial}{\partial t}\Psi\big(x,t\big) = \hat{H}\,\Psi\big(x,t\big) \tag{11.75}$$

for wave functions $\Psi(x,t)$ with $x,t \in \mathbb{R}$. We saw that the general idea of the separation of variables ansatz

$$\Psi\big(x,t\big) = \psi(x)\cdot\phi(t) \tag{11.76}$$

leads to two eingenvalue/eigenstate problems

$$i\,\hbar\,\frac{d}{dt}\,\phi(t) = E_n\,\phi(t) \tag{11.77}$$

$$\hat{H}\,\psi(x) = E\,\psi(x) \tag{11.78}$$

where the former is independent of the problem Hamiltonian $\hat{H}$ and the latter is not. We also saw that this generally means that former is easy to solve analytically irrespective of the given problem but, depending on the nature of $\hat{H}$, the latter may be difficult if not impossible to solve analytically. In other words, the challenge usually consists in solving time-independent Schrödinger equations as in (11.78) and often requires computer assistance. Next, we therefore discuss a simple technique for this purpose.

For simplicity we will stick with our use case of 1D quantum particles moving in various energy landscapes. Also, to reduce notational clutter, we will henceforth work in *atomic units* and simply let $m = 1$ and $\hbar = 1$. Plugging these into (11.25), we obtain the following time-independent Schrödinger equation for our particular setting

$$\left[-\frac{1}{2}\,\frac{d^2}{dx^2} + \hat{V}(x)\right]\psi(x) = E\,\psi(x) \tag{11.79}$$

We further recall that all the scenarios we studied above assumed that the movement of the quantum particle is confined to a one-dimensional interval $[-w/2, +w/2]$ of length $w$ which will simplify our following reasoning.

Consider this: We may discretize the given interval $[-w/2, +w/2]$ by imposing discrete grid of $N$ equally spaced points

$$-\frac{w}{2} \leq x_j \leq +\frac{w}{2} \tag{11.80}$$

such that the distance between any pair of neighboring grid points amounts to

$$\delta x = \left| x_j - x_{j\pm 1} \right| = \frac{w}{N-1} \,. \tag{11.81}$$

This discretization of the domain of the continuous position variable $x$ then allows us to *approximate* or to *represent* the continuous spatial wave function $\psi(x)$ in terms of an $N$-dimensional vector

$$\boldsymbol{\psi} = \begin{bmatrix} \psi(x_1) \\ \psi(x_2) \\ \vdots \\ \psi(x_N) \end{bmatrix} = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_N \end{bmatrix} . \tag{11.82}$$

Next, we discretize the second order derivative operator that occurs in (11.79). To this end, we note that our domain discretization allows us to approximate first order derivatives of the wave function $\psi$ at grid points $x_j$ in terms of a finite difference, namely

$$\frac{d}{dx}\psi(x_j) \approx \frac{\psi(x_{j-1}) - \psi(x_j)}{\delta x} \tag{11.83}$$

Applying this idea twice in a row allows us to discretize the second order derivative like this

$$\frac{d^2}{dx^2}\psi(x_j) \approx \frac{\frac{\psi(x_{j-1})-\psi(x_j)}{\delta x} - \frac{\psi(x_j)-\psi(x_{j+1})}{\delta x}}{\delta x} \tag{11.84}$$

$$= \frac{\psi(x_{j-1}) - 2\,\psi(x_j) + \psi(x_{j+1})}{\delta x^2} \,. \tag{11.85}$$

Hence, if we now express (11.85) in terms of the entries $\psi_j$ of the vector $\boldsymbol{\psi}$ defined in (11.82), we get this discrete approximation of the continuous kinetic energy operator

$$-\frac{1}{2}\frac{d^2}{dx^2}\psi(x_j) \approx \frac{-\psi_{j-1} + 2\,\psi_j - \psi_{j+1}}{2\,\delta x^2} \,. \tag{11.86}$$

Using the discrete approximation of the second spatial derivative in (11.86), we can therefore represent the continuous operator $\hat{T}$ in terms of a **tridiagonal matrix** of size $N \times N$

$$\boldsymbol{T} = \frac{1}{2\,\delta x^2} \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix} \tag{11.87}$$

and obtain the following approximation for how $\hat{T}$ acts on $\psi(x)$

$$\hat{T}\,\psi(x) = -\frac{1}{2}\frac{d^2}{dx^2}\,\psi(x) \approx \boldsymbol{T}\,\boldsymbol{\psi}\ . \tag{11.88}$$

In an even simpler manner manner, we can also represent the continuous potential energy operator $\hat{V}$ in (11.79) as a diagonal $N \times N$ matrix

$$\boldsymbol{V} = \frac{1}{2} \begin{bmatrix} \hat{V}(x_1) & & \\ & \ddots & \\ & & \hat{V}(x_N) \end{bmatrix} \tag{11.89}$$

and thus obtain the following approximation for how it acts on the spatial component of the wave function

$$\hat{V}\,\psi(x) \approx \boldsymbol{V}\,\boldsymbol{\psi}\ . \tag{11.90}$$

Putting all these considerations together, a discretized version of the Hamiltonian $\hat{H}$ of the quantum harmonic oscillator becomes

$$\boldsymbol{H} = \boldsymbol{T} + \boldsymbol{V}\ . \tag{11.91}$$

Looking at this $N \times N$ matrix $\boldsymbol{H}$, we note that it is real valued and symmetric because it is a sum of two real valued and symmetric matrices. All in all, a discretized version of the Schrödinger equation in (11.79) can now be written in terms of finitely sized matrices and vectors, namely

$$\boldsymbol{H}\,\boldsymbol{\psi} = E\,\boldsymbol{\psi} \tag{11.92}$$

which we recognize as a conventional matrix-vector eigenvalue/eigenvector problem.

This is good news because we already know how to solve problems like this using `numpy` functions provided in the `linalg` module.

Next, we put them to use and look at ideas for how to implement the above solution strategy. For all our upcoming code snippets to work, we therefor require the following

```
import numpy as np
import numpy.linalg as la
```

To begin with, we best set the parameters of a problem to be solved. To keep coding efforts low, we therefore once again consider the simple setting of a 1D particle in an infinite potential well where $\hat{V}(x) = 0$ for all $-\frac{w}{2} \leq x \leq +\frac{w}{2}$.

Setting up the parameters for this problem and computing numpy arrays of grid points $x_j$ and corresponding potential energies $\hat{V}(x_j)$ is as easy as:

```
w = 1.6
N = 1001
xs = np.linspace(-w/2, +w/2, N)
vs = np.zeros_like(xs)
```

Given these, we need to compute the Hamiltonian matrix $\boldsymbol{H}$ in (11.91). This can be accomplished, say, by function computeHamiltonian in code snippet 11.1. Its logic is as follows: Given the array xs of equally spaced grid points, the grid point distance $\delta x$ in (11.81) is computed into variable dx so that we can next implement matrix $\boldsymbol{T}$ in (11.87).

To this end, we first compute two arrays d0 and d1 of sizes $N$ and $N - 1$, respectively. Looking at these, is is clear that they contain the numbers on the main- and first sub-diagonal of matrix $\boldsymbol{T}$. Given those, we next invoke the numpy function diag and exploit the (rarely used) fact that it comes with two parameters v and k. Parameter v is an array of values to be set on *a* diagonal of a matrix and parameter k is an integer (..., –1, 0, +1, ...) indicating *which* diagonal is to be set. The default (k=0) is to consider the main diagonal, positive or negative choices of k indicate sub-diagonals above or below the main diagonal. In other words, those who know numpy well do not need for loops to implement *band matrices*.

Given array vs, matrix $\boldsymbol{V}$ in (11.89) can easily be implemented using just a simple call of diag. Once arrays matT and matV are in place, we simply add them and return the result.

All in all, we may thus use

```
matH = computeHamiltonian(xs, vs)
```

to get an array representing the discretized Hamiltonian for our problem. To next determine its eigenvalues and eigenvectors, we will apply function solveTISE in code snippet 11.2. Note that this function's name alludes to solving time-independent Schrödinger equations (TISEs) because, by computing the eigenvalues and eigenstates of a discrete Hamiltonian, it does just that.

The logic behind this function is straightforward. Since the kind of Hamiltonians were are dealing with are real valued and symmetric, they are real analogs of complex valued Hermitian matrices. We therefore work with the numpy function eigh which is specialized on solving Hermitian eigenvalue/eigenvector problems.

```
def computeHamiltonian(xs, vs):
    N = len(xs)

    dx = xs[1] - xs[0]

    d0 = -2 * np.ones(N)
    d1 =  1 * np.ones(N-1)

    matT  = np.diag(d0) + np.diag(d1, +1) + np.diag(d1, -1)
    matT /= -(2 * dx**2)

    matV = np.diag(vs)

    return matT + matV
```

**Code 11.1** Simple `numpy` code to compute discretized Hamiltonian operators for problems involving 1D quantum particles.

```
def solveTISE(matH, num=10):
    enrgs, waves = la.eigh(matH)

    enrgs = enrgs[  :num]
    waves = waves[:,:num]
    waves = waves / np.sqrt(np.sum(waves**2, axis=0))

    return enrgs, waves
```

**Code 11.2** Simple `numpy` code for computing the bottom eigenenergies and eigenstates of a discrete Hamiltonian operator.

In code snippet 11.2, we store the eigenvalues $E_n$ of $\boldsymbol{H}$ in a 1D array `enrgs` and its eigenvectors $\boldsymbol{\psi}_n$ in an array `waves`. Note that `solveTISE` has an additional parameter `num` which indicates how many bottom eigenvalues and eigenvectors to return. The next couple of lines of code thus truncate arrays `enrgs` and `waves` correspondingly. Next, only for good measure and to safeguard any downstream processing against numerical imprecision, we then make sure that the latter are normalized to unit length such that $\|\boldsymbol{\psi}_n\| = 1$ and finally return the results.

In short, calling, say

```
enrgs, waves = solveTISE(matH, num=5)
probs = np.abs(waves)**2
```

first determines the bottom five eigenvalues and eigenvectors of our Hamitonian in a fraction of a second and then uses the latter to compute the corresponding discretized squared norms or probability amplitudes $|\psi_n(x_j)|^2$.

How will these look like? Well, those who are so inclined may run the following snippet to get a visualization.

```
import matplotlib.pyplot as plt

num = probs.shape[1]
for j, n in enumerate (reversed(range(num))):
    plt.subplot(num, 1, j+1)
    plt.plot(xs, probs[:,n])
    plt.axis('off')
plt.show()
```

This will plot a picture similar to the one in Fig. 11.3 albeit not as fanciful which visually confirms that our numerical solution scheme works appropriately.

Having said that, we must point out that our method is a bit simplistic given the high standards of modern numerical computing. For instance, if high resolution was needed, we would have to work with a large number $N$ of grid points which may considerably slow down our solver. Also, its numerical precision is low compared to more sophisticated methods. Readers interested in details about those may consult the tutorials in [3, 4] or read the next chapter in which we will return to some of these issues.

## 11.8 Exercises

**11.1** Let $A$ be some Hermitian operator with eigenvalues $\lambda$ and eigenstates $|u\rangle$ such that $A|u\rangle = \lambda|u\rangle$. Prove the following: If $\lambda$ is an eigenvalue of $A$, the its is also an eigenvalues of $A^2$.

**11.2** For our 1D particle trapped in an infinite potential well scenario, we saw that, if the particle is in basis state $\Psi_n(x,t)$, then its expected energy is $\langle \hat{H} \rangle = E_n$. Use you result from the precious exercise to compute the standard deviation

$$\sigma_{\hat{H}} = \sqrt{\langle \hat{H}^2 \rangle - \langle \hat{H} \rangle^2} \tag{11.93}$$

from this expected value.

**11.3** Reconsider the scenario of a 1D particle trapped in an infinite potential well and assume that the particle is in the following *superposition* state

$$\Psi\big(x,t\big) = \tfrac{1}{\sqrt{2}}\,\Psi_1\big(x,t\big) + \tfrac{1}{\sqrt{2}}\,\Psi_2\big(x,t\big) = \tfrac{1}{\sqrt{2}}\,\psi_1(x)\,\phi_1(t) + \tfrac{1}{\sqrt{2}}\,\psi_2(x)\,\phi_2(t)\;. \tag{11.94}$$

Compute its expected energy $\langle \hat{H} \rangle$ and the standard deviation $\sigma_H = \sqrt{\langle \hat{H}^2 \rangle - \langle \hat{H} \rangle^2}$ from this expected energy. Compare your standard deviation result to your result from the previous exercise. What do you observe?

**11.4** Reconsider the scenario of a 1D particle trapped in an infinite potential well and recall that solutions to the corresponding time-independent Schrödinger equation are given by

$$\psi_n(x) = \sqrt{\frac{2}{w}} \sin\left(\frac{n\,\pi}{w}\,x\right) \tag{11.95}$$

where $0 \leq x \leq w$.

Now consider two such solutions $\psi_j(x)$ and $\psi_k(x)$ with $j \neq k$ and show that they are orthogonal. That is, show that

$$\langle \psi_j \,|\, \psi_k \rangle = \int_0^w \psi_j^*(x)\,\psi_k(x)\,dx = 0\,. \tag{11.96}$$

**Hint:** This is surprisingly easy if you make use of the following trigonometric identity: $2\sin(a)\sin(b) = \cos(a-b) - \cos(a+b)$.

**11.5** Consider the same setting as in the previous exercise but this time compute the value of $\langle \psi_j \mid \psi_j \rangle$. What do you find?

**11.6** Once again reconsider the scenario of a 1D particle trapped in an infinite potential well. For simplicity assume that the particle has mass $m = 1$ and set the reduced Planck constant to $\hbar = 1$.

Under these simplifying assumptions, measurements of the energy of the particle will result in

$$E_n = \frac{n^2\pi^2}{2\,w^2}\,. \tag{11.97}$$

Let $w = 1.6$ and compute these values for $1 \leq n \leq 5$.

**11.7** Run our `numpy` codes for solving Schrödinger equations. Recall that they compute an array `enrgs` containing the energy values of the bottom five eigenstates of a 1D particle trapped in a well of width $w = 1.6$.

Compare the entries of this array to your results from the previous exercise. What do you observe?

**11.8** Adapt our `numpy` codes for solving Schrödinger equations to the scenario of a 1D particle facing a potential barrier. Experiment with different choices of the parameters $b$ and $V_0$ of this barrier. Plot your result and ponder what you observe.

**11.9** A 1D *quantum harmonic oscillator* is a 1D quantum particle which is located at $-\frac{w}{2} \leq x \leq +\frac{w}{2}$ and subject to a potential

$$\hat{V}(x) = \tfrac{1}{2}\,m\,\omega^2\,x^2\,. \tag{11.98}$$

For simplicity let $m = 1, \omega = 1$ and adapt our `numpy` codes for solving Schrödinger equations to solve the time-independent Schrödinger equation for this system. Plot your result and ponder what you observe.

**11.10** Can you think of an idea for how to extend our `numpy` codes for solving Schrödinger equations from 1D settings to 2D settings? If so, solve the time-independent Schrödinger equation for a 2D particle trapped in an in finite 2D potential well.

**Hint:** The time-independent Schrödinger equation for a 2D particle is given by

$$-\frac{\hbar^2}{2\,m}\left[\frac{\partial^2\psi(x,y)}{\partial x^2}+\frac{\partial^2\psi(x,y)}{\partial y^2}\right]+\hat{V}(x,y)\,\psi(x,y)=E\,\psi(x,y) \qquad (11.99)$$

and it is a common idea to consider an ansatz like this

$$\psi(x,y)=\xi(x)\cdot\eta(y)\;. \qquad (11.100)$$

Say you were to discretize these expressions. Would there be a connection to vector outer products as discussed in Chap. 3?

