O'REILLY®

Deutsche Ausgabe

Deep Learning

Grundlagen & Implementierung

Neuronale Netze mit Python und PyTorch programmieren

Seth Weidman

Übersetzung von Jørgen W. Lang

Deep Learning – Grundlagen und Implementierung

Neuronale Netze mit Python und PyTorch programmieren

Seth Weidman

Deutsche Übersetzung von Jørgen W. Lang

O'REILLY®

Seth Weidman

Lektorat: Alexandra Follenius
Übersetzung: Jørgen W. Lang
Korrektorat: Sibylle Feldmann, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Karen Montgomery, Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-96009-136-3
PDF 978-3-96010-378-3
ePub 978-3-96010-379-0
mobi 978-3-96010-380-6

1. Auflage

Wieblinger Weg 17
69123 Heidelberg

Authorized German translation of the English edition of *Deep Learning from Scratch*,
ISBN 9781492041412 © 2019 Seth Weidman. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.
O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

PEFC zertifiziert
Das Papier für dieses Buch stammt aus nachhaltig bewirtschafteten Wäldern und kontrollierten Quellen.
PEFC
PEFC/04-31-0810 www.pefc.de

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: kommentar@oreilly.de.

5 4 3 2 1 0

Inhalt

Einführung

Wenn Sie schon einmal versucht haben, etwas über neuronale Netze und Deep Learning zu erfahren, ist Ihnen vermutlich aufgefallen, wie viele verschiedene Quellen es dazu gibt – von Blogposts über MOOCs (*Massive Open Online Courses*, z. B. die von Coursera und Udacity angebotenen) unterschiedlicher Qualität bis hin zu Büchern. Zumindest hatte ich diesen Eindruck, als ich vor einigen Jahren begann, mich mit diesem Thema zu beschäftigen. Trotzdem ist es recht wahrscheinlich, dass Ihnen bei den bisher bekannten Erklärungen neuronaler Netze etwas fehlt. Mir ging es am Anfang genauso: Die verschiedenen Erklärungen wirkten, als versuchten Blinde, Teile eines Elefanten zu beschreiben (*https://oreil.ly/r5YxS*), ohne dass einer von ihnen den ganzen Elefanten beschreibt. Deshalb habe ich dieses Buch geschrieben.

Die vorhandenen Quellen zu neuronalen Netzen lassen sich in zwei Kategorien einteilen. Einige sind konzeptuell und mathematisch und enthalten sowohl die Abbildungen, die man typischerweise in Erklärungen neuronaler Netze findet – mit Kreisen, die durch Pfeile verbunden sind –, als auch ausführliche mathematische Erläuterungen der Vorgänge, damit Sie »die Theorie verstehen«. Das wohl beste Beispiel hierfür ist das sehr gute Buch *Deep Learning* von Ian Goodfellow et al. (MIT Press).

