Kapitel 1
Einführung

Der Nachteil der Intelligenz besteht darin, dass man ununterbrochen gezwungen ist, dazuzulernen.
– George Bernard Shaw

Bevor es in den folgenden Kapiteln an die mathematischen, technischen und inhaltlichen Details geht, klärt die vorliegende Einführung vor allem zwei Dinge:

- Wie läuft die Fachinformatik-Ausbildung ab (unter besonderer Berücksichtigung der Fachrichtung *Daten- und Prozessanalyse*)?
- Was bedeuten Begriffe wie *künstliche Intelligenz*, *Machine Learning* und *Datenanalyse*?

Das Motto dieses Kapitels passt also gleich doppelt, denn in diesem Buch geht es sowohl darum, dass Sie selbst im Rahmen Ihrer Ausbildung Neues lernen, als auch darum, dass hier Software entwickelt wird, die aus Daten lernt und Computer befähigt, Aufgaben zu lösen, für die Menschen Intelligenz benötigen.

1.1 Die Ausbildung im Überblick

Den dualen Ausbildungsberuf *Fachinformatiker*in* gibt es seit 1996, als Arbeitgeberverbände und Gewerkschaften unter Beratung des Bundesinstituts für Berufsbildung (BiBB) eine zeitgemäße Neuordnung der IT-Berufe beschlossen. Davor konnten sich Interessierte nur zu EDV-Kaufleuten oder zu Büromaschinenelektroniker*innen ausbilden lassen – die Berufsbilder hatten also einen elektrotechnischen beziehungsweise kaufmännischen Schwerpunkt.

Die Aufgabe von Fachinformatiker*innen ist es gemäß Definition in der Ausbildungsordnung, »fachspezifische Anforderungen in komplexe Hard- und Softwaresysteme umzusetzen«. Das ist sehr allgemein formuliert; in der Praxis können die konkreten Inhalte je nach Branche des Ausbildungsbetriebs und natürlich je nach konkreter Fachrichtung der Ausbildung sehr unterschiedlich sein.

Was besagte Fachrichtungen angeht, gab es lange Zeit nur zwei: *Anwendungsentwicklung* und *Systemintegration*. Anwendungsentwickler*innen kümmern sich um Softwareentwicklung, also Programmierung, während es bei Systemintegrator*innen um die Bereitstellung der IT-Infrastruktur, also die System- und Netzwerkadministration geht.

Im August 2020 trat eine weitere Modernisierung der IT-Ausbildungsordnung in Kraft, bei der zwei weitere Fachinformatik-Spezialisierungen hinzukamen. Die *digitale Vernetzung* kümmert sich anders als die Systemintegration weniger um den einzelnen Arbeitsplatz und seine Software, sondern mehr um das große Ganze der Netzwerkinfrastruktur; Stichworte wie *Virtualisierung*[1], *Cloud Computing* und *Internet of Things* (IoT) aus Administrationssicht gehören in diesen Bereich.

Die *Daten- und Prozessanalyse*, Gegenstand des vorliegenden Buchs, kümmert sich um zwei verschiedene Themen: Die Datenanalyse verwendet statistische Verfahren zur weitgehend automatisierten Verarbeitung von Daten, während die Geschäftsprozessanalyse die Arbeitsabläufe in Unternehmen modelliert, untersucht und optimiert.

1.1.1 Ablauf der Ausbildung

Die Fachinformatik-Ausbildung ist eine *duale Berufsausbildung*, das heißt, sie findet in Ausbildungsbetrieb und Berufsschule (60 Tage pro Ausbildungsjahr) statt. Der Berufsschulunterricht wird entweder im regelmäßigen Wochenunterricht oder im Blockunterricht erteilt, das heißt an einem oder zwei Tagen pro Woche (außer in den Schulferien) oder aber in mehrwöchigen Blöcken, in denen nur die Berufsschule besucht wird.

Der Ausbildungsgang dauert im Regelfall drei Jahre, kann jedoch in Einzelfällen auf Antrag bei der IHK um sechs oder zwölf Monate verkürzt werden, wenn bestimmte Voraussetzungen vorliegen (Abitur oder andere Vorkenntnisse).

In allen vier Fachrichtungen beginnen die Ausbildungen mit gemeinsamen Grundlagen, spezialisieren sich jedoch später zunehmend. Mit welchen konkreten Systemen, Programmiersprachen und Anwendungsprogrammen gearbeitet wird, kann in verschiedenen Betrieben stark voneinander abweichen. Umso wichtiger ist es, neben spezifischen Implementierungen auch die allgemeinen Konzepte zu erlernen. Dieses Buch vermittelt daher stets besagte Konzepte, bevor die konkrete Arbeit mit ausgewählter (und in der Praxis weit verbreiteter) Software erläutert wird.

1 Für viele unbekannte Begriffe können Sie in Anhang A, »Glossar«, kurze Definitionen nachschlagen.

1.1.2 Die Abschlussprüfung

Ein weiterer Punkt der Neuordnung von 2020 neben den neuen Ausbildungsgängen ist, dass die Abschlussprüfung aufgeteilt wurde. Seitdem kommt nämlich die sogenannte *gestreckte Prüfung* zur Anwendung: Ungefähr in der Mitte der dreijährigen Ausbildungszeit wird eine 90 Minuten dauernde schriftliche Prüfung abgelegt, deren Bewertung 20 % der Gesamtprüfungsnote ausmacht. Das Oberthema (und die jeweilige Prüfung eines Jahrgangs) ist für alle IT-Berufe identisch: »Einrichtung eines IT-gestützten Arbeitsplatzes« mit Fragen aus den Bereichen Hard- und Software sowie Betriebsorganisation, Wirtschaft, Ergonomie, IT-Sicherheit und Datenschutz.

Vor der Neuordnung wurde zur Halbzeit der Ausbildung dagegen eine *Zwischenprüfung* durchgeführt. Die Teilnahme war verpflichtend, aber das Ergebnis rein informativ.

Wie gehabt findet der Rest der Prüfungen am Ende der Ausbildungszeit statt. Er ist in eine praktische Projektarbeit und mehrere schriftliche Prüfungsteile gegliedert.

Die *Projektarbeit* soll 40 Stunden dauern und ist üblicherweise in die Phasen Planung, Durchführung, Test und Abnahme unterteilt. (In der Fachrichtung Anwendungsentwicklung beträgt die Projektdauer allerdings 80 Stunden, weil umfangreichere Programmierung oft länger dauert als die Projekttätigkeiten der anderen Ausbildungsgänge.) Das Thema der Projektarbeit muss vor Beginn der Arbeit daran bei der IHK eingereicht und von dieser genehmigt werden.

Die Bewertung der Projektarbeit macht 50 % der Gesamtnote aus und ist wiederum je zur Hälfte in die Beurteilung der schriftlichen *Projektdokumentation* und diejenige eines *Fachgesprächs* zur Prüfung (15 Minuten freier Vortrag und 15 Minuten Fragen der Prüfer*innen) unterteilt.

Die drei schriftlichen Teile machen schließlich je 10 % der Prüfungsnote aus. Zwei von ihnen sind *berufsspezifische Aufgaben*, die offene Fragen zu je einem größeren Gesamt-Anwendungsfall stellen. In der Fachrichtung Daten- und Prozessanalyse haben sie die Themen »Durchführung einer Prozessanalyse« beziehungsweise »Sicherstellen der Datenqualität«. In der Anwendungsentwicklung sind es dagegen »Planen eines Softwareprojektes« und »Entwicklung und Umsetzung von Algorithmen«.

Der letzte Prüfungsteil ist eine 60 Minuten dauernde, für alle IT-Berufe identische schriftliche Prüfung zu den Themen Wirtschaft und Soziales einschließlich Arbeits- und Ausbildungsrecht. Die Fragen sind unzusammenhängend, teils offen und teils Multiple Choice.

In Tabelle 1.1 sehen Sie noch einmal alle Prüfungsteile und ihre Gewichtung für die Gesamtnote im Überblick.

Prüfungsteil	Anteil an der Gesamtnote
Schriftliche Prüfung »Einrichtung eines IT-gestützten Arbeitsplatzes« in der Mitte der Ausbildungszeit	20 %
Projektarbeit – Dokumentation	25 %
Projektarbeit – Fachgespräch	25 %
Berufsspezifische Aufgabe I	10 %
Berufsspezifische Aufgabe II	10 %
Wirtschaft und Soziales	10 %

Tabelle 1.1 Alle Teile der Abschlussprüfung in der Fachinformatikausbildung und ihre Gewichtung in der Bewertung

1.2 Datenanalyse und künstliche Intelligenz

Da die Themen Datenanalyse, künstliche Intelligenz und Machine Learning in diesem Buch kapitelübergreifend behandelt werden, erhalten Sie an dieser Stelle vorab eine Einführung in diese Themengebiete.

1.2.1 Datenanalyse

Die *Datenanalyse* (engl. *data science*) verwendet im Wesentlichen statistische Verfahren, um Daten zu verarbeiten. Deren Grundfunktionalität ist seit Jahrzehnten oder gar Jahrhunderten bekannt, aber erst die Leistungsfähigkeit moderner Computersysteme erlaubt ihre Anwendung auf große Datenmengen, wie sie etwa als Messdaten in der Forschung oder als Protokolldaten bei der Auswertung von Besuchen großer Websites oder den begonnenen und vollendeten Transaktionen in Online-Shops aufkommen. Solche enormen Datenmengen werden als *Big Data* bezeichnet.

Das weitgehend automatische Durchsuchen und Kategorisieren der Datenmengen wird als *Data Mining* bezeichnet. Verwenden Computerprogramme die statistischen Methoden, um selbst zu ermitteln, wie sie mit den Daten umzugehen haben, spricht man von *Machine Learning*. (Manchmal wird auch der deutsche Begriff *maschinelles Lernen* verwendet.)

Es werden verschiedene Arten des Machine Learnings unterschieden:

- **Überwachtes Lernen (supervised learning)** durchläuft zunächst eine Trainingsphase. Mithilfe von Datensätzen, für die bereits das korrekte Ergebnis bekannt ist (Trainingsdaten), wird eine Funktion kalibriert. Es folgt eine Testphase, in der die Funktion auf einen nicht für das Training verwendeten Teil der vorhandenen Datensätze angewendet wird, um herauszufinden, wie genau die Funktion arbeitet. Wenn die Genauigkeit den Erwartungen entspricht, kann die Funktion produktiv zur Prognose der Ergebnisse für neue Datensätze verwendet werden.
- **Unüberwachtes Lernen (unsupervised learning)** kommt zum Einsatz, wenn keine Trainingsdaten vorhanden sind, sondern nur Produktivdaten, aus denen der Algorithmus selbst Informationen herauslesen soll. Am bekanntesten sind Clustering-Verfahren, bei denen die Daten selbstständig in mehrere Gruppen (*Cluster* genannt) unterteilt werden.
- **Verstärkendes Lernen (reinforcement learning)** arbeitet am ehesten wie Tiertraining oder gar menschliches Lernen: Korrekte Ergebnisse werden »belohnt«, um ihre Häufigkeit zu steigern. Die »Belohnung« besteht in diesem Fall in einem verstärkenden Faktor, der die Häufigkeit dieser Ergebnisse erhöht.

In Kapitel 7, »Machine Learning«, wird die Implementierung von Verfahren des überwachten und des unüberwachten Lernens beschrieben. Die in Kapitel 8, »Künstliche neuronale Netzwerke«, beschriebenen Konstrukte implementieren typischerweise Verfahren des überwachten Lernens, können aber auch Aspekte des verstärkenden Lernens aufweisen.

1.2.2 Kurze Geschichte der künstlichen Intelligenz

Seit Jahrhunderten ist die menschliche Fantasie von der Idee künstlicher Menschen besessen. Die Vorstellungen wurden konkreter, je weiter sich Mechanisierung, Elektronik und später auch Computertechnik entwickelten. Besonders in der Science-Fiction seit den 1950ern wimmelt es von Robotern[2] und von Computern mit menschlicher oder übermenschlicher Intelligenz. Manche dieser Geschichten sind utopisch und erstrebenswert, andere auch durchaus beängstigend.

In der Praxis stehen besonders zwei Faktoren der Entwicklung einer maschinellen Intelligenz mit menschenähnlichen Fähigkeiten entgegen:

2 Das Wort »Roboter« wurde vom tschechischen Wort für »Arbeiter« entlehnt und stammt aus einem Theaterstück, das der Schriftsteller Karel Čapek in den 1920ern veröffentlichte.

1. Das menschliche Gehirn ist wesentlich komplexer als jedes noch so weit entwickelte Computersystem.
2. Es ist noch immer nicht bekannt, wie Intelligenz und Bewusstsein überhaupt entstehen, sodass man sie auch nicht in Form einer Software nachbilden kann, die praktisch als »Baby« beginnt, selbstständig zu lernen, wie die Welt funktioniert.

Nichtsdestotrotz gibt es eine Bezeichnung für das bisher noch nicht eingetroffene Ereignis, dass eine Software etwas entwickelt, das sich mit dem menschlichen Bewusstsein vergleichen lässt: *Singularität*.

Ernsthafte Forschungsarbeiten zur *künstlichen Intelligenz* (*KI*, engl. *Artificial Intelligence* oder kurz *AI*) begannen in den 1950er-Jahren. Der Aufsatz »Computing Machinery and Intelligence« von *Alan Turing* bildete eine Art Startschuss. Daraus stammt der berühmte *Turing-Test*, den der Autor selbst »The Imitation Game« nennt. Vereinfacht gesagt kommuniziert ein Mensch über ein Terminal mit einem Computer und einem anderen Menschen, und wenn er nicht mehr unterscheiden kann, welcher Gesprächspartner der Mensch ist, kann man davon sprechen, dass sich der Computer intelligent verhält.

Im Jahr 1956 fand am Dartmouth College ein Sommerworkshop zum Thema künstliche Intelligenz statt, der als formaler Beginn der KI-Forschung gilt. Der 1954 verstorbene Turing erlebte ihn leider nicht mehr.

