Teil 1

Von null auf Bilderkennung

Dieser Teil gibt eine Einführung in überwachtes Lernen. Innerhalb von nur zwei Kapiteln werden wir ein erstes Machine-Learning-System programmieren, das wir anschließend Schritt für Schritt weiterentwickeln, bis es leistungsfähig genug ist, um handgeschriebene Ziffern zu erkennen.

Ja, Sie haben richtig gelesen: Auf den nächsten *etwa 100 Seiten* werden wir ein Programm zur Bilderkennung schreiben. Was noch besser ist: Wir werden dazu keine Machine-Learning-Bibliothek verwenden. Abgesehen von einigen Allzweckfunktionen für arithmetische Berechnungen und grafische Darstellung schreiben wir den ganzen Code selbst.

Es ist sehr unwahrscheinlich, dass Sie in Ihrer zukünftigen Karriere jemals Machine-Learning-Algorithmen von Grund auf selbst schreiben müssen. Aber es einmal zu tun, um die Grundlagen richtig zu verstehen, ist von unschätzbarem Wert. Sie werden genau wissen, was jede einzelne Zeile des fertigen Programms tut. Danach wird Machine Learning für Sie nie wieder wie schwarze Magie wirken.

1

Einführung in Machine Learning

Softwareentwickler erzählen sich gern Veteranengeschichten. Sobald ein paar von uns in einer Kneipe zusammensitzen, fragt einer: »An was für Projekten arbeitet ihr gerade?« Dann nicken wir heftig und hören uns die amüsanten und teilweise furchtbaren Geschichten der anderen an.

Bei einem dieser abendlichen Geplänkel Mitte der 90er erzählte mir eine Freundin von dem unmöglichen Auftrag, an dem sie gerade arbeitete. Ihre Vorgesetzten wünschten sich von ihr ein Programm, das Röntgenaufnahmen analysieren und dadurch Krankheiten erkennen konnte, etwa eine Lungenentzündung.

Meine Freundin warnte die Geschäftsleitung vor, dass das ein hoffnungsloses Unterfangen sei, aber man wollte ihr nicht glauben. Wenn ein Radiologe das leisten konnte, so argumentierten die Manager, warum dann nicht auch ein Visual-Basic-Programm? Sie stellten ihr sogar einen Radiologen zur Seite, damit sie lernte, wie er vorging, und dies in Code umsetzen konnte. Diese Erfahrung bestärkte sie jedoch nur in ihrer Meinung, dass Radiologie menschliches Urteilsvermögen und menschliche Intelligenz erforderte.

Wir lachten über die Sinnlosigkeit dieser Aufgabe. Ein paar Monate später wurde das Projekt aufgegeben.

Doch kehren wir nun in die Gegenwart zurück. 2017 veröffentlichte ein Forschungsteam der Stanford University einen Algorithmus, um Lungenentzündung anhand von Röntgenbildern zu erkennen.[1] Er erfüllte nicht nur seine Aufgabe, sondern war sogar zuverlässiger als ein professioneller Radiologe. Das hatte als unmöglich gegolten. Wie hatten die Forscher es geschafft, diesen Code zu schreiben?

Die Antwort lautet: gar nicht. Anstatt Code zu schreiben, setzten sie Machine Learning ein. Sehen wir uns an, was das bedeutet.

Programmierung und Machine Learning im Vergleich

Das folgende Beispiel zeigt den Unterschied zwischen Machine Learning (oder kurz ML) und gewöhnlicher Programmierung. Stellen Sie sich vor, Sie sollen ein Programm erstellen, das Videospiele spielt. Bei der traditionellen Programmierung würden Sie dazu Code wie den folgenden schreiben:

```
enemy = get_nearest_enemy()
if enemy.distance() < 100:
  decelerate()
  if enemy.is_shooting():
    raise_shield()
  else:
    if health() > 0.25:
      shoot()
    else:
      rotate_away_from(enemy)
else:
  # ... und noch viel mehr Code
```

Und so weiter. Der Großteil des Codes würde aus einer Riesenmenge von `if... else`-Anweisungen vermischt mit Befehlen wie `shoot()` bestehen.

Moderne Sprachen bieten uns zwar Möglichkeiten, diese hässlichen, verschachtelten `if`-Anweisungen durch angenehmere Konstruktionen wie Polymorphismus, Mustererkennung oder ereignisgestützte Aufrufe zu ersetzen, aber das Grundprinzip der Programmierung bleibt unverändert: Sie sagen dem Computer, wonach er Ausschau halten und was er tun soll. Dabei müssen Sie jede mögliche Bedingung aufführen und jede mögliche Aktion definieren.

Mit dieser Vorgehensweise sind wir weit gekommen, aber sie hat auch einige Nachteile. Erstens dürfen Sie nichts auslassen. Wahrscheinlich können Sie sich Dutzende oder gar Hunderte von besonderen Situationen vorstellen, die Sie in dem Videospielprogramm berücksichtigen müssen. Was geschieht, wenn sich ein Gegner nähert, sich aber ein Power-up zwischen Ihnen und ihm befindet, das Sie vor

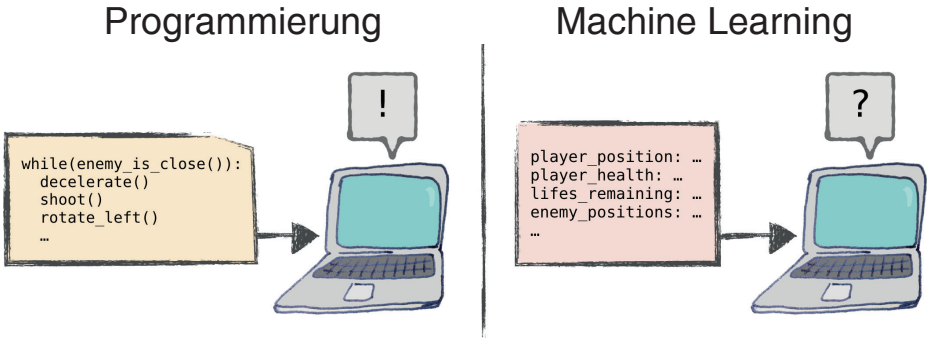
1. *news.stanford.edu/2017/11/15/algorithm-outperforms-radiologists-diagnosing-pneumonia*

seinen Schüssen schützen kann? Ein menschlicher Spieler wird eine solche Situation schnell erkennen und zu seinem Vorteil nutzen. Kann ein Programm das auch? Das hängt ganz von dem Programm ab. Wenn Sie diesen Sonderfall beim Schreiben des Codes berücksichtigt haben, dann kann das Programm damit umgehen. Allerdings wissen wir, wie schwer es ist, selbst in so eng umrissenen Aufgabenfeldern wie der Buchhaltung jegliche Sonderfälle abzudecken. Wenn Sie sämtliche Sonderfälle auf so komplexen Gebieten wie dem Spielen von Videospielen, dem Fahren eines Lkw oder dem Erkennen eines Bilds auflisten wollen, kann ich Ihnen dazu nur viel Glück wünschen.

Es reicht aber nicht nur, all diese Fälle aufzulisten; Sie müssen auch wissen, wie Sie dabei jeweils eine Entscheidung fällen. Das ist die zweite große Einschränkung bei der Programmierung, die in manchen Fachgebieten schon das Aus bedeutet. Betrachten Sie zum Beispiel eine Aufgabe aus dem Bereich des *maschinellen Sehens* wie das bereits erwähnte Erkennen einer Lungenentzündung anhand einer Röntgenaufnahme.

Wir wissen nicht *genau*, wie ein Radiologe eine Lungenentzündung erkennt. Wir haben zwar eine grobe Vorstellung davon, dass er nach undurchsichtigen Bereichen sucht, aber wir wissen nicht, wie sein Gehirn solche undurchsichtigen Bereiche erkennt und bewertet. Manchmal kann ein solcher Experte selbst nicht erklären, wie er zu der Diagnose gekommen ist, sondern nur vage Begründungen geben wie: »Ich weiß aus Erfahrung, dass eine Lungenentzündung nicht so aussieht.« Da wir nicht wissen, wie solche Entscheidungen ablaufen, können wir einen Computer auch nicht anweisen, sie zu fällen. Dieses Problem stellt sich bei allen typisch menschlichen Aufgaben, etwa dem Verkosten von Bier oder dem Verstehen eines Satzes.

Machine Learning dagegen stellt das Prinzip der traditionellen Programmierung auf den Kopf: Der Computer erhält keine *Anweisungen*, sondern *Daten*, und wird aufgefordert, selbst herauszufinden, was er tun soll.



Dass der Computer selbst etwas herausfinden soll, klingt wie Wunschdenken. Allerdings gibt es tatsächlich einige Möglichkeiten, um das zu erreichen. Für alle diese Vorgehensweisen ist nach wie vor die Ausführung von Code erforderlich, allerdings ist dieser Code im Gegensatz zur herkömmlichen Programmierung keine schrittweise Anleitung, um das vorliegende Problem zu lösen. Beim Machine Learning teilt der Code dem Computer mit, wie er die Daten verarbeiten soll, um das Problem selbst zu lösen.

Betrachten wir als Beispiel wieder einen Computer, der herausfinden soll, wie man ein Videospiel spielt. Stellen Sie sich einen Algorithmus vor, der durch Versuch und Irrtum zu spielen lernt. Zu Anfang gibt er zufällige Befehle: schießen, bremsen, drehen usw. Er merkt sich, wann die Befehle zum Erfolg führen, etwa zu einem höheren Punktestand, aber auch, wann sie in einem Fehlschlag wie dem Tod der Spielfigur resultieren. Außerdem registriert er den Spielzustand: wo sich die Gegner, die Hindernisse und die Power-ups befinden, wie viele Gesundheitspunkte die Figur hat usw.

Wenn der Algorithmus später einem ähnlichen Spielzustand begegnet, wird er mit höherer Wahrscheinlichkeit erfolgreiche Aktionen durchführen. Eine Vielzahl solcher Trial-and-Error-Durchläufe kann aus dem Programm einen fähigen Spieler machen. Mit dieser Vorgehensweise erreichte ein System im Jahr 2013 geradezu übermenschliche Fähigkeiten bei einer Reihe alter Atari-Spiele.[2]

Diese Form von ML wird als *bestärkendes Lernen* oder *Reinforcement Learning* bezeichnet. Es funktioniert fast genauso wie Hundetraining: »Gutes« Verhalten wird belohnt, sodass der Hund es häufiger an den Tag legt. (Ich habe das auch mit meiner Katze versucht, aber bislang ohne Erfolg.)

Reinforcement Learning ist jedoch nur eine Möglichkeit, mit der ein Computer die Lösung eines Problems herausfinden kann. Der Schwerpunkt dieses Buchs liegt dagegen auf einer anderen Form von Machine Learning, nämlich der wahrscheinlich am häufigsten verwendeten. Sehen wir uns das genauer an.

Überwachtes Lernen

Unter den verschiedenen Ansätzen des ML hat das *überwachte Lernen* (*Supervised Learning*) bisher die beeindruckendsten Ergebnisse hervorgebracht. Damit lassen sich auch Aufgaben wie die Diagnose einer Lungenentzündung lösen.

Ausgangspunkt des überwachten Lernens ist eine Reihe von *Beispielen*, die jeweils mit *Labels* versehen sind, aus denen der Computer etwas lernen kann:

2. *deepmind.com/research/publications/playing-atari-deep-reinforcement-learning*

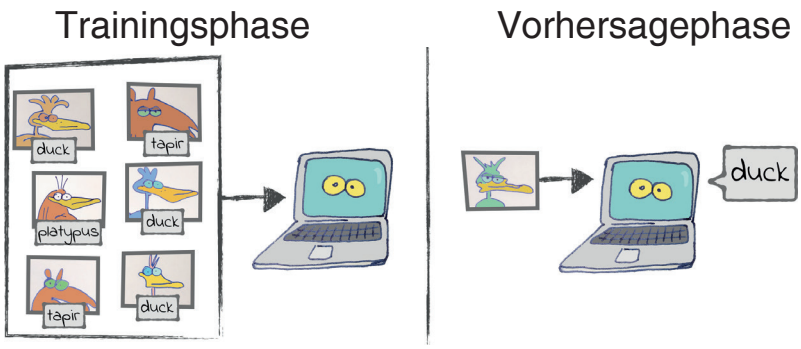
Zweck	Beispiel	Label
Ein System, das aus dem Bellen eines Hundes die Hunderasse bestimmt	`.wav`-Datei mit Hundegebell	Die Hunderasse, also etwa »Windhund«, »Beagle« usw.
Ein System, das eine Lungenentzündung erkennen kann	Röntgenaufnahme	Ein boolesches Flag: 1, wenn die Aufnahme eine Lungenentzündung zeigt, anderenfalls 0
Ein System, das die Einnahmen einer Eisdiele anhand des Wetters vorhersagt	Aufgezeichnete Temperaturen an einem vergangenen Tag	Die aufgezeichneten Einnahmen an diesem Tag
Ein System, das erkennt, welche Stimmung der Tweet eines Politikers vermittelt	Ein Tweet	Eine Stimmung wie »entrüstet«, »verärgert« oder »wütend«

Wie Sie sehen, können als Beispiele viele verschiedene Dinge genutzt werden: Daten, Text, Klänge, Videos usw. Es gibt auch zwei Arten von Labels: *Numerische Labels* sind einfach nur Zahlen, wie es etwa bei dem Temperatur-Eisdielenumsätze-Umrechner der Fall ist. *Kategorielabels* dagegen stehen wie bei dem Hunderassendetektor für eine Kategorie (oder Klasse) aus einer vordefinierten Menge. Wenn Sie Ihre Fantasie spielen lassen, werden Ihnen noch viele weitere Möglichkeiten einfallen, um etwas – seien es Zahlen oder Kategorien – aus etwas anderem zu bestimmen.

Nehmen wir an, dass wir bereits eine Reihe solcher Beispiele mit Labels zusammengestellt haben. Daran schließen sich nun die beiden folgenden Phasen des überwachten Lernens an:

1. In der ersten Phase speisen wir die mit Labeln gekennzeichneten Beispiele in einen Algorithmus ein, der Muster erkennen kann. Ein solcher Algorithmus kann beispielsweise bemerken, dass alle Aufnahmen, die eine Lungenentzündung zeigen, gemeinsame Merkmale aufweisen, etwa undurchsichtige Bereiche, die in den Aufnahmen ohne Lungenentzündung nicht zu finden sind. Dies ist die sogenannte *Trainingsphase*, in der der Algorithmus die Beispiele immer wieder durchsieht, um zu lernen, solche Muster zu erkennen.
2. Wenn der Algorithmus weiß, wie eine Lungenentzündung aussieht, gehen wir zur *Vorhersagephase* über, in der wir die Früchte unserer Arbeit ernten. Wir zeigen dem trainierten Algorithmus eine Röntgenaufnahme *ohne Label*, woraufhin uns der Algorithmus sagt, ob sie Anzeichen einer Lungenentzündung aufweist oder nicht.