Andere Quellen enthalten dicht gepackte Codeblöcke, die einen mit der Zeit abnehmenden Abweichungswert und damit ein neuronales Netz beim »Lernen« zu zeigen scheinen. So definiert das folgende Beispiel aus der PyTorch-Dokumentation tatsächlich ein einfaches neuronales Netz und trainiert es mit zufällig erzeugten Daten:

```
# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# Randomly initialize weights
```

```
w1 = torch.randn(D_in, H, device=device, dtype=dtype)
w2 = torch.randn(H, D_out, device=device, dtype=dtype)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum().item()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # Update weights using gradient descent
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Solche Erklärungen geben natürlich kaum einen Einblick in das, was wirklich passiert – also die zugrunde liegenden mathematischen Prinzipien, die einzelnen Bestandteile des hier gezeigten neuronalen Netzes, ihre Zusammenarbeit und so weiter.[1]

Was *wäre* also eine gute Erklärung der Bestandteile neuronaler Netze? Um eine Antwort zu finden, hilft es, sich anzusehen, wie andere Konzepte der Computerwissenschaft erklärt werden: Wenn Sie etwas über Sortieralgorithmen lernen möchten, gibt es beispielsweise Bücher mit folgendem Inhalt:

- Eine Erklärung des Algorithmus in verständlicher Sprache.
- Eine visuelle Darstellung der Funktionsweise des Algorithmus, so wie Sie es in einem Coding-Interview auf einem Whiteboard notieren würden.
- Eine mathematische Erklärung der Funktionsweise des Algorithmus.[2]
- Pseudocode, der den Algorithmus implementiert.

Diese Elemente findet man in Erklärungen neuronaler Netze selten bis nie gemeinsam, obwohl es mir offensichtlich erscheint, dass eine vollständige Erklärung neu-

1 Aus Fairnessgründen müssen wir sagen, dass dieses Beispiel für die PyTorch-Bibliothek eigentlich für diejenigen gedacht war, die neuronale Netze bereits verstehen, aber nicht als lehrreiche Anleitung. Tatsächlich folgen aber viele Tutorials diesem Stil und zeigen nur den Code mit ein paar kurzen Erläuterungen.

2 Im Fall von Sortieralgorithmen geht es speziell darum, warum der Algorithmus mit einer korrekt sortierten Liste beendet wird.

ronaler Netze auf diese Weise vorgenommen werden sollte. Das Buch ist also der Versuch, diese Lücke zu füllen.

Um neuronale Netze zu verstehen, braucht man mehr als ein Gedankenmodell

Ich bin kein Wissenschaftler und habe auch keinen Doktortitel. Aber ich habe Data Science unterrichtet: Ich habe bei einigen »Data Science Bootcamps« für ein Unternehmen namens Metis unterrichtet und bin ein Jahr lang für Metis um die Welt gereist. Während dieser Zeit führte ich ein- bis fünftätige Workshops für Unternehmen verschiedener Branchen durch, in denen ich den Mitarbeitern maschinelles Lernen und die Grundprinzipien der Softwareentwicklung beibrachte. Ich habe das Unterrichten immer geliebt. Dabei war ich von der Frage fasziniert, wie man technische Konzepte am besten vermitteln kann.

In jüngerer Zeit lag mein Fokus hauptsächlich auf Konzepten des maschinellen Lernens und der Statistik. Bei neuronalen Netzen scheint mir die größte Herausforderung die Vermittlung des richtigen »mentalen Modells« dessen zu sein, was ein neuronales Netz ausmacht. Das gilt besonders, weil man für das Verständnis neuronaler Netze nicht nur ein, sondern *mehrere* Gedankenmodelle braucht, die jeweils verschiedene (aber wichtige) Aspekte der Funktionsweise beleuchten. Zur Illustration sehen Sie unten vier korrekte Antworten auf die Frage »Was ist ein neuronales Netz?«:

- Ein neuronales Netz ist eine mathematische Funktion, die Eingaben übernimmt und Ausgaben erzeugt.
- Ein neuronales Netz ist ein Rechengraph, durch den mehrdimensionale Arrays geleitet werden.
- Ein neuronales Netz besteht aus »Schichten« (»Layers«[3]), die man sich jeweils als eine bestimmte Anzahl von »Neuronen« vorstellen kann.
- Ein neuronales Netz ist ein universeller Funktionsapproximator, der theoretisch die Lösung eines beliebigen Problems des überwachten Lernens darstellt.

Viele von Ihnen kennen diese Definitionen (oder zumindest ein paar davon) und haben bereits ein Grundverständnis ihrer Bedeutung und der Implikationen für die Funktionsweise neuronaler Netze. Für ein umfassendes Verständnis müssen wir jedoch *alle* diese Definitionen verstehen und ihre Verbindungen erkennen können. Worin besteht die gedankliche Verbindung zwischen einem neuronalen Netz als Rechengraph und dem Konzept der »Schichten«?

3 Anm. d. Ü.: Wir werden die Begriffe »Schicht« und »Layer« in diesem Buch synonym verwenden. In manchen Fällen bezeichnet »Layer« gleichzeitig die Schicht selbst und den für ihre Implementierung nötigen Code.

Um diese Zusammenhänge zu verdeutlichen, werden wir die Konzepte von Grund auf in Python implementieren und miteinander verknüpfen. Sie erstellen funktionierende neuronale Netze, die Sie direkt auf Ihrem Laptop trainieren können. Auch wenn wir einen Großteil unserer Zeit mit Implementierungsdetails verbringen werden, geht es bei der Implementierung dieser Modelle in Python darum, die *Konzepte zu verfestigen* und unser *Verständnis davon zu präzisieren*. Was wir hier *nicht* wollen, ist, möglichst knappen Code schreiben oder eine besonders leistungsstarke neuronale Netzwerkbibliothek zu programmieren.

Ich möchte, dass Sie nach dem Lesen dieses Buchs ein solides Verständnis aller dieser mentalen Modelle haben (und was das für die *Implementierung* neuronaler Netze bedeutet). Das soll Ihnen das Lernen verwandter Konzepte und die Durchführung weiterer Projekte in diesem Bereich erleichtern.

Kapitelstruktur

Die ersten drei Kapitel sind die wichtigsten und könnten jeweils selbst ein eigenes Buch füllen.

1. In Kapitel 1, *Grundbausteine*, zeige ich Ihnen, wie mathematische Funktionen zu einer Folge von Operationen verkettet werden können, um einen Rechengraphen zu erstellen. Außerdem erfahren Sie, wie wir anhand dieser Darstellung die Ableitungen der Ausgaben dieser Funktionen bezogen auf die Eingaben berechnen können. Hierfür verwenden wir die Kettenregel aus der Differenzialrechnung. Am Ende des Kapitels stelle ich eine sehr wichtige Operation vor: die Matrizenmultiplikation. Ich zeige, wie sie als mathematische Funktion dargestellt werden kann, während es trotzdem möglich ist, die für das Deep Learning nötigen Ableitungen zu berechnen.
2. In Kapitel 2, *Erste Modelle*, wenden wir die Bausteine aus Kapitel 1 direkt an, um Modelle zu erstellen und zu trainieren, die ein Problem aus der richtigen Welt lösen. Genauer gesagt, verwenden wir sie, um Modelle für lineare Regression und neuronale Netze zu entwickeln, mit denen anhand echter Daten Hauspreise vorhergesagt werden können. Ich zeige Ihnen, dass neuronale Netze leistungsfähiger sind als die lineare Regression, und gebe Ihnen einige Anhaltspunkte für den Grund. Der »First-Principles-Ansatz« (der grundbegriffbasierte Ansatz) für die Erstellung der Modelle in diesem Kapitel soll Ihnen einen guten Eindruck von der Funktionsweise neuronaler Netze vermitteln. Er zeigt aber auch die begrenzten Fähigkeiten eines schrittweisen und nur auf Grundbegriffen basierenden Ansatzes. Dies ist die Motivation für Kapitel 3.