Die Anfänge der KI fielen in eine Zeit optimistischer Wissenschafts- und Forschungsgläubigkeit, deren Höhepunkt die erste Mondlandung am 21. Juli 1969 war. Entsprechend große Erwartungen hatten die beteiligten Wissenschaftler*innen; sie rechneten damit, binnen weniger Jahre die Leistungen des menschlichen Geistes nachbilden oder sogar übertreffen zu können. Doch es zeichnete sich immer deutlicher ab, dass *starke KI* (engl. *strong AI* oder *Artificial General Intelligence*) nichts ist, das sich einfach so programmieren lässt.

Zu Anfang waren es praktische Beschränkungen, die der Forschung das Leben schwer machten. Geschwindigkeit, Speicherkapazität und Rechenleistung der verwendeten Computer reichten einfach nicht aus, um die theoretisch vorhandenen Ideen für vielversprechende Algorithmen umzusetzen. Mit heutiger Technik sind diese Probleme gelöst, und doch kann noch immer niemand wissen, ob Maschinen tatsächlich »denken« können.

Gelehrte neigen dazu, die trotz des Nichterreichens der starken KI erzielten Erfolge kleinzureden. Beispielsweise schreibt der Physiker, Kognitions- und KI-Forscher *Douglas Hofstadter* in seinem berühmtesten Buch, »Gödel, Escher, Bach – An Eternal Golden Braid« (1979): *»Sometimes it seems as though each new step towards AI, rather than producing something which everyone agrees is real intelligence, merely reveals*

what real intelligence is not.« (»Manchmal scheint es so, dass jeder neue Schritt in Richtung KI nicht zu etwas führt, bei dem sich alle einig sind, dass es echte Intelligenz ist, sondern lediglich offenbart, was echte Intelligenz nicht ist.«)

Im selben Buch zitiert Hofstadter den Informatiker *Larry Tesler*, der im bekannten Computerforschungszentrum XEROX PARC das Konzept Copy-and-Paste erfand, mit einem ähnlichen, kürzer gefassten Satz: *»AI is whatever hasn't been done yet«* (»KI ist immer das, was noch nicht erreicht wurde«).

Aber vielleicht dauert die Entwicklung der starken KI nur etwas länger als gedacht, denn ebenfalls von Hofstadter stammt das folgende selbstreferenzielle und rekursive »Gesetz«: *»Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.«* (»Hofstadters Gesetz: Es dauert immer länger als erwartet, selbst wenn man Hofstadters Gesetz beachtet.«)

1.2.3 Künstliche Intelligenz in der Praxis

So gut wie alle in der Praxis erzielten Erfolge in der künstlichen Intelligenz gehören zur sogenannten *schwachen KI* (engl. *weak AI*), die man auch als domänenspezifische KI bezeichnen könnte: Statt einer universell verwendbaren Nachbildung des menschlichen Gehirns werden Algorithmen entwickelt, die die verschiedensten Aufgaben mit möglichst wenig Anleitung durch Menschen lösen können. Es gibt zahlreiche praktische Anwendungsgebiete für solche Algorithmen. Hier folgt eine willkürliche Auswahl:

- automatisierte Kategorisierung oder Prognose verschiedener Daten
- automatische Übersetzungen
- Spracherkennung
- Verarbeitung (geschriebener) natürlicher Sprache
- Handschriftenerkennung
- Bilderkennung
- Erzeugung von Bildern, Texten und Tönen/Musik
- autonomes Spielen von Brett-, Karten- oder Videospielen
- Hilfsarbeiten in Medizin und Pflege
- weitgehend autonome Auskunftssysteme im E-Commerce und im Einzelhandel
- Robotik mit zahlreichen Aspekten (bis auf die rein mechanischen)
- zum Teil oder vollständig autonom fahrende Fahrzeuge
- Wachdienst-, Polizei- und Militäraufgaben

In diesem Buch lernen Sie diverse Grundalgorithmen kennen, die zur praktischen künstlichen Intelligenz gezählt werden. In Kapitel 5, »Algorithmen und Datenstrukturen«, sind es die informierte Suche (am konkreten Beispiel des A*-Algorithmus), Bedingungserfüllungsprobleme und genetische Algorithmen, und in Kapitel 7 und in Kapitel 8 kommen verschiedene Verfahren des Machine Learnings und die damit verwandten künstlichen neuronalen Netze zur Sprache.

1.2.4 Interdisziplinäre KI-Forschung

Die Buchautoren und KI-Spezialisten *Stuart J. Russell* und *Peter Norvig* haben frühere Definitionen künstlicher Intelligenz in die vier Kategorien eingeteilt, die Sie in Tabelle 1.2 sehen.

Menschliches Denken	*Rationales Denken*
Versuche, einer Nachbildung des menschlichen Gehirns mithilfe der Erforschung seiner Funktionalität immer näher zu kommen	Möglichst autonome Anwendung der Gesetze der Logik (Aussagelogik, Prädikatenlogik, höherwertige Logik)
Menschliches Handeln	*Rationales Handeln*
Implementierung von Algorithmen, die den Turing-Test bestehen und verwandte Anforderungen an möglichst menschenähnliches Verhalten erfüllen	»Intelligente Agenten« implementieren, d. h. Programme, die bestimmte Aufgaben lösen, für die Intelligenz benötigt wird

Tabelle 1.2 Die vier Kategorien der künstlichen Intelligenz gemäß Russell und Norvig

Konkrete Problemlösungen gehören typischerweise in mehr als eine dieser Kategorien. Nicht umsonst werden die vier Felder in einer zweidimensionalen Matrix auf den Achsen menschlich-rational und Denken-Handeln angeordnet, denn es sind verschiedene Teilgebiete der KI-Forschung, die sich zum Teil überschneiden.

Allgemein lässt sich feststellen, dass die künstliche Intelligenz mit all ihren Facetten keineswegs nur ein Teilgebiet der Informatik ist, sondern auch andere Bereiche der Wissenschaft und Technik berührt. Zunächst einmal bilden verschiedene Teildisziplinen der Mathematik die Grundlagen, auf denen die Algorithmen in der Praxis basieren: Stochastik, Kombinatorik, lineare Algebra und Analysis. Die wichtigsten Informationen dazu erhalten Sie in Kapitel 2, »Mathematische Grundlagen«.

Als Nächstes sind Neurologie und Psychologie interessante Inspirations- und Erkenntnisquellen für die KI, denn ihre Forschungsgebiete sind die »Hardware« des menschlichen Gehirns beziehungsweise die »Software« des Geistes und des Denkens. Auch die Physik – zum Beispiel Optik, Akustik und Mechanik – spielt eine Rolle, besonders in den KI-Teilgebieten Robotik und Sensorik. (Letztere ist ein Ersatz für die menschlichen Sinne, übertrifft diese sogar bisweilen rein technisch betrachtet.)

Eine theoretische Grundlage der meisten anderen Wissenschaften – und über die Logik untrennbar mit Mathematik und Informatik verbunden – ist schließlich die Philosophie. Auch diverse weitere Teilgebiete der theoretischen Philosophie stellen Fragen, deren Beantwortungsversuche auch die KI betreffen. Besonders die Epistemologie (Erkenntnistheorie), die zu verstehen versucht, was Erkenntnis und Wissen sind und wie ihr Erwerb funktioniert, sowie die Philosophie des Geistes, des Bewusstseins und der Wahrnehmung spielen eine Rolle. Auch Sprachphilosophie und allgemeinere Theorien der symbolischen Repräsentation geistiger Inhalte sind eine Betrachtung wert, weil auch Software, die »Intelligenz simuliert«, sich fragen muss, wie sie diese Repräsentation regelt.

Hinzu tritt die Ethik als Teil der praktischen oder angewandten Philosophie. Wie bei jedem wissenschaftlichen und technischen Fortschritt muss die Frage gestellt werden, ob alles, was machbar ist, tatsächlich in die Tat umgesetzt werden sollte. Beispielsweise sollten sowohl Regierungen als auch Unternehmen ihre Bemühungen um Datenschutz verstärken und sich stets vor Augen halten, dass die sichersten Daten diejenigen sind, die gar nicht erst erhoben werden.

Sobald es darum geht, dass Algorithmen autonom Entscheidungen treffen, wird der entsprechende Bereich der Ethik als *Maschinenethik* bezeichnet. Diese beginnt bei der häufig gewonnenen Erkenntnis, dass Algorithmen menschliche Vorurteile übernehmen, die in den aufgrund ebenso menschlicher Kriterien gesammelten Daten bereits vorhanden sind. Ein triviales Beispiel: Gesichtserkennungsalgorithmen, die etwa zur Authentifizierung eingesetzt werden, weisen die beste Erkennungsrate bei weißen Männern auf, weil von diesen viel mehr Testdaten vorliegen als von anderen Gruppen (denn genau diese Gruppe arbeitet am häufigsten in Technologieunternehmen). Bei der automatisierten (Aus-)Sortierung von Bewerbungen haben sich bereits ähnliche Vorurteile gezeigt, sodass es potenziell auch mit der nächsten Generation Angestellter kaum besser wird.

Noch schwerwiegender sind ethische Erwägungen bei Systemen, die mit der bewegten menschlichen Umwelt interagieren, etwa bei autonomen Fahrzeugen. Im Extremfall geht es darum, zu entscheiden, mit welchen von mehreren Personen oder Fahrzeugen eine Kollision stattfinden soll, wenn sich diese nicht mehr ganz vermeiden lässt.

Ein bekanntes Gedankenexperiment für das zugrunde liegende moralische Dilemma ist das sogenannte *Trolley-Problem*, das auf die Philosophinnen *Philippa Foot* und *Judith Jarvis Thomson* zurückgeht: Ein Güterwaggon rollt auf fünf Menschen zu, die auf den Eisenbahnschienen liegen. Wird rechtzeitig eine Weiche gestellt, kann der Waggon auf ein Gleis umgelenkt werden, wo er nur einen Menschen überrollen würde. Machen Sie sich den Versuch einer Beantwortung nicht zu leicht: Wenn Sie den Hebel nicht ziehen, ist es »nur« unterlassene Hilfeleistung, aber wenn Sie ihn ziehen, führen Sie den Tod der einzelnen Person aktiv herbei. Verschiedene Ethikschulen finden unterschiedliche Antworten auf derartige Fragen.

In der Maschinenethik ist zudem kein Mensch zugegen, der eine Entscheidung treffen muss, sondern ein Algorithmus. Die Menschen, die ihm die Grundlagen dafür einprogrammieren, müssen über solche ethischen Fragen beraten; immer häufiger wird auch der Ruf nach gesetzlicher Regulierung laut.

Die vielleicht fürchterlichste vorstellbare KI-Anwendung sind autonome Waffensysteme, die selbstständig entscheiden, ob sie ein Ziel erfassen und beschießen sollen. Idealerweise sollte es ein internationales Abkommen geben, das ihre Existenz ohne Ausnahme verbietet. Da Rüstungsunternehmen jedoch großen Einfluss auf Regierungen nehmen, weil sich mit ihren Geschäften Milliarden verdienen lassen, ist es leider gut vorstellbar, dass ein solches Abkommen nicht zustande kommen wird.

Der Science-Fiction-Autor und Biochemiker *Isaac Asimov* sah solche Probleme kommen und formulierte die drei *Robotergesetze*, die solche Auswüchse ausschließen sollen:

1. Ein Roboter darf einen Menschen nicht verletzen und auch nicht durch Untätigkeit zulassen, dass ein Mensch zu Schaden kommt.
2. Ein Roboter muss einem Menschen gehorchen, es sei denn, das erste Gesetz würde verletzt.
3. Ein Roboter muss seine eigene Existenz beschützen, es sei denn, das erste oder zweite Gesetz würde verletzt.

Asimov ließ jedoch ein Schlupfloch für eine mögliche Umgehung des ersten Gesetzes für Notfälle. Die intelligenten Roboter in seinen Geschichten entwickeln eines Tages das nullte Gesetz: Ein Roboter darf die Menschheit nicht verletzen und auch nicht durch Untätigkeit zulassen, dass die Menschheit zu Schaden kommt. Die drei bereits vorhandenen Gesetze hängen vom nullten ab. Unter dem nullten Gesetz wären Roboter im Grunde verpflichtet, Genozid, Kriege oder die Klimakatastrophe zu verhindern, selbst wenn sie dafür einzelne Menschen beseitigen müssten.

Allerdings gehen Asimovs Geschichten davon aus, dass die Roboter mit ihren »Positronengehirnen« die Stufe der starken KI erreicht haben, sich ihrer selbst und ihres Handelns also bewusst sind. Die bisher in der Praxis vorhandenen KI-Implementierungen könnten sich ein abstraktes Konzept wie »die Menschheit« noch nicht einmal vorstellen.

1.2.5 Sprachen und Tools für künstliche Intelligenz

Dieses Buch verwendet für alle KI- und Machine-Learning-Implementierungen ausschließlich die Programmiersprache *Python* mitsamt diversen Modulen, die in der täglichen Praxis eingesetzt werden. Eine Einführung in die Sprache erhalten Sie in Kapitel 3, »Programmierkurs mit Python«.

Natürlich gibt es auch viele andere Programmiersprachen und Tools, die für Machine Learning, künstliche Intelligenz und Datenanalyse verwendet werden. Wichtige aktuelle Beispiele sind die verbreitete Statistiksprache *R* oder die ziemlich neue Programmiersprache *Julia*. Auch die viele Jahrzehnte alte funktionale Sprache *Lisp* kommt immer noch zum Einsatz, wenn auch seltener für die eher mathematisch orientierten Machine-Learning-Algorithmen und häufiger für logischer orientierte KI-Verfahren.

Für das Prototyping, also die ersten Testentwürfe neuer Algorithmen, wird oft auch spezifische Mathematiksoftware eingesetzt, zum Beispiel *Matlab* oder dessen Open-Source-Nachbau *Octave*, der etwas weniger umfangreich ist. Produktiv genutzte KI-Software wird dagegen durchaus auch in Compilersprachen wie *C++* implementiert, weil diese einen Performancevorteil gegenüber Skriptsprachen wie Python haben.