Auch das folgende System zur Erkennung von Tierarten ist ein Beispiel für überwachtes Lernen. Hier sind die einzelnen Eingaben die Bilder von Tieren mit der Angabe der Art als Label. In der Trainingsphase zeigen wir dem Algorithmus Bilder mit Labels und in der Vorhersagephase Bilder ohne Labels, wobei der Algorithmus die Labels rät:



Ich habe bereits gesagt, dass der Computer beim Machine Learning etwas aus den Daten »herausfindet«. Das überwachte Lernen ist ein Beispiel eines solchen Vorgangs. Bei der herkömmlichen Programmierung führen Sie den Computer mit Ihrem Code von der Eingabe zur Ausgabe. Beim überwachten Lernen dagegen stellen Sie dem Computer Beispiele von Ein- und Ausgaben zur Verfügung, anhand derer er selbst herausfindet, wie er von den einen zu den anderen kommt.

Möglicherweise hat dieser grobe Überblick über das überwachte Lernen mehr Fragen aufgeworfen als beantwortet. Wir haben gesagt, dass ein Programm zum überwachten Lernen Muster und gemeinsame Merkmale in den Daten »erkennt«. Aber wie macht es das? Schauen wir uns also genauer an, wie dieses Zauberkunststück tatsächlich funktioniert.

Die Mathematik hinter dem Zaubertrick

Um die Beziehung zwischen Daten und ihren Labels zu erkennen, nutzt ein System zum überwachten Lernen das mathematische Prinzip der *Näherungsfunktion* aus. Sehen wir uns an einem konkreten Beispiel an, was das bedeutet.

Nehmen wir an, Sie haben eine Solaranlage auf Ihrem Dach und wollen ein System zum überwachten Lernen konstruieren, das lernt, in welchem Maße diese Anlage Energie erzeugt, und die in Zukunft generierten Energiemengen vorhersagen kann.

Der Ertrag einer Solaranlage hängt von verschiedenen Variablen ab, darunter der Tageszeit, dem Wetter usw. Besonders wichtig scheint die Tageszeit zu sein, weshalb Sie sich darauf konzentrieren. In der typischen Vorgehensweise für über-

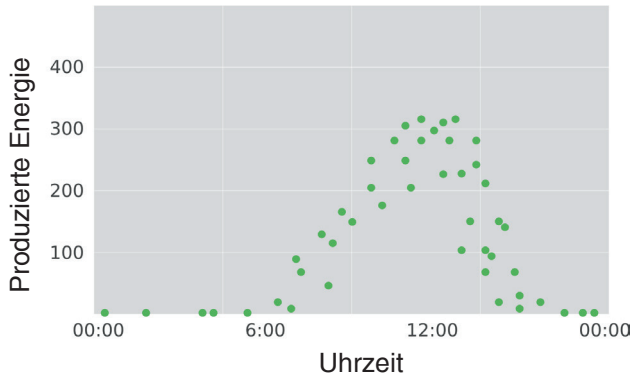
wachtes Lernen sammeln Sie Beispiele für die an verschiedenen Tageszeiten produzierte Energie. Nach einigen Wochen, in denen Sie zufällige Stichproben aufgenommen haben, liegt Ihnen die folgende Tabelle vor:

Uhrzeit	Energie (Wattstunden)
09:01	153
11:48	280
05:20	0

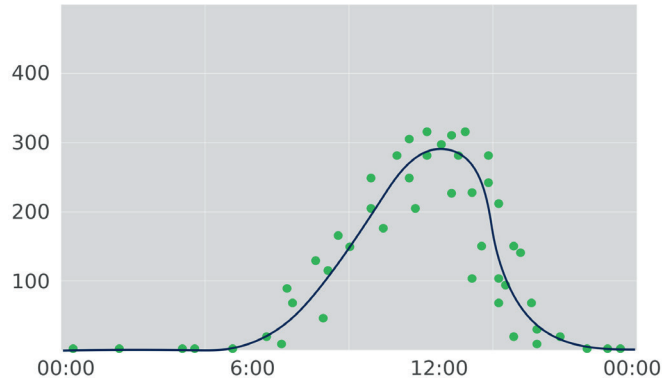
... usw.

Jede Zeile dieser Tabelle ist ein Beispiel, das jeweils aus einer *Eingabevariablen* (der Uhrzeit) und einem *Label* (der produzierten Energie) besteht. Das ist genauso wie bei dem System zum Erkennen von Tieren, bei denen ein Bild als Eingabe diente und der Name des Tiers als Label angegeben wurde.

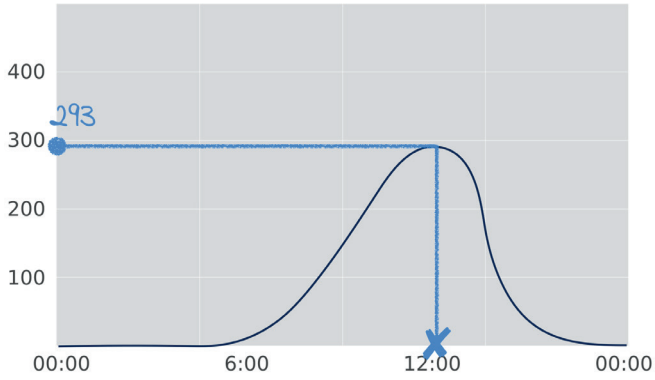
Wenn Sie die Beispiele wie folgt in ein Diagramm einzeichnen, können Sie unmittelbar ablesen, in welchem Zusammenhang die Uhrzeit zur produzierten Energie steht:



Unser menschliches Gehirn erkennt auf den ersten Blick, dass die Solaranlage nachts keine Energie produziert und dass der Ertrag um die Mittagszeit einen Spitzenwert annimmt. Einem System zum überwachten Lernen fehlt eine Luxuseinrichtung wie unser Gehirn, allerdings kann es die Daten verstehen, indem es sie durch eine Funktion annähert, wie Sie im folgenden Diagramm sehen:



Die passende Näherungsfunktion zu den Beispielen zu finden, ist der schwierige Teil. Dies ist, was ich früher als »Trainingsphase« bezeichnet habe. Die anschließende Vorhersagephase dagegen ist einfach: Das System kümmert sich dann nicht mehr um die Beispiele, sondern nutzt die Funktion, um die von der Solaranlage produzierte Energie vorherzusagen, etwa zur Mittagszeit an beliebigen Tagen:



Das war es, was ich weiter vorn mit der Aussage meinte, beim überwachten Lernen werde das Prinzip der Näherungsfunktion genutzt: Das System erhält reale Daten, die gemeinhin unsauber und unvollständig sind. In der Trainingsphase nähert es diese komplizierten Daten durch eine relativ einfache Funktion an, die es dann in der Vorhersagephase nutzt, um unbekannte Daten zu prognostizieren.

Als Programmierer sind Sie es gewohnt, an all die verschiedenen Dinge zu denken, die schiefgehen können. Wahrscheinlich sind Ihnen deshalb bereits zahlreiche Möglichkeiten eingefallen, um unser Beispiel komplizierter zu machen. Erstens hängt der Ertrag einer Solaranlage noch von anderen Variablen ab als der Uhrzeit, etwa der Bewölkung und der Jahreszeit. Wenn wir auch diese Variablen erfassen, haben wir eine mehrdimensionale Wolke aus Datenpunkten, die wir nicht mehr in einem Diagramm wie dem vorherigen grafisch darstellen können. Außer-

dem wollen wir im Fall unserer Solaranlage ein *numerisches* Label voraussagen. Wahrscheinlich fragen Sie sich, wie sich die hier gezeigte Methode auch für nicht numerische Labels wie Tiernamen, also für *Kategorielabels*, umsetzen lässt.

All diese Dinge werden wir uns in diesem Buch noch ansehen. Für den Augenblick wollen wir jedoch nur festhalten, dass das Grundprinzip des überwachten Lernens unabhängig von allen zusätzlichen Komplikationen der hier gegebenen Beschreibung folgt: Wir gehen von einer Reihe von Beispielen aus und suchen eine Näherungsfunktion dafür.

Moderne Systeme für überwachtes Lernen sind sehr gut darin, diese Annäherung vorzunehmen. Sie können selbst komplizierte Beziehungen wie die zwischen Röntgenbildern und einer medizinischen Diagnose annähern. Die Annäherungsfunktion dafür sieht für Menschen zwar wahnsinnig kompliziert aus, ist für solche Systeme aber nichts Ungewöhnliches.

Und das ist auch schon kurz und bündig, worum es beim überwachten Lernen geht. Die Einzelheiten ... nun, die sehen wir uns im Verlaufe dieses Buchs an. Im Folgenden wollen wir daher unsere Computer einrichten, damit wir mit dem Programmieren beginnen können.

Und was ist mit unüberwachtem Lernen?

Es gibt verschiedene Spielarten des Machine Learning. In den meisten Lehrbüchern werden drei erwähnt: bestärkendes Lernen (Reinforcement Learning), überwachtes Lernen (Supervised Learning) und als dritte Variante das *unüberwachte Lernen* (*Unsupervised Learning*). Wir haben zur Einführung in die Grundlagen von ML Reinforcement Learning herangezogen und werden uns im Rest dieses Buchs mit überwachtem Lernen beschäftigen. Was aber ist unüberwachtes Lernen?

Theoretisch besteht der Unterschied zwischen überwachtem und unüberwachtem Lernen darin, dass die Beispieldaten in letzterem Fall keine Labels haben. In der Praxis hat unüberwachtes Lernen jedoch nicht viel mit unserer üblichen Vorstellung von »Lernen« zu tun, sondern wirkt eher wie eine äußerst anspruchsvolle Art von Datenverarbeitung.

Als Beispiel für unüberwachtes Lernen stellen Sie sich vor, dass Sie Marktforschung für einen Onlineshop betreiben und dabei alle Daten über die Kunden vorliegen haben: wie viel sie ausgeben, wie oft sie den Shop besuchen usw. Ein Algorithmus für unüberwaches Lernen kann Ihnen helfen, sinnvolle Zusammenhänge aus diesen Daten abzulesen, indem er ähnliche Kunden zu Gruppen zusammenfasst – ein Vorgang, der als *Clustering* bezeichnet wird.

Unüberwachtes Lernen kann sehr praktisch sein, steht heutzutage aber nicht so sehr im Rampenlicht wie die anderen Varianten von ML. Für dieses Buch musste ich mich auf ein Thema beschränken und habe mich deshalb auf überwachtes Lernen konzentriert.

Das System einrichten

Den ersten Teil dieses Buchs nimmt ein ML-Tutorial ein. Wir fangen dabei bei null an und haben am Ende ein funktionierendes Programm für maschinelles Sehen, das uns später als Ausgangspunkt dient, um die höheren Weihen zu empfangen: neuronale Netze in Teil II und Deep Learning in Teil III.

Am besten ist es, das Tutorial gleich bei der Lektüre durchzuarbeiten, also den angegebenen Quellcode auszuführen und die Übungsaufgaben zu lösen, die am Ende der meisten Kapitel stehen. Es ist jedoch auch möglich, erst alles zu lesen, um sich einen Überblick zu verschaffen, und erst dann zur Tastatur zu greifen. Beide Vorgehensweisen sind geeignet, wobei Programmierer jedoch gewöhnlich die erste bevorzugen.

Es dauert auch nicht lange, Ihr System so einzurichten, dass Sie den angegebenen Code ausführen können. ML erfordert zwar gewöhnlich eine hohe Rechenleistung, doch der Code in diesem Buch läuft auch auf einem normalen Laptop sehr gut. Sie müssen lediglich etwas Software installieren.

Vor allem benötigen Sie Python, die am häufigsten für ML verwendete Sprache, in der auch die Beispiele in diesem Buch geschrieben sind. Machen Sie sich keine Sorgen, falls Sie noch nie in Python programmiert haben – Sie werden überrascht sein, wie gut lesbar diese Sprache ist. Wenn Sie Schwierigkeiten haben sollten, den Python-Code zu verstehen, lesen Sie Anhang 1, »*Grundlagen von Python*«. Darin erfahren Sie alles, was Sie für dieses Buch benötigen.

Hinweis für erfahrene Pythonistas

Wenn Sie sich mit Python auskennen, werden Sie bemerken, dass der Code in diesem Buch teilweise von den üblichen Konventionen abweicht. Beispielsweise habe ich bewusst auf einige sprachspezifische Konstruktionen wie die Listennotation verzichtet, die den Code für Einsteiger unnötig kompliziert machen. In demselben Bemühen, mich möglichst verständlich auszudrücken, verwende ich auch manche Fachbegriffe etwas lockerer, z. B. »Funktion« statt »Methode«. Für meine Verstöße gegen die Python-Etikette möchte ich mich bereits im Voraus entschuldigen.

Fangen wir an! Als Erstes müssen Sie prüfen, ob Python bei Ihnen installiert ist. Führen Sie folgenden Befehl aus:

```
python3 --version
```

Wenn Sie Python 3 nicht haben, dann beschaffen Sie es sich, bevor Sie weiterlesen. In Anhang 1, »*Grundlagen von Python*«, erfahren Sie, wie Sie die Sprache installieren.

Achtung: Auf manchen Systemen können Sie Python 3 auch mit `python` ohne die angehängte `3` ausführen, während dieser Befehl auf anderen Systemen stattdessen Python 2 ausführt. Um Verwirrung und mysteriöse Fehler aufgrund der Verwendung einer älteren Python-Version zu verhindern, sollten Sie immer den Befehl `python3` verwenden.

Wenn Sie die Sprache installiert haben, müssen wir noch ein Wort über die Bibliotheken verlieren. Für den Anfang benötigen Sie drei davon. An erster Stelle steht dabei NumPy, eine Bibliothek für wissenschaftliche Berechnungen. Außerdem verwenden wir zwei Bibliotheken zur Ausgabe von Diagrammen. Matplotlib ist der De-facto-Standard für Diagramme in Python. Seaborn baut auf Matplotlib auf und dient vor allem dazu, Diagrammen ein angenehmes Erscheinungsbild zu verleihen.

Um diese Bibliotheken zu installieren, können Sie entweder `pip` verwenden, den offiziellen Paketmanager von Python, oder Conda, einen anspruchsvollen Umgebungsmanager, der insbesondere im Umfeld von ML sehr beliebt ist. Im Abschnitt »*Pakete mit Conda installieren*« auf *Seite 340* werden die Unterschiede zwischen `pip` und Conda genauer beleuchtet. Im Zweifelsfall nehmen Sie `pip`.