3. In Kapitel 3, *Deep Learning von Grund auf*, verwenden wir die Bausteine aus dem grundbegriffbasierten Ansatz der ersten beiden Kapitel, um allgemeinere Komponenten zu erstellen, aus denen alle Deep-Learning-Modelle bestehen: `Layer` (Schicht), `Model` (Modell), `Optimizer` und so weiter. Wir beenden dieses Kapitel, indem wir ein Deep-Learning-Modell von Grund auf mit dem Daten-

satz aus Kapitel 2 trainieren und zeigen, dass es leistungsfähiger ist als unser einfaches neuronales Netz.

4. Es gibt einige theoretische Garantien dafür, dass ein neuronales Netz einer bestimmten Architektur tatsächlich eine gute Lösung für einen gegebenen Datensatz findet, wenn es mit den in diesem Buch verwendeten Standardtrainingsmethoden trainiert wird. In Kapitel 4, *Techniken zur Verbesserung des Trainings*, behandeln wir die häufigsten »Trainingstricks« zur Erhöhung der Wahrscheinlichkeit, dass ein neuronales Netz eine gute Lösung findet. Nach Möglichkeit geben wir Ihnen mathematische Hinweise darauf, warum dies funktioniert.
5. In Kapitel 5, *CNNs – Faltungsbasierte neuronale Netze*, behandle ich die Grundideen der *Convolutional Neural Networks* (»faltende« neuronale Netze, CNNs), die auf die Interpretation von Bilddaten spezialisiert sind. Hierzu gibt es bereits eine Vielzahl von Erklärungen. Daher konzentriere ich mich auf die absolut wichtigen Grundlagen von CNNs und ihre Unterschiede zu regulären neuronalen Netzen. Insbesondere gehe ich darauf ein, wie es bei CNNs dazu kommt, dass jede Neuronenschicht in »Feature Maps« strukturiert ist. Ich zeige, wie diese Schichten über Faltungsfilter (Convolutional Filters) verbunden sind. Wie die regulären Schichten eines neuronalen Netzes werden wir auch die Faltungsschichten (Convolutional Layers) von Grund auf programmieren, um ihre Funktionsweise besser zu verstehen.
6. In den ersten fünf Kapiteln haben wir eine Miniaturbibliothek für neuronale Netze erstellt, die neurale Netze als eine Reihe von Schichten (`Layer`) definiert, die ihrerseits aus einer Reihe von Operationen bestehen. Diese leiten die Eingaben weiter und schicken Gradienten zurück. In der Praxis werden neuronale Netze jedoch meist anders implementiert: Sie verwenden die sogenannte *automatische Differenzierung* (Automatic Differentiation). Zu Beginn von Kapitel 6, *RNNs – Rekurrente neuronale Netze*, gebe ich Ihnen eine kurze Einführung in das automatische Differenzieren und verwende diese Technik, um das Hauptthema dieses Kapitels vorzustellen: Rekurrente neuronale Netze (RNNs). Diese Architektur wird typischerweise verwendet, um Daten zu verstehen, deren Datenpunkte als Folgen (sequenziell) auftreten, zum Beispiel Zeitreihen oder natürliche Sprachdaten. Ich erkläre die Funktionsweise »normaler« RNNs sowie zweier Varianten: *GRUs* und *LSTMs* (die wir natürlich ebenfalls von Grund auf implementieren). Dabei zeige ich, welche Dinge alle RNN-Varianten *gemeinsam* haben und welche *Unterschiede* es gibt.
7. Zum Abschluss zeige ich Ihnen in Kapitel 7, *PyTorch*, wie die in den Kapiteln 1 bis 6 behandelten Dinge mithilfe der hochperformanten Open-Source-Bibliothek für neuronale Netze PyTorch implementiert werden können. Um mehr über neuronale Netze zu erfahren, ist es unabdingbar, ein Framework zu lernen. Ohne ein solides Verständnis der Funktionsweise neuronaler Netze werden Sie auf lange Sicht aber auch mit dem besten Framework nicht weiter-

kommen. Ich möchte Ihnen mit diesem Buch helfen, hochperformante neuronale Netze zu schreiben (indem ich Ihnen PyTorch beibringe), und Sie gleichzeitig auf das langfristige Lernen und das erfolgreiche Anwenden des Wissen vorbereiten (indem ich Ihnen vorher die Grundlagen vermittle). Wir beschließen das Buch mit einem kurzen Blick auf die Verwendung neuronaler Netze für das unüberwachte Lernen.

Mein Ziel war es, das Buch zu schreiben, das mir gefehlt hat, als ich vor einigen Jahren damit begann, mich mit diesem Thema zu befassen. Ich hoffe, dieses Buch hilft Ihnen. Auf geht's!

In diesem Buch verwendete Konventionen

Die folgenden typografischen Konventionen werden in diesem Buch verwendet:

Kursiv
: Kennzeichnet neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateiendungen.

`Konstante Zeichenbreite`
: Wird für Programmlistings und für Programmelemente in Textabschnitten wie Namen von Variablen und Funktionen, Datenbanken, Datentypen und Umgebungsvariablen sowie für Anweisungen und Schlüsselwörter verwendet.

`Konstante Zeichenbreite, fett`
: Kennzeichnet Befehle oder anderen Text, den der Nutzer wörtlich eingeben sollte.

`Konstante Zeichenbreite, kursiv`
: Kennzeichnet Text, den der Nutzer je nach Kontext durch entsprechende Werte ersetzen sollte.

Der Satz des Pythagoras lautet: $a^2 + b^2 = c^2$.

Dieses Symbol steht für einen allgemeinen Hinweis.

Codebeispiele

Weiterführendes Material (Codebeispiele, Übungen etc.) steht im GitHub-Repository zu diesem Buch (*https://oreil.ly/deep-learning-github*) zum Download bereit.

Dieses Buch dient dazu, Ihnen beim Erledigen Ihrer Arbeit zu helfen. Im Allgemeinen dürfen Sie die Codebeispiele aus diesem Buch in Ihren eigenen Programmen und der dazugehörigen Dokumentation verwenden. Sie müssen uns dazu nicht um Erlaubnis fragen, solange Sie nicht einen beträchtlichen Teil des Codes reproduzie-

ren. Beispielsweise benötigen Sie keine Erlaubnis, um ein Programm zu schreiben, in dem mehrere Codefragmente aus diesem Buch vorkommen. Wollen Sie dagegen eine CD-ROM mit Beispielen aus Büchern von O'Reilly verkaufen oder verteilen, benötigen Sie eine Erlaubnis. Eine Frage zu beantworten, indem Sie aus diesem Buch zitieren und ein Codebeispiel wiedergeben, benötigt keine Erlaubnis. Eine beträchtliche Menge Beispielcode aus diesem Buch in die Dokumentation Ihres Produkts aufzunehmen, bedarf hingegen einer Erlaubnis.

Wir freuen uns über Zitate, verlangen diese aber nicht. Ein Zitat enthält Titel, Autor, Verlag und ISBN, beispielsweise: *Deep Learning – Grundlagen und Implementierung* von Seth Weidman (O'Reilly). Copyright 2019 Seth Weidman, ISBN 978-3-96009-136-3.«

Wenn Sie glauben, dass Ihre Verwendung von Codebeispielen über die übliche Nutzung hinausgeht oder außerhalb der oben vorgestellten Nutzungsbedingungen liegt, kontaktieren Sie uns bitte unter *kommentar@oreilly.de*.

Danksagungen des Autors

Ich möchte mich bei meiner Lektorin Melissa Potter und dem Team bei O'Reilly bedanken für ihre unzähligen Rückmeldungen und Antworten auf meine vielen Fragen während des gesamten Prozesses.