## References

1. Teschl, G.: Ordinary Differential Equations and Dynamical Systems. AMS (2012)
2. Perko, L.: Differential Equations and Dynamical Systems, 3rd edn. Springer (2013)
3. Bauckhage, C.: Numerically Solving Schrödinger Equations (1). Tech. rep., Lamarr Institute for ML and AI (2022). Available on researchgate.net
4. Bauckhage, C.: Numerically Solving Schrödinger Equations (2). Tech. rep., Lamarr Institute for ML and AI (2022). Available on researchgate.net

# Chapter 12
# Adiabatic Quantum Computing

## 12.1 Introduction

Finally, it is time to look at quantum computing. But why took it so long? Well, one of the main challenges for people who are beginning to learn about quantum computing is to wrap their heads around the meaning and implications of the following general statement.

While the basic units of information on a digital computer are *bits* and the mathematics that governs their behavior is *Boolean algebra*, the basic units of information on a quantum computer are *quantum bits* or *qubits* and the mathematics that governs their behavior is *linear algebra* in *complex tensor product spaces* which, moreover, has to comply with the *axioms of quantum mechanics*.

For us, the challenge will be minor by now. We already know about tensor spaces, the axioms of quantum mechanics, the role of Schrödinger equations, and the concept of wave functions and their superposition. We already know about Hermitian- and unitary operators and the notion of the Hamiltonian of a system, its eigenstates and eigenenergies. Even better, from our study of Hopfield nets and their interpretation in terms of quantum mechanical concepts, we already know that we may think of their micro states as two-dimensional vectors and of their macro states as tensor products. All these ideas will return now that we will be studying quantum computing.

Having said this, we note that there exist several quantum computing paradigms two of which stand out as of the writing of this book because corresponding hardware solutions are already being commercialized.

First and foremost, there is **quantum gate computing**. Its foundations date back to pioneering work by [1–5] and it conceptualizes quantum computing in terms of *quantum circuits* in which unitary operators or *quantum gates* manipulate quantum bits. In a sense, this generalizes today's omnipresent digital circuits in which logical operators manipulate classical bits. Quantum computers which follow this paradigm are being build and promoted by companies such as IBM, Google, IonQ, or Rigetti to name but a few. As is stands right now, they have reached a readiness level which

C. Bauckhage and R. Sifa, *Quantum Computing from Hopfield Nets*,
Cognitive Technologies, https://doi.org/10.1007/978-3-031-99402-9_12

demonstrates that quantum gate computing is technically feasible but still lacks the kind of robustness and scale that would allow practitioners to harness quantum advantages for real world applications.

And then there is **adiabatic quantum computing** based on ideas due to Farhi and colleagues [6, 7]. This paradigm is tailored towards solving combinatorial optimization problems and exploits a quantum mechanical principle known as the *adiabatic theorem* which, rather loosely speaking, describes a form of quantum tunneling. Quantum computing devices which follow this paradigm are being build, promoted, and sold by D-WAVE and have by now reached a readiness level which demonstrates that adiabatic quantum computing is technically feasible but not yet to an extent of robustness and scale that would provide advantages for real world applications

In this chapter, we will be concerned with adiabatic quantum computing because, given our study so far, we can almost immediately understand its underlying concepts. Indeed, adiabatic quantum computing and Hopfield nets for problem solving are very closely related concepts [8] and all the previous chapters have collectively prepared us to see why and how this is.

However, before we look at adiabatic quantum computing in particular, we will scrutinize the axiomatic foundations of quantum computing in general. This, too, will mainly consist of paraphrasing or re-contextualizing material we already know of. While this may sound less exciting, the following section is indeed the pinnacle of this book. Everything we studied up until now was to prepare us for its content and everything that will follow afterwards are but applications of its content.

Not only are we going to discuss and extensively comment on the fundamental postulates of quantum computing but we will also tie them back to our earlier studies on QUBO solving and Hopfield nets. We shall introduce the notions of *qubits* and *qubit systems* and their mathematical representations in terms of vectors and tensor products. Given these, we revisit the notions of the *evolution* of quantum systems and their *measurement* and tie them closer to qubits and qubit systems. In all of this, particular attention will be given to the fact that qubit systems can exist in a *quantum superposition* of exponentially many basis states which is one of the central advantages of quantum computing as it allows for a kind of parallelism which is generally infeasible on digital computers.

Our subsequent study of adiabatic quantum computing will illustrate this claim by means of two simple QUBO solving examples. We will state the *adiabatic theorem*, closely tie it to qubit systems, discuss the general ideas behind adiabatic quantum evolution for QUBO solving, and present simulation results which visualize how qubit systems can explore the search space of a QUBO or, equivalently, the state space of a Hopfield net in parallel. Speaking about simulations of adiabatic quantum computing, we furthermore present and discuss code snippets which show how to accomplish them. These, too, will only be based on concepts we are already familiar with but also take them to another level.

Finally, we conclude this chapter with a few quite technical remarks about adiabatic quantum computing and its relation to quantum gate computing. These will be good to know for practitioners even if we gloss over intricate physical details and detailed mathematical derivations.

## 12.2 The Axioms of Quantum Computing

In Chap. 10, we discussed the axioms of quantum mechanics and what they imply for the behaviors of quantum systems. Since quantum computing is but applied quantum mechanics, these axioms also hold for anything we could ever do with a quantum computer. And yet, it may be beneficial to see quantum computing as an area with its own, slightly more specialized quantum mechanical axioms which emphasize the specific nature of *qubits* and *qubit systems*.

Next, we therefore state and extensively comment upon four axioms which we will refer to as *our* axioms because they are not canonical. Compared to the postulates of quantum mechanics as a whole, they are tailored to the particular quantum systems considered in quantum information processing.

Indeed, qubits and qubit systems are the simplest kind of quantum systems we could ever think of and our first axiom of quantum computing illustrates this.

### *12.2.1 Qubits*

**Definition 12.1** (*1st axiom of quantum computing*) A **qubit** is a *two-state quantum system* which may exist in a *superposition* of two *orthonormal basis states*

$$\left|\psi\right\rangle = \alpha_0 \left|0\right\rangle + \alpha_1 \left|1\right\rangle \tag{12.1}$$

$$= \alpha_0 \begin{bmatrix}1\\0\end{bmatrix} + \alpha_1 \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}\alpha_0\\\alpha_1\end{bmatrix} . \tag{12.2}$$

The two coefficients $\alpha_0, \alpha_1 \in \mathbb{C}$ obey the *Born rule* or *normalization condition*

$$\left|\alpha_0\right|^2 + \left|\alpha_1\right|^2 = 1 . \tag{12.3}$$

In other words, a qubit is a unit length vector $\left|\psi\right\rangle$ in the Hilbert space $\mathbb{H} = \mathbb{C}^2$.

Did this definition really say that qubits are vectors? Does that make sense? After all, so far we always said that states of quantum systems are *represented* in terms of Hilbert space vectors.

Well, the conundrum resolves if we distinguish between **physical qubits** and **logical qubits**. The former are physical manifestations of two-state quantum systems; common examples include the spin of an electron (with basis states `up` / `down`) or the polarization of a photon (with basis states `vertical`/`horizontal`). The latter describe the mathematical essence of two-state quantum systems.

People who build and engineer quantum computers have to worry about physical qubits; people who design quantum algorithms usually think in terms of logical qubits. Since we count ourselves among the latter, we blissfully ignore the technical challenges associated with manipulating various instances of physical qubits and instead focus exclusively on logical qubits. And these are really just two-dimensional complex valued vectors.

Nevertheless, we still need the notion of instantiations or *states* of a qubit. This is because, in quantum computing, qubits are dynamical objects which evolve over time $t$. More generally, we should therefore write

$$\big|\psi(t)\big\rangle = \alpha_0(t)\,\big|0\big\rangle + \alpha_1(t)\,\big|1\big\rangle \tag{12.4}$$

and point out that this encodes the time dependency of the state of a qubit in the coefficients $\alpha_0(t)$ and $\alpha_1(t)$ rather than in the basis states $|0\rangle$ and $|1\rangle$. With respect to the latter, we further note that they form the so called **computational basis**

$$\mathbb{B} = \Big\{\,|0\rangle, |1\rangle\Big\} \tag{12.5}$$

which we recognize as the Boolean domain we studied at the end of Chap. 9. Here, we just expressed it using the Dirac notation.

But what does it actually mean to say that a qubit can exist in a superposition of its basis states? Nobody really knows! But let us dig deeper anyway.

First of all, when we studied wave functions of quantum particles in Chap. 11, we saw that the phenomenon of *quantum superposition* follows from taking the *Schrödinger equation* as an axiom of quantum mechanics. This still holds true for qubits; however, to simplify things, we now declared superposition to be axiomatic, too.

Second of all, we probably should point out that the right hand side of (12.1) is a very simple, discrete and only two-dimensional quantum wave function. To confirm this, we can perform a calculation similar to those we did in Chap. 11. Given (12.1), we do indeed have

$$\begin{aligned}
\big\langle\psi\,\big|\,\psi\big\rangle = \big|\psi\big\rangle^{\dagger}\,\big|\psi\big\rangle &= \Big[\alpha_0^*\big\langle 0\,\big| + \alpha_1^*\big\langle 1\big|\Big]\Big[\alpha_0\big|0\big\rangle + \alpha_1\big|1\big\rangle\Big] && (12.6)\\
&= \alpha_0^*\alpha_0\,\underbrace{\big\langle 0\,\big|\,0\big\rangle}_{=1} + \alpha_0^*\alpha_1\,\underbrace{\big\langle 0\,\big|\,1\big\rangle}_{=0} + \alpha_1^*\alpha_0\,\underbrace{\big\langle 1\,\big|\,0\big\rangle}_{=0} + \alpha_1^*\alpha_1\,\underbrace{\big\langle 1\,\big|\,1\big\rangle}_{=1} && (12.7)\\
&= \alpha_0^*\alpha_0 + \alpha_1^*\alpha_1 && (12.8)\\
&= \big|\alpha_0\big|^2 + \big|\alpha_1\big|^2 && (12.9)\\
&= 1\,. && (12.10)
\end{aligned}$$

Note that this calculation simply replaced the integration over continuous basis functions we used in Chap. 11 by summation over the discrete *basis functions* $|0\rangle$ and $|1\rangle$ and straightforwardly exploited that the latter are orthonormal.

Qubits are therefore very simple quantum systems indeed and our calculation shows that we can think of them as quantum mechanical wave functions. This then implies that everything we saw in the previous chapters also applies to qubits; things just become a bit simpler.

Of course it is anything but intuitive to say that a qubit is a quantum mechanical wave function. While a classical bit is *either* in state `on` *or* in state `off` and can thus be represented as a binary number $z \in \{0, 1\}$ or a bipolar number $s \in \{\pm 1\}$, a qubit which may exist in a superposition of vectors $\{|0\rangle, |1\rangle\}$ can somehow be in states `on` and `off` simultaneously.

This paradoxical behavior of quantum systems is essentially a consequence of their wave like nature and simply does not manifest in our everyday environments. Indeed, it appears so strange that people had to come up with crazy thought experiments such as *Schrödinger's cat* to try to break it down in terms of objects and occurrences we are all familiar with. And yet, it has been confirmed in numerous physics experiments and can be exploited for information processing.

Attentive readers may of course have realized that there is a catch and we shall return to this catch once we discuss *measurements* of qubit systems.

Why did we just mention qubit systems? Because, just as classical information processing, quantum information processing becomes more interesting and useful once we consider larger ensembles of qubits. Speaking about those, here is a notational convention which we will adhere to throughout the remainder of this book:

**Notation**

We henceforth denote individual qubits by

$$\left|\psi\right\rangle \quad \text{or} \quad \left|\psi_k\right\rangle$$

and systems comprised of one or more qubits by

$$\left|\Psi\right\rangle \quad \text{or} \quad \left|\Psi_j\right\rangle .$$

Note that this convention is *not* commonly used in the quantum computing literature but emphasizes the nature of the objects that appear in all the following equations.

To clarify the notion of a system comprised of one or more qubits, we next state our second axiom of quantum computing. While it may look innocent, it will finally reveal why our whole study began with QUBOs, continued with Hopfield nets, and led to where we are now.

### *12.2.2 Qubit Systems*

**Definition 12.2** (*2nd axiom of quantum computing*) The *joint state* of two qubit systems $|\Psi_1\rangle$ and $|\Psi_2\rangle$ can be modeled as a tensor product

$$\big|\Psi\big\rangle = \big|\Psi_1\big\rangle \otimes \big|\Psi_2\big\rangle . \tag{12.11}$$

Let us connect this axiom to material we studied in Chaps. 3, 9, and 10. To this end, let us consider the simplest possible example of a non-trivial qubit system, namely a system comprising only two qubits. To be as general as possible, let us further assume that both these qubits are in some superposition state. In short, let us consider

$$\big|\Psi\big\rangle = \big|\psi_1\big\rangle \otimes \big|\psi_2\big\rangle = \Big[\alpha_0 \big|0\big\rangle + \alpha_1\big|1\big\rangle\Big] \otimes \Big[\beta_0 \big|0\big\rangle + \beta_1\big|1\big\rangle\Big] . \tag{12.12}$$

Given what we worked out about the algebra of tensor products in Chap. 3, we know that we can expand the expression on the right hand side as

$$\big|\Psi\big\rangle = \alpha_0\beta_0 \big|0\big\rangle \otimes \big|0\big\rangle + \alpha_0\beta_1 \big|0\big\rangle \otimes \big|1\big\rangle + \alpha_1\beta_0 \big|1\big\rangle \otimes \big|0\big\rangle + \alpha_1\beta_1 \big|1\big\rangle \otimes \big|1\big\rangle \tag{12.13}$$

$$\equiv \gamma_0 \big|0\big\rangle \otimes \big|0\big\rangle + \gamma_1 \big|0\big\rangle \otimes \big|1\big\rangle + \gamma_2 \big|1\big\rangle \otimes \big|0\big\rangle + \gamma_3 \big|1\big\rangle \otimes \big|1\big\rangle . \tag{12.14}$$

To better understand the significance of this expression, we next rewrite the kets $|0\rangle$ and $|1\rangle$ in terms of the two-dimensional vectors they signify. This provides us with

$$\big|\Psi\big\rangle = \gamma_0\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} + \gamma_1\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} + \gamma_2\begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} + \gamma_3\begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} \tag{12.15}$$

$$\equiv \gamma_0\begin{bmatrix}1\\0\\0\\0\end{bmatrix} + \gamma_1\begin{bmatrix}0\\1\\0\\0\end{bmatrix} + \gamma_2\begin{bmatrix}0\\0\\1\\0\end{bmatrix} + \gamma_3\begin{bmatrix}0\\0\\0\\1\end{bmatrix} . \tag{12.16}$$

Despite its simplicity, there are several remarkable general properties of qubit systems which we can deduce from this example.

First of all, we have seen this specific qubit system before, namely in Exercise 10.11. We therefore already know that the coefficient $\gamma_j \in \mathbb{C}$ which we substituted in (12.14) are guaranteed to obey the Born rule. That is, their amplitudes $|\gamma_j|^2$ are guaranteed to sum to one so that $|\Psi\rangle$ is guaranteed to be a physically meaningful quantum system.

Note that this property of the coefficients of this two qubit tensor product state generalizes to tensor products of more than just two qubits which can be shown by inductively extending the proof from Exercise 10.11.

Second of all, while the individual qubits $|\psi_1\rangle$ and $|\psi_2\rangle$ are two-dimensional vectors in a superposition of two basis states, the system $|\Psi\rangle$ is a four-dimensional vector in a superposition of four basis states.

Given what we know from Chap. 3, this exponential growth of the dimension of the state spaces of qubit systems generalizes to tensor products of more than just two qubits. This insight is so important that we shall state it as a theorem.

**Theorem 12.1** *The joint state of a system of* $1 \leq k \leq n$ *qubits* $|\psi_k\rangle \in \mathbb{C}^2$ *can be modeled as a tensor product*

$$\big|\Psi\big\rangle = \big|\psi_1\big\rangle \otimes \big|\psi_2\big\rangle \otimes \cdots \otimes \big|\psi_n\big\rangle \in \mathbb{C}^{2^n} . \tag{12.17}$$

*Since each individual qubit can be in a superposition of two basis vectors, the whole system can be in a superposition of up to* $2^n$ *basis vectors.*

But why or how does this statement about qubit systems relate to our study of QUBO solving and Hopfield nets?

Well, compare the theorem to the content of Definition 9.2 in Chap. 9 where we studied Hopfield nets from the perspective of quantum mechanical models.