Zur Installation der Bibliotheken mit `pip` führen Sie die folgenden Befehle aus:

```
pip3 install numpy==1.15.2
pip3 install matplotlib==3.1.2
pip3 install seaborn==0.9.0
```

Das war es auch schon! Wenn Sie lieber Conda verwenden, richten Sie sich nach den Anweisungen in der Datei `readme.txt`, die Sie im Wurzelverzeichnis des Quellcodes finden.

Schließlich brauchen Sie auch noch eine Programmierumgebung. In vielen ML-Tutorials wird Jupyter Notebook verwendet, womit Sie den Code im Browser bearbeiten und ausführen können. Für die Beispiele in diesem Buch ist es jedoch nicht unbedingt erforderlich, Jupyter Notebook einzusetzen. Als Entwickler sind Sie ja schon damit vertraut, Programme zu schreiben und auszuführen. Nutzen Sie also einfach den Texteditor oder die IDE Ihrer Wahl. Für den Fall, dass Sie Jupyter bereits kennen und verwenden, finden Sie im Verzeichnis `notebooks` die Jupyter-Version des Codes zu diesem Buch.

Gehen wir die Liste noch einmal durch: Sie haben Python, drei wichtige Bibliotheken und Ihren bevorzugten Editor. Mehr brauchen Sie nicht, um anfangen zu können.

Als Nächstes wollen wir damit beginnen, ein Programm zu schreiben, das lernen kann.

Gewöhnen Sie sich an ML

Als Entwickler sind Sie es gewohnt, rasch zu lernen. Bei ML jedoch wagen Sie sich auf völlig neues Terrain vor. Ich will es nicht beschönigen: Die nächsten drei oder vier Kapitel sind schwer. Bei der Lektüre werden Sie sich womöglich wie ein blutiger Anfänger vorkommen, was zwar durchaus spannend, aber manchmal auch sehr frustrierend sein kann.

Ich kenne dieses Gefühl, aber ich kann Ihnen versichern, dass es sich lohnt, diese Hürden zu überwinden. Ich kann mich noch gut daran erinnern, wie faszinierend es war, als ich zum ersten Mal ein Machine-Learning-Programm ausführte, das eine korrekte Vorhersage machte. Halten Sie durch, und schon bald werden auch Sie dieses Glücksgefühl erleben!

3

Am Gradienten entlang

Im letzten Kapitel haben wir bereits etwas geschafft, auf das wir stolz sein können: Wir haben Code geschrieben, der lernen kann. Sollten wir diesen Code jedoch von einem Informatiker überprüfen lassen, so würde dieser ihn als mangelhaft einstufen. Insbesondere beim Anblick der Funktion `train()` würde er den Kopf schütteln. »Für diese einfache Aufgabe mag der Code in Ordnung sein«, mag der gestrenge Informatiker sagen, »aber er lässt sich nicht auf reale Probleme skalieren.«

Womit er recht hätte. In diesem Kapitel werden wir zwei verschiedene Dinge tun, damit es nicht zu einer solchen Kritik kommt. Erstens verzichten wir darauf, den Code einem Informatiker zur Überprüfung vorzulegen. Zweitens schauen wir uns die Probleme der jetzigen Implementierung von `train()` genauer an und lösen sie mit einem der grundlegenden Prinzipien des Machine Learnings, nämlich einem Algorithmus, der als *Gradientenverfahren*, *Gradientenabstieg* oder *Verfahren des steilsten Abstiegs* bekannt ist. Ebenso wie unser bisheriger Code von `train()` dient dieses Verfahren dazu, das Minimum der Verlustfunktion zu finden, allerdings schneller, genauer und allgemeiner als mit dem Code aus dem vorherigen Kapitel.

Der Gradientenabstieg ist nicht nur für unser kleines Programm nützlich. Ohne dieses Verfahren werden Sie beim Machine Learning nicht weit kommen.

Dieser Algorithmus wird uns in verschiedener Form das ganze Buch hindurch begleiten.

Schauen wir uns als Erstes das Problem an, das wir mit dem Gradientenverfahren lösen wollen.

Unser Algorithmus bringt es nicht

Unser Programm kann erfolgreich Pizzaverkaufszahlen vorhersagen. Aber warum sollten wir es dabei belassen? Vielleicht können wir mit demselben Code ja auch andere Dinge vorhersagen, etwa Bewegungen auf dem Aktienmarkt. Damit könnten wir über Nacht reich werden! (Um Ihnen die Enttäuschung zu ersparen: Nein, das funktioniert nicht.)

Allerdings stoßen wir schon bald auf ein Problem, wenn wir versuchen, unser Programm zur linearen Regression auf eine andere Aufgabe anzuwenden. Unser Code verwendet ein einfaches Geradenmodell mit zwei Parametern, nämlich dem Gewicht `w` und dem Bias `b`. In der Praxis sind jedoch meistens komplexe Modelle mit mehr Parametern erforderlich. Denken Sie zum Beispiel an das Ziel, das wir uns für Teil I dieses Buchs gesetzt haben, nämlich ein System zu konstruieren, das Bilder erkennen kann. Ein Bild ist viel komplizierter als eine einzelne Zahl, weshalb wir auch ein Modell mit weit mehr Parametern brauchen als für das Pizzaprogramm.

Wenn wir unserem Modell weitere Parameter hinzufügen, geht seine Leistung jedoch in den Keller. Betrachten Sie dazu noch einmal die Funktion `train()` aus dem letzten Kapitel:

```
def train(X, Y, iterations, lr):
    w = b = 0
    for i in range(iterations):
        current_loss = loss(X, Y, w, b)
        print("Iteration %4d => Loss: %.6f" % (i, current_loss))

        if loss(X, Y, w + lr, b) < current_loss:
            w += lr
        elif loss(X, Y, w - lr, b) < current_loss:
            w -= lr
        elif loss(X, Y, w, b + lr) < current_loss:
            b += lr
        elif loss(X, Y, w, b - lr) < current_loss:
            b -= lr
        else:
            return w, b

raise Exception("Couldn't converge within %d iterations" % iterations)
```

Bei jeder Iteration modifiziert dieser Algorithmus entweder `w` oder `b` und sucht nach den Werten, bei denen der Verlust möglichst gering ist. Das allerdings kann schiefgehen, denn wenn wir `w` optimieren, kann das den durch `b` verursachten Verlust erhöhen und umgekehrt. Um dieses Problem zu vermeiden und so nah wie möglich an den kleinstmöglichen Verlust heranzukommen, könnten wir beide Parameter *gleichzeitig* modifizieren. Je mehr Parameter wir haben, umso wichtiger wird das.

Um `w` und `b` gemeinsam zu optimieren, müssen wir alle möglichen Kombinationen ausprobieren: sowohl `w` als auch `b` vergrößern, `w` vergrößern und `b` verkleinern, `w` vergrößern und `b` unverändert lassen, `w` verkleinern und ... usw. usf. Die Gesamtzahl möglicher Kombinationen einschließlich des Falls, bei dem alle Parameter unverändert bleiben, ist 3 hoch die Anzahl der Parameter. Bei zwei Parametern sind das 3^2 gleich 9 Kombinationen.

Es hört sich nicht weiter schlimm an, `loss()` pro Iteration neunmal aufzurufen. Bei zehn Parametern haben wir es allerdings schon mit 3^{10} Kombinationen zu tun, also fast 60.000 Aufrufen pro Iteration. Eine Anzahl von zehn Parametern ist auch alles andere als übertrieben. Weiter hinten in diesem Buch verwenden wir Modelle mit *Hunderttausenden* von Parametern. Bei solch riesigen Modellen käme ein Algorithmus, der jede Parameterkombination ausprobiert, nicht mehr von der Stelle. Wir sollten uns daher lieber gleich von diesem langsamen Code verabschieden.

Außerdem weist `train()` in der jetzigen Form ein noch gewichtigeres Problem auf: Die Funktion ändert die Parameter in Schritten, die genauso groß sind wie die Lernrate. Bei großer `lr` ändern sich die Parameter schnell, was zwar den Trainingsvorgang beschleunigt, das Endergebnis aber weniger genau macht, da jeder Parameter jetzt ein Vielfaches des großen `lr` betragen muss. Um die Genauigkeit zu verbessern, brauchen wir eine kleine `lr`, die allerdings zu einem langsameren Training führt. Geschwindigkeit und Genauigkeit gehen jeweils auf Kosten des anderen, wobei wir jedoch beides brauchen.

Aus diesen Gründen ist unser bisheriger Code nichts weiter als ein Hack. Wir müssen ihn durch einen besseren Algorithmus ersetzen – einen, der `train()` sowohl schnell als auch genau macht.

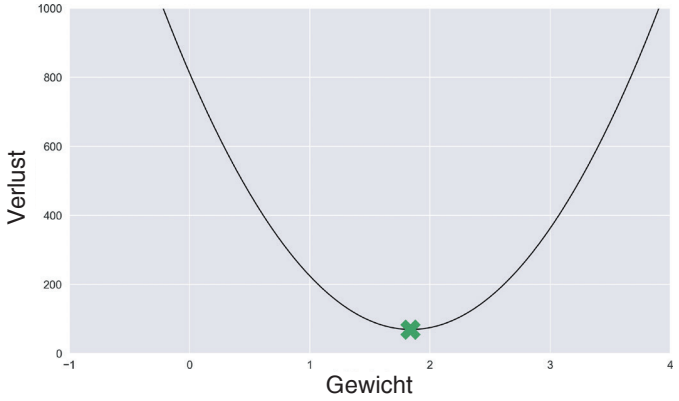
Das Gradientenverfahren

Wir brauchen einen besseren Algorithmus für `train()`. Die Aufgabe dieser Funktion besteht darin, die Parameter zu finden, bei denen der Verlust minimal wird. Schauen wir uns also `loss()` genauer an:

```
def loss(X, Y, w, b):
    return np.average((predict(X, w, b) - Y) ** 2)
```

Betrachten Sie die Argumente dieser Funktion. `X` und `Y` enthalten die Eingabevariablen und die Labels, ändern sich also nicht von einem Aufruf von `loss()` zum nächsten. Zur Vereinfachung wollen wir `b` vorübergehend auch konstant auf 0 setzen. Unsere einzige Variable ist damit `w`.

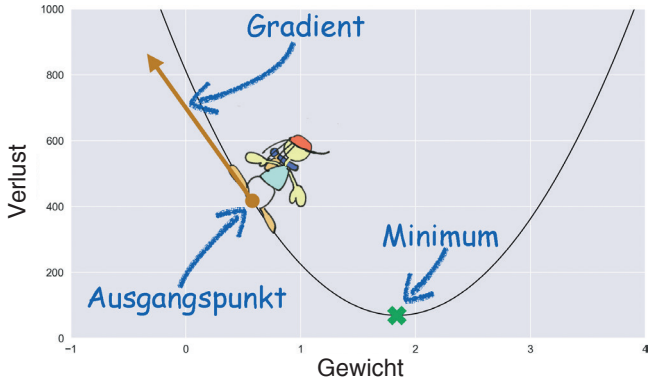
Wie ändert sich nun der Verlust mit der Veränderung von `w`? Ich habe ein kleines Programm geschrieben, das den Verlauf von `loss()` für `w` im Bereich von –1 bis 4 ausgibt und das Minimum mit einem grünen Kreuz markiert. Das sehen Sie in der folgenden Abbildung. (Den Code finden Sie wie üblich im Begleitmaterial zu diesem Buch.)



Eine hübsche Kurve – nennen wir sie die *Verlustkurve*. Sinn und Zweck von `train()` besteht darin, die markierte Stelle unten in dieser Kurve zu finden, also den Wert von `w`, der zu einem minimalen Verlust führt. Bei diesem `w` nähert sich unser Modell den Datenpunkten am besten an.

Stellen Sie sich diese Kurve als ein Tal vor, an dessen Hang irgendwo eine Wanderin steht, die zu ihrem Lager an der gekennzeichneten Stelle unterwegs ist. Allerdings ist es so dunkel, dass sie nur den Boden in unmittelbarer Nähe ihrer Füße sehen kann. Um das Lager zu finden, kann sie sich eines ganz einfachen Verfahrens bedienen: Sie geht immer in die Richtung des steilsten Abstiegs. Sofern es in dem Gelände keine Löcher oder Klippen gibt – was bei unserer Verlustfunktion nicht der Fall ist –, führt jeder Schritt die Wanderin näher an ihr Lager heran.

Um dieses Prinzip in Code umsetzen zu können, müssen wir die *Steigung* der Verlustkurve bestimmen. Ein Maß dafür ist der *Gradient*. Vereinbarungsgemäß ist der Gradient an einem gegebenen Punkt der Kurve ein »Pfeil«, der bergauf weist.



Um den Gradienten zu bestimmen, verwenden wir ein mathematisches Werkzeug: die *Ableitung* des Verlusts nach dem Gewicht, geschrieben `∂L/∂w`. Formal ausgedrückt, bestimmt die Ableitung an einem gegebenen Punkt, wie stark sich `L` an diesem Punkt bei kleinen Abweichungen von `w` ändert. Was geschieht mit dem Verlust, wenn wir das Gewicht ein winziges bisschen erhöhen? In dem vorstehenden Diagramm ist die Ableitung negativ, da der Verlust abnimmt. Bei positiver Ableitung dagegen steigt der Verlust. Am Minimum der Kurve, also an dem mit dem Kreuz markierten Punkt, ist die Kurve eben und die Ableitung damit null.

Beachten Sie, dass unsere Wanderin in die dem Gradienten *entgegengesetzte* Richtung gehen muss, um das Minimum zu erreichen. An einem Punkt mit negativer Ableitung wie in dem Bild muss sie sich also in positiver Richtung bewegen. Ihre Schrittweite muss proportional zum Betrag der Ableitung sein. Ist die Ableitung betragsmäßig groß, verläuft die Kurve steil. Das Lager ist dann noch weit entfernt. Daher kann die Wanderin vertrauensvoll große Schritte machen. Wenn sie sich dem Lager nähert, wird die Ableitung jedoch kleiner und damit auch ihre Schrittweite.

Dieser Algorithmus ist das *Gradientenverfahren* oder *Verfahren des steilsten Abstiegs*. Zu seiner Implementierung ist ein bisschen Mathematik gefordert.