Ein besonderes Dankeschön geht an die vielen Menschen, deren Arbeit, die technischen Konzepte des maschinellen Lernens einem breiteren Publikum zugänglich zu machen, mich direkt beeinflusst habt. Ich hatte das Glück, einige dieser Menschen persönlich kennenlernen zu dürfen: In zufälliger Reihenfolge sind dies Brandon Rohrer, Joel Grus, Jeremy Watt und Andrew Trask.

Ich möchte mich außerdem bei meinem Chef bei Metis bedanken und meinem Director bei Facebook. Beide haben mir sehr dabei geholfen, Zeit zu finden für die Arbeit an diesem Projekt.

Würdigung und besonderer Dank gebührt Mat Leonhard, der für kurze Zeit mein Koautor war, bevor wir uns entschieden, eigene Wege zu gehen. Mat half mir, den Code für die Mini-Bibliothek zu diesem Buch zu organisieren – `lincoln` –, und gab mir Feedback zu den ersten beiden Kapiteln, wobei er für große Abschnitte dieser Kapitel seine eigenen Versionen schrieb.

Schließlich möchte ich mich noch bei meinen Freunden Eva und John bedanken, die mich inspiriert und dazu motiviert haben, ins kalte Wasser zu springen und tatsächlich mit dem Schreiben zu beginnen. Ich bedanke mich außerdem bei meinen vielen Freunden in San Francisco, die es ausgehalten haben, dass ich ständig Sorgen und Vorbehalte bezüglich dieses Buchs hatte und für viele Monate kaum für gemeinsame Unternehmungen verfügbar war. Sie haben mich unerschütterlich unterstützt, als ich sie brauchte.

KAPITEL 1
Grundbausteine

> Merken Sie sich diese Formeln nicht. Wenn Sie die Konzepte verstehen, können Sie Ihre eigene Schreibweise erfinden.
>
> *– John Cochrane, Investments Notes 2006 (https://oreil.ly/33CVXjg)*

Dieses Kapitel erklärt Ihnen einige grundsätzliche Gedankenmodelle, die für das Verständnis neuronaler Netze notwendig sind. Insbesondere kümmern wir uns um *verschachtelte mathematische Funktionen und ihre Ableitungen*. Wir beginnen mit möglichst einfachen Bausteinen und arbeiten uns von dort aus vor, um zu zeigen, dass komplexe Funktionen aus einer »Kette« von Einzelfunktionen erstellt werden können. Selbst wenn eine dieser Teilfunktionen eine Matrizenmultiplikation mit mehreren Eingaben ist, können wir die Ableitung ihrer Ausgaben bezogen auf die Eingaben berechnen. Das Verständnis der Funktionsweise dieses Prozesses ist essenziell, um neuronale Netze als Ganzes zu verstehen, die wir in Kapitel 2 genauer betrachten werden.

Mit dem wachsenden Verständnis dieser Grundbausteine neuronaler Netze werden wir jedes neu eingeführte Konzept systematisch auf drei Arten beschreiben:

- Mathematisch – in der Form einer oder mehrerer Gleichungen.
- Als Code – mit so wenig zusätzlicher Syntax wie möglich (was Python zu einer idealen Wahl macht).
- Als erklärendes Diagramm – wie Sie es beispielsweise während eines »Coding Interviews« auf ein Whiteboard zeichnen würden.

Wie Sie in diesem Kapitel sehen werden, liegt eine der Herausforderungen zum Verständnis neuronaler Netze in der Anwendung verschiedener Gedankenmodelle. Jede der drei Sichtweisen ist für sich genommen nicht ausreichend. Nur gemeinsam bekommen wir ein vollständiges Bild davon, warum und wie verschachtelte mathematische Funktionen auf die ihnen eigene Weise funktionieren. Grundsätzlich habe ich hierzu eine ziemlich klare Meinung: Jeder Versuch, die Grundbausteine neuronaler Netze nicht aus *allen drei Perspektiven* zu erklären, ist meiner Meinung nach unvollständig.

Nachdem das klar ist, können wir unsere ersten Schritte unternehmen. Wir beginnen mit besonders einfachen Grundbausteinen, um zu zeigen, wie wir verschiedene Konzepte aus diesen drei Perspektiven verstehen können. Unser erster Baustein ist ein einfaches, aber äußerst wichtiges Konzept: die mathematische Funktion.

Funktionen

Was ist eine *Funktion*, und wie wird sie beschrieben? Bei neuronalen Netzen gibt es hierfür mehrere Wege. Keiner dieser Wege ist für sich genommen vollständig. Daher will ich gar nicht erst versuchen, Ihnen das alles in einem einzigen knackigen Satz zu präsentieren. Stattdessen gehen wir die drei Gedankenmodelle (Mathematik, Code, Diagramm) nacheinander durch. Sie werden die Rolle der im Vorwort beschriebenen »blinden Menschen« einnehmen, die jeweils einen Teil des Elefanten beschreiben.

Mathematik

Hier sehen Sie zwei Beispiele für Funktionen in mathematischer Schreibweise:

- $f_1(x) = x^2$
- $f_2(x) = max(x, 0)$

In dieser Schreibweise haben wir zwei Funktionen namens f_1 und f_2. Sie übernehmen eine Zahl x als Eingabe und wandeln diese in x^2 (im ersten Fall), bzw. $max(x, 0)$ (im zweiten Fall) um.

Diagramme

Auch grafisch können Funktionen dargestellt werden. Dafür führt man folgende Schritte aus:

1. Zeichnen Sie einen x-y-Graphen (wobei x für die horizontale Achse steht und y für die vertikale Achse).
2. Plotten Sie eine Anzahl von Punkten. Dabei stehen die x-Koordinaten der Punkte (üblicherweise in regelmäßigen Abständen) für die Eingaben der Funktion über einen bestimmten Bereich; die y-Koordinaten stehen für die Ausgaben der Funktionen über den gegebenen Bereich.
3. Verbinden Sie die geplotteten Punkte miteinander.

Dieses Verfahren wurde zuerst von dem französischen Philosophen René Descartes[1] angewandt und ist in vielen Bereichen der Mathematik sehr nützlich – besonders in der Differenzialrechnung. Abbildung 1-1 zeigt die grafische Darstellung der zwei Funktionen.

1 Daher kommt übrigens auch der Name »kartesisches« Koordinatensystem.

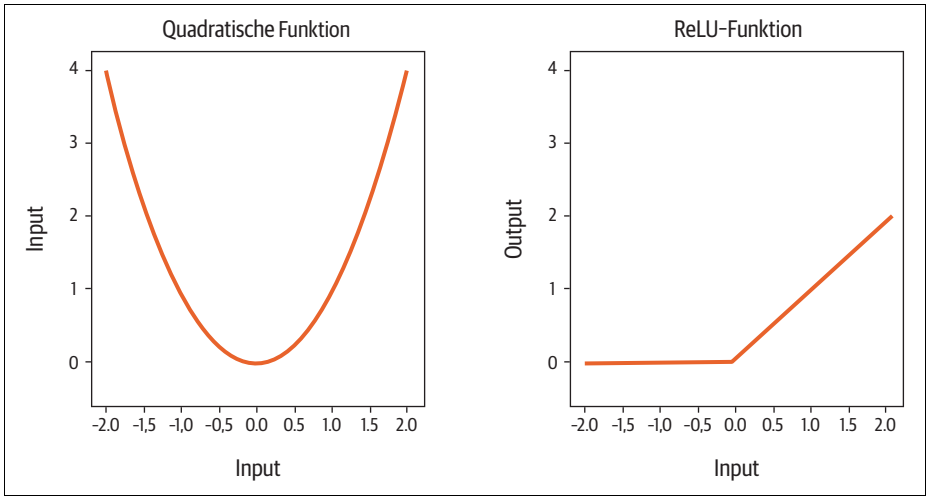


Abbildung 1-1: Zwei stetige, größtenteils differenzierbare Funktionen im kartesischen Koordinatensystem

Eine weitere Art, Funktionen darzustellen, sehen Sie in der folgenden Abbildung. Zum Lernen der Differenzialrechnung ist sie nicht so gut geeignet, umso mehr hilft sie aber bei der Visualisierung von Deep-Learning-Modellen. Wir können uns die Funktionen als Kästen vorstellen, die Zahlen als Eingabe übernehmen und Zahlen als Ausgabe erzeugen – wie kleine Fabriken, die jeweils eigene Regeln für die Behandlung der Eingaben besitzen. Abbildung 1-2 zeigt die Beschreibung der Funktionen als allgemeine Regeln und deren Anwendung auf die jeweiligen Eingaben.

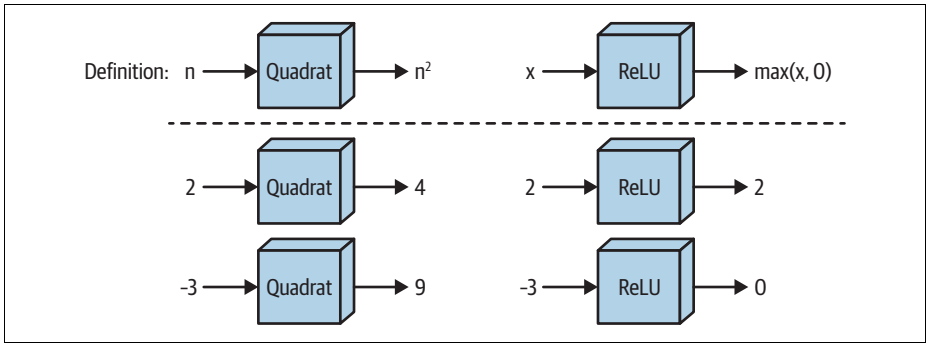


Abbildung 1-2: Eine weitere Darstellungsweise von Funktionen

Code

Außerdem können wir diese Funktionen als Programmiercode ausdrücken. Vorher sollten wir aber noch etwas über *NumPy* sagen, die Python-Bibliothek, die wir zum Schreiben unserer Funktionen verwenden.

Codewarnung 1: NumPy

NumPy ist eine beliebte Python-Bibliothek für schnelle numerische Berechnungen, die intern größtenteils in C geschrieben ist. Hierbei werden die Daten, mit denen wir in neuronalen Netzen arbeiten, grundsätzlich in einem *Array* mit einer bis vier Dimensionen gespeichert. Anhand von NumPy-`ndarray`-Objekten können wir intuitiv und schnell Operationen an diesen Arrays durchführen. Hier ein einfaches Beispiel: Speichern wir unsere Daten als (möglicherweise verschachtelte) Python-Listen, könnten wir sie mit normalem Code nicht einfach addieren oder multiplizieren – mit `ndarrays` dagegen schon:

```
print("Python list operations:")
a = [1,2,3]
b = [4,5,6]
print("a+b:", a+b)
try:
    print(a*b)
except TypeError:
    print("a*b has no meaning for Python lists")
print()
print("numpy array operations:")
a = np.array([1,2,3])
b = np.array([4,5,6])
print("a+b:", a+b)
print("a*b:", a*b)
```

```
Python list operations:
a+b: [1, 2, 3, 4, 5, 6]
a*b has no meaning for Python lists

numpy array operations:
a+b: [5 7 9]
a*b: [ 4 10 18]
```

`ndarrays` besitzen viele Merkmale, die man von einem `n`-dimensionalen Array üblicherweise erwartet: Jedes `ndarray` hat `n` Achsen (oder *Dimensionen*), deren Indizierung bei 0 beginnt. Die erste Achse hat also den Index `0`, die zweite `1` und so weiter. Da wir häufig mit zweidimensionalen Arrays arbeiten, können wir uns die Achse 0 (`axis = 0`) als *Zeilen* und die Achse 1 (`axis = 1`) als *Spalten* vorstellen, wie in Abbildung 1-3 gezeigt:

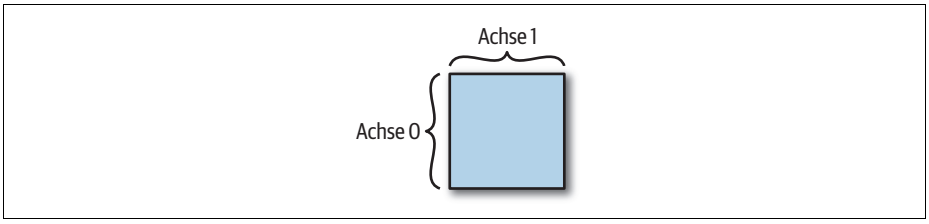


Abbildung 1-3: Ein zweidimensionales NumPy-Array, bei dem Achse 0 als Zeilen und Achse 1 als Spalten betrachtet werden

NumPy-ndarrays unterstützen die intuitive Verwendung von Funktionen entlang dieser Achsen. Eine Berechnung der Summe entlang der Achse 0 (die *Zeilen* des 2-D-Arrays) »verflacht« (collapse) das Array entlang dieser Achse. Es wird ein Array mit einer Dimension weniger als das Ausgangsarray zurückgegeben. Bei einem 2-D-Array entspricht dies der Berechnung der Summen der einzelnen Spalten:

```
print('a:')
print(a)
print('a.sum(axis=0):', a.sum(axis=0))
print('a.sum(axis=1):', a.sum(axis=1))
```

```
a:
[[1 2]
 [3 4]]
a.sum(axis=0): [4 6]
a.sum(axis=1): [3 7]
```

Außerdem unterstützen NumPy-ndarrays das Addieren eines eindimensionalen Arrays zur letzten Achse. Für ein zweidimensionales Array `a` mit `Z` Zeilen und `S` Spalten können wir also ein eindimensionales Array `b` der Länge `C` addieren. NumPy führt die Addition auf leicht nachvollziehbare Weise aus, indem es die Elemente jeder Zeile von `a` hinzufügt: [2]

```
a = np.array([[1,2,3],
              [4,5,6]])

b = np.array([10,20,30])

print("a+b:\n", a+b)
```

```
a+b:
[[11 22 33]
 [14 25 36]]
```

Codewarnung 2: Funktionen mit Type Checking

Wenn wir in diesem Buch Code schreiben, möchte ich die behandelten Konzepte so präzise und klar wie möglich vermitteln. Das wird im Verlauf des Buchs schwieriger, wenn wir Funktionen mit vielen Argumenten als Teil komplexer Klassen schreiben. Daher verwenden wir durchgehend Funktionen mit Typsignaturen. In Kapitel 3 initialisieren wir unsere neuronalen Netze beispielsweise wie folgt:

```
def __init__(self,
             layers: List[Layer],
             loss: Loss,
             learning_rate: float = 0.01) -> None:
```

2 Dadurch können wir unsere Matrizenmultiplikation später auf einfache Weise mit einem Bias (Verschiebung, mehr dazu in Kapitel 2 im Abschnitt »Lineare Regression« auf Seite 45) versehen.

Diese Typsignatur vermittelt bereits, wofür die Klasse verwendet wird. Sehen Sie sich im Gegensatz dazu die folgende Typsignatur an, die wir ebenfalls benutzen *könnten*:

```
def operation(x1, x2):
```

Diese Typsignatur gibt keinen Hinweis darauf, welche Operationen hier tatsächlich ausgeführt werden. Das lässt sich nur durch die Ausgabe der einzelnen Objekttypen herausfinden. Wir müssten versuchen, anhand der Namen `x1` und `x2` zu *erraten*, was in dieser Funktion passiert. Daher ist es besser, wir nutzeneine Typsignatur wie diese:

```
def operation(x1: ndarray, x2: ndarray) -> ndarray:
```

Hier wird sofort klar, was passiert: Diese Funktion übernimmt zwei `ndarrays`, kombiniert diese offenbar und gibt das Ergebnis dieser Kombination aus. Weil typisierte Funktionen deutlich klarer sind, werden wir sie durchgehend in diesem Buch verwenden.

Grundsätzliche Funktionen in NumPy

Mit diesen Voraussetzungen im Hinterkopf können wir die zuvor definierten Funktionen in NumPy schreiben:

```
def square(x: ndarray) -> ndarray:
    '''
    Jedes Element des Eingabearrays quadrieren.
    '''
    return np.power(x, 2)

def leaky_relu(x: ndarray) -> ndarray:
    '''
    Die "Leaky ReLU"-Funktion auf jedes Element des Arrays anwenden.
    '''
    return np.maximum(0.2 * x, x)
```

In NumPy können viele auf `ndarrays` angewandte Funktionen entweder als `np.`*`funktionsname`*`(ndarray)` oder als `ndarray.`*`funktionsname`* geschrieben werden. Die oben gezeigte `relu`-Funktion könnte beispielsweise als `x.clip(min=0)` geschrieben werden. Um möglichst konsistent zu bleiben, verwenden wir nach Möglichkeit die Schreibweise `np.funktionsname(ndarray)`. Spezialtricks und Abkürzungen wie die Verwendung von *`ndarray`*`.T` für die Transposition eines zweidimensionalen Arrays werden wir vermeiden. Stattdessen schreiben wir – ausführlicher, aber auch eindeutiger – `np.transpose(`*`ndarray`*`, (1, 0))`.

Wenn Sie sich darauf einlassen können, dass Mathematik, ein Diagramm und Code dasselbe Konzept beschreiben können, sind Sie bereits auf einem guten Weg, die Art flexiblen Denkens zu entwickeln, die für das wahre Verständnis des Deep Learning gebraucht wird.

Ableitungen

Neben Funktionen sind *Ableitungen* ebenfalls ein äußerst wichtiges Konzept für das Verständnis von Deep Learning, das viele von Ihnen vermutlich schon kennen. Auch Ableitungen können auf verschiedene Weise dargestellt werden. Allgemein gesagt, kann die Ableitung einer Funktion an einem bestimmten Punkt als »Änderungsrate« der Ausgaben dieser Funktion, bezogen auf diesen Punkt, betrachtet werden. Zum besseren Verständnis schauen wir auch die Ableitungen aus den bekannten drei Blickwinkeln an.

Mathematik

Zuerst werden wir uns dem Konzept mathematisch nähern: Wir wollen herausfinden, wie stark sich die Änderung eines Eingabewerts a auf den Ausgabewert f auswirkt. Diese Änderungsrate können wir als Grenzwert beschreiben:

$$\frac{df}{du}(a) = \lim_{\Delta \to 0} \frac{f(a+\Delta) - f(a-\Delta)}{2 \times \Delta}$$

Diesem Grenzwert kann sich numerisch genähert werden, indem wir für Δ einen sehr kleinen Wert verwenden, z. B. 0,001. Dann können wir die Ableitung wie folgt berechnen:

$$\frac{df}{du}(a) = \frac{f(a+0.001) - f(a-0.001)}{0.002}$$

Das ist zwar akkurat, gibt uns aber noch kein umfassendes Gedankenmodell der Ableitungen. Sehen wir sie uns also zusätzlich als Diagramm an.

Diagramme

Zuerst auf die bekannte Art: Wenn wir eine Tangente für die kartesische Darstellung der Funktion f zeichnen, ist die Ableitung von f an Punkt a einfach die *Steigung* dieser Linie bei a. Wie in der mathematischen Beschreibung gibt es auch hier zwei Möglichkeiten, die Steigung dieser Linie zu berechnen. Im ersten Fall benutzt man die Differenzialrechnung, um den Grenzwert zu berechnen. Im zweiten Fall nimmt man einfach die Steigung der Linie am Schnittpunkt von f und a - 0,001 und a + 0,001. Die zweite Methode sehen Sie in Abbildung 1-4. Wenn Sie Differenzialrechnung gelernt haben, sollte Ihnen das bekannt vorkommen.

Wie gesagt, wir können uns Funktionen auch als Minifabriken vorstellen, bei denen die Ein- und Ausgaben durch eine Linie miteinander verbunden sind. Die Ableitung entspricht also der Antwort auf die Frage: Um welchen Faktor ändert sich die Ausgabe basierend auf den »Regeln« der Fabrik, wenn wir die Eingabe a um einen kleinen Wert anheben oder absenken? Die Darstellung sehen Sie in Abbildung 1-5.

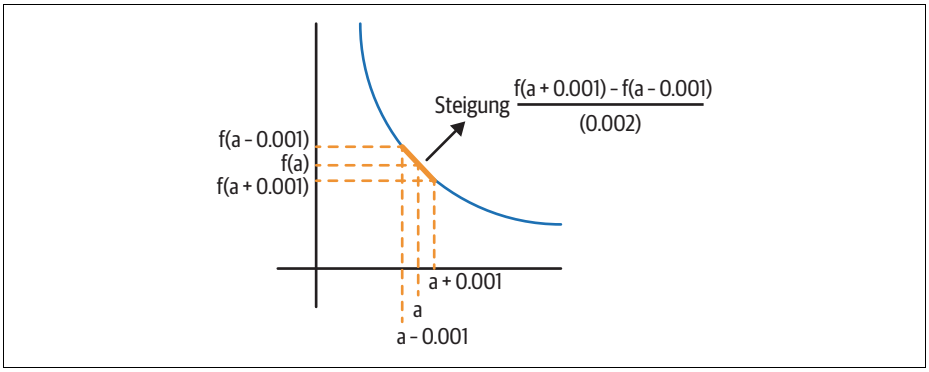


Abbildung 1-4: Ableitungen als Steigungen

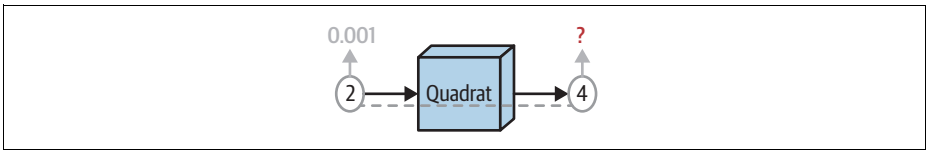


Abbildung 1-5: Eine weitere Art, Ableitungen visuell darzustellen

Wie sich zeigt, ist die zweite Darstellung für das Verständnis von Deep Learning wichtiger als die erste.

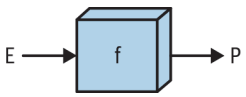
Code

Abschließend können wir die Näherung der gerade gesehenen Ableitung programmieren:

```
from typing import Callable

def deriv(func: Callable[[ndarray], ndarray],
          input_: ndarray,
          delta: float = 0.001) -> ndarray:
    '''
    Ermittelt die Ableitung einer Funktion "func" für jedes Element
    im Array "input_".
    '''
    return (func(input_ + delta) - func(input_ - delta)) / (2 * delta)
```

Manchmal sagen wir, etwas sei eine Funktion von etwas anderem, zum Beispiel »*P* ist eine Funktion von *E*« (die Buchstaben sind absichtlich zufällig gewählt). Damit meinen wir: Es gibt eine Funktion *f*, etwa *f(E)* = *P*, oder auch: Die Funktion *f* übernimmt *E* Objekte und erzeugt *P* Objekte. Oder wir können sagen: »*P ist definiert als* das Ergebnis der Anwendung von *f* auf *E*.«



Das könnten wir folgendermaßen in Code ausdrücken:

```
def f(input_: ndarray) -> ndarray:
    # eine oder mehrere Operationen
    return output

P = f(E)
```

Verschachtelte Funktionen

Und damit kommen wir zu einem Thema, das ebenfalls für das Verständnis neuronaler Netze unabdingbar ist: *verschachtelte* (nested) *Funktionen*. Was ist hier mit »verschachtelt« gemeint? Angenommen, wir hätten zwei Funktionen, f_1 und f_2, dann dient die Ausgabe von f_1 als Eingabe von f_2, wodurch beide Funktionen miteinander »verkettet« oder ineinander »verschachtelt« werden.

Diagramm

Am einfachsten lassen sich verschachtelte Funktionen wieder als Minifabriken oder als »Kastendiagramm« (die zweite Darstellungsform aus dem Abschnitt »Funktionen« auf Seite 2) ausdrücken.

Wie in Abbildung 1-6 gezeigt, erhält die erste Funktion eine Eingabe x. Diese wird (innerhalb von f_1) umgewandelt wieder ausgegeben. Danach wird diese Ausgabe als Eingabe für f_2 verwendet, erneut umgewandelt, und wir erhalten unsere endgültige Ausgabe y.

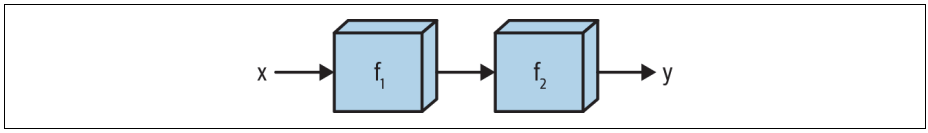


Abbildung 1-6: Verschachtelte Funktionen, natürlich dargestellt

Mathematik

Wir sollten auch die weniger intuitive mathematische Darstellung nicht vergessen:

$$f_2(f_1(x)) = y$$

Das ist weniger intuitiv, weil verschachtelte Funktionen »von außen nach innen« gelesen werden. Tatsächlich laufen die Operationen aber »von innen nach außen« ab. So wird $f_2(f_1(x)) = y$ beispielsweise als »f2 von f1 von x« gelesen. Das heißt: »Wende zuerst f_1 auf x an, dann wende f_2 auf das Ergebnis der vorherigen Operation (der Anwendung von f_1 auf x) an.«

Code

Auch dies wollen wir der Verständlichkeit halber als Code ausdrücken. Zuerst definieren wir die Datentypen für verschachtelte Funktionen:

```
from typing import List

# Eine Funktion übernimmt ein ndarray als Argument und erzeugt ein ndarray.
Array_Function = Callable[[ndarray], ndarray]

# Eine Kette (Chain) ist eine Liste mit Funktionen.
Chain = List[Array_Function]
```

Danach definieren wir, wie die Daten die Kette durchlaufen, zuerst eine Kette mit zwei Funktionen:

```
def chain_length_2(chain: Chain,
                   a: ndarray) -> ndarray:
    '''
    Wertet zwei Funktionen nacheinander als "Kette" aus.
    '''
    assert len(chain) == 2, \
    "Length of input 'chain' should be 2"

    f1 = chain[0]
    f2 = chain[1]