There, we only dealt with micro states $|z_k\rangle \in \{|0\rangle, |1\rangle\}$ and saw that their tensor products result in high dimensional one-hot vectors $|x_j\rangle$ which provide an alternative representation of the macro states of a Hopfield net. Here we are now dealing with qubits $|\psi_k\rangle$ which may exist in a *superposition* of their basis states $\{|0\rangle, |1\rangle\}$. Tensor products of such qubits can therefore result in states $|\Psi\rangle$ which are in a *superposition* of high dimensional one-hot vectors such as in (12.16).

Just think about what this generalization could mean for QUBO solving! If we tackled this task using a Hopfield net of $n$ neurons, the network would have $2^n$ macro states each of which could be represented as a $2^n$-dimensional one-hot vector. Crucially, however, the Hopfield net could only ever be in one such state at a time and thus would have to (cleverly) explore its state space in a sequential manner to solve the given QUBO.

On the other hand and crucially, a quantum system composed of $n$ qubits also corresponds to a $2^n$-dimensional vector but it can furthermore exist in a superposition of $2^n$ one-hot vectors. In other words, at any point in time, *a quantum system of $n$ qubits can exist in a state that represents all the possible macro states of a Hopfield net simultaneously!* Clever quantum algorithm could exploit such a superposition to solve QUBOs in a parallel manner.

But how could an $n$ qubit superposition state possibly look like? Well, consider this: When we work with a conventional Hopfield net, each of its $n$ neurons can be either `active` or `inactive` at any point in time. Since we represent these micro states in terms of bipolar numbers $s_k \in \{\pm 1\}$, the respective macro state of the network thus corresponds to a bipolar vector $\boldsymbol{s} \in \{\pm 1\}^n$.

However, if the micro states of a Hopfield net were qubits $|\psi_k\rangle$ instead, then each neuron could somehow be `active` and `inactive` simultaneously. A qubit state in which both these properties occur in equal shares is given by

$$\big|\psi_k\big\rangle = \frac{1}{\sqrt{2}}\Big[\big|0\big\rangle + \big|1\big\rangle\Big] \tag{12.18}$$

and the joint state of a system of $n$ such qubits amounts to

$$\begin{aligned}\big|\Psi\big\rangle &= \big|\psi_1\big\rangle \otimes \big|\psi_2\big\rangle \otimes \cdots \otimes \big|\psi_n\big\rangle &(12.19)\\ &= \frac{1}{\sqrt{2^n}} \sum_{z_1=0}^{1} \sum_{z_2=0}^{1} \cdots \sum_{z_n=0}^{1} \big| z_1\big\rangle \otimes \big| z_2\big\rangle \otimes \cdots \otimes \big| z_n\big\rangle \,. &(12.20)\end{aligned}$$

Note what we did here! We applied a very clever notational trick commonly used in the quantum computing literature.

**Notation**

It is often useful to write the computational basis states

$$\Big\{\, |0\rangle, |1\rangle \Big\}$$

in terms of expressions which involve binary variables

$$\Big\{\, |z\rangle \;\Big|\; z \in \{0, 1\} \Big\} \,.$$

---

Now, while the expression in (12.20) may look daunting, it really just generalizes the summation in (12.13) from a system of 2 qubits to a system of $n$ qubits. In a sense, it is even simpler because all $2^n$-dimensional basis states $|z_1\rangle \otimes |z_2\rangle \otimes \cdots \otimes |z_n\rangle$ in (12.20) have the same coefficient $1/\sqrt{2^n}$ whereas the $2^2$-dimensional basis states $|z_1\rangle \otimes |z_2\rangle$ in (12.13) each came with their own coefficients $\alpha_{z_1}\beta_{z_2}$.

Having said this, readers may now verify that the joint state $|\Psi\rangle$ which we computed in (12.20) is nothing but the $2^n$-dimensional vectors of all ones scaled by a factor of $1/\sqrt{2^n}$. But this is to say that $|\Psi\rangle$ is indeed a superposition

$$\big|\Psi\big\rangle = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} \big|\Psi_j\big\rangle \tag{12.21}$$

where each of the $2^n$ basis states $|\Psi_j\rangle$ is a $2^n$-dimensional one-hot vector.

Again, attentive readers may already see another catch and we will return to this catch once we discuss *measurements* of qubit systems.

One more thing: Looking, say, at (12.17), why did we write the joint state of the qubit system as $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle$ and not as $|\psi_n\rangle \otimes \cdots \otimes |\psi_2\rangle \otimes |\psi_1\rangle$ or any other permutation of the individual factors?

Well, this is just a modeling choice! With respect to modeling the joint state of a qubit system, all tensor products of all possible permutations of individual qubits would be *equivalent but different*. This is not uncommon in mathematical modeling of physical phenomena and harkens back to our discussion of ordered- and unordered sets and non-commutative multiplications at the very beginning of this book.

What we need to reason about a qubit system are the exponentially many coefficients of its exponentially many high dimensional basis states. Nature does not care how we do this. In fact, nobody knows how and where she represents the exponential amount of information contained in qubit systems; she just does. However, the mathematical tools we (successfully) use to describe her behavior require us to enumerate qubits and to multiply them in a certain order. Again, this is a restriction imposed by our linear algebraic toolbox rather than by nature.

In short, when working with mathematical expressions to model physical phenomena, we may have several options, must decide for one of them, and henceforth use it consistently.

### 12.2.3 Qubit Evolution

**Definition 12.3** (*3rd axiom of quantum computing*) The evolution of the state a qubit system is governed by the *Schrödinger equation*

$$i\,\hbar \frac{d}{dt} \big|\Psi(t)\big\rangle = \hat{H}(t)\,\big|\Psi(t)\big\rangle \tag{12.22}$$

where the Hermitian operator $\hat{H}$ represents the Hamiltonian of the system.

Compared to our first and second axioms of quantum computing, there is not much we can say about this third axiom because the Schrödinger equation is what it is.

However, generalizing our discussion from Chap. 9, we can and should say something about the Hamiltonian $\hat{H}(t)$ of a qubit system $|\Psi(t)\rangle$ at time $t$.

Consider a qubit system

$$\big|\Psi(t)\big\rangle = \sum_{j=0}^{2^n-1} \alpha_j(t)\,\big|\Psi_j\big\rangle = \sum_{j=0}^{2^n-1} \big|\Psi_j(t)\big\rangle \tag{12.23}$$

whose time-dependent high dimensional basis states $|\Psi_j(t)\rangle$ span the space $\mathbb{C}^{2^n}$. We already know that we may use such a collection of states to expand the $2^n \times 2^n$ Hamiltonian operator $\hat{H}(t)$ as

$$\hat{H}(t) = \sum_{j=0}^{2^n-1} E_j(t) \left|\Psi_j(t)\right\rangle\!\left\langle\Psi_j(t)\right| \tag{12.24}$$

where the $E_j(t)$ denote its eigenenergies at time $t$. Indeed, if we focus on the very special case where

$$a_j(t) = \begin{cases} 1 & \text{if } j = l \\ 0 & \text{otherwise ,} \end{cases} \tag{12.25}$$

that is, if we focus on the case where $|\Psi(t)\rangle$ is not in some superposition but in one of its basis state $|\Psi(t)\rangle = |\Psi_l(t)\rangle$, then the right hand side of the Schrödinger equation in (12.22) becomes

$$\hat{H}(t)\left|\Psi_l(t)\right\rangle = \sum_{j=0}^{2^n-1} E_j(t) \left|\Psi_j(t)\right\rangle\!\left\langle\Psi_j(t)\,\middle|\,\Psi_l(t)\right\rangle = E_l(t)\left|\Psi_l(t)\right\rangle . \tag{12.26}$$

Note that physicists call such a state $|\Psi_l(t)\rangle$ an **instantaneous eigenstate** of the system $|\Psi(t)\rangle$ because it is determined by the system's Hamiltonian $\hat{H}(t)$ at time instance $t$.

## 12.2.4 *Qubit Measurement*

**Definition 12.4** (*4th axiom of quantum computing*) *Measuring* the state of an $n$ qubit system which is in a superposition of $2^n$ basis states

$$\left|\Psi\right\rangle = \sum_{j=0}^{2^n-1} \alpha_j \left|\Psi_j\right\rangle \tag{12.27}$$

will cause the system to *collapse* to one of its basis states. The *probability* for collapse from $|\Psi\rangle$ to $|\Psi_j\rangle$ is given by

$$p\Big(\left|\Psi\right\rangle \mapsto \left|\Psi_j\right\rangle\Big) = \left|\alpha_j\right|^2 . \tag{12.28}$$

This axiom has dire consequence for quantum computing and quantum algorithm design. Consider, for instance, the case of just a single qubit ($n = 1$). As there are infinitely many $\alpha_0, \alpha_1 \in \mathbb{C}$ such that $|\alpha_0|^2 + |\alpha_1|^2 = 1$, a qubit can exist in infinitely many states $|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$.

However, we can never observe the *true* state of a qubit. Any observation or measurement we make, will cause it to collapse to either basis state $|0\rangle$ with probability $|\alpha_0|^2$ or to basis state $|1\rangle$ with probability $|\alpha_1|^2$.

All we can therefore ever extract from a single quantum bit is a single classical bit worth of information. In other words, measuring the state of a qubit is to transform quantum information into classical information.

Consequently, if we measure all the qubits of an $n$ qubit system, they will all probabilistically collapse $|\psi_k\rangle \mapsto |z_k\rangle$ where $|z_k\rangle \in \{|0\rangle, |1\rangle\}$. Their joint state will thus probabilistically collapse to some $|\Psi_j\rangle = |z_1\rangle \otimes |z_2\rangle \otimes \cdots \otimes |z_n\rangle$ even if the system was in a superposition state prior to measurement.

In other words, while a system of $n$ qubits can exist in a superposition of up to $2^n$ different states and thus represent a vast amount of information simultaneously, we can only ever get some particular configuration of its constituent qubits out of it.

All of this implies the following: *Quantum computing algorithm design is the quest for methods or processes that manipulate a qubit system such that it reaches a joint state which, when measured, is likely to collapse to a state that represents a solution to the problem the algorithm is supposed to tackle.*

Not only does this sound abstract, it usually really is abstract. However, the next section will show that we are by now well prepared to wrap our heads around one such kind of process.

## 12.3 QUBO Solving via Adiabatic Evolution

To make a long story short, adiabatic quantum computing is a technique for bipolar QUBO solving which we recall is the problem of finding

$$\boldsymbol{s}_* = \underset{\boldsymbol{s} \in \{\pm 1\}^n}{\operatorname{argmin}} -\tfrac{1}{2}\, \boldsymbol{s}^\intercal \boldsymbol{W} \boldsymbol{s} + \boldsymbol{b}^\intercal \boldsymbol{s} \ . \tag{12.29}$$

In a very loose sense, it takes the idea of (mean field) annealing the macro states $\boldsymbol{s} \in \{\pm 1\}^n$ of Hopfield nets to the idea of annealing the energy functions of Hopfield nets instead. Since adiabatic quantum computing is one of the paradigms of quantum information processing, this annealing of energy functions will obviously have to involve qubits and quantum mechanical operators. In order for our claim to make sense, there thus must exist an conceptual link between Hopfield states and energies on the one hand and qubit systems and quantum operators on the other hand.

But we already know about this link! We previously summarized it in Theorem 9.3 and repeat it here in a more quantum mechanical language.

A Hopfield net $(\boldsymbol{W}, \boldsymbol{b})$ with macro states $\boldsymbol{s}_j$ and energy function $E(\boldsymbol{s}_j)$ and its associated Hamiltonian operator $\hat{H}$ with eigenstates $|\Psi_j\rangle$ and eigenenergies $E_j$ are but two sides of the same coin.

Given this connection between the energy functions of Hopfield nets or, just as well, the objective functions of bipolar QUBOs and quantum Hamiltonian operators, the seminal idea of Farhi and colleagues [6, 7] was to harness the *adiabatic theorem* for quantum QUBO solving. Paraphrasing its original form due to Born and Fock [9], this theorem reads:

**Theorem 12.2** (The adiabatic theorem) *A quantum system will remain in its instantaneous eigenstate if it is perturbed slowly enough and there is a gap between its instantaneous eigenvalue and the rest of the spectrum of the system's Hamiltonian.*

Reading this, there is obviously a lot to unpack. However, we will refrain from an in-depth analysis since proving this quantum mechanical principle would take us too far afield. Instead, we refer to the beautiful and all in all easy to follow proof by Sakurai [10]. Moreover and more importantly, the theorem's statement is too general and too obscure for our purposes. Let us therefore rephrase it in terms that are closer to our present discussion.

**Theorem 12.3** (The adiabatic theorem w.r.t. qubit systems) *If an evolving qubit system starts at time $t = 0$ in one of the eigenstates $|\Psi_l(0)\rangle$ of some Hamiltonian operator $\hat{H}(0)$ which gradually changes to another Hamiltonian operator $\hat{H}(T)$, then, at time $t = T$, the system will end up in the corresponding eigenstate $|\Psi_l(T)\rangle$.*

*While $|\Psi_l(0)\rangle$ and $|\Psi_l(T)\rangle$ will generally be different quantum states because they are eigenstates of different operators*

$$\hat{H}(0)\,\big|\Psi_l(0)\big\rangle = E_l(0)\,\big|\Psi_l(0)\big\rangle \tag{12.30}$$

$$\hat{H}(T)\,\big|\Psi_l(T)\big\rangle = E_l(T)\,\big|\Psi_l(T)\big\rangle\,, \tag{12.31}$$

*their energy levels "l" will be the same. That is, if $|\Psi_l(0)\rangle$ is the $l$-th eigenstate of $\hat{H}(0)$, then $|\Psi_l(T)\rangle$ is the $l$-th eigenstate of $\hat{H}(T)$.*

To see how this quantum mechanical principle allows for QUBO solving, we recall an insight from Chap. 9. There we said, that, if $\hat{H}_P$ is the Hamiltonian associated with a Hopfield net or bipolar QUBO problem with parameters $(\boldsymbol{W}, \boldsymbol{b})$, then the task of finding a minimum Hopfield energy state that solves the QUBO problem can be formalized as

$$\begin{aligned} \left|\Psi_{P*}\right\rangle &= \underset{j}{\operatorname{argmin}}\ E_j \\ &\qquad \text{s.t.} \quad \hat{H}_P \left|\Psi_j\right\rangle = E_j \left|\Psi_j\right\rangle . \end{aligned} \tag{12.32}$$

Now, consider the following idea: If we could think of a **beginning Hamiltonian**

$$\hat{H}_B = \hat{H}(0) \tag{12.33}$$

whose minimum energy eigenstate or *ground state* $|\Psi_{B*}\rangle$ is easy to "prepare", then we could slowly evolve it to the **problem Hamiltonian**

$$\hat{H}_P = \hat{H}(T) \tag{12.34}$$

whose ground state $|\Psi_{P*}\rangle$ we want to find.

Indeed, given "appropriate" operators $\hat{H}_B$ and $\hat{H}_P$, we can define the following time-dependent Hamiltonian

$$\hat{H}(t) = \left(1 - \frac{t}{T}\right) \hat{H}_B + \frac{t}{T}\, \hat{H}_P \ . \tag{12.35}$$

Looking at this definition, we recognize an idea we previously saw in Chap. 8 where we used annealing algorithms to run Hopfield nets. There, we considered a linear annealing schedule to slowly transit from high to low temperatures; here, we are now using it to slowly transit from the beginning to the problem Hamiltonian.

Given this time-dependent Hamiltonian $\hat{H}(t)$, we can next "solve" the Schrödinger equation

$$i\,\hbar\, \frac{d}{dt} \left|\Psi(t)\right\rangle = \hat{H}(t) \left|\Psi(t)\right\rangle \tag{12.36}$$

at time points $0 \leq t \leq T$. If we start this process in the initial state $|\Psi(0)\rangle = |\Psi_{B*}\rangle$, then it will end in the the final state $|\Psi(T)\rangle = |\Psi_{P*}\rangle$.