Im Bereich Big Data wird spezifische Software verwendet, um die gewaltigen Datenmengen zu verwalten, die dort verarbeitet werden. Eines der bekanntesten Programme dafür ist *Apache Hadoop*. Die Software hält die Daten auf beliebig vielen Serverrechnern vor, kann sie bei Bedarf wieder auffinden und sorgt dabei für Stabilität und Ausfallsicherheit. Auch Datenbankserver – Open-Source-Systeme wie MySQL, PostgreSQL und kommerzielle wie Oracle und Microsoft SQL Server – sind in der Lage, Daten auf praktisch beliebig viele physische Rechner zu verteilen.

Auch professionelle Cloud-Computing-Dienste wie Amazon Web Services oder Microsoft Azure bieten Dienstleistungen rund um Machine Learning und KI an. Bei Azure bilden diese Themen inzwischen sogar einen Schwerpunkt, der durch spezielle Microsoft-Zertifizierungen unterstützt wird.

Neben der Software kann auch die Hardware speziell an die Anforderungen der Datenanalyse angepasst sein, und die entsprechende Software muss diese unterstüt-

zen. Zum Beispiel sind die Prozessoren moderner Grafikkarten (*Graphics Processing Unit* oder kurz *GPU*) besonders gut geeignet, um die umfangreichen Berechnungen für bestimmte Machine-Learning-Algorithmen und neuronale Netzwerke mit der höchsten möglichen Geschwindigkeit auszuführen, denn genau wie in Echtzeit gerenderte 3D-Grafik wird dafür unter anderem möglichst schnelle Fließkomma-Matrixmultiplikation benötigt.

Damit auch tatsächlich die GPU statt der CPU verwendet wird, müssen die verwendeten Programmierbibliotheken dafür optimiert sein. Solche Optimierungen gehen über den Fokus dieses Buchs hinaus, aber die Dokumentation der Tools und Sprachen kann Ihnen Auskunft darüber geben, ob sie GPU-fähig sind.

Kapitel 5

Algorithmen und Datenstrukturen

Es gibt ein unfehlbares Rezept, eine Sache gerecht unter zwei Menschen aufzuteilen: Einer von ihnen darf die Portionen bestimmen, und der andere hat die Wahl.
– Gustav Stresemann

Das Schema, nach dem ein Computerprogramm ein Problem löst, wird als *Algorithmus* bezeichnet. Der Begriff ist älter als die Informatik und wird auch für mathematische Berechnungsverfahren – insbesondere solche mit mehreren Schritten – eingesetzt. Das Wort selbst ist vom Namen des persisch-arabischen Mathematikers *Muhammad ibn Musa al-Chwarizmi* abgeleitet, der im 8. bis 9. Jahrhundert in Bagdad wirkte. Sein Hauptwerk ist ein Buch, von dessen Titel das Wort »Algebra« abgeleitet wurde.

Eng mit Algorithmen verknüpft sind die *Datenstrukturen*. Sie bestimmen unter anderem, in welcher Reihenfolge Daten verarbeitet werden, und haben so großen Einfluss auf die Arbeitsweise verschiedener Algorithmen.

In diesem Kapitel lernen Sie einige klassische Datenstrukturen und Algorithmen kennen, die zur Lösung verschiedener Probleme mithilfe der Computerprogrammierung dienen.

Berechenbarkeit und Halteproblem

Ein wichtiges Kriterium für die Gültigkeit von Algorithmen ist die *Berechenbarkeit*. Das bedeutet: Ist das Problem, das der Algorithmus lösen soll, überhaupt durch Berechnung lösbar? Das ist einerseits eine praktische Frage, denn theoretische Berechenbarkeit nützt nichts, wenn die tatsächliche Ausführung Jahre oder noch länger benötigen würde. Es kommt also auf die *Komplexität* eines Algorithmus an, die für verschiedene Algorithmen in diesem Kapitel exemplarisch angegeben wird.

Andererseits gibt es jedoch auch Probleme, die überhaupt nicht durch Berechnung lösbar sind. Dieser Ansicht waren Mathematiker*innen nicht immer: Im 19. und bis zu Beginn des 20. Jahrhunderts glaubten viele von ihnen, dass es möglich sein

müsse, die gesamte Mathematik aus wenigen Axiomen herzuleiten. Besonders der deutsche Mathematiker *David Hilbert* und die britischen Philosophen *Bertrand Russell* und *Alfred North Whitehead* versuchten dies.

1931 bewies der junge österreichische Mathematiker *Kurt Gödel* jedoch, dass man in jeder formalen Sprache, wie die Mathematik eine ist, Sätze formulieren kann, die sich innerhalb dieser Sprache weder beweisen noch widerlegen lassen, obwohl von einer höheren Warte (außerhalb des Systems) klar ist, ob sie es sind oder nicht. Diese *Unvollständigkeitssätze* bedeuten nicht, dass die Mathematik unbrauchbar wäre, sondern nur, dass sie sich in verschiedene, zueinander inkompatible Richtungen erweitern lässt. Für die Geometrie ist Ähnliches schon viel länger bekannt.

Mit der Berechenbarkeit verwandt ist das *Halteproblem*: Ist es möglich, einen Algorithmus zu entwickeln, der für jeden anderen Algorithmus in endlicher Zeit überprüfen kann, ob jener in endlicher Zeit mit seiner Arbeit fertig wird? Der amerikanische Mathematiker *Alonzo Church* und der britische Informatikpionier *Alan Turing* beantworteten diese Frage in dem nach ihnen benannten Satz mit einem klaren Nein.

Solche Ausflüge an die Grenzen der Mathematik, Informatik und Philosophie sind besonders interessant, wenn Sie sich mit dem Thema künstliche Intelligenz beschäftigen. Im Anhang werden einige Bücher empfohlen, die sich näher damit beschäftigen.

5.1 Listen durchsuchen und sortieren

Viele Probleme der Informatik werden mit verschiedenen Arten der Suche gelöst. Ob nun bestimmte Textstellen in einer umfangreichen Datei, ein Weg durch ein Labyrinth oder die Lösung einer Gleichung gesucht werden: Verschiedene Grundformen der Suche sind verallgemeinerbar und können jeweils mehrere dieser Aufgaben lösen.

Wie die Suchalgorithmen konkret aussehen, hängt von der Beschaffenheit der zu durchsuchenden Menge ab, die als *Suchraum* bezeichnet wird. Für das Durchsuchen von Listen gibt es weniger Möglichkeiten als für andere Datenstrukturen, die in späteren Abschnitten vorgestellt werden.

Wir betrachten zunächst die beiden grundlegenden Suchverfahren für Listen. Danach gehe ich noch kurz auf Sortieralgorithmen ein.

5.1.1 Lineare Suche

Die *lineare Suche* ist die einfachste Form der Suche. Sie durchsucht den Suchraum, der in diesem Fall die Form einer sequenziellen Liste hat, vom Anfang bis zum Ende, um nach einem bestimmten Objekt Ausschau zu halten. (*Objekt* wird als allgemeiner Begriff für eine Zahl, einen String oder jede andere Art von Daten verwendet.) Dies kann aufgrund verschiedener Fragen geschehen:

- Kommt das gesuchte Objekt überhaupt in der Datenmenge vor?
- An welcher Stelle kommt das Objekt zum ersten Mal vor?
- Wie oft kommt das Objekt vor?
- An welchen verschiedenen Stellen kommt das Objekt vor?

Sie haben in Kapitel 3, »Programmierkurs mit Python«, bereits den Operator `in` kennengelernt, der eine Liste oder andere iterierbare Datenstrukturen nach einem Element durchsucht. Er beantwortet die Frage, ob das gesuchte Objekt im Suchraum vorhanden ist, und entspricht so dem einfachsten Anwendungsfall der linearen Suche. Dennoch wird die lineare Suche in diesem Abschnitt manuell implementiert, um zu zeigen, wie Sie von der Beschreibung eines Algorithmus über verschiedene formale Darstellungen zum fertigen Python-Skript gelangen.

In Alltagssprache formuliert, funktioniert der Algorithmus zur linearen Suche für all diese Anwendungsfälle wie folgt:

1. Beginne beim ersten Element.
2. Vergleiche das aktuelle Element mit dem gesuchten Objekt. Sind sie identisch? Falls ja, ist hier je nach Anwendungsfall entweder Schluss oder die Fundstelle wird gespeichert.
3. Wenn ein weiteres Element vorhanden ist, mache mit diesem weiter, ansonsten Ende.
4. Falls der Algorithmus noch nicht beendet wurde, zurück zu Schritt 2.

Alltagssprache ist nur eine Möglichkeit, Algorithmen darzustellen, und zwar nicht gerade die übersichtlichste. Geeigneter ist beispielsweise ein *Flussdiagramm*. In Abbildung 5.1 sehen Sie ein Beispiel für den Anwendungsfall der einfachen Überprüfung, ob das gesuchte Objekt vorhanden ist. Es handelt sich um ein vereinfachtes Flussdiagramm, weil nur vier Symboltypen vorkommen: ein Oval für den Start, Rechtecke für auszuführende Anweisungen, Rauten für Ja-Nein-Entscheidungen und abgerundete Rechtecke für das Ende. In der ISO-Norm 5807, die einen einheitlichen Standard für Flussdiagramme beschreibt, gibt es noch weitere Symbole, zum Beispiel ein eigenes Symbol für die Ausgabe.

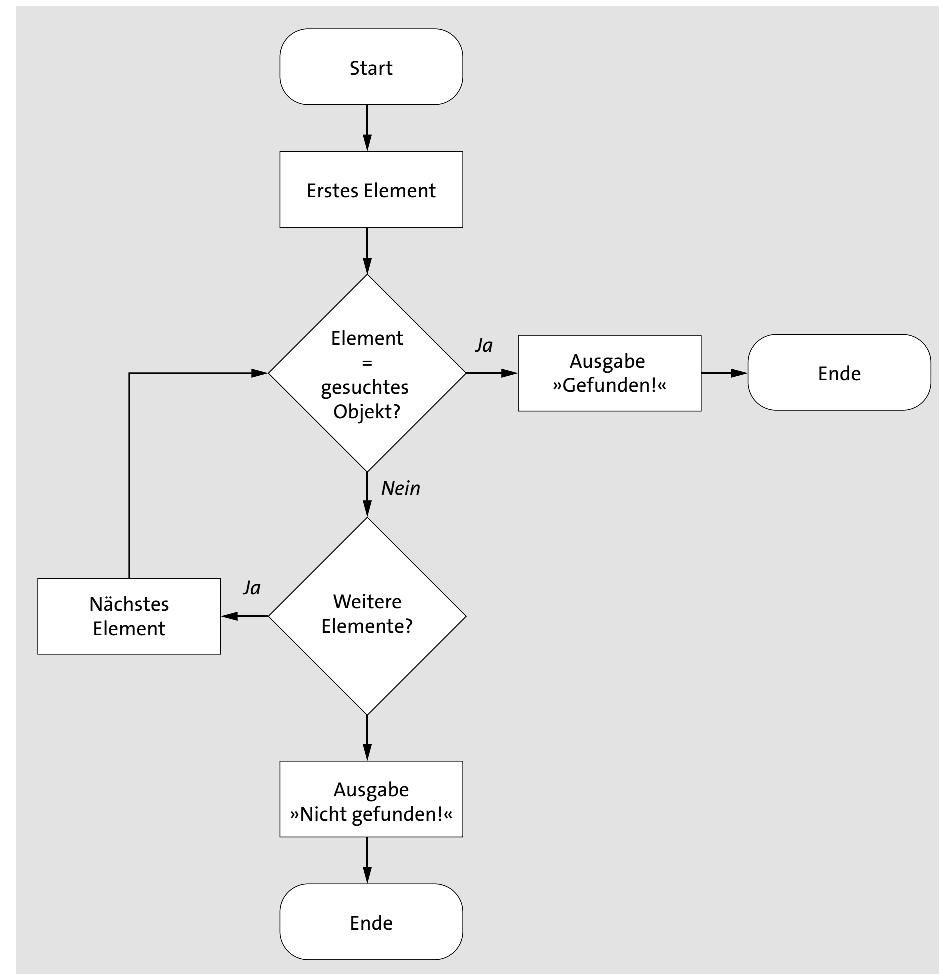

Abbildung 5.1 Lineare Suche mit der genauen Aufgabenstellung »Ist das gesuchte Objekt vorhanden?« als Flussdiagramm

Es gibt durchaus berechtigte Kritik an dieser Darstellungsform, denn das Flussdiagramm gibt die Struktur des Programms nicht korrekt wieder. Besonders Schleifen werden einfach als Verzweigungen angezeigt, wie Sie am Ja-Ausgang der Fallentscheidung »Weitere Elemente?« sehen. Andere Darstellungsformen sind in dieser Hinsicht besser geeignet, zum Beispiel das in Abbildung 5.2 gezeigte *Struktogramm* oder *Nassi-Shneiderman-Diagramm* (benannt nach seinen Erfindern), diesmal für die Frage, wie oft das gesuchte Objekt vorkommt.

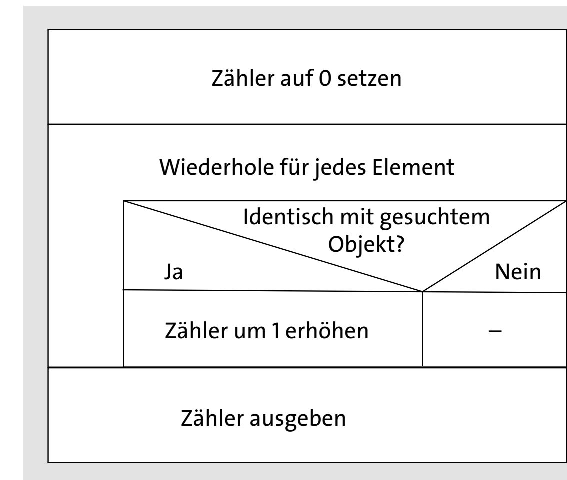

Abbildung 5.2 Nassi-Shneiderman-Diagramm (Struktogramm) für die lineare Suche mit dem Anwendungsfall »Wie oft kommt das gesuchte Objekt im Suchraum vor?«

Wie Sie sehen, lassen sich die Schleife (»Wiederhole für jedes Element«) und die Fallentscheidung (»Identisch mit gesuchtem Objekt?«) einfach voneinander unterscheiden, und es ist auch intuitiv verständlich, wie sich mehrere dieser Konstrukte ineinander verschachteln lassen.