    return f2(f1(x))
```

Noch ein Diagramm

Stellen wir die verschachtelte Funktion als Kastendiagramm dar, bilden die Einzelfunktionen eine zusammengesetzte Operation oder *Kompositfunktion*[3], die wir einfach als $f_1 f_2$ ausdrücken können, wie in Abbildung 1-7 gezeigt.

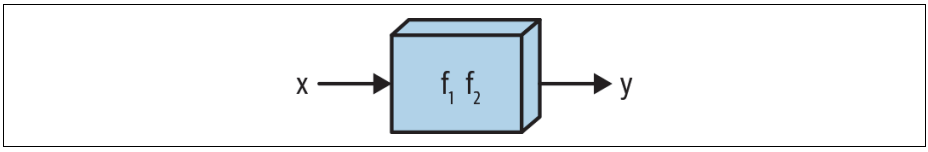


Abbildung 1-7: Eine andere Art, sich verschachtelte Funktionen vorzustellen

Außerdem besagt ein Theorem der Differenzialrechnung, dass eine komposite Funktion, die aus »größtenteils differenzierbaren« Funktionen besteht, ebenfalls differenzierbar ist! Daher können wir uns $f_1 f_2$ einfach als weitere Funktion vorstellen, deren Ableitungen wir berechnen können. Die Berechnung von Ableitungen kompositer Funktionen ist für das Training von Deep-Learning-Modellen essenziell.

3 Das Wort »Kompositfunktion« leitet sich vom mathematischen Prinzip der »Komposition« her, also der Verkettung oder Verknüpfung mehrerer Teilfunktionen. Weitere Informationen finden Sie z. B. in der Wikipedia unter *https://de.wikipedia.org/wiki/Komposition_(Mathematik)*.

Allerdings brauchen wir eine Formel, um die Ableitungen der Kompositfunktion, bezogen auf die Ableitungen ihrer einzelnen Bestandteile (der Einzelfunktionen), berechnen zu können. Darauf kommen wir im folgenden Abschnitt.

Die Kettenregel

Die *Kettenregel* ist ein mathematisches Theorem, mit dem wir die Ableitungen von Kompositfunktionen berechnen können. Aus mathematischer Perspektive sind Deep-Learning-Modelle ebenfalls Kompositfunktionen. Die Überlegungen zu ihren Ableitungen sind wichtig, um diese Modelle zu trainieren, wie wir in den folgenden Kapiteln sehen werden.

Mathematik

Mathematisch gesehen, besagt das Theorem in einer eher wenig intuitiven Form, dass für einen gegebenen Wert x gilt:

$$\frac{df_2}{du}(x) = \frac{df_2}{du}\big(f_1(x)\big) \times \frac{df_1}{du}(x)$$

Dabei ist u einfach eine Platzhaltervariable, die für die Eingabe an eine Funktion steht.

Um die Ableitung einer Funktion f mit je einer Ein- und Ausgabe zu beschreiben, können wir die Funktion, die für *die Ableitung dieser Funktion* steht, als $\frac{df}{du}$ notieren. Dabei ist es egal, ob Sie anstelle von u eine andere Platzhaltervariable verwenden, denn $f(x) = x2$ und $f(y) = y2$ sind gleichbedeutend.

Später werden wir uns mit Funktionen beschäftigen, die *mehrere* Eingaben entgegennehmen, etwa x und y. Wenn wir so weit sind, ist es sinnvoll, $\frac{df}{dx}$ zu schreiben, was eine andere Bedeutung hat als $\frac{df}{dy}$.

Daher bezeichnen wir alle Ableitungen unten mit einem u: f_1 und f_2 sind beides Funktionen, die eine Eingabe übernehmen und eine Ausgabe erzeugen. In solchen Fällen (wenn Funktionen je eine Ein- und eine Ausgabe haben) werden wir u in der Schreibweise für die Ableitung verwenden.

Diagramm

Die vorherige Formel gibt uns keinen besonders intuitiven Zugang zur Kettenregel. Dafür ist ein Kastendiagramm deutlich hilfreicher. Lassen Sie uns überlegen, was die Ableitung im einfachen Fall von f_1 f_2 sein *sollte*:

Intuitiv betrachtet, sollte die Ableitung der Kompositfunktion in Abbildung 1-8 ein Produkt der Ableitungen der Einzelfunktionen sein. Stellen wir uns vor, wir über-

geben der ersten Funktion den Wert 5, und die Berechnung der Ableitung der ersten Funktion bei $u = 5$ ergibt den Wert 3, also $\frac{df_1}{du}(5) = 3$.

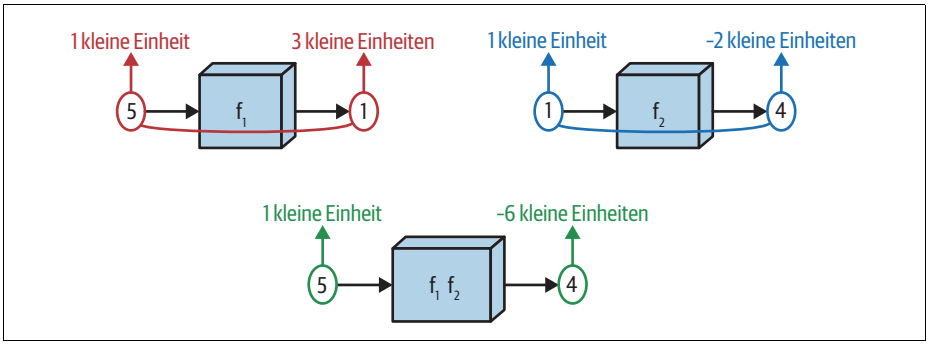


Abbildung 1-8: Eine Illustration der Kettenregel

Nehmen wir weiter an, dass der *Wert* der Funktion im ersten Kasten 1 beträgt, sodass gilt: $f_1(5) = 1$. Jetzt berechnen wir die Ableitung der zweiten Funktion f_2 bei diesem Wert, also $\frac{df_2}{du}(1)$ und erhalten den Wert –2.
Wenn wir uns beide Funktionen als tatsächlich miteinander verbunden vorstellen, bewirkt eine Änderung der Eingabe für Kasten 2 um eine Einheit in dessen Ausgabe eine Änderung von –2 Einheiten. Ändern wir die Eingabe für Kasten 2 um 3 Einheiten, sollte sich die Ausgabe um $-2 \times 3 = -6$ Einheiten verändern Aus diesem Grund ist in der Formel für die Kettenregel das Endergebnis letztlich ein Produkt:

$\frac{df_2}{du}(f_1(x))$ *multipliziert mit* $\frac{df_1}{du}(x)$.

Anhand des Diagramms und der mathematischen Formel können wir unter Verwendung der Kettenregel herausfinden, was die Ableitung der Ausgabe einer verschachtelten Funktion bezogen auf ihre Eingabe sein sollte. Wie sähen die Codeanweisungen für die Berechnung dieser Ableitung aus?

Code

Jetzt wollen wir den Code dazu schreiben und zeigen, dass die Berechnung von Ableitungen auf diese Weise tatsächlich »korrekt aussehende« Ergebnisse ergibt. Wir verwenden hier die `square`-Funktion aus dem Abschnitt »Grundsätzliche Funktionen in NumPy« auf Seite 6 zusammen mit `sigmoid`, einer weiteren für das Deep Learning wichtigen Funktion.

```
def sigmoid(x: ndarray) -> ndarray:
    '''
    Die sigmoid-Funktion auf jedes Element des Eingabe-ndarrays anwenden.
    '''
    return 1 / (1 + np.exp(-x))
```