Note our quoted use of the terms *prepare* and *solve*. Below, we will see that we can simulate this kind of adiabatic evolution of a quantum system on a digital computer. If we are dealing with $n$ qubit systems, this will require us to explicitly set up and work with number arrays of exponential sizes $2^n$ for states and $2^n \times 2^n$ for operators. This is of course only feasible for up to very moderate choices of $n$.

However, when working with an adiabatic quantum computer such as produced by D-WAVE, we would not have to worry about such limitations. While current technical challenges still limit the number of qubits we could consider, we would not have to worry about the exponential sizes of $|\Psi(t)\rangle$ and $\hat{H}(t)$. Rather, we would only have to dial the linearly many knobs of the machine for it to prepare physical manifestations of qubits and operators and to physically run the process. While any hardware details are too intricate for us to discuss in this book, the take home message is that, on quantum computers, nature takes care of the evolution of qubit systems and of the exponential sizes of quantum states and operators.

This brings us to our third quoted term and the question of how to come up with *appropriate* operators $\hat{H}_B$ and $\hat{H}_P$?

For the latter, we actually already saw the answer in Chap. 9 and therefore only need to translate our earlier ideas into the language of quantum computing. Hence, we recall that we previously worked with three curious matrices $\boldsymbol{I}$, $\boldsymbol{X}$, and $\boldsymbol{Z}$. In quantum computing these are known as (some of) the **Pauli matrices** and written as

$$I = \begin{bmatrix} +1 & 0 \\ 0 & +1 \end{bmatrix}, \quad X = \begin{bmatrix} 0 & +1 \\ +1 & 0 \end{bmatrix}, \text{and} \quad Z = \begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix}. \tag{12.37}$$

Now, in Chap. 9, we saw that, if we are given a Hopfield net or bipolar QUBO with $\boldsymbol{W} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{b} \in \mathbb{R}^n$, then the associated problem Hamiltonian of size $2^n \times 2^n$ can be computed as

$$\hat{H}_P = -\frac{1}{2} \sum_{k=1}^{n} \sum_{l=1}^{n} w_{kl} \, Z_k \, Z_l - \sum_{k=1}^{n} b_k \, Z_k \tag{12.38}$$

which indeed only requires quadratic efforts in $n$ if we let a quantum computer handle the exponentially large operators $Z_k$.

We therefore recall that these operators were defined as

$$Z_k = \underbrace{I \otimes I \otimes \cdots \otimes I}_{k-1 \text{ terms}} \otimes Z \otimes \underbrace{I \otimes I \otimes \cdots \otimes I}_{n-k \text{ terms}} \tag{12.39}$$

which means that they are indeed of size $2^n \times 2^n$ but only involve linearly many factors. We also note that, now that we understand them as quantum operators acting on qubits systems $|\Psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle$, they are commonly referred to as the Pauli spin matrix $Z$ acting on the $k$-th qubit.

But what about the beginning Hamiltonian? Well, Farhi and colleagues propose to simply set it to

$$\hat{H}_B = -\sum_{k=1}^{n} X_k \tag{12.40}$$

where $X_k$ denotes the Pauli spin matrix $X$ acting on the $k$-th qubit.

To see why this may be an operator whose ground state is easily prepared on a quantum computer, we observe that the two matrices $I$ and $X$ in (12.37) can be computed as $I = |0\rangle\langle 0| + |1\rangle\langle 1|$ and $X = |0\rangle\langle 1| + |1\rangle\langle 0|$. We furthermore recall the very special superposition state $1/\sqrt{2}|0\rangle + 1/\sqrt{2}|1\rangle$ of a single qubit from (12.18) and encourage the reader to verify that the following hold true

$$\tfrac{1}{\sqrt{2}}\Big[|0\rangle\langle 0| + |1\rangle\langle 1|\Big]\Big[|0\rangle + |1\rangle\Big] = \tfrac{1}{\sqrt{2}}\Big[|0\rangle + |1\rangle\Big] \tag{12.41}$$

$$\tfrac{1}{\sqrt{2}}\Big[|0\rangle\langle 1| + |1\rangle\langle 0|\Big]\Big[|0\rangle + |1\rangle\Big] = \tfrac{1}{\sqrt{2}}\Big[|0\rangle + |1\rangle\Big]. \tag{12.42}$$

**Algorithm 1** Adiabatic quantum evolution for bipolar QUBO solving

```
Input: QUBO or Hopfield net parameters $\boldsymbol{W} \in \mathbb{R}^{n \times n}$ and $\boldsymbol{b} \in \mathbb{R}^n$

  prepare the beginning- and problem Hamiltonians
      $\hat{H}_B = - \sum_{k=1}^{n} X_k$
      $\hat{H}_P = -\frac{1}{2} \sum_{k=1}^{n} \sum_{l=1}^{n} w_{kl} \, Z_k \, Z_l - \sum_{k=1}^{n} b_k \, Z_k$
  prepare an $n$ qubit system in the ground state of $\hat{H}_B$
      $\big|\Psi(0)\big\rangle = \big|\Psi_{B*}\big\rangle$
  gradually evolve the system from time $t = 0$ to time $t = T$ under a time-dependent Haltonian
      $i \, \hbar \frac{d}{dt} \big|\Psi(t)\big\rangle = \Big[\Big(1 - \frac{t}{T}\Big) \hat{H}_B + \frac{t}{T} \hat{H}_P\Big] \big|\Psi(t)\big\rangle$
  at time $T$, perform a measurement of the system to very likely obtain ground state $|\Psi_{P*}\rangle$ of $\hat{H}_P$
```

But if this is so, then readers should furthermore verify that the scaled vector of all ones from (12.20) is an eigenstate of any $X_k$ and, consequently, of $\hat{H}_B$. This state is indeed easily prepared on a quantum computer; while we are not yet going to discuss the whys and hows, we can at least hint at our brief discussion of Hadamard operators at the end of Chap. 9.

All in all, we can therefore summarize the idea of adiabatic quantum evolution for bipolar QUBO solving as shown in Algorithm 1.

But how would an adiabatic evolution of a system of qubits look like if we could watch it? Well, on a real quantum computer, we could never watch the evolution of a qubit system

$$\big|\Psi(t)\big\rangle = \sum_{j=0}^{2^n-1} \alpha_j(t) \, \big|\Psi_j\big\rangle \tag{12.43}$$

because there is nothing to see in the literal sense of the word and because any observation would destroy the system's superposition state. But we can simulate adiabatic quantum computing on digital computers and visualize the evolution of the amplitudes $|\alpha_j(t)|^2$ of superposition states.

So let us do just that with respect to two exemplary QUBO problems. In particular, let us consider two tiny subset sum problems of problem size $n = 8$, namely

$$\text{SSP 1} : \Big(\big\{12, 7, 6, 10, 3, 5, 9, 7\big\}, 11\Big) \tag{12.44}$$

$$\text{SSP 2} : \Big(\big\{12, 7, 6, 10, 3, 5, 9, 6\big\}, 11\Big) \,. \tag{12.45}$$

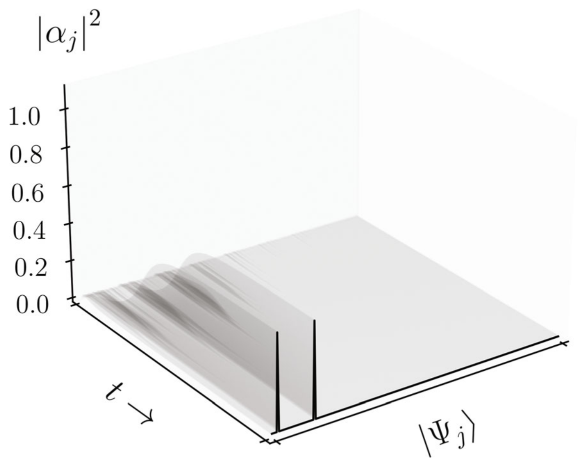


**Fig. 12.1** Visualization of the adiabatic evolution of an 8 qubit system that solves the SSP in (12.44)

Can you spot the minute difference between both SSPs? This minute difference causes the first SSP to have just one solution and the second SSP to have two solutions.

If we represent these solutions in terms of binary indicator vectors, then they are given by

$$z_* = [0, 0, 1, 0, 0, 1, 0, 0]^\intercal \tag{12.46}$$

for the first SSP and by

$$z_{1*} = [0, 0, 1, 0, 0, 1, 0, 0]^\intercal \tag{12.47}$$

$$z_{2*} = [0, 0, 0, 0, 0, 1, 0, 1]^\intercal \tag{12.48}$$

for the second SSP and we note that these representations also answer the question about the slight difference between both problems.

For the following, we simply refer to our discussion in Chap. 5 where we showed how to formalize plain vanilla SSPs like these in terms of binary- and bipolar QUBOs. In other words, we next simply assume the parameters $\boldsymbol{W} \in \mathbb{R}^{8\times 8}$ and $\boldsymbol{b} \in \mathbb{R}^8$ of bipolar QUBO formulations of our SSPs to be given.

If we use them in Algorithm 1 and then digitally simulate the evolution of the respective 8 qubit systems for a sequence of $O(\sqrt{2^8}) = O(16)$ little time steps, we obtain results as shown in Figs. 12.1 and 12.2.

Both systems begin their evolution in a state $|\Psi(0)\rangle$ which is in a superposition of $2^8 = 256$ basis states $|\Psi_j\rangle$. Each of these basis states represents a potential solution to the respective QUBO. Initially, all these basis states have the same but small probability amplitude of $|\alpha_j(0)|^2 = 1/256$ for being observed upon measurement. However, over time $t$, their coefficients $\alpha_j(t)$ begin to grow or shrink which makes some basis states more likely to be measured than others.

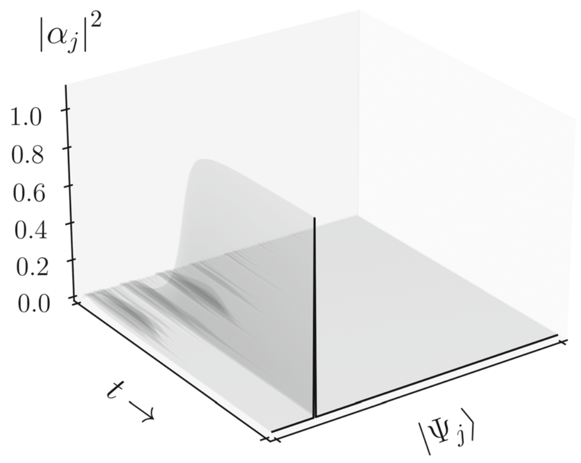


**Fig. 12.2** Visualization of the adiabatic evolution of an 8 qubit system that solves the SSP in (12.45)

At the end of its evolution, the first system is in a state $|\Psi(T)\rangle$ which coincides with one of its basis states $|\Psi_l\rangle$. This state has a probability of $|\alpha_l(T)|^2 = 1$ for being measured; for all the other basis states this probability has dropped to zero. The second system ends up in a state which is in a superposition of two basis states. The probabilities for either of these to be observed upon measurement are $|\alpha_{l_1}(T)|^2 = |\alpha_{l_2}(T)|^2 = 0.5$; for all other basis states, they are zero.

Scrutinizing the basis states with non-zero measurement probabilities, we realize that they amount to

$$\big|\Psi_{16}\big\rangle = \big|0\big\rangle \otimes \big|0\big\rangle \otimes \big|1\big\rangle \otimes \big|0\big\rangle \otimes \big|0\big\rangle \otimes \big|1\big\rangle \otimes \big|0\big\rangle \otimes \big|0\big\rangle \tag{12.49}$$

in case of the qubit systems which solves the SSP in (12.44) and to

$$\big|\Psi_{16}\big\rangle = \big|0\big\rangle \otimes \big|0\big\rangle \otimes \big|1\big\rangle \otimes \big|0\big\rangle \otimes \big|0\big\rangle \otimes \big|1\big\rangle \otimes \big|0\big\rangle \otimes \big|0\big\rangle \tag{12.50}$$

$$\big|\Psi_{5}\big\rangle = \big|0\big\rangle \otimes \big|0\big\rangle \otimes \big|0\big\rangle \otimes \big|0\big\rangle \otimes \big|0\big\rangle \otimes \big|1\big\rangle \otimes \big|1\big\rangle \otimes \big|0\big\rangle \tag{12.51}$$

in case of the qubit systems which solves the SSP in (12.45). Comparing these quantum states to the binary vectors in (12.46)–(12.48), is also becomes clear what it means for the state of a quantum system to represent the solution to a QUBO.

In both processes, our qubit systems have evolved to a joint state for which a measurement causes the individual qubits $|\psi_1\rangle, |\psi_2\rangle, \ldots, |\psi_n\rangle$ to collapse to either one of the two single qubit basis states $|0\rangle$ or $|1\rangle$. As we can see, the resulting qubit instances $|z_1\rangle, |z_2\rangle, \ldots, |z_n\rangle$ can therefore be understood as quantum analogs of the entries of the classical binary indicator vectors $\boldsymbol{z} = [z_1, z_2, \ldots, z_n]^\intercal$ in (12.46)–(12.48). In short, after measurement, the state of qubit $|\psi_k\rangle$ will be $|z_k\rangle \in \{|0\rangle, |1\rangle\}$ and thus indicate whether or not integer $x_k$ contributes to the solution of the respective SSP.

```
import numpy as np
import scipy.sparse as sprs
import scipy.sparse.linalg as sprsla
from functools import reduce

opI = sprs.csr_matrix([[+1, 0], [ 0,+1]])
opX = sprs.csr_matrix([[ 0,+1], [+1, 0]])
opZ = sprs.csr_matrix([[+1, 0], [ 0,-1]])

def tprod(os):
    return reduce(sprs.kron, os)
```

**Code 12.1** Preparatory imports and definitions for the simulation of adiabatic quantum computing.

## 12.4 Simulating Adiabatic Quantum Evolution

Next, we discuss how we simulated the processes in Figs. 12.1 and 12.2. Given all our code snippets and coding exercises so far, the following should be straightforward. But let us start with two disclaimers anyway.

First of all, simulating the behavior of $n$ qubit systems will be taxing if neigh impossible for standard laptop- or desktop computers once $n \gtrsim 25$. Our following code snippets should thus be used considerately.

Second of all, we will need to implement the operators $\hat{H}_P$ and $\hat{H}_B$ in (12.38) and (12.40). These are large but only finitely sized matrices. More interestingly, they also are *sparse* matrices which means that they contain many more zero entries than non-zero ones. We can exploit this to lessen the burden for our digital computers and will work with methods for sparse matrix manipulation provided in the `scipy` modules `sparse` and `sparse.linalg`. Using these, $\hat{H}_P$ and $\hat{H}_B$ will not be represented in terms of dense $2^n \times 2^n$ arrays but in terms of specialized data structures. In particular, we will be using the *compressed sparse row matrix representation* (`csr_matrix`) which we found to work well for our purposes.

With that being said, code snippet 12.1 performs the imports we require. It also sets up three sparse matrices `opI`, `opX`, and `opZ` which represent the $2 \times 2$ Pauli matrices in (12.37). Finally, it defines a function `tprod` which takes a list `os` of quantum operators $O_1, O_2, \ldots, O_n$ and returns their tensor product $O_1 \otimes O_2 \otimes \cdots \otimes O_n$. Looking at this function, we note our use of `reduce` from the `python` standard library `functools` which cumulatively applies the sparse Kronecker product `sprs.kron` to the elements of the input list `os`.