Ein wenig Mathematik

Als Erstes übersetzen wir unsere Formel für den mittleren quadratischen Fehler in die gute, alte mathematische Schreibweise:

$$L = \frac{1}{m}\sum_{i=1}^{m}(wx_i + b) - y_i)^2$$

Quadrat
Fehler
Mittelwert

Falls Ihnen diese Schreibweise nicht bekannt vorkommt: Das Symbol S ist das Summenzeichen, und das `m` steht für die Anzahl der Beispiele. Diese Formel bedeutet: »Summiere die quadrierten Fehler aller Beispiele von Beispiel 1 bis Beispiel `m` und dividiere das Ergebnis durch die Anzahl der Beispiele.«

Bei den verschiedenen `x` und `y` handelt es sich um die Eingabevariablen und Labels, also um Konstanten. Auch `m` ist konstant, da sich die Anzahl der Beispiele nicht ändert. Da wir `b` vorübergehend auf 0 gesetzt haben, ist auch dieser Wert konstant. Wir werden `b` in Kürze wieder benutzen, aber vorläufig ist `w` der einzige Wert in der Formel, der sich ändert.

Nun müssen wir Betrag und Richtung des Gradienten bestimmen, also die Ableitung von `L` nach `w`. Wenn Sie sich noch an die Analysis in der Oberstufe erinnern, können Sie die Ableitung selbst berechnen. Wenn nicht, ist das aber auch kein Beinbruch. Jemand anderes hat die Arbeit schon für uns erledigt:

$$\frac{\partial L}{\partial w} = \frac{2}{m} \sum_{i=1}^{m} x_i((wx_i + b) - y_i)$$

Die Ableitung des Verlusts sieht ähnlich aus wie der Verlust selbst, allerdings ist die Quadrierung verloren gegangen. Außerdem wird jeder Summand mit `x` und das Endergebnis mit 2 multipliziert. In diese Formel können wir nun beliebige Werte für `w` eingeben und erhalten den Gradienten an diesem Punkt als Ergebnis.

Im Code sieht diese Formel wie folgt aus. Auch hier ist `b` wieder auf 0 fixiert.

03_gradient/gradient_descent_without_bias.py

```
def gradient(X, Y, w):
    return 2 * np.average(X * (predict(X, w, 0) - Y))
```

Mit der Formel für den Gradienten können wir nun `train()` so umschreiben, dass die Funktion das Gradientenverfahren anwendet.

Abwärts

Mit den Änderungen für das Gradientenverfahren sieht `train()` wie folgt aus:

03_gradient/gradient_descent_without_bias.py

```
def train(X, Y, iterations, lr):
    w = 0
    for i in range(iterations):
        print("Iteration %4d => Loss: %.10f" % (i, loss(X, Y, w, 0)))
        w -= gradient(X, Y, w) * lr
    return w
```

Diese Version von `train()` ist viel knapper als die vorherige. Bei Anwendung des Gradientenverfahrens brauchen wir keine `if`-Anweisungen mehr. Wir müssen lediglich `w` initialisieren und dann wiederholt in die dem Gradienten entgegengesetzte Richtung gehen (da der Gradient aufwärts zeigt, wir uns aber abwärts bewegen wollen). Der Hyperparameter `lr` ist immer noch da, gibt jetzt aber an, wie groß jeder einzelne Schritt im Verhältnis zum Gradienten sein soll.

Außerdem müssen wir entscheiden, wann wir aufhören wollen. Die alte Version von `train()` endete, wenn die Höchstzahl der Iterationen erreicht oder es nicht mehr möglich war, den Verlust weiter zu verringern. Beim Gradientenverfahren dagegen kann der Verlust theoretisch immer kleiner werden und sich in immer winzigeren Schritten dem Minimum nähern, ohne es jemals zu erreichen. Wann sollten wir diesen Vorgang abbrechen?

Wir können aufhören, wenn der Gradient ausreichend klein geworden ist, da das bedeutet, dass wir dem Minimum schon sehr nahe gekommen sind. Der vorstehende Code dagegen verfolgt einen weniger raffinierten Ansatz: Wenn Sie `train()` aufrufen, geben Sie an, wie viele Iterationen die Funktion durchlaufen soll. Mehr Iterationen bedeuten einen geringeren Verlust, aber da der Verlust in immer geringem Maße sinkt, ist irgendwann ein Punkt erreicht, an dem eine größere Präzision den Aufwand nicht mehr lohnt.

Weiter hinten in diesem Buch (in Kapitel 15, »*Entwicklung*«) erfahren Sie, wie Sie sinnvolle Werte für Hyperparameter wie `iterations` und `lr` auswählen. Vorläufig habe ich einfach verschiedene Werte ausprobiert und mich für diejenigen entschieden, die zu einem ausreichend geringen Verlust und genügender Genauigkeit führten:

```
X, Y = np.loadtxt("pizza.txt", skiprows=1, unpack=True)
w = train(X, Y, iterations=100, lr=0.001)
print("\nw=%.10f" % w)
```

Bei der Ausführung dieses Codes ergibt sich Folgendes:

```
Iteration    0 => Loss: 812.8666666667
Iteration    1 => Loss: 304.3630879787
Iteration    2 => Loss: 143.5265791020
...
Iteration   98 => Loss: 69.1209628275
Iteration   99 => Loss: 69.1209628275

w=1.8436928702
```

Wie beabsichtigt wird der Verlust mit jedem Durchlauf geringer. Nach 100 Iterationen kommt das Verfahren dem Minimum so nahe, dass wir schon keinen Unterschied mehr zwischen den beiden letzten Verlustwerten erkennen können. Unser Algorithmus scheint seine Aufgabe wie erwartet zu erfüllen.

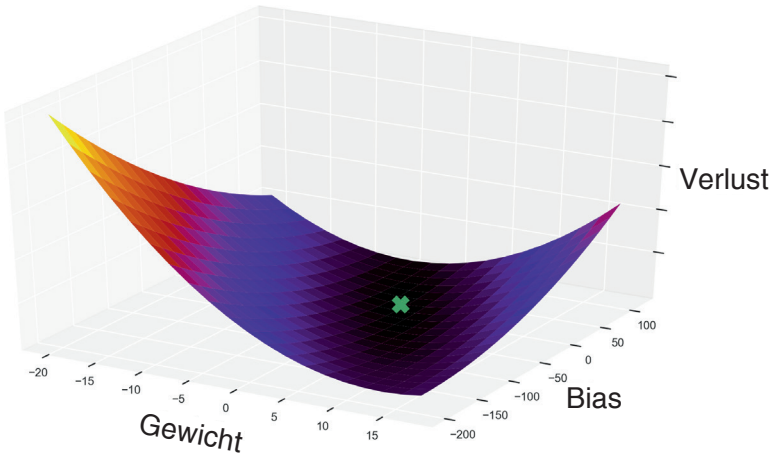
Aber halt – wir haben ja nur den Parameter `w` verwendet! Was passiert, wenn wir `b` wieder mit ins Spiel nehmen?

Die dritte Dimension

Sehen wir uns die Verlustfunktion noch einmal in der mathematischen Schreibweise an:

$$L = \frac{1}{m}\sum_{i=1}^{m}((wx_i + b) - y_i)^2$$

Bis jetzt haben wir bis auf `w` alle Werte in dieser Formel als Konstanten behandelt. Vor allem haben wir `b` auf 0 festgelegt. Wenn wir `b` wieder zu einer Variablen machen, dann ist der Verlust keine zweidimensionale Kurve mehr, sondern eine Oberfläche:

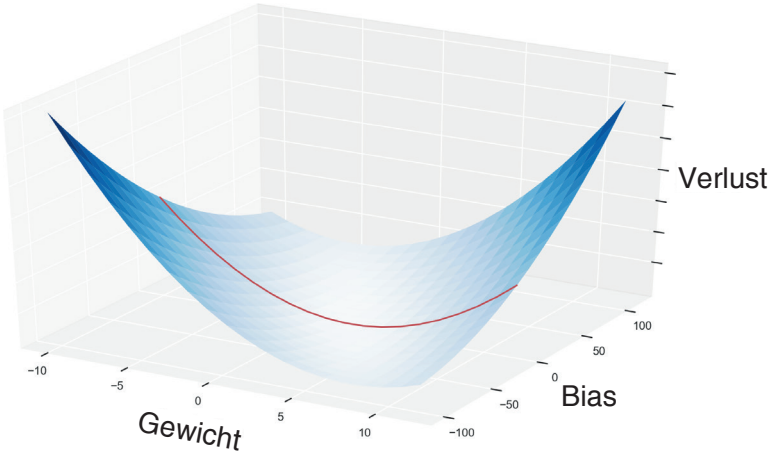


Jetzt lebt unsere Wanderin nicht mehr im Plattland, sondern kann sich in drei Dimensionen bewegen. Die Werte von `w` und `b` bilden die beiden horizontalen Achsen und die Werte des Verlusts die vertikale. Anders ausgedrückt steht jeder Punkt auf dieser Oberfläche für den Fehler in einer Geraden unseres Modells. Wir wollen nun die Gerade mit dem geringsten Fehler finden: den mit dem Kreuz markierten Punkt.

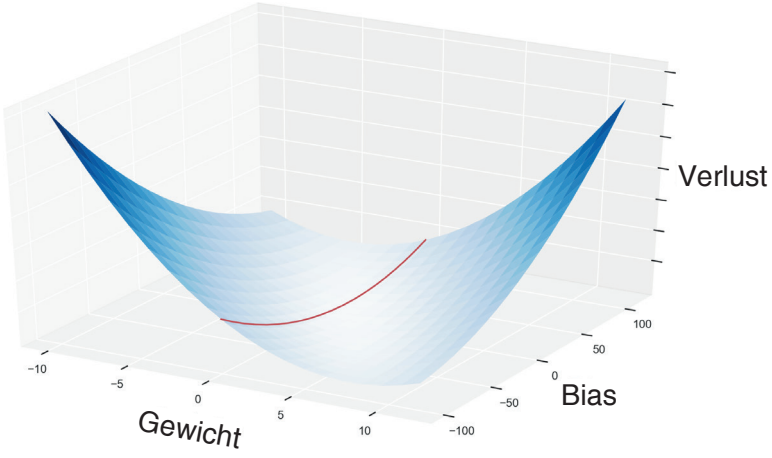
Auch wenn der Verlust jetzt eine Oberfläche ist, können wir das Minimum immer noch mit dem Gradientenverfahren erreichen, wobei wir allerdings den Gradienten einer Funktion mit mehreren Variablen berechnen müssen. Das können wir mit der Technik der *partiellen Ableitung* erreichen.

Partielle Ableitung

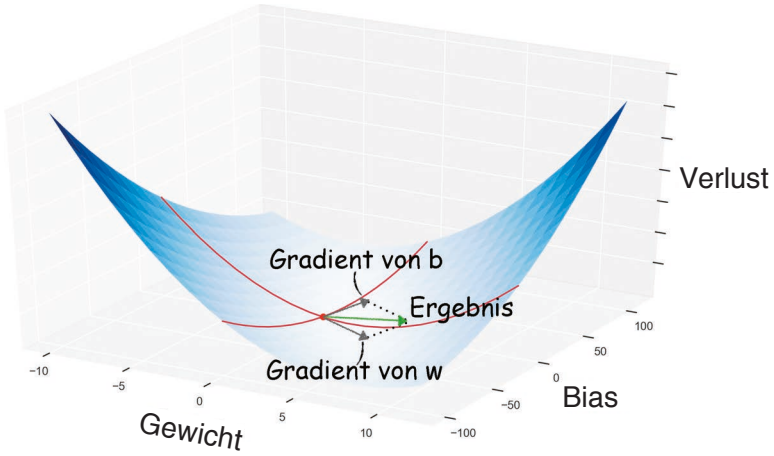
Was ist eine partielle Ableitung, und wie kann sie uns weiterhelfen? Stellen Sie sich vor, dass Sie die Funktion mit einem Samuraischwert aufschlitzen und dann die Ableitung des Schlitzes berechnen. (Es muss kein Samuraischwert sein, aber damit sieht es besonders cool aus.) Beispielsweise haben wir bisher `b` auf 0 festgelegt und den Verlust damit wie folgt aufgeschlitzt:



Dieser Schlitz ist genau die Verlustkurve, die wir bei der ersten Anwendung des Gradientenverfahrens behandelt haben. Sie sieht hier ein bisschen flachgedrückt aus, da ich andere Intervalle und Maßstäbe für die Achsen verwendet habe, aber es ist genau dieselbe Funktion. Für jeden Wert von `b` gibt es eine solche Kurve, deren einzige Variable `w` ist. Ebenso gibt es aber auch für jeden Wert von `w` eine Kurve mit der Variablen `b`. Für `w = 0` sieht sie wie folgt aus:



Für jeden dieser eindimensionalen »Schlitze« können wir den Gradienten berechnen, wie wir es bei unserer ursprünglichen Kurve getan haben. Das Schöne daran ist, dass wir die Gradienten dieser Schlitze kombinieren können, um die Gradienten der Oberfläche zu erhalten, wie die folgende Abbildung zeigt:



Mithilfe von partiellen Ableitungen können wir unser Problem mit zwei Variablen in zwei Probleme mit einer Variablen aufteilen. Wir brauchen also keinen neuen Algorithmus, um das Gradientenverfahren auf einer Oberfläche anzuwenden, sondern können diese Oberfläche einfach mithilfe partieller Ableitungen aufteilen und auf jedem Teil unser bisheriges Verfahren nutzen.

Mathematischer Hintergrund: Ableitungen und Analysis

Der Zweig der Mathematik, der sich mit Gradienten, Ableitungen und partiellen Ableitungen beschäftigt, ist die *Analysis*. Wenn Sie sich intensiver damit beschäftigen wollen, werfen Sie einen Blick auf die Khan Academy.[1] Auch zu diesem Thema finden Sie dort weit mehr Informationen, als Sie zum Verständnis dieses Buchs benötigen.

Um die partiellen Ableitungen konkret zu berechnen, leiten Sie die Funktion jeweils nach einer der Variablen ab (in unserem Fall also `w` oder `b`) und tun dabei so, als ob dies die *einzige* Variable in der Funktion und jeder andere Wert konstant wäre. Die Hälfte dieser Arbeit haben wir bereits erledigt, als wir `b` auf 0 gesetzt und `L` nach `w` abgeleitet haben:

1. *www.khanacademy.org/math/differential-calculus*

$$\frac{\partial L}{\partial w} = \frac{2}{m} \sum_{i=1}^{m} x_i((wx_i + b) - y_i)$$

Jetzt müssen wir das Gleiche noch einmal tun, dabei aber `w` als Konstante betrachten und `L` nach `b` ableiten. Wenn Sie mit Analysis vertraut sind, können Sie sich selbst daran versuchen. Ansonsten finden Sie das Ergebnis hier:

$$\frac{\partial L}{\partial b} = \frac{2}{m} \sum_{i=1}^{m} ((wx_i + b) - y_i)$$

Wie funktioniert das Gradientenverfahren nun auf einer zweidimensionalen Verlustfläche? Unsere Wanderin steht an einem Punkt mit einem gegebenen Wert von `w` und `b` und hat die Formeln für die Berechnung der partiellen Ableitung von `L` nach `w` und `b`. Sie gibt die aktuellen Werte von `w` und `b` in diese Formeln ein und erhält dadurch einen Gradienten für jede Variable. Anschließend wendet sie das Gradientenverfahren auf beide Gradienten an – und schon kann sie dem Gradienten der Oberfläche folgend absteigen.