Bei größeren Softwareprojekten dominieren andere Darstellungsformen, beispielsweise die verschiedenen Diagrammtypen der speziell für die objektorientierte Programmierung entwickelten *Unified Modeling Language* (UML). Diese kommt in Kapitel 9, »Geschäftsprozessanalyse«, kurz zur Sprache.

Eine weitere Darstellungsform für Algorithmen im Entwurfsstadium ist der *Pseudocode*. Bei ihm handelt es sich um einen Kompromiss zwischen natürlicher Sprache und einer Schreibweise, die an eine möglichst generische Programmiersprache angelehnt ist. Da Programmiersprachen jedoch sehr unterschiedlich sind, gibt es viele verschiedene Pseudocode-Varianten, die unter anderem von Pascal, BASIC oder C inspiriert wurden.

Da Python mit einem besonderen Augenmerk auf natürliche Lesbarkeit entwickelt wurde, können Sie sich diesen Schritt meist sparen, wenn Sie diese Sprache verwenden. Sie sollten aber darüber Bescheid wissen, weil viele Lehrbücher über Algorithmen Pseudocode statt Implementierungen in einer konkreten Sprache enthalten.

Hier sehen Sie den Anwendungsfall »An welcher Stelle kommt das Objekt zum ersten Mal vor?« in (Python-ähnlichem) Pseudocode:

```
Für jedes Element/Index im Suchraum:
    Wenn Element == GesuchtesObjekt:
        Rückgabe: Index
Rückgabe: nichts
```

Aus diesem Beispiel dürfte klar werden, warum die Pseudocode-Schreibweise für die Python-Entwicklung nicht sonderlich nützlich ist, denn in Python sieht das Ganze fast genauso aus (hier als einfache Funktion):

```
def linear_search(search_space, object):
    for index, element in enumerate(search_space):
        if element == object:
            return index
    return None
```

Das Beispiel können Sie direkt in `ipython` oder ein Jupyter Notebook eingeben und dann testen, beispielsweise so:

```
: search_space = [1, 3, 5, 7, 9]
: print(linear_search(search_space, 7))
3
: print(linear_search(search_space, 6))
None
```

Nützlicher ist jedoch eine Variante der Funktion, die in der Lage ist, alle denkbaren Anwendungsfälle der linearen Suche auf einmal abzubilden. In Listing 5.1 sehen Sie diese Funktion mitsamt einigen Tests. Ausführliche Erläuterungen finden Sie in den Kommentaren.

```
def linear_search(search_space, search_object, func = None,
                  find_all = False):
    # Zähler zuerst auf 0 setzen
    counter = 0
    # Schleife über jedes Element mit Index
    for index, element in enumerate(search_space):
        # Ist das aktuelle Element das gesuchte Objekt?
        if search_object == element:
            # Zähler um 1 erhöhen
            counter += 1
            # Evtl. Verarbeitungsfunktion mit Index der Fundstelle aufrufen
            if func is not None:
                func(index)
```



```
            # Falls nur ein Vorkommen gesucht, True zurückgeben
            if not find_all:
                return True
    # Falls nur ein Vorkommen gesucht, False zurückgeben
    if not find_all:
        return False
    # Ansonsten Zähler zurückgeben
    return counter

# Konkrete Verarbeitungsfunktion, die nur den Index der Fundstelle ausgibt
def print_index(index):
    print(index)

# Hauptprogramm zum Test aller Anwendungsfälle
if __name__ == '__main__':
    search_space = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    print("Existiert das Element?")
    print("7?")
    print(linear_search(search_space, 7))
    print("11?")
    print(linear_search(search_space, 11))
    print("Wie oft kommt das Element vor?")
    print("7?")
    print(linear_search(search_space, 7, find_all = True))
    print("10?")
    print(linear_search(search_space, 10, find_all = True))
    print("11?")
    print(linear_search(search_space, 11, find_all = True))
    print("Erstes Vorkommen von 7 bei Index?")
    linear_search(search_space, 7, func = print_index)
    print("Erstes Vorkommen von 11 (kommt nicht vor!) bei Index?")
    print(linear_search(search_space, 11, func = print_index))
    print("Alle Vorkommen von 7?")
    linear_search(search_space, 7, func = print_index, find_all = True)
```

Listing 5.1 »linear_search.py« implementiert alle hier diskutierten Anwendungsfälle der linearen Suche.

Die Ausgabe des Programms sieht folgendermaßen aus:

```
Existiert das Element?
7?
True
11?
False
Wie oft kommt das Element vor?
7?
2
10?
1
11?
0
Erstes Vorkommen von 7 bei Index?
6
Erstes Vorkommen von 11 (kommt nicht vor!) bei Index?
False
Alle Vorkommen von 7?
6
16
```

Dadurch, dass die Ausgabe der Indizes in eine optional als Argument übergebene Verarbeitungsfunktion ausgelagert wurde, lässt sich die Funktion sogar noch vielseitiger anwenden als ursprünglich konzipiert. Denn natürlich können Sie auch eine Funktion übergeben, die etwas anderes mit den Indizes anstellt.

Nun folgt ein interaktives Beispiel, das in einem String nach dem Zeichen `"."` sucht und die Indizes in einer Liste sammelt. Diese wird anschließend verwendet, um die Fundstellen durch `","` zu ersetzen, also das englische durch das deutsche Dezimalzahlformat. Dazu wird der String vorübergehend in eine Liste umgewandelt, weil Sie den Indexoperator bei Strings nicht zur Wertzuweisung verwenden können.

Stellen Sie sicher, dass sich *linear_search.py* im aktuellen Verzeichnis befindet, bevor Sie `ipython` starten:

```
: from linear_search import linear_search
: indices = [] # Liste der Indizes
: def collect_indices(index):
:     indices.append(index)

: def repl_dot_comma(numbers):
:     numbers_list = list(numbers)
:     for i in indices:
```



```
:         numbers_list[i] = ','
:     return ''.join(numbers_list)

: numbers = """1.1
: 2.5
: 3.1415926"""
: linear_search(numbers, '.', func = collect_indices, find_all = True)
3
: indices
[1, 5, 9]
: print(repl_dot_comma(numbers))
1,1
2,5
3,1415926
```

Diese Aktion ist natürlich endlos viel umständlicher als `re.sub("\.", ",", numbers)`, zeigt aber, wie flexibel die Funktion zur linearen Suche ist. Unter anderem kann sie neben Listen und Strings auch alle anderen iterierbaren Objekte durchsuchen.

5.1.2 Binärsuche

Ein wesentlicher Nachteil der linearen Suche ist ihre relative Langsamkeit. Bei einer Liste mit `n` Elementen müssen im ungünstigsten Fall `n` Vergleiche durchgeführt werden. Der Namensbestandteil »linear« deutet bereits darauf hin, dass dies der Fall ist. In der sogenannten *O-Notation* (engl. *big O notation*) wird die Komplexität linearer Suchalgorithmen mit $O(n)$ angegeben, was entsprechend als *lineare Komplexität* bezeichnet wird.[1]

Wenn es lediglich darum geht, zu überprüfen, ob ein bestimmtes Element im Suchraum vorkommt oder nicht, gibt es unter bestimmten Umständen ein Verfahren, das im Durchschnitt schneller arbeitet: die *Binärsuche*. Dabei wird der Suchraum wiederholt halbiert. Die notwendigen Voraussetzungen sind:

1. Die Elemente brauchen ein Ordnungskriterium, müssen sich also mithilfe von Operatoren wie `<` und `>` miteinander und mit dem gesuchten Objekt vergleichen lassen.
2. Der Suchraum muss in sortierter Form vorliegen. Das heißt, für jedes Element `e[i]` muss `e[i] >= e[i - 1]` gelten.

1 Im IT-Handbuch gehe ich näher auf die Komplexitätstheorie ein.

Die Sortierung des Suchraums dauert natürlich auch eine gewisse Zeit (wobei verschiedene Sortierverfahren wiederum unterschiedliche Komplexitätsklassen haben, wie im Laufe dieses Kapitels noch besprochen wird). Die Binärsuche lohnt sich daher immer dann, wenn nicht nur eine Suche stattfinden soll. Ihre Komplexität ist logarithmisch und wird entsprechend als $O(\log n)$ angegeben.

Der Ablauf der Binärsuche ist wie folgt:

1. Zunächst wird der gesamte (sortierte) Suchraum betrachtet.
2. Aus dem aktuellen Suchraum wird das mittlere Element ausgewählt. (Bei einer geraden Anzahl muss konsistent entschieden werden, ob in jeder Runde das Element links oder das Element rechts der leeren Mitte verwendet wird.)
3. Wenn das ausgewählte Element dem gesuchten Objekt entspricht, ist klar, dass es im Suchraum vorhanden ist, und die Suche ist beendet.
4. Wenn das gesuchte Objekt kleiner als das aktuelle Element ist und es noch kleinere Element gibt, wird der Bereich links vom aktuellen Element zum neuen aktuellen Suchraum.
5. Ist das gesuchte Objekt dagegen größer als das aktuelle Element und sind noch größere Elemente vorhanden, wird der Bereich rechts vom aktuellen Element zum neuen aktuellen Suchraum.
6. Wenn die Bedingungen aus Schritt 4 und 5 nicht zutreffen, steht fest, dass das gesuchte Objekt nicht im Suchraum vorkommt, und die Suche ist beendet.
7. Der bisherige Ablauf ab Schritt 2 wird mit dem verkleinerten Suchraum wiederholt.

In Listing 5.2 sehen Sie eine ausführlich kommentierte Implementierung der Binärsuche nach dem obigen Schema.

```
def binary_search(search_space, search_object):
    # Vorerst Endlosschleife
    while True:
        # Index des mittleren Elements (unter der Mitte bei gerader Anzahl)
        center = len(search_space) // 2
        # Aktuelles Element
        current = search_space[center]
        # Gefunden?
        if current == search_object:
            return True
        # Kleiner als aktuelles Element, und gibt es noch kleinere?
        if search_object < current and center > 0:
```



```
            # Anfang bis ausschließlich Mitte ist neuer Suchraum
            search_space = search_space[0:center]
        # Größer als aktuelles Element, und gibt es noch größere?
        elif search_object > current and center < len(search_space) - 1:
            # Mitte + 1 bis Ende ist neuer Suchraum
            search_space = search_space[center + 1:]
        else:
            # Alles durchsucht, nichts gefunden
            return False

if __name__ == '__main__':
    list1 = [3, 7, 9, 13, 24, 37, 42, 49, 53]
    list2 = [3, 7, 9, 13, 24, 37, 42, 49, 53, 60]
    search_objects = [3, 53, 9, 49, 60, 61]
    for search_object in search_objects:
        print(f"{search_object} in {list1}? {binary_search(list1,
        search_object)}")
        print(f"{search_object} in {list2}? {binary_search(list2,
        search_object)}")
```

Listing 5.2 »binary_search.py« sucht in einer bereits sortierten Liste binär nach einem Objekt.

Hier sehen Sie die Ausgabe des Hauptprogramms:

```
3 in [3, 7, 9, 13, 24, 37, 42, 49, 53]? True
3 in [3, 7, 9, 13, 24, 37, 42, 49, 53, 60]? True
53 in [3, 7, 9, 13, 24, 37, 42, 49, 53]? True
53 in [3, 7, 9, 13, 24, 37, 42, 49, 53, 60]? True
9 in [3, 7, 9, 13, 24, 37, 42, 49, 53]? True
9 in [3, 7, 9, 13, 24, 37, 42, 49, 53, 60]? True
49 in [3, 7, 9, 13, 24, 37, 42, 49, 53]? True
49 in [3, 7, 9, 13, 24, 37, 42, 49, 53, 60]? True
60 in [3, 7, 9, 13, 24, 37, 42, 49, 53]? False
60 in [3, 7, 9, 13, 24, 37, 42, 49, 53, 60]? True
61 in [3, 7, 9, 13, 24, 37, 42, 49, 53]? False
61 in [3, 7, 9, 13, 24, 37, 42, 49, 53, 60]? False
46 in [3, 7, 9, 13, 24, 37, 42, 49, 53]? False
46 in [3, 7, 9, 13, 24, 37, 42, 49, 53, 60]? False
```

Dieses Hauptprogramm zeigt, wie Sie die Implementierung eines Algorithmus gründlich testen: Versuchen Sie stets, alle denkbaren Fälle zu bedenken. In diesem Fall werden zwei Listen mit ungerader beziehungsweise gerader Anzahl von Elementen durchsucht, wobei die gesuchten Objekte von beiden Enden und aus dem Inneren der Listen stammen oder aber in einer oder beiden nicht vorkommen.

Wenn Sie wissen möchten, wie die Binärsuche genau vonstatten geht, können Sie hinter der Zeile `current = search_space[center]` eine zusätzliche Zeile zur Testausgabe einfügen:

```
print(f"Suchraum: {search_space},
mittl. Element: {current} (Index {center})")
```

Hier folgt exemplarisch die Ausgabe für die Suche nach 60 in beiden Listen, wobei der gesuchte Wert nur in der zweiten Liste vorkommt:

```
Suchraum: [3, 7, 9, 13, 24, 37, 42, 49, 53], mittl. Element: 24 (Index 4)
Suchraum: [37, 42, 49, 53], mittl. Element: 49 (Index 2)
Suchraum: [53], mittl. Element: 53 (Index 0)
60 in [3, 7, 9, 13, 24, 37, 42, 49, 53]? False
Suchraum: [3, 7, 9, 13, 24, 37, 42, 49, 53, 60], mittl. Element: 37 (Index 5)
Suchraum: [42, 49, 53, 60], mittl. Element: 53 (Index 2)
Suchraum: [60], mittl. Element: 60 (Index 0)
60 in [3, 7, 9, 13, 24, 37, 42, 49, 53, 60]? True
```

Wie Sie sehen, sind nur je drei Durchläufe nötig. Wenn Sie dieselben Listen und dasselbe Suchelement der linearen Suche übergäben, wären neun beziehungsweise zehn Durchläufe nötig, denn da es sich einmal um ein nicht vorhandenes Element und einmal um das letzte in der Liste handelt, tritt hier jeweils der ungünstigste Fall der linearen Suche ein. Überschlagsweise stimmt daher sogar der Vergleich der Komplexitäten gemäß der Komplexitätsklassen der beiden Algorithmen:

```
: import math
: math.log2(9)
3.169925001442312
: math.log2(10)
3.321928094887362
```

Warum der Zweierlogarithmus und nicht der natürliche? Das liegt daran, dass eben binär gesucht wird, wodurch sich der Suchraum mit jedem Durchlauf genau halbiert. Also ist die Basis des hier zugrunde liegenden Logarithmus 2 und nicht *e*. Beachten Sie aber, dass logarithmische Komplexität unabhängig von der konkreten Basis

immer einfach als $O(\log n)$ angegeben wird. Daher spricht man auch konkreter von einer *Komplexitätsklasse*.