Code snippet 12.2 defines a function `prepareAQC` which takes array representations `matW` and `vecB` of the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ of a QUBO as input and returns sparse matrix representations of the Hamiltonians $\hat{H}_B$ and $\hat{H}_P$ required for adiabatic quantum computing. First, it computes two lists `Xs` and `Zs` which contain sparse matrix representations of the $1 \leq k \leq n$ quantum operators $X_k$ and $Z_k$ needed for computing the beginning- and problem Hamiltonian. Using list `Zs`, it computes a sparse matrix `opHp` representing $\hat{H}_P$ as defined in (12.38) and, using list `Xs`, it computes a sparse matrix `opHb` representing $\hat{H}_B$ as defined in (12.40). Looking at

```python
def prepareAQC(matW, vecB):
    ns = range(len(vecB))
    Xs = [tprod([opX if k==l else opI for k in ns]) for l in ns]
    Zs = [tprod([opZ if k==l else opI for k in ns]) for l in ns]

    opHp  = -0.5 * sum([matW[k,l] * Zs[k] * Zs[l] for k in ns for l in ns])
    opHp -=        sum([vecB[k  ] * Zs[k]          for k in ns])

    opHb = -sum([Xs[j] for j in ns])

    return opHb, opHp
```

**Code 12.2** Simple `scipy` code to compute sparse matrix representations of the two Hamiltonian operators $\hat{H}_P$ and $\hat{H}_B$ according to their definitions in (12.38) and (12.40).

```python
def simulateAQC(opHb, opHp, tmax, dt=0.1):
    stps = np.linspace(0, tmax, int(tmax/dt), endpoint=True)
    amps = []

    for t in stps:
        opH = (1-t/tmax) * opHb + t/tmax * opHp

        try:
            val, psi = sprsla.eigsh(opH, k=1, which='SA')
        except:
            break

        psi = psi[:,0]
        amp = np.abs(psi)**2
        amps.append(amp)

    return np.array(amps)
```

**Code 12.3** Simple `scipy` code for simulating the adiabatic evolution of a qubit system by repeatedly computing the bottom eigenvector of the system's time-dependent Hamiltonian.

this snippet, we emphasize our use of "$-$" signs as prescribed by the definitions in (12.38) and (12.40).

Once `opHb` and `opHp` have been computed, they can be used in procedures for simulating adiabatic evolutions of respective qubit systems. Code snippet 12.3 presents one such procedure. The parameters of function `simulateAQC` represent the Hamiltonians $\hat{H}_B$ and $\hat{H}_P$, the intended duration $T$ of an adiabatic evolution, and an increment $\Delta t$ for gradually stepping from time $t$ to $t + \Delta t$. Given the parameters for $T$ and $\Delta t$, the function first sets up an iterator `stps` containing time steps and an empty list `amps` in which to store amplitudes $|\alpha_j(t)|^2$ of evolving basis states $\alpha_j(t)\,|\Psi_j\rangle$.

**Note:** *What follows next is strictly speaking cheating*. We do not actually simulate adiabatic evolutions by solving corresponding Schrödinger equations. This could be done using `scipy.integrate` methods such as `ode` or `solve_ivp` but would be slow and brittle and therefore typically require careful fiddling with tolerance parameters and the like.

However, we can afford to cheat as our main interest is in visualizing the adiabatic evolution of a qubit system that starts out in a *ground state* of a known beginning Hamiltonian $\hat{H}(0) = \hat{H}_B$. Taking the adiabatic theorem by its word and boldly assuming that there is a gap between the ground state and the rest of the spectrum of $\hat{H}(t)$, we proceed as follows: We iterate over the time points `t` in `stps`. In each iteration, we first compute a sparse matrix `opH` representing $\hat{H}(t)$ as defined in (12.35). Given this matrix, we then determine its bottom eigenvalue and eigenvector. This is possible using the `scipy.sparse.linalg` method `eigsh` which is tailored to complex Hermitian matrices and whose parameters `k` and `which` allow for control of how many and what kind of einegnvalue / eigenvectors pairs are to be determined. Here, we opt for just one pair where the eigenvalue is of smallest size. Also note our use of `try` and `except` which are intended to catch potential numerical instabilities.

In each iteration, the bottom eigenvector of $\hat{H}(t)$ is stored in array `psi` and the squared norms of its entries are stored in array `amp` which is furthermore appended to list `amps`. Once the loop has terminated, `amps` is turned into a 2D `numpy` array and returned.

Finally, assuming we had appropriate array representations `matW` and `vecB` of the parameters of a given bipolar QUBO, we may simply use something like

```
matHb, matHp = prepareAQC(matW, vecB)
amps = simulateAQC(matHb, matHp, tmax=2**(len(vecB)/2), dt=0.05)
```

to get an array `amps` whose rows we may "pretty plot" in order to create visualizations similar to the ones in Figs. 12.1 and 12.2.

## 12.5 Caveats and Unjustified Criticism

Let us conclude our study of adiabatic quantum computing by discussing several caveats practitioners should be aware of. The following remarks will be fairly technical and we shall gloss over their derivation. Since this may sound unsatisfactory, we note that Theorem 12.2 is a statement about a *physical principle* rather than a *mathematical theorem*. In other words, the adiabatic theorem states an observation about nature which requires mathematical modeling; its *proof* is therefore not a *mathematical proof* but consists of physically plausible reasoning [10] and we do not want to dive too deep into this physics.

First of all, what is with the *gap* mentioned in Theorem 12.2? Well, let us again narrow down the answer to the $n$ qubits setting in adiabatic quantum computing. Considering the Hamiltonian $\hat{H}(t)$ in (12.35), we may refer to its eigenenergies as $E_0(t) \leq E_1(t) \leq \cdots \leq E_{2^n-1}(t)$ where $E_0(t)$ is the energy of its *ground state* $|\Psi_0(t)\rangle$ and $E_1(t)$ is the energy of its *first excited state* $|\Psi_1(t)\rangle$.

Defining

$$g_{\min} = \min_t \big(E_1(t) - E_0(t)\big) \,, \tag{12.52}$$

one can show [6, 10] that the phenomenon described by the adiabatic theorem will only hold if

$$\max_t \frac{\left\langle \Psi_1(t) \,\middle|\, \frac{d}{dt}\hat{H}(t) \,\middle|\, \Psi_0(t) \right\rangle}{g_{\min}^2} \ll 1 \tag{12.53}$$

which means that $g_{\min}$ must not be too small. This is no trivial matter because, when we discussed a quantum tunneling example in Chap. 11, we literally saw that energy gaps between ground- and first excited states of a quantum system can be small.

This, in turn, may impact attempts of QUBO solving via adiabatic quantum computing. Indeed, recall our study in Chap. 4 where we visualized energy landscapes of different QUBO models and saw that energy differences between global minimizers and other states can be small. However, we also said that corresponding challenges can be overcome by carefully re-parameterizing the respective QUBO model [11, 12]. In short, if practitioners know what they are doing, then adiabatic quantum computing is very widely applicable.

Second of all, the Hamiltonian $\hat{H}(t)$ in (12.35) can have several ground states. We actually saw this for the example of the simple SSP in (12.45) which has two solutions so that the corresponding problem Hamiltonian $\hat{H}_P$ has two eigenstates of minimum energy. With respect to (12.52), this implies $g_{\min} = 0$ so that (12.53) no longer makes sense. When adiabatic quantum computing was proposed in the early 2000s, this apparent breakdown of the adiabatic theorem caused some controversy in the physics community but was quickly resolved [13]. Indeed, by carefully revising traditional proofs of the adiabatic theorem, one can show that a quantum system can adiabatically evolve to states $|\Psi(T)\rangle$ which are in a superposition of several eigenstates of $\hat{H}(T)$ and we literally saw this in Fig. 12.2.

Third of all, what about our choice of a running time of $T \in O(\sqrt{2^n})$ in our above practical experiments? Well, one can show [6, 10] that, for

$$T \in O\left(\frac{1}{g_{\min}^2}\right), \tag{12.54}$$

it is exceedingly likely that $\langle \Psi_1(t) \,|\, \frac{d}{dt}\hat{H}(t) \,|\, \Psi_0(t)\rangle \to 0$ as $t \to T$. This dictates how long, or, vice verse, how slowly an adiabatic evolution has to run for the adiabatic theorem to hold, namely on a scale inversely proportional to the gap $g_{\min}$.

One can also show [14] that the size of $g_{\min}$ is inversely proportional to the square root of the number of eigenstates states $|\Psi_j\rangle$ of $\hat{H}_P$ with eigenenergies $E_j$ close to the ground state energy $E_0$. Moreover, for settings where this number is much smaller than the total number of eigenstates, one can show [15] that $T \in O(\sqrt{2^n})$ is the smallest possible running time for a successful adiabatic evolution. But this corroborates our claim from Chap. 1 where we said that adiabatic quantum computing can successfully solve combinatorial problems quadratically faster than exhaustive searches which are the only classical methods that can guarantee success.

Finally, we should mention that it is a popular trope among parts of the quantum computing community to see adiabatic quantum computing as inferior to quantum gate computing. The argument goes that adiabatic quantum computing merely performs a form a quantum annealing which only allows for QUBO solving and is thus not as universal as quantum gate computing. However, both paradigms are indeed equivalent in the sense that it only requires polynomial efforts and resources to transit from one model to the other [16]. Crucially, this may require a more general view on adiabatic quantum computing than the one we discussed here. Indeed, the problem Hamiltonians $\hat{H}_P$ we were concerned with are of simple diagonal structure (recall our earlier discussion in Chap. 9) but, in general, we could also consider adiabatic processes with more intricate Hamiltonian operators which would allow for applications beyond QUBO solving. In short, if we wanted to then everything we can do with a quantum gate computer can be ported to an adiabatic quantum computer and everything we can do with an adiabatic quantum computer can be ported to a quantum gate computer.

## 12.6 Exercises

**12.1** Note that our code for simulating adiabatic quantum computation does not include functionalities for simulating quantum measurements. So, implement those for yourself. Proceed as follows: Given the amplitudes $|\alpha_j(T)|^2$ at the end of an adiabatic evolution, sample an integer

$$j \sim |\alpha_j(T)|^2 \tag{12.55}$$

by using, say, the `numpy.random` method `choice`. Carefully read the man pages of this function to make sure you use it to its full potential. Once $j$ is available, you can think of it as the index of the basis state

$$\left|\Psi_j\right\rangle = \left|z_1\right\rangle \otimes \left|z_2\right\rangle \otimes \cdots \otimes \left|z_n\right\rangle \tag{12.56}$$

which results from a measurement performed on $|\Psi(T)\rangle$. To visualize or print this basis state $|\Psi_j\rangle$, you may feed the integer $j$ into function `num2bits` from Chap. 9 to turn it into a binary vector $z$ whose entries $z_j$ represent the states of the single qubit basis vectors $|z_j\rangle$.

**12.2** Create your own (small) instances of all of the QUBO models we discussed in Chap. 5 and then solve them via simulated adiabatic quantum computing. Use your result form the previous exercise to simulate a quantum measurement of the final state of the corresponding qubit system and see if your results make sense, i.e. see if they represent a solution to the QUBO at hand.

**12.3** While their main interest is in genuine quantum computing, Farhi et al. [7] point out that digitally simulated adiabatic quantum computing is yet another algorithm for solving generally NP-hard QUBO problems, albeit a not at all efficient one. They therefore present experiments in which they terminate a simulated adiabatic quantum evolution once there is any amplitude $|a_j(t)|^2$ which exceeds a value of $1/8$. Adapt our code snippets correspondingly and repeat your above experiments to see if such an early stopping is capable of solving QUBOs.

## References

1. Manin, Y.: Computable and Noncomputable (in Russian). Kibernetika. Sovetskoye Radio, Moscow (1980)
2. Benioff, P.: The Computer as a Physical System: A Microscopic Quantum Mechanical Hamiltonian Model of Computers as Represented by Turing Machines. J. of Statistical Physics **22**(5) (1980). https://doi.org/10.1007/BF01011339
3. Feynman, R.: Simulating Physics with Computers. Int. J. of Theoretical Physics **21** (1982). https://doi.org/10.1007/BF02650179
4. Deutsch, D.: Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. Proc. of the Royal Society London A **400**(1818) (1985). https://doi.org/10.1098/rspa.1985.0070
5. Feynman, R.: Quantum Mechanical Computers. Foundations of Physics **16**(6) (1986). https://doi.org/10.1007/BF01886518
6. Farhi, E., Goldstone, J., Gutmann, S., Sipser, M.: Quantum Computation by Adiabatic Evolution. arXiv:quant-ph/0001106 (2000)
7. Farhi, E., Goldstone, J., Gutmann, S., Lapan, J., Lundgren, A., Preda, D.: A Quantum Adiabatic Evolution Algorithm Applied to Random Instances of an NP-Complete Problem. arXiv:quant-ph/0104129 (2001)
8. Bauckhage, C., Sanchez, R., Sifa, R.: Problem Solving with Hopfield Networks and Adiabatic Quantum Computing. In: Proc. Int. Joint Conf. on Neural Networks. IEEE (2020). https://doi.org/10.1109/IJCNN48605.2020.9206916
9. Born, M., Fock, V.: Beweis des Adiabatensatzes. Zeitschrift für Physik **51**(3–4) (1928). https://doi.org/10.1007/BF01343193
10. Sakurai, J., Napolitano, J.: Modern Quantum Mechanics, 3rd edn. Cambridge University Press (2020)
11. Gerlach, T., Mücke, S.: Investigating the Relation Between Problem Hardness and QUBO Properties. In: I. Miliou, N. Piatkowski, P. Papapetrou (eds.) Advances in Intelligent Data Analysis, *LNCS*, vol. 14642. Springer (2024). https://doi.org/10.1007/978-3-031-58553-1_14
12. Mücke, S., Gerlach, T., Piatkowski, N.: Optimum-preserving QUBO Parameter Compression. Quantum Macine Intelligence **7**(1) (2025). https://doi.org/10.1007/s42484-024-00219-3
13. Sarandy, M., Wu, L., Lidar, D.: Consistency of the Adiabatic Theorem. Quantum Information Processing **3**(6) (2004). https://doi.org/10.1007/BF01343193
14. Amin, M.: Effect of Local Minima on Adiabatic Quantum Optimization. Physical Review Letters **100**(13) (2008)
15. Roland, J., Cerf, N.: Quantum Search by Local Adiabatic Evolution. Physical Review A **65**(4) (2002)
16. Aharonov, D., et al.: Adiabatic Quantum Computation Is Equivalent to Standard Quantum Computation. In: Proc. Symp. on Foundations of Computer Science. IEEE (2004). https://doi.org/10.1109/FOCS.2004.8

# Chapter 13
# An Outlook to Quantum Gate Computing

## 13.1 Introduction

This final chapter does what its title promises: It provides an outlook to the possibilities of quantum gate computing. Having spent almost an entire book on working out that adiabatic quantum computing is, loosely speaking, like running Hopfield nets on steroids, we now depart from our story line of problem solving as energy minimization. This is necessary because the ideas behind quantum gate computing differ from most of what we studied so far. Nevertheless, our study up to this point has us prepared for what will follow.

To begin with, we provide a different view on how to understand the *evolution* of qubit systems. That is, we will redefine our third axiom of quantum computing and restate it in a manner that is closer to the spirit of quantum gate computing. Indeed, this paradigm posits that quantum algorithms are *sequences of unitary operators* which, one after the other, act on the joint state of a qubit system. However, there still is a connection to the Hamiltonian point of view we assumed thus far and, to see it clearly, we will prove one of our claims in Chap. 10 where we said that *every unitary operator is an exponential of some Hermitian operator*.

Given this connection, we then look at *quantum circuits* which are the quantum computing analog of logic circuits in digital computing. We will see that a quantum circuit consists of quantum gates interconnected by quantum wires where the latter represent the states of qubits on which the former perform unitary operations. While this sounds similar to what happens in digital circuits, quantum circuits can only involve *reversible operations* and we briefly discuss why this is.

As a practical example for this discussion, we consider the problem of devising quantum circuits which compute univariate Boolean functions. While these are of limited practical utility, they played a key role in seminalwork by David Deutsch [1] who showed that there exist computational problems for which quantum computing

C. Bauckhage and R. Sifa, *Quantum Computing from Hopfield Nets*,
Cognitive Technologies, https://doi.org/10.1007/978-3-031-99402-9_13

algorithms can be exponentially faster than their classical counterparts. To see this for ourselves, we finally look at *Deutsch's problem* and *Deutsch's algorithm* and discuss how to simulate the latter using plain vanilla `numpy` code.

## 13.2 A Different View on Qubit Evolution

In the preceding chapters, we have been working with the Schrödinger equation

$$i\,\hbar\,\frac{d}{dt}\,\big|\Psi(t)\big\rangle = \hat{H}(t)\,\big|\Psi(t)\big\rangle \tag{13.1}$$

for which we recall that $|\Psi(t)\rangle$ represents the state of a quantum system at time $t$ and that $\hat{H}(t)$ is the time-dependent Hamiltonian of said system. Now, that we are interested in quantum gate computing, we better point out an alternative form of (13.1) which we actually already saw it in Chap. 10 when we discussed unitary operators.