Das reicht erst einmal an Mathematik für dieses Kapitel. Jetzt wollen wir den Algorithmus in Code umsetzen.

Die Probe aufs Exempel

In der Version mit zwei Variablen sieht unser Code für das Gradientenverfahren wie folgt aus, wobei die geänderten Zeilen wiederum durch kleine Pfeile gekennzeichnet sind:

03_gradient/gradient_descent_final.py

```
import numpy as np

def predict(X, w, b):
    return X * w + b

def loss(X, Y, w, b):
    return np.average((predict(X, w, b) - Y) ** 2)

def gradient(X, Y, w, b):   <
    w_gradient = 2 * np.average(X * (predict(X, w, b) - Y))   <
    b_gradient = 2 * np.average(predict(X, w, b) - Y)   <
    return (w_gradient, b_gradient)   <

def train(X, Y, iterations, lr):
    w = b = 0   <
```

```
    for i in range(iterations):
        print("Iteration %4d => Loss: %.10f" % (i, loss(X, Y, w, b)))   <
        w_gradient, b_gradient = gradient(X, Y, w, b)   <
        w -= w_gradient * lr   <
        b -= b_gradient * lr   <
    return w, b   <

X, Y = np.loadtxt("pizza.txt", skiprows=1, unpack=True)
w, b = train(X, Y, iterations=20000, lr=0.001)   <
print("\nw=%.10f, b=%.10f" % (w, b))
print("Prediction: x=%d => y=%.2f" % (20, predict(20, w, b)))
```

Die Funktion `gradient()` gibt jetzt die partiellen Ableitungen des Verlusts nach `w` und `b` zurück. Anhand dieser Werte verändert `train()` gleichzeitig `w` und `b`. Außerdem habe ich die Anzahl der Iterationen heraufgesetzt, da das Programm jetzt zwei Variablen optimieren muss und daher länger benötigt, um in die Nähe des Minimums zu gelangen.

Um diese neue Version des Programms unter gleichartigen Bedingungen mit derjenigen aus dem vorherigen Kapitel zu vergleichen, führen wir zunächst Letztere mit einer großen Anzahl von Iterationen und einer ziemlich niedrigen `lr` von 0,0001 aus, sodass wir eine Genauigkeit auf vier Dezimalstellen erhalten:

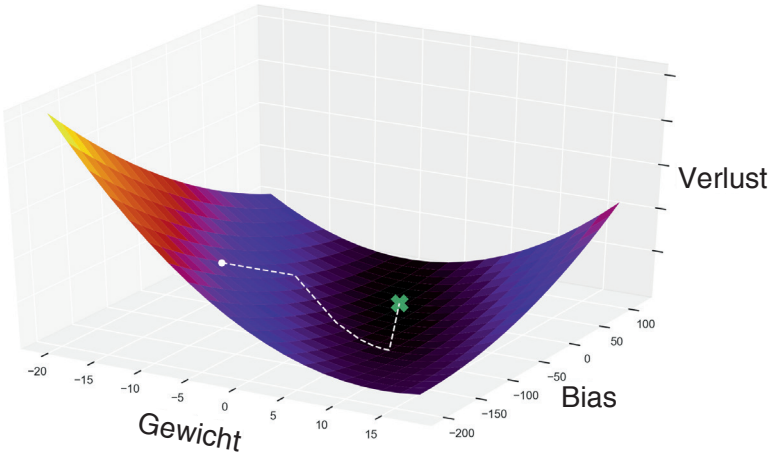
```
...
Iteration 157777 => Loss: 22.842737

w=1.081, b=13.171
Prediction: x=20 => y=34.80
```

Unsere neue Implementierung mit Gradientenverfahren nähert sich dem Ergebnis auf spiralförmigem Kurs. Nach nur 20.000 Iterationen erhalten wir folgendes Resultat:

```
...
Iteration 19999 => Loss: 22.8427367616
w=1.0811301700, b=13.1722676564
Prediction: x=20 => y=34.79
```

Höhere Genauigkeit mit einem Zehntel an Iterationen – großartig! Für unser Pizzavorhersageproblem mag dieses schnellere und genauere ML-Programm zu viel des Guten sein, denn schließlich kauft niemand eine Hundertstel Pizza. Der Geschwindigkeitsgewinn jedoch wird sich bei anspruchsvolleren Problemen noch als äußerst wichtig erweisen.

Abschließend habe ich noch ein kurzes Visualisierungsprogramm geschrieben, um den Pfad auszugeben, den der Algorithmus von einem willkürlichen Ausgangspunkt zum Minimum des Verlusts nimmt. Sie wissen inzwischen zwar schon, wie das Gradientenverfahren abläuft, aber nichts geht über eine grafische Darstellung:



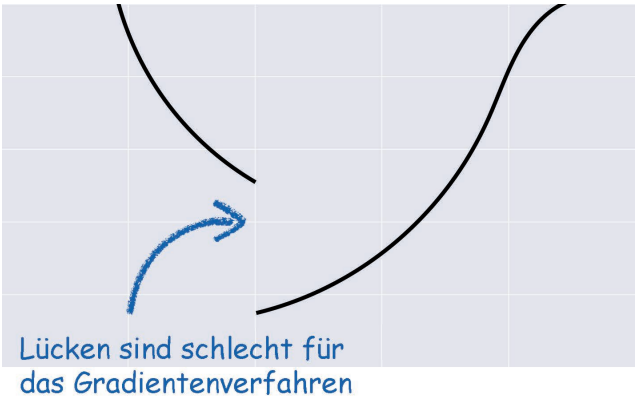
Die Wanderin hat nicht den kürzesten Weg zum Lager genommen, da sie die Route schließlich nicht im Voraus kannte. Stattdessen hat sie sich bei jedem Schritt an der Verlustfunktion orientiert. Nach zwei abrupten Richtungswechseln hat sie endlich die Talsohle erreicht und ist dem sanft absteigenden Pfad zum Lager gefolgt.

Probleme beim Gradientenverfahren

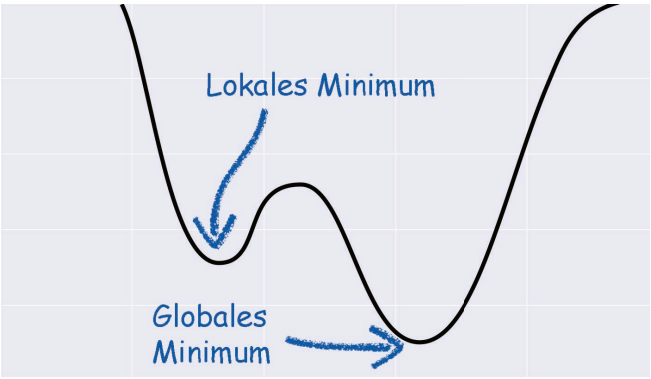
Bei der Anwendung des Gradientenverfahrens gibt es keine Erfolgsgarantie. Die gefundene Route ist nicht unbedingt die kürzeste. Es kann auch sein, dass wir am Lager vorbeigehen und wieder zurückmarschieren müssen oder dass wir uns sogar vom Lager entfernen.

Außerdem gibt es einige unglückselige Fälle, bei denen das Gradientenverfahren das Ziel völlig verfehlt. Einer davon hat mit der Lernrate zu tun. Wir werden ihn uns in der praktischen Übung am Ende dieses Kapitels noch genauer ansehen. Die meisten Probleme beim Gradientenverfahren gehen jedoch auf die Form der Verlustoberfläche zurück.

Mit etwas Fantasie lassen sich durchaus Oberflächen denken, auf denen unsere Wanderin auf dem Weg zum Lager zu Fall kommt. Was passiert, wenn die Verlustoberfläche wie in der folgenden Abbildung einen abrupten Absturz enthält, über dem die Wanderin wie Wile E. Coyote mit den Beinen rudernd in der Luft hängt?



Ein anderes Problem besteht darin, dass die Wanderin statt des *globalen Minimums*, zu dem sie unterwegs ist, nur ein *lokales Minimum* wie in der folgenden Abbildung erreicht.



Am Boden des lokalen Minimums beträgt die Steigung null. Wenn das Gradientenverfahren dort hingelangt, steckt es fest.

Langer Rede kurzer Sinn: Das Gradientenverfahren funktioniert gut, solange die Verlustoberfläche bestimmte Eigenschaften aufweist. Mathematisch ausgedrückt, muss eine gute Verlustfunktion *konvex* sein (keine lokalen Minima aufweisen), *stetig* (frei von Lücken und Abstürzen) und *differenzierbar* (glatt, also ohne Spitzen und andere eigentümliche Stellen, an denen es nicht möglich ist, die Ableitung zu berechnen). Unsere jetzige Verlustfunktion erfüllt alle drei Voraussetzungen, ist also ideal für den Gradientenabstieg geeignet. Wir werden das Verfahren später noch auf andere Funktionen anwenden, die wir dabei zunächst auf die genannten Voraussetzungen überprüfen müssen.

Das Gradientenverfahren ist auch der Hauptgrund dafür, dass wir den Verlust als mittleren quadratischen Fehler implementiert haben. Wir hätten auch den mittleren *Absolutwert* des Fehlers nehmen können, allerdings ist diese Funktion nicht gut für das Gradientenverfahren geeignet, da sie beim Wert 0 eine nicht differenzierbare Spitze aufweist. Außerdem werden die Fehlerwerte durch das Quadrieren noch größer, was dazu führt, dass die Oberfläche sehr steil wird, wenn wir uns vom Minimum entfernen. Umgekehrt bringt dieser steile Verlauf es natürlich mit sich, dass sich das Gradientenverfahren dem Minimum rasant nähert. Aufgrund der Glätte und Steilheit ist der mittlere quadratische Fehler sehr gut für dieses Verfahren geeignet.

Zusammenfassung

In diesem Kapitel haben wir uns mit dem *Gradientenverfahren* beschäftigt, dem am häufigsten verwendeten Algorithmus zur Minimierung des Verlusts. Wie kompliziert unser Modell und die Datenmenge auch sein mögen, das Gradientenverfahren funktioniert immer auf die gleiche Weise: Es geht immer einen Schritt in die entgegengesetzte Richtung des Verlust*gradienten*, bis dieser Gradient sehr klein geworden ist. Um den Gradienten zu finden, berechnen wir die *partiellen Ableitungen* des Verlusts nach `w` und `b`.

Das Gradientenverfahren hat jedoch seine Grenzen. Da es auf Ableitungen basiert, muss die Verlustfunktion glatt und lückenlos sein, damit die Ableitung überall berechnet werden kann. Des Weiteren ist es möglich, dass das Verfahren in einem *lokalen Minimum* stecken bleibt und das *globale Minimum* nicht mehr erreicht. Um diese Probleme zu vermeiden, verwenden wir glatte Verlustfunktionen mit einem einzigen Minimum.

Das Gradientenverfahren ist nicht das Nonplusultra der Algorithmen zur Verlustminimierung. Forscher suchen nach anderen Algorithmen, die für bestimmte Umstände besser geeignet sind. Es gibt auch Variationen des regulären Verfahrens, von denen wir in diesem Buch noch einige kennenlernen werden. Nichtsdestoweniger ist das Gradientenverfahren im modernen ML von entscheidender Bedeutung und wird es auch noch lange bleiben.

Wappnen Sie sich jetzt für eine Herausforderung! Zu Beginn des Kapitels habe ich gesagt, dass es durch Anwendung des Gradientenverfahrens möglich wird, unseren Code auch auf interessantere Modelle zu übertragen und eine Annäherung an kompliziertere Datenmengen zu erreichen. Im nächsten Kapitel werden wir uns ein solches Modell ansehen.

Praktische Übung: Über das Ziel hinaus

Kommen wir noch einmal auf die Lernrate zurück. Im letzten Beispiel dieses Kapitels haben wir eine Lernrate von 0,001 verwendet. Wenn Sie die Lernrate erhöhen, werden Sie feststellen, dass der Verlust schließlich zu *steigen* beginnt anstatt weiter zu sinken. Können Sie sich denken, warum das so ist?

Wenn Sie den Grund durch abstraktes Nachdenken nicht herausfinden können, versuchen Sie, die Verlustfunktion aufzuzeichnen. Was geschieht bei einer sehr großen Lernrate? Die Antwort finden Sie im Verzeichnis `03_gradient/solution`.

7

Die große Herausforderung

Im letzten Kapitel haben wir ein System für maschinelles Sehen entwickelt – allerdings nur ein ziemlich elementares, nämlich einen *binären Klassifizierer*, der die Daten in zwei Klassen einteilt: »Fünf« und »keine Fünf«. Dieses Programm wollen wir nun weiterentwickeln, sodass es jegliche Ziffern aus der MNIST-Datenmenge erkennt.

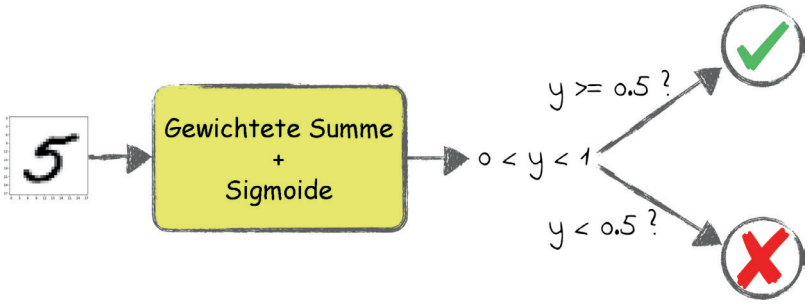
Anders als bei der binären Klassifizierung müssen wir bei dieser Aufgabe mehr als zwei Klassen berücksichtigen, weshalb wir hier von einer *Multiklassen-* oder *Mehrklassen-Klassifizierung* sprechen. Keine Sorge, es gibt ein ganz einfaches Rezept dafür: Wir erstellen für jede Klasse einen binären Klassifizierer und kombinieren anschließend all diese Zweiklassen-Klassifizierer zu einem einzigen Mehrklassen-Klassifizierer.