5.1.3 Listen sortieren

Nicht nur für die Binärsuche ist es oftmals notwendig, Listen zu sortieren. Die zuständigen Python-Anweisungen `sort(liste)` und `liste.sorted()` haben Sie bereits kennengelernt. Dennoch lohnt sich ein kurzer Blick darauf, wie Sortierverfahren hinter den Kulissen implementiert werden. Es gibt übrigens endlos viele Sortierverfahren; hier sollen zwei der bekanntesten genügen.

Bubblesort

Der vielleicht einfachste und intuitivste Sortieralgorithmus wird als *Bubblesort* bezeichnet, weil er an das Aufsteigen von Luftblasen im Wasser erinnert: Benachbarte Elemente werden so lange miteinander vertauscht, bis alle an der richtigen Position sind.

Formal lässt sich der Algorithmus wie folgt zusammenfassen:

1. Bis auf Weiteres davon ausgehen, dass die Liste bereits sortiert ist.
2. Erstes bis vorletztes Element mit dem zweiten bis letzten vergleichen.
3. Ist ein Element größer als sein Nachfolger, beide vertauschen und feststellen, dass die Liste noch nicht sortiert ist.
4. Beenden, falls die Liste noch immer als sortiert gilt.
5. Zurück zu Schritt 1.

In Listing 5.3 finden Sie die Implementierung von Bubblesort gemäß dieser Beschreibung mitsamt zwei Testfällen im Hauptprogramm:

```
def bubblesort(unsorted):
    # Vorerst Endlosschleife
    while True:
        # Bis auf Weiteres gilt die Liste als sortiert
        is_sorted = True
        # Erstes bis vorletztes Element
        for i in range(0, len(unsorted) - 1):
            # Aktuelles Element größer als sein Nachfolger?
            if unsorted[i] > unsorted[i + 1]:
                # Elemente vertauschen
                unsorted[i], unsorted[i + 1] = unsorted[i + 1], unsorted[i]
```

```
                # Feststellung: Liste ist noch nicht sortiert
                is_sorted = False
        # Falls hier sortiert, Ende
        if is_sorted:
            break

if __name__ == '__main__':
    list1 = [7, 2, 9, 1, 8, 4, 6, 3, 5, 0, 9]
    list2 = ['Katze', 'Hund', 'Elefant', 'Maus', 'Affe', 'Giraffe']
    bubblesort(list1)
    print(list1)
    bubblesort(list2)
    print(list2)
```

Listing 5.3 »bubblesort.py« implementiert den Bubblesort-Algorithmus und probiert ihn an zwei Beispielen aus.

Beachten Sie, dass die Funktion `bubblesort()` keine Rückgabe hat. Das liegt daran, dass sie die übergebene Liste selbst sortiert, da diese ein veränderliches Objekt ist. Wenn Sie stattdessen eine sortierte Liste zurückgeben und das Original in Ruhe lassen möchten, müssen Sie zunächst eine Kopie der ursprünglichen Liste erstellen, etwa mit `unsorted[:]`.

Die Ausgabe der Beispiele dürfte wenig überraschend sein:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 9]
['Affe', 'Elefant', 'Giraffe', 'Hund', 'Katze', 'Maus']
```

Wenn Sie einen Zähler einbauen, der jeden Durchlauf der inneren Schleife zählt, kommen Sie bei der ersten Liste auf 100 und bei der zweiten auf 25 Durchläufe. Die erste Liste hat 11 Elemente, die zweite 6 – die Anzahl der Durchläufe scheint konsistent $(n - 1)^2$ bei n Durchläufen zu liegen. Die Komplexitätsklasse von Bubblesort ist also zweifellos $O(n^2)$.

Quicksort

Wenn Bubblesort sozusagen die Sortierentsprechung der linearen Suche ist, müsste es auch ein Äquivalent zur Binärsuche geben, bei dem der zu sortierende Bereich jeweils geteilt wird. In der Tat existiert ein solcher Algorithmus; er wird *Quicksort* genannt, weil er wesentlich schneller ist als Bubblesort.

Schematisch funktioniert Quicksort folgendermaßen:

1. Willkürlich ein Element aus der aktuellen Liste herauspicken. Dieses Element wird *Pivot* genannt. Auf Deutsch heißt es oft Median, obwohl es mit dem gleichnamigen statistischen Maß nichts zu tun hat. In der nachfolgenden Implementierung wird jeweils das erste Element aus der Liste verwendet, aber es kann wie gesagt jedes sein.
2. Wenn die aktuelle Liste nur ein Element hat, wird sie ohne weitere Aktion zurückgegeben.
3. Drei Teillisten anlegen, um Elemente zu sammeln, die kleiner als das Pivot, größer als das Pivot und identisch mit dem Pivot sind.
4. Alle Elemente der aktuellen Liste durchgehen und je nach Ergebnis des Vergleichs mit dem Pivot in eine der drei Listen einsortieren.
5. Auf die Liste der kleineren und die Liste der größeren Elemente werden die Schritte ab 1 jeweils erneut angewendet.
6. Die Ergebnisse aus Schritt 5 werden in der Reihenfolge »sortierte kleinere Elemente«, »gleiche Elemente«, »sortierte größere Elemente« zu einer neuen Liste zusammengefasst und zurückgegeben.

Wie Schritt 5 nahelegt, wird Quicksort am häufigsten rekursiv implementiert, da auf die jeweiligen Teillisten dieselben Schritte angewendet werden wie auf die ursprüngliche Liste. In Listing 5.4 sehen Sie die fertige Implementierung mit denselben Beispielen wie für Bubblesort.

```
def quicksort(unsorted):
    # Teillisten
    less = []
    equal = []
    greater = []

    # Mehr als ein Element in der aktuellen Teilliste
    if len(unsorted) > 1:
        # Vergleichselement (willkürlich)
        pivot = unsorted[0]
        for element in unsorted:
            if element < pivot:
                # Kleinere Elemente in eigene Liste
                less.append(element)
            elif element > pivot:
                # Größere Elememte in eigene Liste
                greater.append(element)
```

```
            else:
                # Gleiche Elemente in eigene Liste
                equal.append(element)
        # Teillisten sortieren und Gesamtliste zusammenstellen
        return quicksort(less) + equal + quicksort(greater)
    # Falls nur ein Element, einfach die Teilliste zurückgeben
    return unsorted

if __name__ == '__main__':
    list1 = [7, 2, 9, 1, 8, 4, 6, 3, 5, 0, 9]
    list2 = ['Katze', 'Hund', 'Elefant', 'Maus', 'Affe', 'Giraffe']
    quicksort(list1)
    print(list1)
    quicksort(list2)
    print(list2)
```

Listing 5.4 »quicksort.py« implementiert den Quicksort-Algorithmus und wendet ihn auf zwei Beispiele an.

Wenn Sie für die im Listing verwendeten Beispiele einen Zähler mitlaufen lassen, benötigt das Sortieren der ersten Liste 29 Durchläufe und das Sortieren der zweiten Liste 13 Durchläufe. Das ist erheblich besser als die 100 beziehungsweise 25 Runden, die Bubblesort benötigt.

Wie ist es um die Komplexitätsklasse von Quicksort bestellt? Die lineare Suche hat die Komplexität $O(n)$ und das darauf basierende Bubblesort $O(n^2)$. Könnte es also sein, dass die Komplexität von Quicksort $O(n \cdot \log n)$ beträgt, weil der Algorithmus ebenso wiederholt halbiert wie die Binärsuche, deren Komplexitätsklasse $O(\log n)$ ist? Probieren Sie es aus:

```
: import math
: 11 * math.log2(11)
38.05374780501027
: 6 * math.log2(6)
15.509775004326936
```

Die Ergebnisse sind etwas größer als die tatsächlichen Werte 29 und 13, weil die Komplexitätsklasse den ungünstigsten Fall angibt. Überschlagsweise stimmen sie jedoch. Quicksort ist also tatsächlich schneller als Bubblesort. Dennoch sollten Sie die eingebauten Sortierfunktionen von Python verwenden, wann immer es geht.

Kapitel 8
Künstliche neuronale Netzwerke

Auch ist das Suchen und Irren gut, denn durch Suchen und Irren lernt man.
– Johann Wolfgang Goethe

In der Praxis werden die verschiedenen Arten des Machine Learnings oft durch *künstliche neuronale Netzwerke* (KNN, engl. *artificial neural networks* oder kurz ANN) ausgeführt. In diesem Kapitel erhalten Sie zunächst einen Überblick darüber, was künstliche neuronale Netzwerke überhaupt sind, wie sie arbeiten und welche Arten es gibt. Anschließend wird ein einfaches KNN manuell (also nur mit Python und NumPy) geschrieben, und zum Schluss lernen Sie ein *scikit-learn*-Untermodul für neuronale Netzwerke und die darauf spezialisierten Python-Module *TensorFlow* und *Keras* kennen.

8.1 Einführung und Überblick

Ein künstliches neuronales Netzwerk wird oft als unvollkommene Nachbildung eines natürlichen Netzwerks, also des Gehirns eines Tieres oder gar eines Menschen bezeichnet. Das ist übertrieben. Man kann eher davon sprechen, dass einige Eigenschaften künstlicher neuronaler Netzwerke von der Natur inspiriert wurden, ohne ihren Leistungen nahe zu kommen. Ob es grundsätzlich auszuschließen ist, dass Software je das Niveau der Natur erreicht, ist seit Jahrzehnten Gegenstand philosophischer Diskussionen, die bisher ohne Endergebnis geblieben sind (siehe Kapitel 1, »Einführung«).

8.1.1 Natürliche und künstliche neuronale Netzwerke

Das Nervensystem von Tieren und Menschen, dessen Steuerzentrale bei höheren Tieren das Gehirn ist, besteht aus *Neuronen* (Nervenzellen). In Abbildung 8.1 sehen Sie eine Schemazeichnung eines Neurons.

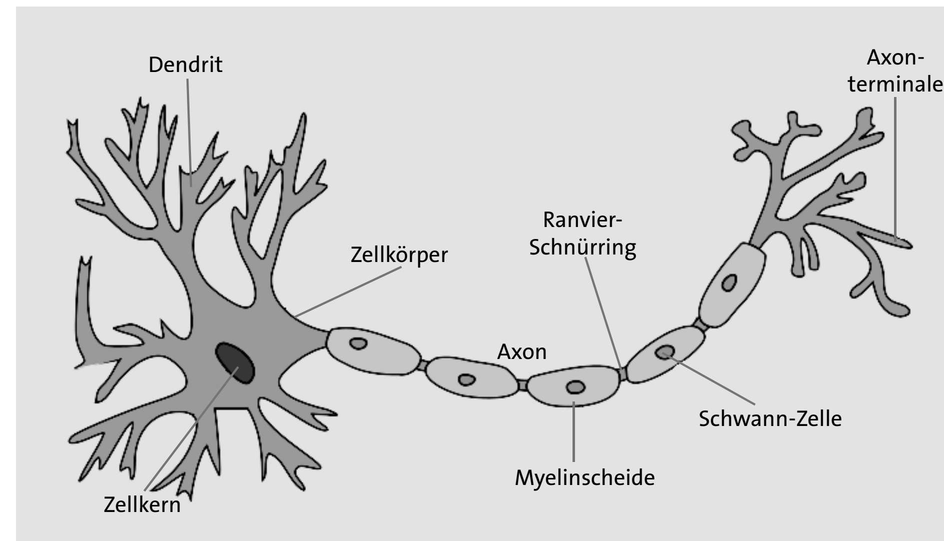

Abbildung 8.1 Schematische Darstellung eines Neurons mit seinen diversen Bestandteilen (Quelle: Wikipedia-User Quasar Jarosz, Lizenz CC BY-SA 3.0.)

Über die Dendriten erhält die Nervenzelle Signale von anderen (oft sehr vielen) Neuronen in Form elektrochemischer Impulse. Das Axon fasst alle eingehenden Signale zusammen, und wenn sie ein bestimmtes Potenzial überschreiten, beginnt das Neuron seinerseits zu »feuern«, also Signale auszusenden. Diese Signale verzweigen in die Axonterminale, die wiederum mit den Dendriten anderer Nervenzellen verbunden sind.

Ein künstliches neuronales Netzwerk besteht aus *künstlichen Neuronen*, die ebenfalls mehrere Eingangssignale zu einem Ausgabewert verrechnen. Ein KNN besteht aus mehreren Schichten, die durchlaufen werden: aus einer *Eingabeschicht* (engl. *input layer*), die die ursprünglichen Daten entgegennimmt, aus einer oder mehreren *versteckten Schichten* (engl. *hidden layers*), die sie weiterverarbeiten, und aus einer *Ausgabeschicht* (engl. *output layer*), aus der das Ergebnis abgelesen werden kann.

In Abbildung 8.2 sehen Sie ein Modell eines künstlichen neuronalen Netzwerks mit einer Eingabeschicht aus vier künstlichen Neuronen, einer einzelnen versteckten Schicht mit sechs von ihnen und einer Ausgabeschicht mit vier Neuronen. Jedes Neuron einer Schicht ist mit jedem der folgenden verbunden.

Das ist bei vielen, aber nicht bei allen neuronalen Netzwerken der Fall. Das Modell entspricht jedoch demjenigen, das im nächsten Hauptabschnitt manuell implementiert wird.

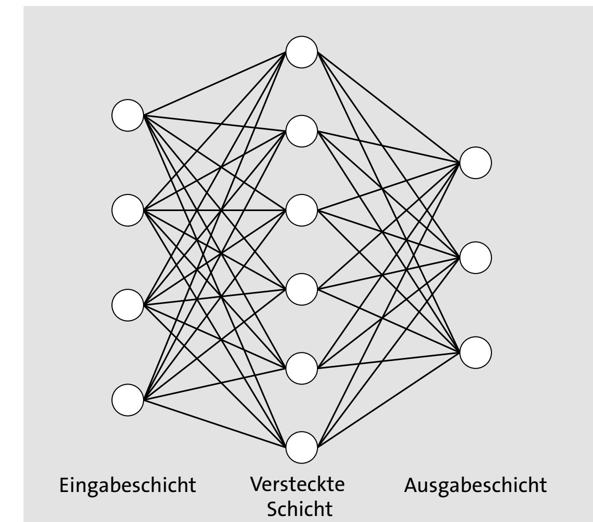

Abbildung 8.2 Modell eines einfachen künstlichen neuronalen Netzwerks: eine Eingabeschicht mit vier Neuronen, eine einzelne versteckte Schicht mit sechs Neuronen und eine Ausgabeschicht mit drei künstlichen Neuronen

8.1.2 Arten künstlicher neuronaler Netzwerke

Es gibt nicht nur eine Sorte neuronaler Netzwerke, sondern viele verschiedene. Einige der wichtigsten werden im Folgenden mit ihren wesentlichen Eigenschaften kurz vorgestellt.

Feedforward-Netzwerke

Das *Feedforward-Netzwerk* ist die einfachste Form eines künstlichen neuronalen Netzwerks. Ein seit den 1960er-Jahren bekannter Vorläufer dieses Konstrukts wurde *Perzeptron* genannt. Im Feedforward-Netzwerk erfolgt der Datenfluss bei der Verarbeitung der Daten ausschließlich in eine Richtung: von der Eingabeschicht über die aufeinanderfolgenden versteckten Schichten (falls überhaupt mehrere vorhanden sind) bis zur Ausgabeschicht.

Jedes künstliche Neuron verfügt über so viele *Gewichte* (Fließkommawerte), wie es Eingänge hat. Aus den Eingabewerten und den Gewichten wird das Skalarprodukt gebildet, das anschließend durch eine sogenannte *Aktivierungsfunktion* modifiziert wird. Eine sehr typische Aktivierungsfunktion ist die Sigmoid-Funktion, die Sie bereits im vorigen Kapitel im Zusammenhang mit der logistischen Regression kennengelernt haben. In Abbildung 8.3 wird das Modell eines einzelnen Neurons im

Feedforward-Netzwerk gezeigt. Wie Sie sehen, wird aus den Vektoren $\vec{w}$ (Gewichte des Neurons) und $\vec{i}$ (Eingabewerte aller Neuronen der vorigen Schicht) das Skalarprodukt gebildet. Das Ergebnis wird durch die besagte Aktivierungsfunktion modifiziert und anschließend an alle Neuronen der nächsten Schicht weitergegeben.

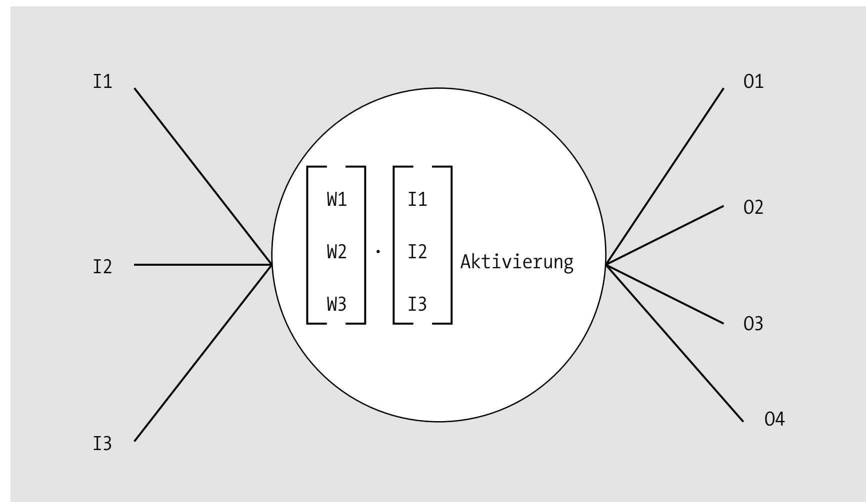

Abbildung 8.3 Modell eines einzelnen Neurons im Feedforward-Netzwerk

Ein Feedforward-Netzwerk lernt durch einen Vorgang, der als *Backpropagation* bezeichnet wird und dem Gradientenabstieg ähnelt. Dabei werden Trainingsdaten zunächst vorwärts durch das Netzwerk geleitet. Die Ausgabewerte werden von den Sollwerten abgezogen und anschließend rückwärts durch das Netzwerk geschickt. Dadurch werden die Gewichte bei jedem Lernschritt etwas besser an die Trainingsdaten angepasst. Genau wie bei den Regressionsverfahren kommt auch hier eine Lernrate zum Einsatz, die für ausreichend kleine Anpassungsschritte sorgt.

Der Hauptanwendungszweck einfacher Feedforward-Netzwerke ist die Klassifikation, die bei ihnen ähnlich funktioniert wie bei der logistischen Regression. Der wichtigste Vorteil des neuronalen Netzwerks ist, dass es beliebig viele Kategorien unterscheiden kann und nicht nur zwei wie ein einzelner logistischer Klassifizierer. Außerdem kann das Netzwerk eine hohe Performance erreichen, wenn in einer Trainingsrunde (bei neuronalen Netzwerken *Epoche* genannt) statt der einfachen Punktprodukte alle Trainingsdaten gleichzeitig in einer Matrixmultiplikation mit den Gewichten der jeweiligen Schicht multipliziert werden und wenn eine gute Bibliothek für lineare Algebra wie NumPy zum Einsatz kommt.

Rekurrente neuronale Netzwerke

Anders als bei einem Feedforward-Netzwerk fließen Daten in einem *rekurrenten neuronalen Netzwerk* (engl. *recurrent neural network*) nicht nur vorwärts, sondern es gibt diverse Formen der Rückkopplung, bei der Neuronen einer Schicht Daten an die Neuronen einer früheren Schicht weitergeben. Auf diese Weise lösen rekurrente Netzwerke Probleme, die nicht aus einzelnen, unzusammenhängenden Datensätzen bestehen, sondern ganze Sequenzen von Daten beinhalten. Das bekannteste Anwendungsbeispiel ist die Erkennung von Bildern, handgeschriebenen Texten oder gesprochener Sprache.

Eine bekannte Implementierung eines rekurrenten Netzwerks ist das LSTM (*Long Short-Term Memory*), bei dem ein »Kurzzeitgedächtnis« seine Informationen länger als bei anderen Implementierungen behält. Dazu werden statt der gewöhnlichen künstlichen Neuronen LSTM-Module verwendet, die nicht nur einen Eingang (*Input-Gate*) und einen Ausgang (*Output-Gate*) besitzen, sondern auch das sogenannte *Forget-Gate*, das bestimmt, wie lange ein Wert in der Zelle verbleibt. Für die verschiedenen Gates werden verschiedene Aktivierungsfunktionen und verschiedene Vektor- und Matrixoperationen verwendet.

LSTMs können nicht nur vorhandene Daten interpretieren, sondern auch »kreativ« neue generieren, beispielsweise Texte oder Bilder.

Convolutional Neural Networks

Ein *Convolutional Neural Network* (der deutsche Begriff, der jedoch nicht genutzt wird, wäre »faltendes neuronales Netzwerk«) besitzt nicht nur Schichten, die Daten mithilfe von Gewichten modifizieren, sondern auch solche, die sogenannte *Faltungsfunktionen* anwenden. Eine Faltungsfunktion beschreibt allgemein gesagt, wie eine zweite Funktion die Form einer dritten modifiziert. Eine typische Anwendung ist die Bilderkennung, denn Faltungsfunktionen sind gut für die Erkennung von Kanten, Kontrasten und anderen Mustern in Bildern geeignet.

Eine solche Vorgehensweise verbessert die Performance der Datenverarbeitung dramatisch, und wenn die konkreten Operationen geschickt gewählt und gut implementiert werden, verringern sie nicht die Genauigkeit des Netzwerks.

Deep-Learning-Netzwerke

Beim *Deep Learning* (der deutsche Begriff »tiefes Lernen« wird nur selten verwendet) werden neuronale Netzwerke mit besonders vielen versteckten Schichten verwendet, die zudem oft jeweils unterschiedliche Aufgaben erledigen. Zudem wird ein sol-

ches Netzwerk besonders gründlich trainiert, das heißt mit möglichst vielen Trainingsdaten und über viele Durchgänge.

Professionelle Deep-Learning-Netzwerke verwenden Millionen von Features und führen in den verschiedenen Schichten oft Milliarden von Operationen durch. Ihre Ergebnisse sind entsprechend beeindruckend. Beispielsweise handelte es sich bei den künstlichen neuronalen Netzwerken, die Weltmeister in Schach und dem japanischen Brettspiel Go besiegten, und auch bei der KI, die das Quizspiel Jeopardy gewann, um Deep-Learning-Konstrukte.

Generative Adversarial Networks

Bei einem *Generative Adversarial Network* (GAN) treten zwei Deep-Learning-Netzwerke gegeneinander an. Eines von ihnen erfüllt eine bestimmte Aufgabe, das andere beurteilt die Lösungen und bringt das erste Netzwerk so dazu, seine Leistung kontinuierlich zu verbessern. Besonders häufig werden Generative Adversarial Networks eingesetzt, um zufällige, aber bestimmten Regeln genügende und daher sinnvoll aussehende Datenmuster zu erzeugen.

Ein besonders beeindruckendes Beispiel für die Arbeit eines GANs können Sie sich unter *https://thispersondoesnotexist.com/* anschauen: Das Netzwerk erzeugt zufällige und oft sehr echt wirkende Porträtfotos von Menschen, die es mit genau diesem Aussehen in Wirklichkeit überhaupt nicht gibt. Selbst die sichtbaren Ausschnitte von Kleidung und verschiedene Hintergründe werden jeweils hinzugefügt. (Letztere enthalten allerdings mitunter Artefakte, die wie verfremdete menschliche Körperteile aussehen, weil offenbar ein signifikanter Anteil der Trainingsdaten aus Gruppenbildern ausgeschnitten wurde.)

Natürlich wurde das Netzwerk mit einer riesigen Menge an Trainingsdaten in Form echter Porträtfotos trainiert, bevor es seine heutigen Leistungen erreicht hat. Wenn Sie die Seite im Browser neu laden, wird jeweils ein neues Porträt generiert. Rechts unten auf der Seite sind Erläuterungen und sogar der Quellcode verlinkt, zudem verwandte Projekte, die zufällige Katzen, Pferde, Kunstwerke oder Molekülmodelle generieren.

8.2 Ein neuronales Netzwerk manuell implementieren

In diesem Abschnitt wird ein künstliches neuronales Netzwerk programmiert, ohne eine spezialisierte Bibliothek zu verwenden. Es kommen also nur NumPy und Teile der Standardbibliothek zum Einsatz. Das Netzwerk ist ein Feedforward-Netzwerk, das

durch Backpropagation lernt. Eine offensichtliche Einschränkung besteht darin, dass es nur eine versteckte Schicht besitzt. Das Netzwerk wird testweise eingesetzt, um zwei verschiedene Klassifikationsprobleme zu lösen.

8.2.1 Mathematische Vorüberlegungen

Die Abläufe in Feedforward-Netzwerken, also die vorwärts gerichtete Klassifikation von Daten und die in umgekehrte Richtung verlaufende Backpropagation, lassen sich sehr effizient durch Matrixmultiplikationen nach dem Falk-Schema beschreiben. Der mathematische Hintergrund dieser Operation wurde in Kapitel 2, »Mathematische Grundlagen«, erläutert, und ihre Ausführung mithilfe von `numpy.dot()` haben Sie in Kapitel 4, »Mit Python-Modulen arbeiten«, kennengelernt.

Die Grundlage der Überlegung bildet das Geschehen in einem einzelnen Neuron. Ein Neuron besitzt, wie bereits erwähnt, so viele Gewichte wie Eingabewerte und berechnet sein Zwischenergebnis (auf das noch die Aktivierungsfunktion angewendet wird) als einfaches Skalarprodukt. Für n Eingabewerte und Gewichte sieht das Ganze also wie folgt aus:

$$\begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} = i_1 w_1 + \cdots + i_n w_n$$

Praktischerweise lassen sich jedoch die Gewichte aller Neuronen einer Schicht als Matrix schreiben. Für m Neuronen und n Eingabewerte ergibt dies eine Gewichtsmatrix der Form $W_{(m;n)}$:

$$W = \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \end{pmatrix}$$

Dies lässt sich ohne Weiteres mit einem n-zeiligen Eingabevektor multiplizieren, was wie folgt aussieht:

$$W \cdot \vec{i} = \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \end{pmatrix} \cdot \begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix}$$

Das Ergebnis ist ein m-zeiliger Vektor, was den Ausgaben aller Neuronen der aktuellen Schicht entspricht. Diese werden als Nächstes durch die Aktivierungsfunktion modifiziert und anschließend an die nächste Schicht weitergegeben, wo sinngemäß derselbe Vorgang stattfindet.

Noch praktischer ist, dass Sie beliebig viele Eingabedatensätze als Matrix darstellen und auf diese Weise mithilfe einer einzigen Falk-Schema-Operation sämtliche Ausgabewerte einer Schicht berechnen können. Das gilt sowohl für Trainingsdaten als auch für Test- oder Produktivdaten.

Eine Datenmenge mit p Datensätzen und n Features hat typischerweise folgendes Format:

$$I_{(p;n)} = \begin{pmatrix} {i^{(1)}}_1 & \cdots & {i^{(1)}}_n \\ \vdots & \ddots & \vdots \\ {i^{(p)}}_1 & \cdots & {i^{(p)}}_n \end{pmatrix}$$

Genau wie bei den Gewichten bilden die Features also die Spalten, doch das Falk-Schema funktioniert nur, wenn eine Matrix mit n Spalten (das sind die Gewichte) mit einer zweiten mit n Zeilen (das sind die Eingabedatensätze) multipliziert wird. Das bedeutet, dass die Matrix transponiert werden muss, sodass sich schließlich folgende Gesamtoperation für eine Schicht ergibt:

$$W \cdot I^T = \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \end{pmatrix} \cdot \begin{pmatrix} {i^{(1)}}_1 & \cdots & {i^{(p)}}_1 \\ \vdots & \ddots & \vdots \\ {i^{(1)}}_n & \cdots & {i^{(p)}}_n \end{pmatrix} = O_{(m;p)}$$

In Abbildung 8.4 wird dies für vier Datensätze mit je zwei Features illustriert, die in eine Schicht mit drei Neuronen überführt werden. Die konkrete Operation, die dort stattfindet, lautet also:

$$W \cdot I^T = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{pmatrix} \cdot \begin{pmatrix} {i^{(1)}}_1 & {i^{(2)}}_1 & {i^{(3)}}_1 & {i^{(4)}}_1 \\ {i^{(1)}}_2 & {i^{(2)}}_2 & {i^{(3)}}_2 & {i^{(4)}}_2 \end{pmatrix} = O_{(3;4)}$$

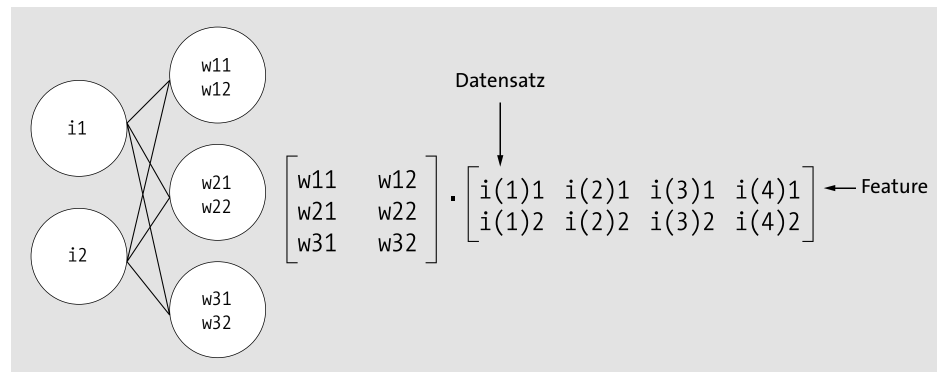

Abbildung 8.4 Schema der Berechnung von Ausgabewerten einer Schicht (vor der Aktivierung) aus mehreren Eingabedatensätzen

8.2.2 Das neuronale Netzwerk implementieren

Nach den Vorüberlegungen zum mathematischen Ablauf der Operationen im neuronalen Netzwerk müssen noch einige weitere Dinge geklärt werden, bevor das Netzwerk implementiert werden kann. Eine wichtige Baustelle sind die Ausgabewerte. Bei Klassifizierungsproblemen, die mit dieser Art von KNN typischerweise gelöst werden, sind die Zielwerte typischerweise verschiedene Kategorien, die beispielsweise als Strings oder als Ganzzahlen dargestellt werden. Die Ausgabe des Netzwerks für einen einzelnen Datensatz ist jedoch ein Vektor. Wenn es beispielsweise drei verschiedene Kategorien wie bei der Iris-Datenmenge gibt, lauten die drei idealen möglichen Ergebnisse wie folgt:

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Die tatsächlichen Resultate einer Klassifikation von Test- oder Produktivdaten bestehen üblicherweise nicht aus Einsen und Nullen, sondern aus Fließkommawerten dazwischen, denn wie bereits bei der logistischen Regression beschrieben, geben sie Wahrscheinlichkeitswerte an. Damit der Klassifikator wie erwartet funktioniert, wird das Maximum der Einzelwerte als 1 interpretiert, die restlichen als 0.

Beim Training werden dagegen die tatsächlich erreichten Werte von den erwarteten abgezogen. Bei der Backpropagation werden die Gewichte dann durch umgekehrte Matrixmultiplikationen mit diesen Fehlern modifiziert, um sie schrittweise so anzupassen, dass möglichst viele Prognosen richtig sind.

Um die erzielten Ergebnisse für Menschen lesbar zu machen, speichert die folgende Implementierung die verschiedenen Kategorien als Liste, auf die der Index des Maximums aus der Klassifikation wiederum als Index angewendet wird, was schließlich die passende Kategorie zurückliefert.

Bleibt noch zu erwähnen, dass die Eingabedaten wieder einer Feature-Skalierung unterzogen werden und dass die Gewichte mit Zufallswerten zwischen –0,5 und 0,5 initialisiert werden, weil dieser Wertebereich ausreichend diverse Ergebnisse für die Sigmoid-Funktion liefert. Die Zufallsmatrizen werden durch die Funktion `np.random.rand()` erzeugt, der nur die gewünschte Zeilen- und Spaltenzahl übergeben werden muss. In Listing 8.1 sehen Sie den kompletten, ausführlich kommentierten Quellcode:

```
import math
import csv
import numpy as np
from random import shuffle
```

```
# Sigmoidfunktion als Aktivierungsfunktion
def sigmoid(x):
    try:
        return 1 / (1 + math.exp(-x))
    except OverflowError:
        return 0

# Künstliches neuronales Netzwerk
class NeuralNetwork:

    # Attribute:
    # - Anzahl Neuronen der Eingabeschicht
    # - Anzahl Neuronen der versteckten Schicht
    # - Anzahl Neuronen der Ausgabeschicht
    # - Lernrate (benannt)
    def __init__(self, i_neurons, h_neurons, o_neurons, learning_rate = 0.1):
        # Grundattribute initialisieren
        self.input_neurons = i_neurons
        self.hidden_neurons = h_neurons
        self.output_neurons = o_neurons
        self.learning_rate = learning_rate
        self.categories = []

        # Gewichte als Zufallswerte initialisieren
        # Gewichte als Zufallswerte initialisieren
        self.input_to_hidden = np.random.rand(
            self.hidden_neurons, self.input_neurons
        ) - 0.5
        self.hidden_to_output = np.random.rand(
            self.output_neurons, self.hidden_neurons
        ) - 0.5
        # Aktivierungsfunktion für NumPy-Arrays
        self.activation = np.vectorize(sigmoid)

    # Daten vorbereiten
    # Attribute:
    # - Daten als zweidimensionale Liste
    # - Anteil, der als Testdaten abgespalten werden soll
    # - Kategorie in der letzten Spalte? Sonst in der ersten
    def prepare(self, data, test_ratio=0.1, last=True):
        if last:
```



```
            x = [line[0:-1] for line in data]
            y = [line[-1] for line in data]
        else:
            x = [line[1:] for line in data]
            y = [line[0] for line in data]
        # Feature-Skalierung (x)
        columns = np.array(x).transpose()
        x_scaled = []
        for column in columns:
            if min(column) == max(column):
                column = np.zeros(len(column))
            else:
                column = (column - min(column)) / (max(column) - min(column))
            x_scaled.append(column)
        x = np.array(x_scaled).transpose()
        # Kategorien extrahieren und als Attribut speichern
        y_values = list(set(y))
        self.categories = y_values
        # Verteilung auf Ausgabeneuronen (y)
        y_spread = []
        for y_i in y:
            current = np.zeros(len(y_values))
            current[y_values.index(y_i)] = 1
            y_spread.append(current)
        y_out = np.array(y_spread)
        separator = int(test_ratio * len(x))
        return x[:separator], y[:separator], x[separator:], y_out[separator:]

    # Ein einzelner Trainingsdurchgang
    # Attribute:
    # - Eingabedaten als zweidimensionale Liste/Array
    # - Zieldaten als auf Ausgabeneuronen verteilte Liste/Array
    def train(self, inputs, targets):
        # Daten ins richtige Format bringen
        inputs = np.array(inputs, ndmin = 2).transpose()
        targets = np.array(targets, ndmin = 2).transpose()
        # Matrixmultiplikation: Gewichte versteckte Schicht * Eingabe
        hidden_in = np.dot(self.input_to_hidden, inputs)
        # Aktivierungsfunktion anwenden
        hidden_out = self.activation(hidden_in)
```

```
        # Matrixmultiplikation: Gewichte Ausgabeschicht * Ergebnis versteckt
        output_in = np.dot(self.hidden_to_output, hidden_out)
        # Aktivierungsfunktion anwenden
        output_out = self.activation(output_in)
        # Die Fehler berechnen
        output_diff = targets - output_out
        hidden_diff = np.dot(self.hidden_to_output.transpose(), output_diff)
        # Die Gewichte mit Lernrate * Fehler anpassen
        self.hidden_to_output += (
            self.learning_rate *
            np.dot(
                (output_diff * output_out * (1.0 - output_out)),
                hidden_out.transpose()
            )
        )
        self.input_to_hidden += (
            self.learning_rate *
            np.dot(
                (hidden_diff * hidden_out * (1.0 * hidden_out)),
                inputs.transpose()
            )
        )

    # Vorhersage für eine Reihe von Testdaten
    # Attribute:
    # - Eingabedaten als zweidimensionale Liste/Array
    # - Vergleichsdaten (benannt, optional)
    def predict(self, inputs, targets = None):
        # Dieselben Schritte wie in train()
        inputs = np.array(inputs, ndmin = 2).transpose()
        hidden_in = np.dot(self.input_to_hidden, inputs)
        hidden_out = self.activation(hidden_in)
        output_in = np.dot(self.hidden_to_output, hidden_out)
        output_out = self.activation(output_in)
        # Ausgabewerte den Kategorien zuweisen
        outputs = output_out.transpose()
        result = []
        for output in outputs:
            result.append(
                self.categories[list(output).index(max(output))]
            )
```



```
        # Wenn keine Zielwerte vorhanden, Ergebnisliste zurückgeben
        if targets is None:
            return result
        # Ansonsten vergleichen und korrekte Vorhersagen zählen
        correct = 0
        for res, pred in zip(targets, result):
            if res == pred:
                correct += 1
        percent = correct / len(result) * 100
        return correct, percent


# Hauptprogramm
if __name__ == '__main__':
    with open('iris_nn.csv', 'r') as iris_file:
        reader = csv.reader(iris_file, quoting=csv.QUOTE_NONNUMERIC)
        irises = list(reader)
    shuffle(irises)
    network = NeuralNetwork(4, 12, 3, learning_rate = 0.2)
    x_test, y_test, x_train, y_train = network.prepare(
      irises, test_ratio=0.2
    )
    for i in range(200):
        network.train(x_train, y_train)
    correct, percent = network.predict(x_test, targets = y_test)
    print(f"{correct} korrekte Vorhersagen ({percent}%).")
```

Listing 8.1 »neural_network.py« implementiert ein einfaches künstliches neuronales Netzwerk.

Das Hauptprogramm wendet das neuronale Netzwerk auf die Iris-Datenmenge an. Damit stehen die Anzahlen der Eingabe- und Ausgabeneuronen fest, nämlich drei beziehungsweise vier. Die Größe der versteckten Schicht (zwölf Neuronen), die Lernrate von 0,2 und die Anzahl der Trainingsrunden (200) haben sich beim Ausprobieren als recht gut geeignet erwiesen. Eine Beispielausgabe des Programms sieht so aus:

```
$ python3 neural_network.py
28 korrekte Vorhersagen (93.33333333333333%).
```

Einige Stellen des Quellcodes sollten Sie sich etwas gründlicher anschauen. Die Methode `prepare()`, mit der die Daten für den Einsatz im Netzwerk vorbereitet wer-

den, erledigt beispielsweise eine Reihe verschiedener Aufgaben. Die Methode erwartet eine Datenmenge, deren Zeilen wie üblich die Datensätze bilden und in deren Spalten die Features und die jeweilige Klassifikation stehen. Da es in der Praxis Datenmengen gibt, bei denen die Kategorie in der letzten Spalte steht (zum Beispiel bei der Iris-Datenmenge), aber auch solche, bei denen sie sich in der ersten Spalte befindet, kann über das benannte Argument `last` angegeben werden, wie es sich bei der aktuellen Datenmenge verhält. Mithilfe des ebenfalls benannten Arguments `test_ratio` können Sie angeben, welcher Anteil der Datenmenge als Testdaten abgespalten werden soll. Die Vorgabe ist 10 %.

Die Methode führt verschiedene Datenextraktions- und Datenumformungsoperationen durch, bevor sie eine Liste von vier Arrays zurückgibt: Testdaten-Features (*x*), Testdaten-Ergebnisse (*y*), Trainingsdaten-Features (*x*), Trainingsdaten-Ergebnisse (*y*). Beachten Sie, dass die beiden Ergebnis-Teilmengen unterschiedliche Formate haben: Während die Ergebnisse für die Trainingsdaten in die beschriebene Matrix umgewandelt werden (wobei auch das Attribut `categories` mit den unterschiedlichen Kategorien befüllt wird), verbleiben die Testergebnisdaten im Originalformat, weil sie nicht vom Algorithmus benötigt werden, sondern nur der Kontrolle dienen.

Die Methode `train()` führt die im vorigen Abschnitt ausführlich beschriebenen Vektoroperationen durch, anschließend deren mithilfe der Lernrate gedämpfte Umkehrung als Backpropagation. Warum die Daten zu Beginn transponiert werden, wurde ebenfalls im Rahmen der mathematischen Grundlagen des Algorithmus beschrieben. Beachten Sie dabei das Argument `ndmin`, das hier zwei Dimensionen erzeugt. Sie können `train()` – und die in dieser Hinsicht ebenso funktionierende Methode `predict()` – also auch mit einzelnen Datensätzen aufrufen.

Wie oft Sie `train()` ausführen möchten, können Sie von außen durch eine Schleife steuern. Optional könnte die Methode so umgeschrieben werden, dass ihr die Anzahl der gewünschten Durchläufe als Argument übergeben wird.

In der Vorhersage- und Testmethode `predict()` werden dieselben Schritte ausgeführt, die `train()` vor der Backpropagation durchführt. Wenn Sie die zugehörigen Soll-Ergebnisse übergeben, werden Anzahl und Prozentsatz der korrekten Ergebnisse zurückgegeben, ansonsten einfach die Liste der Vorhersageergebnisse.

8.2.3 Ein weiteres Anwendungsbeispiel

Um das fertig implementierte neuronale Netzwerk einem weiteren Test zu unterziehen, wird es nun mit einer zweiten Datenmenge verwendet. Es handelt sich um knapp 1.800 auf 8×8 Pixel verkleinerte Abbildungen handgeschriebener Ziffern, die entsprechend als »0« bis »9« klassifiziert werden. Diese *Digits-Datenmenge* gehört

unter anderem zum Lieferumfang von *scikit-learn*. Es handelt sich um eine Teilmenge der berühmten *MNIST-Datenmenge*, die 70.000 Datensätze enthält und in der die Abbildungen der Ziffern 28 × 28 Pixel groß sind. Sie kommt am Ende dieses Kapitels zum Einsatz.

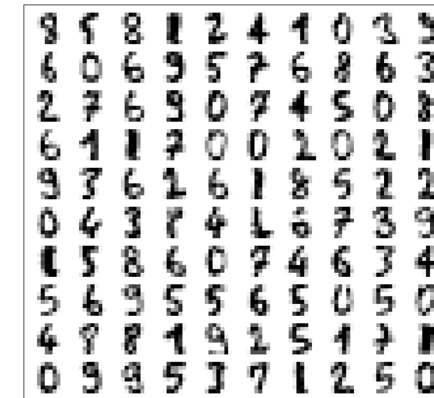

Abbildung 8.5 Zufällige Auswahl von 100 Ziffern aus der Digits-Datenmenge

In Abbildung 8.5 wird eine zufällige Auswahl von 100 Ziffern aus der Digits-Datenmenge gezeigt. Das kurze Skript in Listing 8.2 erzeugt eine solche Darstellung:

```
from sklearn.datasets import load_digits
from skimage.io import imshow
import matplotlib.pyplot as plt
import numpy as np

# Handschriftenerkennung laden
X, y = load_digits(return_X_y=True)
images = X[np.random.choice(len(X), 100, replace=False)]

# Bilder anzeigen
fig, axes = plt.subplots(10, 10, figsize=(8, 8))
ax = axes.ravel()
for i, image in enumerate(images):
    ax[i].imshow(1 - image.reshape([8, 8]), cmap=plt.cm.gray)
    ax[i].axis('off')
fig.tight_layout()
plt.show()
```

Listing 8.2 »draw-digits.py« stellt 100 zufällig gewählte Elemente aus der Digits-Datenmenge als Raster aus 10 × 10 Bildern dar.

Mit der NumPy-Funktion `random.choice()` können Sie zufällige Indizes aus einem Array auswählen. Als erstes Argument wird die Gesamtgröße des Arrays angegeben und als zweites Argument die Anzahl der Elemente. Das benannte Argument `return` ist `True`, wenn sich Werte wiederholen dürfen, und ansonsten `False`. (Das Ganze entspricht dem Ziehen aus einer Urne mit oder ohne Zurücklegen.)

Die `pyplot`-Funktion `subplots()` erzeugt ein Raster aus 10 mal 10 Bildern, in dem die Bilder angezeigt werden. `figsize` regelt die Pixelgröße jedes Anzeigebereichs. Mit der NumPy-Methode `ravel()` wird das zweidimensionale Array verflacht, damit ein einzelner numerischer Index darauf zugreifen kann; mit `reshape()` können Sie dasselbe Ergebnis erzielen, müssen dafür jedoch Argumente angeben.

Die darzustellenden Pixelwerte werden von 1 abgezogen. Dies invertiert die Bilder, sodass dunkle Pixel auf weißem Grund gezeichnet werden und nicht helle auf schwarzem. Mit `axis('off')` wird festgelegt, dass keine Achsenlinien und -beschriftungen gezeichnet werden. Schließlich verringert `tight_layout()` den Abstand zwischen den Bildern.

Das Skript in Listing 8.3 wendet das Feedforward-Netzwerk auf die Datenmenge an:

```
import csv
from neural_network import NeuralNetwork
from random import shuffle

with open('digits.csv', 'r') as digits_file:
    reader = csv.reader(digits_file, quoting=csv.QUOTE_NONNUMERIC)
    digits = list(reader)
shuffle(digits)
network = NeuralNetwork(64, 256, 10, learning_rate = 0.005)
x_test, y_test, x_train, y_train = network.prepare(
    digits, test_ratio = 0.1
)
for i in range(300):
    network.train(x_train, y_train)
correct, percent = network.predict(x_test, targets = y_test)
print(f"{correct} korrekte Vorhersagen ({percent}%).")
```

Listing 8.3 »neural_network_digits.py« wendet das selbst geschriebene neuronale Netzwerk auf die Digits-Datenmenge an.

Wie Sie sehen, wird die Datenmenge nicht aus *scikit-learn*, sondern aus einer lokalen Datei geladen (Sie finden sie in den Listings zu diesem Kapitel). Auch im vorliegenden Fall habe ich die Konfiguration für das Netzwerk (256 Neuronen in der versteckten Schicht, Lernrate 0,005, 300 Epochen) durch Versuch und Fehler ermittelt. Es dauert ein wenig, bis das Ergebnis ausgegeben wird. Eine übliche Ausgabe ist diese hier:


```
$ python3 neural_network_digits.py
165 korrekte Vorhersagen (92.17877094972067%).
```

Dieses Ergebnis ist einigermaßen akzeptabel, aber nicht überwältigend. Professionellere Implementierungen schneiden im Durchschnitt besser ab, wie Sie im nächsten Abschnitt sehen werden.

8.3 Neuronale Netzwerke mithilfe von Python-Modulen einsetzen

Ein künstliches neuronales Netzwerk selbst zu schreiben ist eine interessante Lernerfahrung, aber praxistauglich ist diese Lösung nicht. Python-Module bieten erheblich schnellere und genauere Implementierungen neuronaler Netzwerke. Im Folgenden werden zwei Beispiele kurz vorgestellt.

8.3.1 Das scikit-learn-Modul für einfache neuronale Netzwerke

In *scikit-learn* gibt es eine Klasse namens `MLPClassifier`, die ein in vielerlei Hinsicht konfigurierbares Feedforward-Netzwerk mit beliebig vielen versteckten Schichten bereitstellt. »MLP« steht für *Multi-Layer Perceptron*. Das Skript in Listing 8.4 setzt diese Klasse für die Klassifizierung der Digits-Datenmenge ein:

```
from sklearn.datasets import load_digits
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split

# Digits-Datenmenge laden
X, y = load_digits(return_X_y=True)

# 80 % Trainings-, 20 % Testdaten
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2
)

# KNN erzeugen und mit Trainingsdaten trainieren
ann = MLPClassifier(
```

```
    hidden_layer_sizes = (100,),
    activation = 'logistic',
    max_iter = 1000
)
ann.fit(X_train, y_train)

# Genauigkeit der Vorhersage für die Testdaten testen
accuracy = ann.score(X_test, y_test)
print(f"Genauigkeit: {accuracy}")
```

Listing 8.4 »neural-network-sklearn.py« wendet das »scikit-learn«-Modul für neuronale Netzwerke auf die Digits-Datenmenge an.

Der Ablauf entspricht weitgehend dem Ablauf, der in Kapitel 7, »Machine Learning«, für andere *scikit-learn*-Module gezeigt wurde. Der `MLPClassifier` wird in diesem Beispiel mit drei benannten Argumenten initialisiert: `hidden_layer_sizes` bestimmt die versteckten Schichten und die Anzahlen ihrer Neuronen. (Beachten Sie bei einer einzelnen Schicht, dass Sie wie im Beispiel ein Komma hinzufügen müssen, damit der Wert nicht als Skalar interpretiert wird.) Das Argument `activation` bestimmt die Aktivierungsfunktion; `'logistic'` ist die Sigmoid-Funktion. Das bereits im Rahmen anderer *scikit-learn*-Klassen erwähnte Argument `max_iter` bestimmt die Anzahl der Epochen.

Die Ausgabe des Skripts sieht zum Beispiel so aus:

```
$ python3 neural-network-sklearn.py
Genauigkeit: 0.9861111111111112
```

Die Genauigkeit ist also höher als beim selbst programmierten Netzwerk, weil die *scikit-learn*-Version einige automatische Optimierungen durchführt.

In den Listings zu diesem Kapitel finden Sie zusätzlich das Skript *neural-network-sklearn-mnist.py*, das den `MLPClassifier` auf die MNIST-Datenmenge anwendet. Es erreicht Genauigkeiten um die 97 % und benutzt drei versteckte Schichten mit 64, 128 und wieder 64 Neuronen, 1.000 Epochen und eine Aktivierungsfunktion namens ReLu (*Rectified Linear Unit*), die wie folgt definiert ist:

$$f(x) = \max(0, x)$$

Negative x-Werte werden mit anderen Worten 0, während positive ihren eigenen Funktionswert bilden. Es hat sich in vielen Experimenten gezeigt, dass die Funktion für die Bilderkennung besser geeignet ist als die Sigmoid-Funktion oder ähnliche Aktivierungsfunktionen.

8.3.2 TensorFlow und Keras

Für den professionellen Einsatz neuronaler Netzwerke ist das Python-Modul *TensorFlow* eine der verbreitetsten Lösungen. Wie der Name schon sagt, kann es nicht nur mit Matrizen, sondern auch mit Tensoren rechnen. Das heißt, dass ein TensorFlow-basiertes Netzwerk in einem Schritt mit zweidimensionalen Schichten (Matrizen) arbeiten kann. Das Modul verfügt über sehr viel unterschiedliche Funktionalität, die flexibel kombiniert werden kann, und bietet unter anderem Aspekte von Deep Learning und Convolutional Neural Networks.

In der Praxis wird TensorFlow meist mit dem Modul *Keras* kombiniert, das eine benutzerfreundlichere API für das KNN-Modul bereitstellt. In den Listings zu diesem Kapitel finden Sie einige ausführlich kommentierte Keras-Anwendungsbeispiele. An dieser Stelle soll ein einzelnes Beispielskript genügen, das die MNIST-Datenmenge mithilfe eines Convolutional Neural Networks kategorisiert.

In Listing 8.5 sehen Sie den Quellcode des Programms:

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers

# Kategorien
categories = 10

# Daten laden
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Feature-Skalierung
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Daten in die richtige Form bringen
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)

# Ausgabedaten konvertieren
y_train = keras.utils.to_categorical(y_train, categories)
y_test = keras.utils.to_categorical(y_test, categories)

model = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),
```

```
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(categories, activation="softmax"),
    ]
)

model.summary()

batch_size = 128
epochs = 15

model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
validation_split=0.1)

score = model.evaluate(x_test, y_test, verbose=0)
print("Genauigkeit:", score[1])
```

Listing 8.5 »neural-network-keras-mnist.py« kategorisiert die MNIST-Datenmenge mithilfe der Module »TensorFlow« und »Keras«.

Die MNIST-Datenmenge ist in Keras eingebaut. Datenumformungen wie das Erzeugen der Matrix mit den Zielwerten funktionieren mit dem Modul auch sehr einfach. Betrachten Sie insbesondere den Code, der das Netzwerk instanziiert:

```
model = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
```



```
        layers.Dense(categories, activation="softmax"),
    ]
)
```

Das KNN verwendet insgesamt acht Schichten mit unterschiedlichen Merkmalen. Die eigentliche Bilderkennung erledigt eine doppelte Abfolge von Schichten der Typen `Conv2D` und `MaxPooling2D`. Convolution-Layer führen eine Faltungsoperation durch, was den Kern der Bildunterscheidung ausmacht. Ihre Aktivierungsfunktion ist im vorliegenden Fall ReLu. Max-Pooling-Layer reduzieren die Datenmenge durch das Übernehmen der Maximalwerte aus kleinen quadratischen Bereichen und brauchen keine spezifische Aktivierungsfunktion. Die restlichen Schichten haben folgende Aufgaben: `Flatten` wandelt die Matrix, aus der die Daten bis zu diesem Punkt bestehen, in einen Vektor um, `Dropout` wählt nach dem Zufallsverfahren nur einen Teil der Daten aus (hier 50 %), und `Dense` verdichtet das Ergebnis so, dass es einen Kategorievektor bildet.

Die Methode `summary()` zeigt die Konfiguration des Netzwerks auf der Konsole an, bevor es durch den Aufruf von `fit()` trainiert wird. Das Training erfolgt nicht über die gesamte Trainingsdatenmenge, sondern über zufällige Beispieldaten, die in jeder Epoche neu gewählten werden. Die Anzahl dieser Beispieldaten wird mit dem benannten Argument `batch_size` gewählt.

Die Vorbereitung dauert relativ lange, lohnt sich aber: die Genauigkeit der Bilderkennung mit dieser Konfiguration beträgt über 99 %.

8.4 Übungsaufgaben

Aufgabe 8.1

Erweitern Sie *neural_network.py* so, dass statt einer versteckten Schicht beliebig viele verwendet werden können. Probieren Sie aus, ob zwei oder mehr versteckte Schichten mit unterschiedlichen Anzahlen von Neuronen ein besseres Ergebnis für die Digits-Datenmenge liefern.

Aufgabe 8.2

Probieren Sie die `MLPClassifier`-Implementierung der Digits-Kategorisierung mit der Aktivierungsfunktion ReLu statt Sigmoid aus. Versuchen Sie es auch mit mehreren versteckten Schichten statt einer, und überprüfen Sie, ob die Ergebnisse genauer werden.