There we considered the simple case of a time-independent Hamiltonian $\hat{H}(t) = \hat{H}$ which we claimed made it easy to solve the above differential equation. In short, we claimed that

$$\big|\,\Psi(t)\big\rangle = \int_0^t -\frac{i}{\hbar}\hat{H}\,\big|\,\Psi(\tau)\big\rangle\,d\tau = e^{-i\,t\,\hat{H}/\hbar}\,\big|\,\Psi(0)\big\rangle \equiv U(t,0)\,\big|\,\Psi(0)\big\rangle \tag{13.2}$$

where

$$U(t,0) = e^{-i\,t\,\hat{H}/\hbar} \tag{13.3}$$

is a unitary operator often called the *time evolution operator* or *propagator*. But if this is so, then

$$\big|\,\Psi(t)\big\rangle = U(t,0)\,\big|\,\Psi(0)\big\rangle \tag{13.4}$$

is indeed a valid alternative of stating the Schrödinger equation. Alas, it is not quite yet the form of interest in quantum gate computing.

To get to this form, we could dive into yet another rigorous discussion of quantum operators. But we can also leave the rigor to the physicists [2, 3] and argue more intuitively. We hence recall Exercise 10.8 which showed that the product of any two unitary operators is another unitary operator. Armed with this insight, we may think of the "long term" propagator $U(t,0)$ in (13.3) and (13.4) as a product of a sequence of several "short term" propagators, say

$$U(t,0) = U(t,t_n)\,U(t_n,t_{n-1})\cdots U(t_2,t_1)\,U(t_1,0)\ . \tag{13.5}$$

With this picture in mind, we may then think of the continuous evolution from a quantum state $|\Psi(0)\rangle$ to another quantum state $|\Psi(t)\rangle$ as a sequence of discrete steps

$$\big|\Psi(t_1)\big\rangle = U(t_1, 0)\ \big|\Psi(0)\big\rangle \tag{13.6}$$

$$\big|\Psi(t_2)\big\rangle = U(t_2, t_1)\ \big|\Psi(t_1)\big\rangle \tag{13.7}$$

$$\vdots$$

$$\big|\Psi(t)\big\rangle = U(t, t_n)\ \big|\Psi(t_n)\big\rangle\ . \tag{13.8}$$

If we then focus on qubit systems and ignore long term developments but just look at quantum states $|\Psi\rangle$ and $|\Psi'\rangle$ before and after the application of some propagator $U$, we may redefine our third axiom of quantum computing.

**Definition 13.1** (*Alternative form of the 3rd axiom of quantum computing*) Over a (very) short period of time, the state of a qubit system changes according to some unitary transformation

$$\big|\Psi'\big\rangle = U\ \big|\Psi\big\rangle \tag{13.9}$$

where $U = e^{i\,\hat{K}}$ for some Hermitian operator $\hat{K}$.

This definition now reflects how the evolution of a qubit system is commonly seen in quantum gate computing and of course there is a lot to be said about it. First and foremost, quantum gate computing does indeed consider sequences of unitary operators acting on qubit systems and we shall discuss this view soon.

Further note that the Hermitian operator $\hat{K}$ we mention in the definition is really but "some" Hermitian operator and not the system's overall Hamiltonian $\hat{H}$ in (13.3). We simply consider $\hat{K}$ to be some constant Hamiltonian at some point during the evolution of the system and also subsume the division by $-\hbar$ into $\hat{K}$ for brevity.

Finally, we should verify that $U = e^{i\,\hat{K}}$ is indeed unitary whenever $\hat{K}$ is Hermitian. Since we already promised this verification in Chap. 10, we shall first state our claim formally but slightly simplify matters. While the exponential connection between unitary and Hermitian operators holds in general, the following theorem focuses on the kind of finitely sized operators which occur in the context of evolving qubit systems.

**Theorem 13.1** *Every unitary operator $U \in \mathbb{C}^{2^n \times 2^n}$ such that $U\,U^\dagger = U^\dagger U = I$ can be written as an exponential*

$$U = e^{i\,\hat{K}} \tag{13.10}$$

*where $\hat{K} \in \mathbb{C}^{2^n \times 2^n}$ is a Hermitian operator with $\hat{K} = \hat{K}^\dagger$.*

***Proof*** First, we need to show that, if $\hat{K}$ is Hermitian, then $U = e^{i\,\hat{K}}$ is unitary. This is almost immediate, because, for any Hermitian operator, we have the following conjugate transposed exponential

$$\left[e^{i\,\hat{K}}\right]^{\dagger} = e^{-i\,\hat{K}^{\dagger}} = e^{-i\,\hat{K}}$$

which provides us with

$$e^{i\,\hat{K}}\left[e^{i\,\hat{K}}\right]^{\dagger} = e^{i\,\hat{K}}\,e^{-i\,\hat{K}} = e^{i\,[\hat{K}-\hat{K}]} = e^{i\,0} = I \tag{13.11}$$

$$\left[e^{i\,\hat{K}}\right]^{\dagger} e^{i\,\hat{K}} = e^{-i\,\hat{K}}\,e^{i\,\hat{K}} = e^{i\,[\hat{K}-\hat{K}]} = e^{i\,0} = I \tag{13.12}$$

and therefore establishes $U = e^{i\,\hat{K}}$ is unitary.

Second, we need to show that, if $U = e^{i\,\hat{K}}$ is unitary, then $\hat{K}$ is Hermitian. We therefore note that every unitary operator admits a spectral decomposition

$$U = V\Lambda V^{\dagger} \tag{13.13}$$

where $V$ is yet another unitary operator and $\Lambda = \mathrm{diag}[\lambda_1, \lambda_2, \ldots]$ is a diagonal matrix formed by the eigenvalues $\lambda_j$ of $U$. Since we know from Exercise 10.5 that $|\lambda_j| = 1$, we observe that we may also write $\lambda_j = e^{i\,\theta_j}$ for some $\theta_j \in \mathbb{R}$ and therefore $\Lambda = e^{i\,\Theta}$ where $\Theta = \mathrm{diag}[\theta_1, \theta_2, \ldots]$.

Next, we consider the $k$-th power of $V\Theta V^{\dagger}$ and find that we can expand it as

$$\left[V\Theta V^{\dagger}\right]^{k} = \underbrace{V\Theta V^{\dagger}V\Theta V^{\dagger}\cdots V\Theta V^{\dagger}V\Theta V^{\dagger}}_{k\text{ times}} = V\Theta^{k}V^{\dagger}\,. \tag{13.14}$$

From this it follows that

$$e^{i\,V\Theta V^{\dagger}} = \sum_{k=0}^{\infty}\frac{\left[i\,V\Theta\,V^{\dagger}\right]^{k}}{k!} = V\left[\sum_{k=0}^{\infty}\frac{\left[i\,\Theta\right]^{k}}{k!}\right]V^{\dagger} = Ve^{i\,\Theta}\,V^{\dagger} = V\Lambda V^{\dagger} = U \tag{13.15}$$

which establishes that we can indeed write any unitary as $U = e^{i\,\hat{K}}$ where $\hat{K} = V\Theta V^{\dagger}$. To finally see that $\hat{K}$ is Hermitian, we recall the conjugate transpose of an operator product $[AB]^{\dagger} = B^{\dagger}A^{\dagger}$ which tells us that

$$\hat{K}^{\dagger} = \left[V\Theta V^{\dagger}\right]^{\dagger} = V^{\dagger\dagger}\Theta^{\dagger}V^{\dagger} = V\Theta V^{\dagger} = \hat{K} \tag{13.16}$$

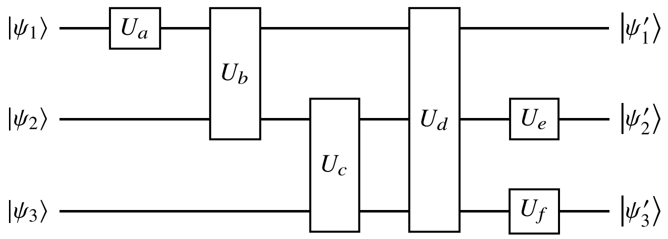
where we used that $\Theta$ is real and diagonal.

Given our redefinition of the third axiom of quantum computing and our brief examination of the nature of unitary operators acting on quantum states, we are now prepared to have at least an initial look at the *circuit model of quantum computing*.

Note, however, that the following will diverge from our overarching story-line of problem solving as energy minimization. The quantum circuit model is instead based on ideas developed by Feynman [4, 5] who conceived quantum computing as a generalization of digital computing which is based on digital circuits. We therefore recall that a digital circuit consists of (Boolean) logic gates interconnected by wires which steer the flow of information; certain bit values enter a gate and certain other bit values leave it. A quantum circuit also consists of gates interconnected by wires but these wires now represent the states of qubits on which the gates perform unitary operations. While this sounds very similar to what happens in digital circuits, quantum circuits do look different and we will need to discuss why this is.

## 13.3 Quantum Circuits in a Nutshell

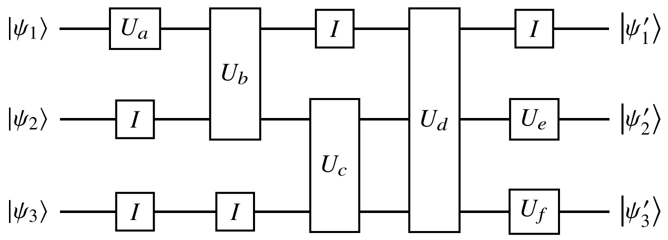
To make a very long story short, let us look at an exemplary, small, and generic quantum circuit such as



(13.17)

What we see here is a system of three qubits which, in quantum gate computing, is also called a **quantum register** and whose initial state is $|\Psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_3\rangle$. Each qubit is associated with a *wire* and there are several generic *quantum gates* along these wires. These gates are nothing but graphical representations of unitary operators; some act on a single qubit, some on a pair of qubits, and one on all three qubits simultaneously. Once all these operators have performed their computations, the qubit register will be in a final state $|\Psi'\rangle = |\psi_1'\rangle \otimes |\psi_2'\rangle \otimes |\psi_3'\rangle$.

Given this short explanation of what is going on in the above circuit, we note that the notion of, say, an operator acting on a single qubit is common but misleading. What the above circuit is acting on is a tensor product state $|\Psi\rangle \in \mathbb{C}^{2^3}$ so that a better but redundant graphical representation would be



(13.18)

Looking at this picture, we can now recognize that, at each computational step of our circuit, something is happening on all its wires. In other words, we can now recognize that each computational step of our circuit manipulates the eight-dimensional state of the whole quantum register.

But can we? Yes, because the two (equivalent) circuits in (13.17) and (13.18) are nothing but visual representations of the following algebraic computation

$$\left|\psi_1'\right\rangle \otimes \left|\psi_2'\right\rangle \otimes \left|\psi_3'\right\rangle = U_5\, U_4\, U_3\, U_2\, U_1 \left[\left|\psi_1\right\rangle \otimes \left|\psi_2\right\rangle \otimes \left|\psi_3\right\rangle\right] \tag{13.19}$$

where

$$U_1 = U_a \otimes I \otimes I \tag{13.20}$$
$$U_2 = U_b \otimes I \tag{13.21}$$
$$U_3 = I \otimes U_c \tag{13.22}$$
$$U_4 = U_d \tag{13.23}$$
$$U_5 = I \otimes U_e \otimes U_f\ . \tag{13.24}$$

Looking at these definitions of the unitary operators $U_j$, we now see that they are to be understood as operators acting on a system of three qubits whose joint quantum state is a vector in $\mathbb{C}^8$. Since this implies that each $U_j \in \mathbb{C}^{8\times 8}$, we once again recognize the exponential nature of qubit register states and of operators acting on them.

We now also get at least an intuition as to why or how it is possible to build quantum computers which accomplish high dimensional state manipulations. While we still can not say much about operators $U_2$, $U_3$, and $U_4$, we note that $U_1$ and $U_5$ are but tensor products of $I$, $U_a$, $U_e$, $U_f \in \mathbb{C}^{2\times 2}$. In other words, in each computational step of our three qubit quantum circuit, each operator seems to involve a tensor product of a maximum of three single qubit operators. Hence, if we assume that it is technically possible to create physical quantum states $|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle \in \mathbb{C}^{2^n}$ (which it is), then the manipulation of such exponentially large states only requires the assembly of linearly many operators. Since this is technically possible, quantum computing is feasible. In other words, quantum computers only need linearly many dials and knobs for us to program them and nature takes care of the rest, i.e. of the exponential dimensionality of the computations we want to perform.

Given what we said so far, it seems as if operators $U_1$ and $U_5$ are a bit boring. Indeed, consider, for example, the action of $U_1$, namely

$$U_1|\Psi\rangle = \Big[U_a \otimes I \otimes I\Big]\Big[|\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_3\rangle\Big] = U_a\,|\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_3\rangle \tag{13.25}$$

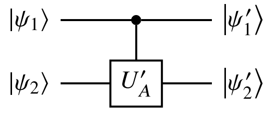
which manipulates the state of qubit $|\psi_1\rangle$ but leaves $|\psi_2\rangle$ and $|\psi_3\rangle$ intact. From the point of view of information processing, such an operation is of limited power as there are no interactions among the qubits involved.

Then what about operators $U_2$, $U_3$, and $U_4$? How could such multiple qubit operators look like and how could they mediate interactions between several qubits?

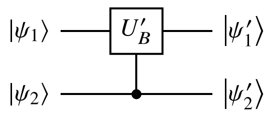
Well, here things become special because due to the unitary nature of quantum gate operators the possibilities for how qubits can interact are rather restricted. Note that this does not mean that quantum computing is not universal. On the contrary, just as a digital computer, a quantum gate computer can perform any conceivable logic operation and consequently any kind of more intricate computation. However, what it means is that quantum computers perform logical operations differently than digital computers and thus require us to think differently.

Indeed, the only way qubits can influence each other is via *controlled operations*. To see how these will look like, we next simplify things to the extent possible and only look at operators involving two qubits; generalizations to more than two qubits are straightforward but beyond the scope of this "outlook" chapter.

To make another very long story short, here are two super simple, generic quantum circuits over two qubits which both involve controlled operators:



(13.26)



(13.27)

What we see here are two unitary operators $U_A, U_B \in \mathbb{C}^{4\times4}$ which both involve a black dot and a vertical wire. In case of $U_A$, the wire ends in $U'_A \in \mathbb{C}^{2\times2}$ and the action of $U_A$ has to be understood as *the state of qubit* $|\psi_1\rangle$ *controls the execution of* $U'_A$. In case of $U_B$, the wire ends in $U'_B \in \mathbb{C}^{2\times2}$ so that *the state of qubit* $|\psi_2\rangle$ *controls the execution of* $U'_B$.

But how can such controlled operations be defined algebraically? Well, letting another fact drop out of the blue sky, we have

$$U_A = |0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes U'_A \tag{13.28}$$
$$U_B = I \otimes |0\rangle\langle 0| + U'_B \otimes |1\rangle\langle 1|\,. \tag{13.29}$$

Note that these operators are fundamentally different from, say, the one in (13.20). While the latter is but a tensor product of individual operators, $U_A$ and $U_B$ are *sums of tensor products* of individual operators.