Sehen wir uns nun an, wie wir diese Idee in Code umsetzen.

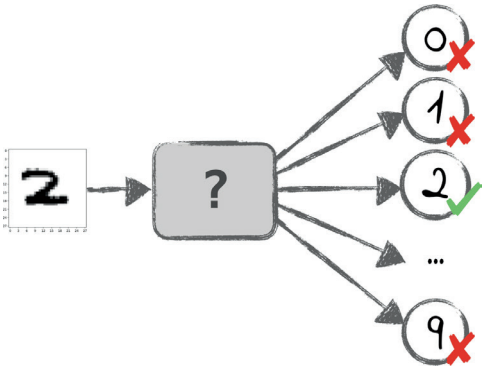
Von zwei zu mehr Klassen

Unser bisheriger binärer Klassifizierer kann eine bestimmte Ziffer erkennen, die hartcodiert vorgegeben ist, in unserem Fall die 5. Die eingespeisten Bilder durchlaufen eine gewichtete Summe und dann eine Sigmoidfunktion, und das Ergebnis

ist eine Zahl zwischen 0 und 1, die wir anschließend auf 0 oder 1 runden, da wir an einem binären Resultat interessiert sind, also an der konkreten Aussage »Ja, das ist eine Fünf« oder »Nein, das ist keine Fünf«. Betrachten Sie dazu die folgende Abbildung:



Unser Ziel besteht nun darin, ein Programm zu schreiben, das ein Bild entgegennimmt und uns sagt, welche der Ziffern 0 bis 9 es darstellt:



Wie kommen wir von unserem bisherigen Programm dorthin? Konzentrieren wir uns zunächst auf den Kasten in der Mitte der ersten Abbildung, der die gewichtete Summe mit anschließender Sigmoidfunktion enthält. Für diese Kombination gibt es keine gängige Bezeichnung; nennen wir sie kurz GSS. Eine GSS ist praktisch ein binärer Klassifizierer ohne den letzten Schritt: Statt 0 oder 1 gibt sie eine Fließkommazahl zwischen 0 und 1 zurück.

Stellen Sie sich jetzt vor, dass wir ein Array aus zehn GSS erstellen, eine pro Klasse von der 0-GSS, die nur Nullen erkennt, bis zur 9-GSS. Wenn wir all diese GSS ausführen, erhalten wir ein Array aus zehn Zahlen:

"0"	"1"	"2"	"3"	"4"	"5"	"6"	"7"	"8"	"9"
0.111	0.005	0.787	0.170	0.001	0.176	0.352	0.001	0.073	0.003

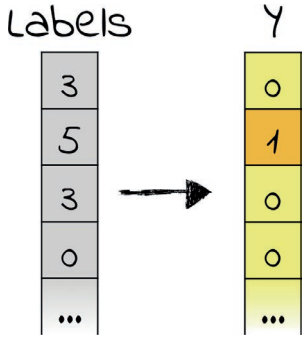
Jede GSS gibt eine Zahl zwischen 0 und 1 zurück, die uns sagt, wie sicher ihre Vorhersage ist. Beispielsweise gibt die 2-GSS in der Abbildung 0,787 zurück, was dem Maximalwert von 1 schon ziemlich nahe kommt. Die 2-GSS ist sich also ziemlich sicher, dass es sich bei dem Bild um eine 2 handelt. Die anderen GSS sind dagegen nicht so überzeugt, dass sie ihre Ziffer erkannt haben. Beispielsweise ist der Rückgabewert der 4-GSS fast 0, sodass wir ziemlich sicher sein können, dass es sich bei dem Bild *nicht* um eine 4 handelt. Insgesamt weist die Vorhersage der 2-GSS die größte Konfidenz von allen auf, sodass es sich bei dem Bild wahrscheinlich um eine 2 handelt.

Das gibt uns einen genaueren Plan, wie wir die Mehrklassen-Klassifizierung angehen müssen: Wir führen zunächst zehn GSS an dem Bild aus, die jeweils auf eine andere Ziffer spezialisiert sind, und entscheiden uns dann für die Ziffer, deren GSS die höchste Konfidenz liefert.

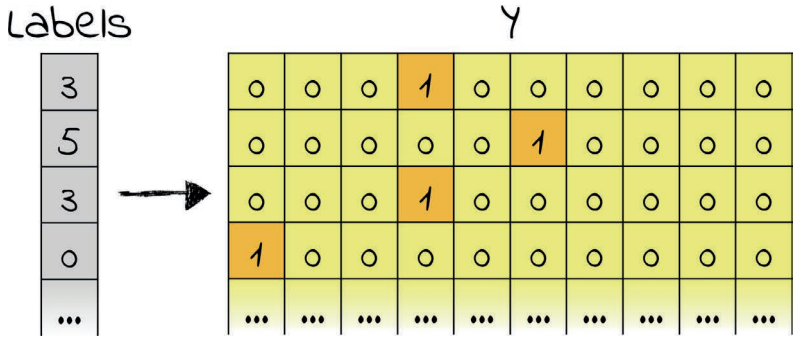
Das könnten wir dadurch implementieren, dass wir zehnmal denselben GSS-Code ausführen, einmal für jede Klasse. Allerdings gibt es eine bessere Möglichkeit.

1-aus-n-Codierung

Als wir im Abschnitt »*Labels laden*« auf *Seite 101* die Labels codiert haben, waren wir nur daran interessiert, die 5 von anderen Ziffern zu unterscheiden. Daher haben wir die 5 durch 1 ersetzt und alle anderen Ziffern durch 0:



Da wir nun zehn solcher Ziffern erkennen müssen, könnten wir dazu zehn solcher codierter Matrizen verwenden, eine pro Klasse. Es gibt aber eine bessere Möglichkeit, nämlich eine große Matrix mit zehn Spalten, in denen jeweils eine Ziffer codiert ist. Das Ergebnis sieht wie folgt aus:



In diesem Beispiel ist das erste Label eine 3, weshalb die vierte Zelle in der ersten Zeile von `Y` den Wert 1 hat. (Es ist die vierte und nicht die dritte Zelle, da wir bei den Ziffern mit 0 anfangen.) Alle anderen Zeilen haben ebenfalls jeweils eine 1 in der Position, die dem ursprünglichen Label entspricht, und überall sonst eine 0.

Diese Vorgehensweise wird als *1-aus-n-Codierung* bezeichnet, da nur jeweils einer der *n* Werte in einer Zeile eine 1 ist.

Können Sie schon erkennen, worauf das hinausläuft? Wir führen unseren Code nicht zehnmal aus, also einmal pro Klasse, sondern nur einmal mit jeweils einer Spalte pro Klasse. Jede Spalte der Matrix enthält praktisch die binäre Codierung für eine der GSS. Dieses Verfahren nutzt die Matrizenmultiplikation und funktioniert genauso gut, als würden wir den Klassifizierer zehnmal ausführen, ist aber viel schneller.

Nicht numerische Labels codieren

Die 1-aus-n-Codierung eignet sich nicht nur für numerische Labels wie die in der MNIST-Datenmenge, sondern für alle Arten von kategorialen Daten. Wenn wir beispielsweise die Labels »Ente«, »Schnabeltier« und »Tapir« hätten, könnten wir sie in einer willkürlichen (aber festen!) Reihenfolge anordnen und als `[1, 0, 0]`, `[0, 1, 0]` und `[0, 0, 1]` codieren. Da die MNIST-Labels zufällig schon Zahlen sind, können wir sie bequem in numerischer Reihenfolge sortieren, allerdings ist das keine Voraussetzung.

Sie können sich die 1-aus-n-Codierung wie ein einfaches Dictionary vorstellen, das ein für Menschen lesbares Label auf eine Zahlenfolge fester Länge aus lauter Nullen und einer Eins abbildet.

Die 1-aus-n-Codierung ist redundant: Statt einer Zahl pro Beispiel haben wir so viele Zahlen, wie es Klassen gibt. Wie wir im nächsten Abschnitt noch sehen werden, ist diese Redundanz jedoch gewöhnlich von Vorteil.

1-aus-n-Codierung in Aktion

Im Abschnitt »*Unsere eigene MNIST-Bibliothek*« auf *Seite 97* haben wir eine Bibliothek geschrieben, um MNIST-Daten zu laden und aufzubereiten. Diese Bibliothek codiert Labels mithilfe der Funktion `encode_fives()`, die wir nun wie folgt ersetzen wollen:

07_final/mnist.py

```
def one_hot_encode(Y):
    n_labels = Y.shape[0]
    n_classes = 10
    encoded_Y = np.zeros((n_labels, n_classes))
    for i in range(n_labels):
        label = Y[i]
        encoded_Y[i][label] = 1
    return encoded_Y
```

`one_hot_encode()` initialisiert eine Matrix aus lauter Nullen mit einer Zeile pro Label und einer Spalte pro Klasse. (`Y.shape[0]` bedeutet »die Anzahl der Zeilen in `Y`«.) Anschließend durchläuft die Funktion die Matrix und ändert die erforderlichen Werte in 1. Damit können wir eine 1-aus-n-Version von `Y_train` erstellen:

```
# 60.000 Labels mit je einer Ziffer von 0 bis 9
Y_train_unencoded = load_labels("../data/mnist/train-labels-idx1-ubyte.gz")

# 60.000 Labels aus jeweils zehn 1-aus-n-codierten Elementen
Y_train = one_hot_encode(Y_train_unencoded)

# 10.000 Labels mit je einer Ziffer von 0 bis 9
Y_test = load_labels("../data/mnist/t10k-labels-idx1-ubyte.gz")
```

Vielleicht wundern Sie sich jetzt, warum wir nur die Trainings-, aber nicht die Testdaten codieren. Diese Frage sorgt oft für Verwirrung, weshalb wir diesen Punkt erst einmal klären müssen. Sehen wir uns dazu die Funktion `classify()` an.

Die Antworten des Klassifizierers decodieren

Überlegen wir noch einmal, wie `classify()` funktioniert. In der Klassifizierungsphase geben die GSS Arrays mit zehn Zahlen zwischen 0 und 1 zurück. Wenn wir das System auffordern, ein Bild zu erkennen, wollen wir diese Arrays aber nicht

sehen, sondern wünschen uns eine klare Aussage wie »3«. Daher müssen wir die Antworten der GSS decodieren, bevor wir sie zurückgeben.

Bisher hat die Funktion `classify()` jedoch nicht viel in Sachen Decodierung unternommen, sondern hat lediglich `forward()` aufgerufen und die Ausgabe gerundet. Jetzt aber hat `classify()` eine viel anspruchsvollere Aufgabe. Sie muss die Ausgabe der GSS wieder in klare Labels zurückverwandeln:

07_final/mnist_classifier.py

```
def classify(X, w):
    y_hat = forward(X, w)
    labels = np.argmax(y_hat, axis=1)
    return labels.reshape(-1, 1)
```

Die erste Zeile von `classify()` berechnet eine Matrix von Voraussagen mit einer Zeile pro Label und einer Spalte pro Klasse. Jede Zeile dieser Matrix enthält zehn Zahlen zwischen 0 und 1.

In der zweiten Zeile ruft die NumPy-Funktion `argmax()` für jede Zeile den Index des Maximalwerts von `y_hat` ab, also den Wert, der 1 am nächsten ist. Standardmäßig sucht `argmax()` das Maximum der gesamten Matrix, weshalb in diesem Code das Argument `axis=1` angegeben wird, um die Funktion auf jede Zeile einzeln anzuwenden. Das Ergebnis ist ein Array aus Indizes, bei denen es sich gleichzeitig um decodierte MNIST-Labels handelt. (Funktionen wie `argmax()` sind etwas gewöhnungsbedürftig. Bevor Sie sie anwenden, sollten Sie sie zunächst in einem interaktiven Python-Interpreter ausprobieren. Auf der anderen Seite sorgen sie für sehr knappen, effizienten Code.)

Schließlich wird das Array `labels` in der letzten Zeile von `classify()` zu einer Matrix mit einer einzigen Spalte umgeformt, die die Spalten enthält. Jetzt wird klar, warum wir die Matrix `Y_test` nicht ebenso wie `Y_tain` 1-aus-n-codiert haben: Da wir `Y_test` mit der Ausgabe des Klassifizierers vergleichen, müssen beide Matrizen das gleiche Erscheinungsbild aufweisen, also nur eine einzige Spalte mit eindeutigen Labels aufweisen.

Wir haben es schon fast geschafft. Es ist nur noch eine weitere Änderung am Code des Klassifizierers erforderlich.

Mehr Gewichte

Bei der Einführung der 1-aus-n-Codierung haben wir die Labelmatrix von eine auf zehn Spalten erweitert. Das müssen wir jetzt auch bei den Gewichten nachholen.

Bis jetzt bestand unsere Gewichtsmatrix aus einer einzigen Spalte sowie einer Zeile pro Eingabevariable. Wir haben sie wie folgt initialisiert:

```
w = np.zeros((X.shape[1], 1))
```

Jetzt brauchen wir zehn Spalten mit Gewichten, eine pro Klasse:

```
w = np.zeros((X_train.shape[1], Y_train.shape[1]))
```

Das neue `w` hat eine Zeile pro Eingabevariable und eine Spalte pro Klasse. Der vorstehende Code bezieht die Anzahl der Variablen und Klassen aus der Anzahl der Spalten in `X` bzw. `Y`.

An dieser Stelle können die Dimensionen all dieser Matrizen schon ziemlich verwirrend wirken. Als ich diesen Abschnitt schrieb, musste ich selbst innehalten, um sie genau zu überprüfen. (»Moment mal, muss `w` jetzt so viele Zeilen haben, wie es Spalten in `X` gibt, oder war es anders herum?«) Nehmen wir uns daher einen Augenblick Zeit, um all diese Zeilen und Spalten einer Plausibilitätsprüfung zu unterziehen.

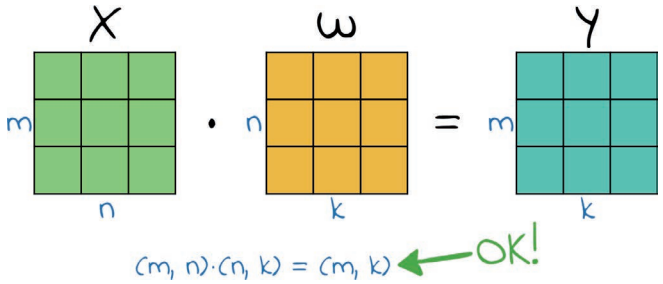
Die Matrixdimensionen überprüfen

Ich erinnere mich noch gern an meine ersten Gehversuche auf dem Gebiet des Machine Learning – bis auf eine frustrierende Erfahrung, die zu vielen Fehlern führte: Es schien, als ob ich die Dimensionen der Matrizen nie richtig hinbekommen würde. Da ich selbst unter so vielen Fehlern aufgrund von nicht zueinanderpassenden Dimensionen gelitten habe, möchte ich Ihnen diese Erfahrung ersparen. Auch auf die Gefahr hin, mich zu wiederholen, gehe ich hier noch einmal auf die Dimensionen unserer Matrizen ein.