To see how controlled operators work, we next consider the action of, say, $U_B$ on the two rather specific register states $|\psi_1\rangle \otimes |0\rangle$ and $|\psi_1\rangle \otimes |1\rangle$. For these, we have

$$\begin{aligned}\Big[I \otimes |0\rangle\langle 0| + U'_B \otimes |1\rangle\langle 1|\Big]\Big[|\psi_1\rangle \otimes |0\rangle\Big] &= |\psi_1\rangle \otimes |0\rangle\langle 0\,|\,0\rangle + U'_B\,|\psi_1\rangle \otimes |1\rangle\langle 1\,|\,0\rangle \\ &= |\psi_1\rangle \otimes |0\rangle \qquad (13.30)\end{aligned}$$

and

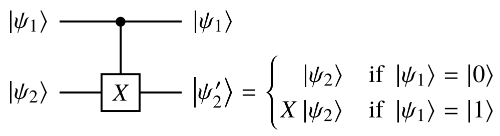
$$\begin{aligned}\Big[I \otimes |0\rangle\langle 0| + U'_B \otimes |1\rangle\langle 1|\Big]\Big[|\psi_1\rangle \otimes |1\rangle\Big] &= |\psi_1\rangle \otimes |0\rangle\langle 0\,|\,1\rangle + U'_B\,|\psi_1\rangle \otimes |1\rangle\langle 1\,|\,1\rangle \\ &= U'_B\,|\psi_1\rangle \otimes |1\rangle\,. \qquad (13.31)\end{aligned}$$

In plain English we therefore find: If qubit $|\psi_2\rangle = |0\rangle$, then nothing happens to both qubits and the resulting qubit register state is $|\psi'_1\rangle \otimes |\psi'_2\rangle = |\psi_1\rangle \otimes |0\rangle$. However, if qubits $|\psi_2\rangle = |1\rangle$, then $U'_B$ will be applied to $|\psi_1\rangle$ and the resulting qubit register state is $|\psi'_1\rangle \otimes |\psi'_2\rangle = U'_B|\psi_1\rangle \otimes |1\rangle$. In short, operator $U'_B$ is conditionally applied to the first qubit

Now, before we look at a simple practical application of the concept of controlled operations, we note that we have already seen the arguably most important instance of a controlled two qubit operator. This was in Exercise 9.7 where we introduced the CNOT operator in a form which we may now write in Dirac notation

$$CX = |0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes X \qquad (13.32)$$

where $X$ denotes the first Pauli matrix. Given what we saw above, we can easily provide a quantum circuit representation of this operator, namely


$|\psi_1\rangle$
$|\psi_1\rangle$
$|\psi_2\rangle$
$X$
$|\psi'_2\rangle = \begin{cases} |\psi_2\rangle & \text{if } |\psi_1\rangle = |0\rangle \\ X\,|\psi_2\rangle & \text{if } |\psi_1\rangle = |1\rangle \end{cases}$

(13.33)

Looking at this circuit, we observe that if the second input qubit is restricted to $|\psi_2\rangle \in \{|0\rangle, |1\rangle\}$, then we will have $|\psi'_2\rangle = X|0\rangle = |1\rangle$ or $|\psi'_2\rangle = X|1\rangle = |0\rangle$ but only if $|\psi_1\rangle = |1\rangle$. It is in this sense, that the controlled $X$ or $CX$ operator in (13.32) and (13.33) performs a controlled NOT or CNOT operation.

## 13.4 The Need for Reversibility

Have you noticed that all our quantum gates in all our quantum circuits had as many input wires as output wires? This is in contrast to logic gates in digital circuits which may have more input wires than output wires.

Consider, for instance, the Boolean function $\texttt{AND} : \{0, 1\}^2 \rightarrow \{0, 1\}$ which takes two inputs $x_1, x_2 \in \{0, 1\}$ and produces a single output $y \in \{0, 1\}$. Its corresponding logic gate will therefore have two input wires and just one output wire.

Why would that be noteworthy? Because it implies that the computation performed by AND are not always reversible. Consider this: If we did not know the values of the inputs $x_1$ and $x_2$ but were told that $\texttt{AND}(x_1, x_2)$ has produced an output of $y = 1$, we can conclude that the input must have been $(x_1, x_2) = (1, 1)$. In other words, in this special case, we can reverse the computation of this Boolean operator. However, if we were told that $\texttt{AND}(x_1, x_2)$ has produced an output of $y = 0$ we can no longer reverse or invert the computation as there are now three possibilities for the input, namely $(x_1, x_2) \in \{(0, 0), (0, 1), (1, 0)\}$.

But again, why would this be noteworthy? Because quantum gates are unitary operators which must have an inverse as $U^\dagger U = UU^\dagger = I$ implies that $U^\dagger = U^{-1}$. In other words, any operation we can ever perform on a quantum computer must be reversible and a necessary (but not sufficient) condition for this is that every quantum gate has as many input wires as it has output wires.

How then can we implement irreversible Boolean functions such as AND on a quantum computer? In short, by introducing **ancilla qubits** whenever necessary. In what follows, we will illustrate the basic idea. However, we will keep things simpler than in our introductory example and focus on the four univariate Boolean functions in Table 13.1.

Looking at this table, we observe that ID and NOT are reversible whereas FALSE and TRUE are not. To be able to implement quantum circuits for the non-reversible functions FALSE and TRUE, we first have to rewrite them as reversible operations. For consistency, we will also rewrite the two already reversible functions ID and NOT correspondingly.

The basic idea is to consider quantum circuits over two qubits, namely an argument qubit $|x\rangle$ with $x \in \{0, 1\}$ and an ancilla qubit $|a\rangle$. If we then assume that the latter will always be prepared in state $|a\rangle = |0\rangle$, we may try to devise operators $U_f \in \mathbb{C}^{4\times 4}$ for $f \in \{\texttt{FALSE}, \texttt{TRUE}, \texttt{ID}, \texttt{NOT}\}$ which realize the I/O behavior in Table 13.2.

Looking at the truth tables shown there, we observe that they are all distinct. That is, there are four different functions and four different tables and as the tables are different, the functions are reversible. We further observe that all these truth tables

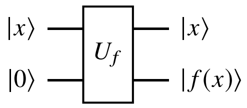
**Table 13.1** Bit truth tables of the four univariate Boolean functions $f : \{0, 1\} \rightarrow \{0, 1\}$

| `FALSE` | | `TRUE` | | `ID` | | `NOT` | |
|---|---|---|---|---|---|---|---|
| input x | output y | input x | output y | input x | output y | input x | output y |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |

**Table 13.2** Qubit truth tables of reversible versions of the four univariate Boolean functions

| `FALSE` | | | | `TRUE` | | | | `ID` | | | | `NOT` | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input | | output | | input | | output | | input | | output | | input | | output | |
| $\vert x\rangle$ | $\vert a\rangle$ | $\vert x'\rangle$ | $\vert a'\rangle$ | $\vert x\rangle$ | $\vert a\rangle$ | $\vert x'\rangle$ | $\vert a'\rangle$ | $\vert x\rangle$ | $\vert a\rangle$ | $\vert x'\rangle$ | $\vert a'\rangle$ | $\vert x\rangle$ | $\vert a\rangle$ | $\vert x'\rangle$ | $\vert a'\rangle$ |
| $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 1\rangle$ | $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 0\rangle$ | $\vert 1\rangle$ |
| $\vert 1\rangle$ | $\vert 0\rangle$ | $\vert 1\rangle$ | $\vert 0\rangle$ | $\vert 1\rangle$ | $\vert 0\rangle$ | $\vert 1\rangle$ | $\vert 1\rangle$ | $\vert 1\rangle$ | $\vert 0\rangle$ | $\vert 1\rangle$ | $\vert 1\rangle$ | $\vert 1\rangle$ | $\vert 0\rangle$ | $\vert 1\rangle$ | $\vert 0\rangle$ |

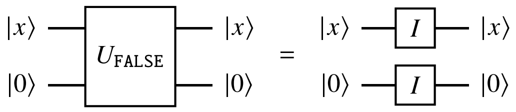
assume that input- and output state of the argument qubit are the same $|x\rangle = |x'\rangle$ and, crucially, that the output state of the ancilla qubit is given by $|a'\rangle = |f(x)\rangle$ with $f(x) \in \{0, 1\}$. In short, we are therefore looking for four quantum operators $U_f$ such that



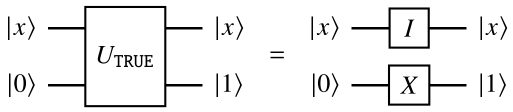
(13.34)

Without further ado, we next simply present circuit representations of the four corresponding operators and very much encourage our readers to verify that these do indeed produce the reversible I/O behaviors in Table 13.2.

For the functions `FALSE` and `TRUE`, the corresponding quantum operators are trivial

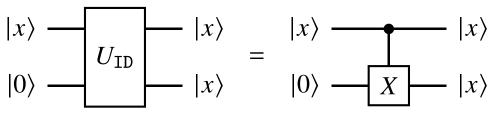


(13.35)



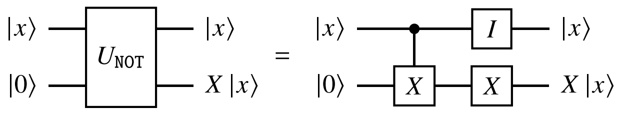
(13.36)

For the function ID, we need a CNOT gate which either leaves the ancilla $|a\rangle$ in state $|0\rangle$ if the argument is $|x\rangle = |0\rangle$ or switches the ancilla $|a\rangle$ from $|0\rangle$ to $|1\rangle$ if the argument is $|x\rangle = |1\rangle$.We thus simply have



(13.37)

For the function NOT, we need an operator that switches the ancilla $|a\rangle$ from $|0\rangle$ to $|1\rangle$ if the argument is $|x\rangle = |0\rangle$ but leaves the ancilla in state $|a\rangle = |0\rangle$ if the argument is $|x\rangle = |1\rangle$. To accomplish this, we simply add another $X$ gate to the ancilla wire of the circuit we just saw and get



(13.38)

In short, it is indeed possible to implement the four univariate Boolean functions in terms of unitary operators acting on a register of two qubits. But why would that be of interest? What could this possibly be good for?

Well, while there are admittedly hardly any serious applications of our operators $U_f$, they played a central role in the first ever demonstration of *quantum supremacy*. That is, they featured prominently in a quantum gate algorithm which solves a certain problem exponentially faster than classically possible. Luckily for us, now that we spent all the above work on deriving the $U_f$, we can actually study this algorithm.

## 13.5 Deutsch's Problem and Algorithm

Let us reconsider the four univariate Boolean functions FALSE, TRUE, ID, and NOT in Table 13.1. Above, we already classified them into two classes, namely into the class of reversible functions and the class of irreversible functions. Interestingly, there is yet another classification scheme we may apply and we therefore state or recall the following.

A Boolean function $f : \{0, 1\}^n \to \{0, 1\}$ is **constant**, if the output for all its $2^n$ possible inputs is always 0 or always 1.

A Boolean function $f : \{0, 1\}^n \to \{0, 1\}$ is **balanced**, if the output for one half of its $2^n$ possible inputs is 0 and for the other half it is 1.

Given these definition, we note that FALSE and TRUE are constant whereas ID and NOT are balanced. With this in mind, let us paraphrase a problem considered in a seminal paper by David Deutsch [1].

Say, we had access to an API that computes some $f \in \{$FALSE, TRUE, ID, NOT$\}$ but we are not told which of the four functions is hidden under the hood. Nevertheless, we are asked to determine if $f$ is constant or balanced. How could we do this?

Well, if the API would run a digital implementation of the black box function $f$, we would obviously have to query it twice with inputs $x = 0$ and $x = 1$ to first learn about the corresponding outputs. Once we know those, we can classify the black box function as either constant or balanced.

However, here is an outrageous claim: If the API would run a quantum implementation of the black box function $f$, we would only need to query it once to determine the class of the function in the black box.

Now, since $1 = \log_2(2)$, and since our claim can indeed proven to be true, we therefore have the following theorem.

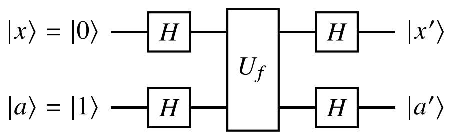
**Theorem 13.2** *A quantum computer can solve Deutsch's problem exponentially faster than a digital computer.*

Silly as Deutsch's problem may be, its (historical) significance must not be understated as the accompanying theorem established the existence of computational problems for which quantum computers have an edge over digital computers. Put differently, it established that there are problems for which quantum computing can drastically outperform digital computing.

But since Deutsch's problem really seems contrived and of little utility, let us look at an everyday analogy of what it is all about: If you were given a coin for a coin tossing game and wanted to determine if it is fair (has sides {`heads`, `tails`}) or rigged (has sides either {`heads`, `heads`} or {`tails`, `tails`}), you would have to examine both sides of the coin to determine whichever is the case. However, if you were given a "quantum coin" which can exist in a superposition of its two sides, you would only have to look at one side to determine whether it is fair or rigged.

Now then, let us look at Deutsch's quantum algorithm for determining the nature of a quantum coin. But be warned, since we are discussing quantum gate computing, the meaning of *algorithm* differs from what most computer scientists would usually think of. Indeed, present day quantum computing is still bit level computing so that quantum algorithms are nothing but quantum circuits.

Letting $U_f \in \{U_{\texttt{FALSE}}, U_{\texttt{TRUE}}, U_{\texttt{ID}}, U_{\texttt{NOT}}\}$ and considering a quantum circuit over an argument qubit $|x\rangle$ and an ancilla qubit $|a\rangle$, we now claim the following: When running the following algorithm



(13.39)

a measurement of the output $|a'\rangle$ will always result in basis state $|1\rangle$. However, a measurement of the output $|x'\rangle$ will result in

$$\big|x'\big\rangle = \begin{cases} |0\rangle & \text{if } U_f \text{ realizes a \textit{constant} function} \\ |1\rangle & \text{if } U_f \text{ realizes a \textit{balanced} function .} \end{cases} \tag{13.40}$$

In other words, on input $|x\rangle \otimes |a\rangle = |0\rangle \otimes |1\rangle$, a single run of Deutsch's algorithm determines the class of the black box function $f$.

As we already indicated, this claim can be proven analytically. However, in what follows, we shall not go down that road. Instead, we will empirically verify the claimed behavior of Deutsch's algorithm by implementing and running `numpy` code which simulates the circuit in (13.39).

To be able to do this, we need two more things. First of all, we need to know what the single qubit operator $H \in \mathbb{C}^{2\times 2}$ is all about. Interestingly, we have already seen it a couple of times before, namely in Chaps. 3 and 9 where we ever so briefly discussed Hadamard operators. Algebraically, the single qubit **Hadamard operator** can be defined in terms of the Pauli operators $X$ and $Z$

$$H = \frac{1}{\sqrt{2}}\big[X + Z\big] \tag{13.41}$$

and its matrix representation reads

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix} . \tag{13.42}$$

Note that the Hadamard operator or, equivalently, the Hadamard gate, is of pivotal importance in quantum gate computing as it maps the computational basis states $|0\rangle$ and $|1\rangle$ to the following *superposition states*

$$H\,\big|0\big\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} +1 \\ +1 \end{bmatrix} = \frac{1}{\sqrt{2}}\Big[\big|0\big\rangle + \big|1\big\rangle\Big] \tag{13.43}$$

$$H\,\big|1\big\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} +1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}}\Big[\big|0\big\rangle - \big|1\big\rangle\Big] \tag{13.44}$$

and we recall that we already discussed the role of state $1/\sqrt{2}|0\rangle + 1/\sqrt{2}|1\rangle$ in Chap. 12.

Second of all, we need an algebraic representation of the circuit in (13.39). That is we need to write it as an operator that maps the circuit's input $|x\rangle \otimes |a\rangle$ to its output $|x'\rangle \otimes |a'\rangle$. Given our discussion so far, it should be easy to see that it amounts to

$$D_f = \big[H \otimes H\big]U_f\big[H \otimes H\big] . \tag{13.45}$$

Looking at this algebraic representation of the circuit in (13.39), we better note the following: We wrote $D$ with a subscript $f$ because the black box operator $U_f$ is a *variable* component of Deutsch's algorithm. It can be any of the four operators in (13.35)–(13.38) but the premise is that we do not know which one. In terms of the language we used above, it is the quantum API we query to classify its nature. The whole point of the algorithm in (13.39) is to figure out if $U_f$ realizes a constant or a balanced Boolean function.

All this can be confusing to quantum computing novices as much of the literature considers problems that have to do with black box functions or *oracles* which can be easily included in diagrammatic representations of quantum algorithms. But if we actually want to implement and run a quantum algorithm, we have to instantiate variable black box operators and can then check if the algorithm works as claimed given the instantiation we are working with. In a sense, the issue with black box operators is that theorists can use them in their work but practitioners can not.