Ein Wort vorab: Für mich war es dabei immer sehr hilfreich, die Matrizen auf Papier zu skizzieren. Probieren Sie es ruhig aus!

- `X` hat die Größe `(m, n)` mit einer Zeile pro *Beispiel* und einer Spalte pro *Eingabevariable*. Der MNIST-Trainingsdatensatz besteht aus 60.000 Beispielen mit je 784 Pixeln. Mit der zusätzlichen Biasspalte erhält `X` die Größe `(60.000, 785)`.
- `Y` ist eine Matrix aus 1-aus-n-codierten Labels. Sie hat eine Zeile pro *Beispiel* und eine Spalte pro *Klasse*. Wenn wir die Anzahl der Klassen `k` nennen, ist `Y` also eine `(m, k)`-Matrix. In unserem Fall beträgt ihre Größe `(60.000, 10)`.
- Die Matrix `w` mit den Gewichten hat die Größe `(n, k)` mit einer Zeile pro *Eingabevariable* und einer Spalte pro Klasse. In unserem Fall beträgt die Größe `(785, 10)`. Seit unserem ersten ML-Programm, in dem `w` noch ein einzelner Parameter war, sind wir ein großes Stück vorangekommen, denn jetzt trainieren wir unser System mit Tausenden von Parametern!

Des Weiteren vergewissern wir uns, ob die gewichtete Summe `X · w = Y` die richtigen Dimensionen aufweist. Das ist in der Tat der Fall:



(Eine Erklärung der Matrizenmultiplikation finden Sie im Abschnitt »*Matrizen multiplizieren*« auf *Seite 61*.)

Die hier genannten Zahlen sind die Größen der Matrizen für das Training. Die MNIST-Testdatenmenge ist kleiner, und darum sind auch die Matrizen für diesen Zweck kleiner. Beispielsweise hat `X` beim Testen die Größe `(10.000, 785)`. Wenn wir ein einzelnes Bild klassifizieren, ist `X` eine `(1, 785)`-Matrix mit einer einzigen Zeile.

Nach dieser mühseligen Überprüfung der Matrixdimensionen sind Sie wahrscheinlich begierig darauf, den Klassifizierer endlich auszuführen. Fangen wir also an.

Der Augenblick der Wahrheit

Das ist der Augenblick, auf den wir gewartet haben: Wir werden unseren Multiklassen-Klassifizierer auf die MNIST-Daten loslassen! Der Klassifiziercode in all seiner Pracht sieht wie folgt aus:

07_final/mnist_classifier.py

```
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def forward(X, w):
    weighted_sum = np.matmul(X, w)
    return sigmoid(weighted_sum)

def classify(X, w):
    y_hat = forward(X, w)
    labels = np.argmax(y_hat, axis=1)
    return labels.reshape(-1, 1)
```

```
def loss(X, Y, w):
    y_hat = forward(X, w)
    first_term = Y * np.log(y_hat)
    second_term = (1 - Y) * np.log(1 - y_hat)
    return -np.sum(first_term + second_term) / X.shape[0]

def gradient(X, Y, w):
    return np.matmul(X.T, (forward(X, w) - Y)) / X.shape[0]

def report(iteration, X_train, Y_train, X_test, Y_test, w):
    matches = np.count_nonzero(classify(X_test, w) == Y_test)
    n_test_examples = Y_test.shape[0]
    matches = matches * 100.0 / n_test_examples
    training_loss = loss(X_train, Y_train, w)
    print("%d - Loss: %.20f, %.2f%%" % (iteration, training_loss, matches))

def train(X_train, Y_train, X_test, Y_test, iterations, lr):
    w = np.zeros((X_train.shape[1], Y_train.shape[1]))
    for i in range(iterations):
        report(i, X_train, Y_train, X_test, Y_test, w)
        w -= gradient(X_train, Y_train, w) * lr
    report(iterations, X_train, Y_train, X_test, Y_test, w)
    return w

import mnist as data
w = train(data.X_train, data.Y_train,
          data.X_test, data.Y_test,
          iterations=200, lr=1e-5)
```

Ich habe die Gelegenheit genutzt, die neue Funktion `report()` zu schreiben, die den Prozentsatz der korrekten Ergebnisse protokolliert. `report()` ähnelt unserer bisherigen Funktion `test()`, wird aber einmal pro Trainingsiteration sowie einmal ganz am Ende aufgerufen. Das Protokoll zeigt uns im Einzelnen, wie gut (oder schlecht) unser Klassifizierer lernt.

Eine kleine Korrektur

Man muss schon Adleraugen haben, um es zu bemerken, aber in dem endgültigen Klassifiziercode gibt es eine winzige Änderung in der letzten Zeile von `loss()`. Bis jetzt wurde darin der Durchschnitt von (`first_term + second_term`) mithilfe der NumPy-Funktion `average()` berechnet. Allerdings ermittelt `average()` den Durchschnitt über alle *Elemente* einer Matrix, während wir den Verlust über alle Beispiele brauchen, also über alle *Zeilen* der Matrix.

→

Bis jetzt hat das keine Rolle gespielt, da (`first_term + second_term`) genau ein Element pro Zeile aufwies. Durch die Umstellung auf 1-aus-n-Codierung hat die Matrix nun aber zehn Elemente pro Zeile, weshalb der ursprüngliche Code einen Verlust berechnen würde, der nur ein Zehntel so groß ist wie der tatsächliche Verlust. Selbst mit diesem Bug würde der Klassifizierer immer noch dieselben Gewichte berechnen, da das Minimum des Verlusts an derselben Stelle verbleibt, wenn Sie den Verlust durch 10 teilen. Allerdings wären die während des Trainings angezeigten Angaben über den Verlust falsch.

Dieser versteckte Bug zeigt ein weiteres Mal, dass die Arbeit mit Matrizen voller Stolpersteine steckt. Zumindest ich bin darüber gestolpert, denn dieser Fehler wäre beinahe durch die Überprüfung des Codes in diesem Buch gerutscht. Bei dem neuen Code tritt das Problem nicht auf, da der Durchschnitt jetzt auf die klassische Weise berechnet wird: Er addiert alle Elemente in der Matrix und teilt das Ergebnis durch die Anzahl der Zeilen.

Wenn wir das Programm ausführen, ergibt sich Folgendes:

```
0 - Loss: 6.93147180559945397249, 9.80%
1 - Loss: 8.43445687508333641347, 68.04%
2 - Loss: 5.51204748892387641490, 68.10%
3 - Loss: 2.95687007359365416903, 68.62%
...
200 - Loss: 0.85863196488041293453, 90.32%
```

Klopfen Sie sich ruhig auf die Schulter, denn wir haben gerade Großartiges geleistet: Unser kleines Programm hat in nur wenigen Minuten gelernt, handgeschriebene Ziffern mit einer Genauigkeit von mehr als 90 % zu erkennen.

Ohne den Weißraum umfasst das gesamte Programm lediglich etwa 35 Zeilen Python-Code. Wenn wir auf die Protokollierung verzichten und die Lesbarkeit opfern, könnten wir es auch auf 20 Zeilen eindampfen. Das Programm ist nicht nur kurz, wir haben auch nicht einmal eine ML-Bibliothek dafür verwendet! In diesen Zeilen lauert nirgendwo ein Rattenschwanz von kompliziertem Code. Wir können den gesamten Vorgang problemlos nachvollziehen.

Zusammenfassung

Blicken wir noch einmal auf unser erstes Abenteuer im Land des Machine Learning zurück:

- In Kapitel 1, »*Einführung in Machine Learning*«, haben Sie erfahren, was Machine Learning und *überwachtes Lernen* eigentlich sind.
- In Kapitel 2, »*Ihr erstes ML-Programm*«, haben Sie einen konkreten Vorstoß in das Gebiet des Machine Learning unternommen und mithilfe der *linearen Regression* eine Variable aus einer anderen vorhergesagt.
- In Kapitel 3, »*Am Gradienten entlang*«, haben wir unser ML-Programm mit einem schnelleren und effizienteren Algorithmus versehen: dem *Gradientenverfahren*.
- In Kapitel 4, »*Hyperräume*«, haben wir das Gradientenverfahren (sowie einige Matrizenoperationen) genutzt, um die *multiple lineare Regression* zu implementieren, die der linearen Regression ähnelt, aber mehrere Eingaben verarbeiten kann.
- In Kapitel 5, »*Ein binärer Klassifizierer*«, sind wir von der linearen Regression zur *Klassifizierung* übergegangen.
- In Kapitel 6, »*Eine Aufgabe aus der Praxis*«, haben wir unseren binären Klassifizierer genutzt, um in der MNIST-Datenmenge eine einzige Ziffer zu erkennen.
- In diesem Kapitel haben wir das Programm auf *Mehrklassen-Klassifizierung* erweitert, sodass es alle MNIST-Zeichen mit einer Genauigkeit von mehr als 90 % erkennen kann.

Wir sind schon ganz schön weit gekommen. Nach einer letzten Übung wollen wir dann im nächsten Kapitel eine kleine Pause beim Programmieren einlegen und stattdessen unser bis dahin erstelltes System in den größeren Zusammenhang einordnen.

Praktische Übung: Minensucher

Wenn Sie die Herausforderung lieben, habe ich hier eine weitere Übung für Sie: Ändern Sie den MNIST-Klassifizierer so ab, dass er die Sonar-Datenmenge[1] verarbeiten kann. Diese Datenmenge (auch als »Mines vs. Rocks« bezeichnet) enthält die Muster von Sonarsignalen, die von zwei verschiedenen Arten von Objekten zurückgeworfen werden, nämlich Metallzylindern, bei denen es sich um Minen

1. *https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks)*

handeln kann, und Felsen. Können Sie den Klassifizierer so trainieren, dass er in der Lage ist, Minen von Felsen zu unterscheiden?

Unterschätzen Sie dieses kleine Projekt nicht. Wenn Sie in Python nicht firm sind, kann es Sie einige Stunden kosten, um das Sonar-Gegenstück zur Bibliothek `mnist.py` zu schreiben, und dabei müssen Sie wahrscheinlich in der Dokumentation sowohl von Python als auch von NumPy nachschlagen. Versuchen Sie sich nur an dieser Übung, wenn Sie auch bereit sind, die dafür erforderliche Zeit aufzubringen.

Wenn Sie die Herausforderung annehmen, dann besuchen Sie als Erstes die Website mit der Sonar-Datenmenge. Laden Sie die Beispiele in `sonar.all-data` herunter und lesen Sie die Dokumentation in `sonar.names`.

Hier noch einige Tipps:

- Es sollte nicht erforderlich sein, irgendetwas außer den Hyperparametern am Code des eigentlichen Klassifizierers zu ändern. Sie sollten nur den Code zum Laden und Aufbereiten der Daten ersetzen müssen.
- Die Sonar-Datenmenge umfasst 208 Beispiele. Wie Sie sie in Trainings- und Testdaten aufteilen, ist Ihre Sache. Ich habe 48 Beispiele für Testzwecke reserviert.
- Die Beispiele in der Sonar-Datenmenge sind geordnet. Zuerst kommen die Felsen, dann die Minen. Bevor Sie die Menge aufteilen, müssen Sie sie erst durcheinanderwürfeln, da Ihre Testdatenmenge sonst nur Beispiele einer einzigen Klasse enthält.
- Denken Sie daran, sowohl `X_train` als auch `X_test` eine Biasspalte hinzuzufügen.
- Denken Sie daran, eine 1-aus-n-Codierung für `Y_train` vorzunehmen, aber nicht für `Y_test`.
- Probieren Sie als Erstes eine Lernrate von 0,01 aus und ändern Sie den Wert, wenn Sie mit dem Ergebnis nicht zufrieden sind. Eine zu große Lernrate kann zu Fehlern bei der Berechnung des Verlusts führen, da Logarithmen und Exponentialfunktionen sehr große (oder kleine) Zahlen hervorrufen.
- Wenn Sie ein System zu lange trainieren, kann seine Genauigkeit *sinken*, anstatt zu steigen. Das ist eine Auswirkung der *Überanpassung*, die wir bereits im Abschnitt »*Trainings- und Testdatensatzsatz*« auf *Seite 96* besprochen haben. Wie sich dieses Problem vermeiden lässt, sehen wir uns in Teil III dieses Buchs an.

- Denken Sie daran, dass dies ein Problem der binären Klassifizierung ist, weshalb Sie nicht unbedingt alle zusätzlichen Feinheiten der Multiklassen-Klassifizierung berücksichtigen müssen, etwa die 1-aus-n-Codierung. Allerdings habe ich die Datenmenge trotzdem 1-aus-n-codiert, damit ich denselben Multiklassen-Klassifizierer wie für die MNIST-Daten anwenden konnte. Alternativ können Sie auch den binären Klassifizierer aus Kapitel 5, »*Ein binärer Klassifizierer*«, verwenden. Statt der 1-aus-n-codierten Labels, die entweder `[1, 0]` oder `[0, 1]` sind, können Sie dann auf die binärcodierten Labels zurückgreifen, also einfach 0 oder 1.

Versuchen Sie, eine Genauigkeit von 75 % oder mehr zu erreichen. Wenn Sie nicht mehr weiterkommen sollten, können Sie sich meine Lösung im Verzeichnis `07_final/solution` ansehen.

11

Das Netz trainieren

Im letzten Kapitel haben wir ein funktionierendes neuronales Netz geschrieben, wobei ich mit »funktionierend« meine, dass der Vorhersagecode fertig ist: Das Netz kann Daten durch das Modell schleusen und Labels ausgeben. Dafür sind allerdings Gewichtungen erforderlich, wobei wir den Code zur Ermittlung dieser Gewichtungen erst noch schreiben müssen. Das werden wir in diesem Kapitel mit der Funktion `train()` erledigen.

In der Frühzeit neuronaler Netze war das Training ein kniffliges Problem. KI-Experten zweifelten sogar daran, ob ein solches Training überhaupt möglich wäre. Die Antwort erhielten sie Anfang der 70er-Jahre, als Forscher eine Möglichkeit herausfanden, um den Gradienten eines Netzes mithilfe des Algorithmus der *Backpropagation* zu berechnen.