## 13.5.1 Simulating Quantum Gate Computing

Having defined operators $H$ and $D_f$, we can start coding. Given everything we discussed in previous chapters and in the present chapter so far, this should be easy.

Indeed, code snippet 13.1 presents straightforward ideas for `numpy` array implementations of the computational basis states and quantum operators which we need to simulate Deutsch's algorithm.

Arrays `ket0` and `ket1` represent the ket vectors or single qubit basis states $|0\rangle$ and $|1\rangle$. Arrays `opI`, `opX`, and `opH` represent the $2 \times 2$ identity operator $I$, the Pauli $X$ operator, and the Hadamard operator $H$. By now, we are also well familiar with the two projectors $P_0 = |0\rangle\langle 0|$ and $P_1 = |1\rangle\langle 1|$ which are implemented as arrays `opP0` and `opP1`. Together with `opX`, these allow us to code the $CX$ operator in terms of array `opCX`. What follows next, are the definitions of four arrays `opUF`, `opUT`, `opUI`, and `opUFN` which represent the operators $U_{\texttt{FALSE}}$, $U_{\texttt{TRUE}}$, $U_{\texttt{ID}}$, and $U_{\texttt{NOT}}$ whose circuit representations we gave in (13.35), (13.36), (13.37), and (13.38).

Looking at this snippet, it is clear that our code really just translates mathematical equations into `numpy` expressions. The only thing worth mentioning is perhaps our use of `np.outer` and `np.kron` to computer outer products and Kronecker products. However, given our code examples in earlier chapters, their roles should also be clear.

Next, before we set out implementing Deutsch's algorithm, we need to address the issue of measuring the quantum state produced by a quantum algorithm. With respect to Deutsch's algorithm, we therefore note the following: Its input will always be $|x\rangle \otimes |a\rangle = |0\rangle \otimes |1\rangle = |\Psi\rangle$ but its output depends on $U_f$ and will be of the form

$$D_f \left|\Psi\right\rangle = \left|x'\right\rangle \otimes \left|a'\right\rangle = \left|\Psi'\right\rangle = \sum_{j=0}^{2^2-1} \alpha_j \left|\Psi_j\right\rangle . \tag{13.46}$$

That is, its output will be a superposition of basis states $|\Psi_j\rangle \in \mathbb{C}^{2^2}$ whose amplitudes $|\alpha_j|^2$ obey the Born rule

```python
import numpy as np

ket0 = np.array([1,0])
ket1 = np.array([0,1])

opI = np.array([[+1, 0], [ 0,+1]])
opX = np.array([[ 0,+1], [+1, 0]])
opH = np.array([[+1,+1], [+1,-1]]) / np.sqrt(2)

opP0 = np.outer(ket0, ket0)
opP1 = np.outer(ket1, ket1)
opCX = np.kron(opP0, opI) + np.kron(opP1, opX)

opUF = np.kron(opI, opI)
opUT = np.kron(opI, opX)
opUI = opCX
opUN = opCX @ opUT
```

**Code 13.1** Quantum states and operators required to simulate Deutsch's algorithm.

```python
import numpy.random as rnd

def measure(amps):
    N = len(amps)
    n = int(np.log2(N))
    k = rnd.choice(np.arange(N), size=1, p=amps)
    vecZ = np.array([(k >> j) & 1 for j in range(n)])[::-1]
    return vecZ
```

**Code 13.2** Simply `numpy` code to simulate quantum measurement and probabilistic state collapse of a qubit register.

$$\sum_{j=0}^{2^2-1} \left|\alpha_j\right|^2 = 1\ . \tag{13.47}$$

Now, recall from our study in Chap. 12 that measuring a quantum state $|\Psi'\rangle$ as in (13.46) will cause it to probabilistically collapse to one of its basis states $|\Psi_j\rangle$ where the probability of collapse to the particular basis state $|\Psi_k\rangle$ is given by

$$p\Big(\left|\Psi'\right\rangle \mapsto \left|\Psi_k\right\rangle\Big) = \left|\alpha_k\right|^2\ . \tag{13.48}$$

In order to faithfully simulate the behavior of a quantum circuit, we also need to simulate this behavior upon measurement. Based on our coding choices so far, this is rather straightforward, too.

Code snippet 13.2 present a function `measure` which realizes the corresponding mechanism. Given an array `amps` containing the the $N = 2^n$ amplitudes $|\alpha_j|^2$ of the basis states of the superposition state of a qubit register, it first uses the length of `amps` to determine the numbers $N$ and $n$. Given $N$, it then randomly samples an integer $0 \leq k \leq N-1$ which represent the index of the basis state $|\Psi_k\rangle$ the system will collapse to. Note that we are using function `choice` provided in `numpy.random` which comes with a parameter p that allows for weighted random sampling. This perfectly fits our current purpose! As we need to sample $k \sim |\alpha_k|^2$, we pass p=amps to choice to simulate collapse to a physically reasonable basis state $|\Psi_k\rangle$.

We furthermore recall from Chap. 12 that such a basis state has to be of the form

$$\big|\Psi_k\big\rangle = \big|z_1\big\rangle \otimes \big|z_2\big\rangle \otimes \cdots \otimes \big|z_n\big\rangle \tag{13.49}$$

where $z_j \in \{0, 1\}$. We also already saw that the corresponding individual qubit instances $|z_1\rangle, |z_2\rangle, \ldots, |z_n\rangle$ can thus be represented in terms of a classical indicator vector $\boldsymbol{z} = [z_1, z_2, \ldots, z_n]^\intercal$ and our function measure does just that. It returns an array vecZ whose entries $z_j \in \{0, 1\}$ represent the basis states $|z_j\rangle$ in which the individual qubits of a quantum register once the overall state of the register has collapsed from $|\Psi'\rangle$ to $|\Psi_k\rangle$.

With all these preparations in place, we can finally simulate Deutsch's algorithm. To this end, we first define its input state $|\Psi\rangle = |0\rangle \otimes |1\rangle$ and a dictionary fdict which maps the names of the four univariate Boolean functions to their corresponding quantum operators

```
ketIn = np.kron(ket0, ket1)
fdict = {'FALSE':opUF, 'TRUE':opUT, 'ID':opUI, 'NOT':opUN}
```

In the following code snippet, we will instantiate and run Deutsch's algorithm four times, namely for each of the four choices of $U_f$, in order to verify that it behaves as claimed.

The snippet iterates over the keys f and values opU_f of dictionary fdict. In each iteration, it first assembles the corresponding operator $D_f = [H \otimes H] U_f [H \otimes H]$ and then computes the output state $|\Psi'\rangle = D_f |\Psi\rangle$.

The amplitudes $|\alpha_j|^2$ of this output state $|\Psi'\rangle = \sum_j \alpha_j |\Psi_j\rangle$ are stored in an array ampltd which is then passed to function measure. Finally, the corresponding measurement result is "pretty" printed to the screen in order to ease interpretation of the behavior of the algorithm.

```
for f, opU_f in fdict.items():
    opD_f = np.kron(opH, opH) @ opU_f @ np.kron(opH, opH)

    ketOut = opD_f @ ketIn
    ampltd = np.abs(ketOut)**2
    result = measure(ampltd).flatten()

    print (f)
    print ("|x_out> = |{0}>, |a_out> = |{1}>".format(*result))
```

Now, if we run this script as often as we like, we will find that it *always* produces the following output

```
FALSE
|x_out> = |0>, |a_out> = |1>
TRUE
|x_out> = |0>, |a_out> = |1>
ID
|x_out> = |1>, |a_out> = |1>
NOT
|x_out> = |1>, |a_out> = |1>
```

and therefore empirically corroborates our claim about the behavior of Deutsch's algorithm.

To be specific, it confirms that the ancilla qubit will always be measured in output state $|a'\rangle = |1\rangle$ and that the output state $|x'\rangle$ of the argument qubit indicates the nature of the function which we probe. In case of the two constant Boolean functions $f \in \{\texttt{FALSE}, \texttt{TRUE}\}$, we get $|x'\rangle = |0\rangle$ and, in case of the two balanced Boolean functions $f \in \{\texttt{ID}, \texttt{NOT}\}$, we get $|x'\rangle = |1\rangle$. Moreover, it confirms that a single computation of $|\Psi'\rangle = D_f|\Psi\rangle$ is enough to determine the nature of $f$.

### 13.5.2 Concluding Remarks

Note that our implementation of a digital simulator of Deutsch's algorithm just used plain vanilla `numpy` functionalities and did not at all require us to work with any kind of fancy or advanced special purpose libraries. This clearly underlines that quantum computing really is nothing but linear algebra in high dimensional, complex valued tensor product spaces. Granted our code was as simple as it was because the setting we considered takes place in a very low dimensional space and does not even pose a need for considering complex numbers.

In short, the basics of quantum computing are surprisingly simple. It really is just about multiplying unitary operators to quantum state vectors where both usually result from tensor products of more elementary operators and vectors. Of course more elaborate problems will require complex valued operators and vectors of a dimensionality which defies the capabilities of digital computers. And the fact that these operators and vectors may have to be complex valued also means that we usually have to think quite carefully about their properties and capabilities as computational devices.

Indeed and obviously, there would be much more to be said even just about Deutsch's algorithm. Why does it actually work? What are the underlying quantum mechanical principles it exploits? Quantum superposition seems to play a role but the real trick is its use of another weird quantum effect called *phase kickback* or *quantum interference*. What is this? How does it work? Can it be generalized to more interesting settings which deal with multivariate Boolean functions?

Just as obviously, there would be much more to be said about quantum gate computing in general. What else can it do? What other kinds of famous algorithms are there? What about quantum Fourier transforms, Shor's quantum prime factorization algorithm [6], or Grover's quantum search algorithm [7]?

Alas, while interesting, intriguing, and definitely worth further study, answers to these kinds of questions are beyond our purpose of providing a mere "outlook" to the possibilities of quantum gate computing. For now, we therefore simply refer those who want to learn more to dedicated resources on quantum gate computing [8].

## 13.6 Exercises

**13.1** Verify that operator $U_A$ in (13.28) performs the following mappings

$$|0\rangle \otimes |\psi_2\rangle \mapsto |0\rangle \otimes |\psi_2\rangle \tag{13.50}$$
$$|1\rangle \otimes |\psi_2\rangle \mapsto |1\rangle \otimes U'_A|\psi_2\rangle \; . \tag{13.51}$$

**13.2** Work with the Dirac notation to prove the following identity

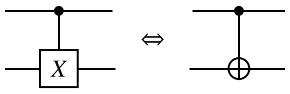
$$\big|0\big\rangle\big\langle 0\big| \otimes I + \big|1\big\rangle\big\langle 1\big| \otimes X = I \otimes I + \big|1\big\rangle\big\langle 1\big| \otimes \big[X - I\big] \; . \tag{13.52}$$

**13.3** Operator $CX$ in (13.32) and (13.33) should better be called $CX_{1\to 2}$ because the first qubit controls the $X$ operation to be performed on the second one. Give an algebraic expression for a corresponding operator $CX_{2\to 1}$ where the second qubit controls the $X$ operation to be performed on the first one.

**13.4** For $x_1, x_2 \in \{0, 1\}$, convince yourself of the following equivalence

$$x'_2 = \begin{cases} x_2 & \text{if } x_1 = 0 \\ \texttt{NOT}(x_2) & \text{if } x_1 = 1 \end{cases} \quad \Leftrightarrow \quad x'_2 = \texttt{XOR}(x_1, x_2) \; . \tag{13.53}$$

To see why this is relevant, we note that it is common to write the XOR function in terms of the infix operator $\oplus$, namely $\texttt{XOR}(x_1, x_2) = x_1 \oplus x_2$. Because of this, most texts on quantum gate computing use a different graphical representation of the $CX$ or CNOT operator. Indeed, we have the following graphical equivalence

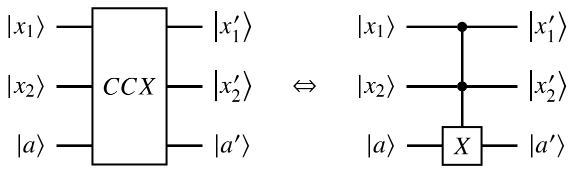


(13.54)

**13.5** Show that the following operators are unitary

$$U_{\texttt{FALSE}} = I \otimes I \tag{13.55}$$
$$U_{\texttt{TRUE}} = I \otimes X \tag{13.56}$$
$$U_{\texttt{ID}} = \big|0\big\rangle\big\langle 0\big| \otimes I + \big|1\big\rangle\big\langle 1\big| \otimes X \tag{13.57}$$
$$U_{\texttt{NOT}} = \Big[I \otimes X\Big]\Big[\big|0\big\rangle\big\langle 0\big| \otimes I + \big|1\big\rangle\big\langle 1\big| \otimes X\Big] \; . \tag{13.58}$$

**13.6** Implement `numpy` code that can simulate the following quantum circuit



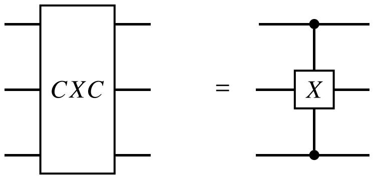
(13.59)

where the algebraic representation of operator $CCX$ is given by

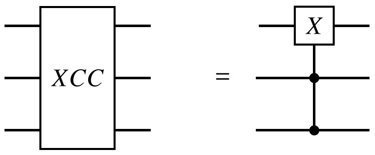
$$CCX = I \otimes I \otimes I + |1\rangle\langle 1| \otimes |1\rangle\langle 1| \otimes [X - I] \,. \tag{13.60}$$

Let $a = 0$, run the circuit for all possible combinations of $x_1, x_2 \in \{0, 1\}$, and read out the respective state of $|a'\rangle$. What do you observe? Does this circuit compute a Boolean function you have seen before?

**13.7** Can you recognize a characteristic structure in the definition of the $CCX$ operator in (13.60)? If so, provide algebraic representations of the following operators

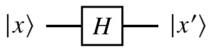


(13.61)



(13.62)

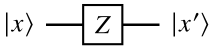
**13.8** Implement `numpy` code that runs and measures the output of the following simple quantum circuit several times



(13.63)

What kinds of measurement do you get on input $|x\rangle = |0\rangle$? What kinds of measurement do you get on input $|x\rangle = |1\rangle$?

**13.9** Implement `numpy` code that runs and measures the output of the following simple quantum circuit



(13.64)

where $Z = |0\rangle\langle 0| - |1\rangle\langle 1|$ is the Pauli $Z$ gate. What kinds of measurement do you get on input $|x\rangle = |1\rangle$?

Note the following: For input state $|x\rangle = |1\rangle$, the circuit will produce the output state

$$|x'\rangle = \Big[|0\rangle\langle 0| - |1\rangle\langle 1|\Big]|1\rangle = |0\rangle\langle 0\,|\,1\rangle - |1\rangle\langle 1\,|\,1\rangle = -|1\rangle\,. \tag{13.65}$$

However, your measurements will always result in the state $+|1\rangle$. Can you explain why this is and why it makes sense?

## References

1. Deutsch, D.: Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. Proc. of the Royal Society London A **400**(1818) (1985). https://doi.org/10.1098/rspa.1985.0070
2. Shankar, R.: Principles of Quantum Mechanics, 2nd edn. Springer (1994)
3. Sakurai, J., Napolitano, J.: Modern Quantum Mechanics, 3rd edn. Cambridge University Press (2020)
4. Feynman, R.: Simulating Physics with Computers. Int. J. of Theoretical Physics **21** (1982). https://doi.org/10.1007/BF02650179
5. Feynman, R.: Quantum Mechanical Computers. Foundations of Physics **16**(6) (1986). https://doi.org/10.1007/BF01886518
6. Shor, P.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: Proc. Symp. on Foundations of Computer Science. IEEE (1994). https://doi.org/10.1109/SFCS.1994.365700
7. Grover, L.: A Fast Quantum Mechanical Algorithm for Database Search. In: Proc. Symp. on Theory of Computing. ACM (1996). https://doi.org/10.1145/237814.237866
8. Chuang, I., Nielsen, M.: Quantum Computation and Quantum Information, 10th anniversary edition edn. Cambridge University Press (2010)

# Index