In Projekten in der Praxis werden Sie die Backpropagation wahrscheinlich niemals selbst implementieren müssen. Moderne ML-Bibliotheken bringen bereits ihre eigenen fertigen Implementierungen mit. Es ist jedoch wichtig, zu verstehen, wie die Backpropagation funktioniert, damit Sie mit allen noch so feinen Auswirkungen umgehen können.

In diesem Kapitel schauen wir uns an, wie Backpropagation funktioniert, und implementieren sie für unser neuronales Netz. Sie werden hier auch lernen, wie

Sie die Gewichte des Netzes auf eine für die Backpropagation geeignete Weise initialisieren. Anschließend führen wir unser Netz zum ersten Mal aus. Kann es die Genauigkeit des Perzeptrons überbieten, und wenn ja, wie sehr?

Eine kleine Warnung vorab

Ich habe schon Einführungen in Backpropagation gelesen, die mit der dreisten Behauptung begannen, Backpropagation sei einfach. Dem kann ich nicht zustimmen. Im Gegenteil, Backpropagation ist ziemlich schwierig. Einfach erscheint sie erst im Rückblick, wenn Sie bereits wissen, wie sie funktioniert.

Um die Anfangsschwierigkeiten zu überwinden, bauen wir in diesem Kapitel vor allem auf Anschauung. Wir beschäftigen uns mit den allgemeinen Prinzipien der Backpropagation, lassen aber einige Details und langwierige Berechnungen aus.

Ich halte diese Erklärungen so einfach wie möglich, aber nicht noch einfacher. Neben Kapitel 4, »*Hyperräume*«, ist dies das mathematiklastigste Kapitel im ganzen Buch. Lassen Sie sich nicht entmutigen, wenn Sie nicht alles auf Anhieb verstehen. Die meisten Leute brauchen einige Zeit, um mit Backpropagation klarzukommen. Mir ist es selbst so ergangen.

Wozu Backpropagation?

Bevor ich die Backpropagation erkläre, wollen wir uns ansehen, warum wir sie überhaupt brauchen.

Zunächst einmal habe ich eine gute Nachricht für Sie: In gewissem Sinne wissen Sie bereits, wie ein neuronales Netz trainiert wird, nämlich ebenso wie ein Perzeptron mithilfe des Gradientenverfahrens. Bei jeder Iteration berechnen Sie den Gradienten des Verlusts und folgen dann diesem Gradienten, um den Verlust zu minimieren (siehe Kapitel 3, »*Am Gradienten entlang*«).

Die weniger gute Nachricht dagegen lautet, dass der Abstieg am Gradienten der einfache Teil der Aufgabe ist. Schwierig ist es dagegen, diesen Gradienten zu berechnen.

Beim Perzeptron haben wir dazu einfach die Ableitungen des Verlusts nach den Gewichten berechnet. (In Wirklichkeit haben wir die Ableitungen in einem Lehrbuch nachgeschlagen, aber es läuft auf dasselbe hinaus.) Im Falle eines neuronalen Netzes kann es jedoch sehr schwer sein, diese Ableitungen zu bestimmen. So sieht beispielsweise der Code zur Berechnung des Verlusts in unserem dreischichtigen Netz wie folgt aus:

```
h = sigmoid(matmul(X, w1))
y_hat = softmax(matmul(h, w2))
L = cross_entropy_loss(Y, y_hat)
```

Stellen Sie sich nun vor, Sie müssten diesen Code in mathematische Formeln übersetzen und diese dann nach `w1` und `w2` ableiten. Selbst für jemanden, der sich in Analysis gut auskennt, bringt das ziemlich viel Arbeit mit sich.

In unserem Beispielnetz könnten wir uns noch auf diese Weise durchkämpfen, um den Gradienten zu erhalten, doch in einem typischen modernen Netz wäre diese Aufgabe noch viel schwieriger. Neuronale Netze in der Praxis können unglaublich kompliziert sein und aus Dutzenden von verzwickt miteinander verbundenen Schichten und Gewichtsmatrizen bestehen. Bei einem solchen großen Netz wäre es schon schwer, die Verlustfunktion zu formulieren, geschweige denn, ihre Ableitungen zu berechnen.

Damit stehen wir vor einem Dilemma: Auf der einen Seite möchten wir in der Lage sein, den Verlustgradienten beliebiger neuronaler Netze zu bestimmen, auf der anderen Seite ist die Berechnung der Ableitungen außer für die einfachsten und leistungsärmsten Netze impraktikabel.

An dieser Stelle kommt nun die Backpropagation ins Spiel. Sie hilft uns aus dem Dilemma, denn mit diesem Algorithmus lassen sich die Verlustgradienten beliebiger neuronaler Netze bestimmen. Sobald wir diese Gradienten haben, können wir ihnen mit unserem bewährten Gradientenverfahren folgen.

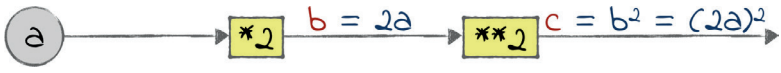
Schauen wir uns nun an, wie die Backpropagation funktioniert.

Von der Kettenregel zur Backpropagation

Die Backpropagation ist eine Anwendung der *Kettenregel*, einer der grundlegendsten Regeln der Analysis. Im Folgenden sehen wir uns an, wie sich diese Regel auf eine einfache und eine kompliziertere netzartige Struktur auswirkt.

Die Kettenregel in einem einfachen Netz

Schauen Sie sich die folgende einfache netzähnliche Struktur an:



Dies ist kein neuronales Netz, da es keine Gewichte hat. Borgen wir uns hierfür einen Begriff aus der Informatik aus und nennen wir dieses Gebilde einen *Rechengraphen*. Er hat eine Eingabe `a`, auf die zwei Operationen folgen, nämlich eine Multiplikation mit 2 und eine Quadrierung. Die Ausgabe der Multiplikation heißt `b`, die Ausgabe des gesamten Graphen `c`.

Nehmen wir nun an, wir möchten $\partial c/\partial a$ berechnen, also den Gradienten von `c` nach `a`. Anschaulich steht dieser Gradient für den Einfluss von `a` auf `c`: Wenn sich `a` ändert, so ändert sich auch `c`, wobei der Gradient angibt, wie sehr. (Wenn Ihnen

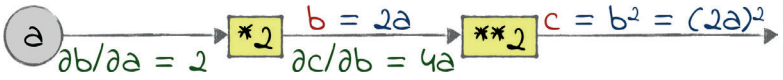
das verwirrend vorkommt, blättern Sie noch einmal zu »*Das Gradientenverfahren*« auf *Seite 43* zurück.)

Bei einem so kleinen Graphen könnten wir ∂c/∂a auf einen Rutsch berechnen, indem wir die Ableitung von c nach a bestimmen. Wie bereits erwähnt, wäre das bei umfangreicheren Graphen jedoch impraktikabel. Stattdessen bestimmen wir den Gradienten mithilfe der Kettenregel, die sich für Graphen beliebiger Größe eignet.

Nach dieser Regel müssen wir zur Berechnung von ∂c/∂a wie folgt vorgehen:

1. Wir durchlaufen den Graphen rückwärts von c nach a.
2. Wir berechnen den *lokalen Gradienten* für jede dazwischen liegende Operation, also die Ableitung der Ausgabe dieser Operation nach deren Eingabe.
3. Wir multiplizieren alle lokalen Gradienten.

Sehen wir uns diesen Vorgang nun an einem praktischen Beispiel an. In unserem Fall enthält der Pfad von c zurück zu a zwei Operationen, nämlich die Quadrierung und die Multiplikation mit 2. Wenn wir die lokalen Gradienten jeweils bei ihren Operationen notieren, erhalten wir Folgendes:



Woher weiß ich, dass ∂b/∂a gleich 2 ist und ∂c/∂b gleich 4a? Nun, auch wenn wir die Kettenregel anwenden, müssen wir die lokalen Gradienten immer noch auf althergebrachte Weise bestimmen, indem wir die Ableitungen ausrechnen. Machen Sie sich keine Sorgen, wenn Sie nicht wissen, wie das geht, denn es gibt Bibliotheken, die das für Sie erledigen. Es reicht, wenn Sie wissen, wie der Gesamtvorgang abläuft.

Anschließend multiplizieren Sie die lokalen Gradienten, um ∂c/∂a zu erhalten:

$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b} \cdot \frac{\partial b}{\partial a} = 4a \cdot 2 = 8a$$

Dank der Kettenregel haben wir hier also unsere Antwort: Der Gradient von c nach a ist 8a. Mit anderen Worten, wenn sich a ein kleines bisschen ändert, so ändert sich c um das Achtfache des aktuellen Werts von a.

Zusammengefasst lautet die Kettenregel also: Um den Gradienten eines beliebigen Knotens y nach einem anderen Knoten x zu berechnen, multiplizieren wir die lokalen Gradienten aller Knoten, die auf dem Weg von y zurück zu x liegen.

Damit können wir auch komplizierte Gradienten durch die Multiplikation vieler einfacher Gradienten bestimmen.

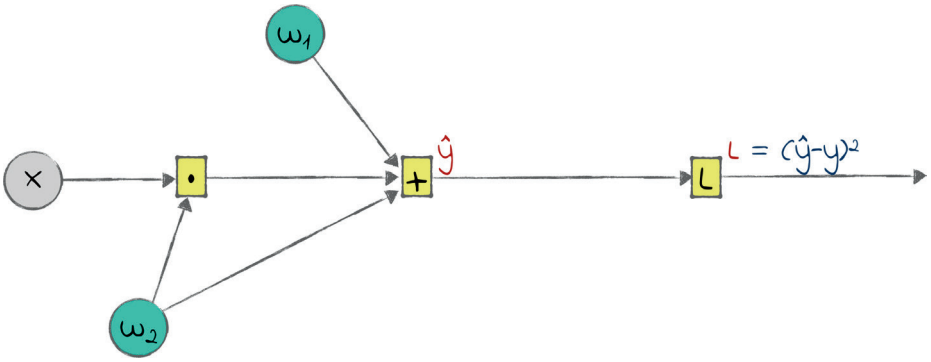
Als Nächstes wenden wir die Kettenregel auf ein neuronales Netz an.

Mathematischer Hintergrund: Die Kettenregel

Im Bereich der Khan Academy zur Analysis finden Sie auch Videos zur Kettenregel.[1] Der Stoff geht auch hier weit über das hinaus, was Sie zum Verständnis dieses Kapitels benötigen, aber wenn Sie sich für Mathematik interessieren, lohnt es sich bestimmt, sich diese Filme anzusehen.

Es wird komplizierter

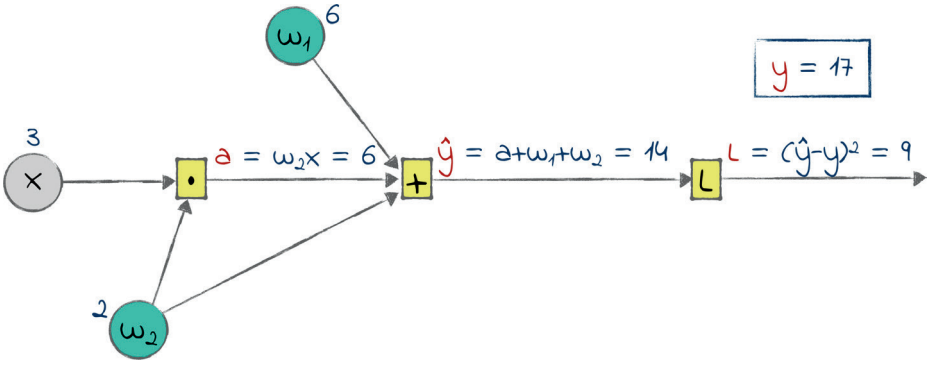
»Backpropagation« heißt im Grunde genommen: »Berechne die Verlustgradienten eines neuronalen Netzes nach den Gewichten mithilfe der Kettenregel.« Betrachten Sie als Beispiel dafür den folgenden zweiten Rechengraphen:



Mit diesem Graphen werden wir natürlich keinen ML-Wettbewerb gewinnen können. Man könnte sogar behaupten, dass er gar kein neuronales Netz darstellt. Er hat jedoch alles, was er braucht, um in diesem Beispiel die Rolle eines neuronalen Netzes zu spielen, nämlich eine Eingabe `x`, eine Ausgabe $\hat{y}$, zwei Gewichte und einen Verlust `L`, der als quadratischer Fehler der Differenz zwischen $\hat{y}$ und der Grundwahrheit `y` berechnet wird.

Stellen Sie sich nun vor, dass Sie dieses Netz mitten im Training kurz vor dem nächsten Schritt des Gradientenverfahrens einfrieren. Nehmen wir an, dass `w1` und `w2` zurzeit den Wert 6 bzw. 2 aufweisen. Außerdem haben wir nur ein einziges Trainingsbeispiel mit `x = 3` und `y = 17`. Aus diesen Zahlen können wir nun die anderen Werte im Graphen berechnen:

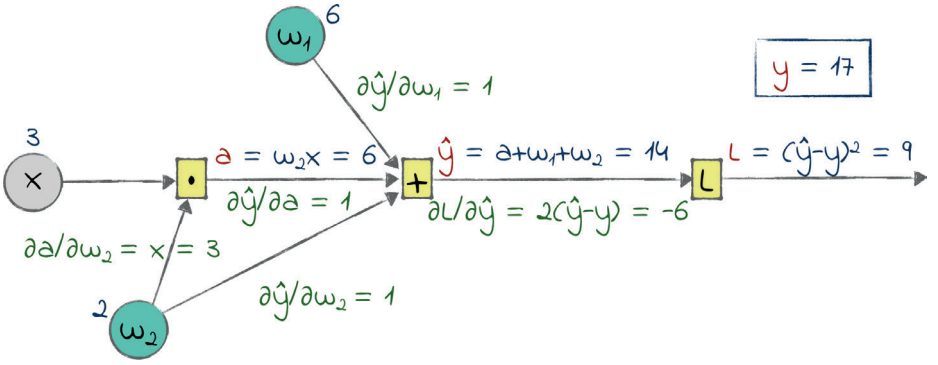
1. *www.khanacademy.org/math/differential-calculus/dc-chain*



Die Ausgabe der Multiplikation habe ich hier `a` genannt, weil ich noch keinen anderen Namen dafür hatte.

Für einen Schritt des Gradientenverfahrens benötigen wir die Gradienten von `L` nach w_1 und w_2, die wir mit der Kettenregel berechnen können. Danach ist $\partial L/\partial w_1$ das Produkt der lokalen Gradienten auf dem Weg von `L` zurück nach w_1. Für $\partial L/\partial w_2$ gilt das Gleiche. Berechnen wir nun also diese lokalen Gradienten.

Im Vergleich zu dem Graphen aus dem vorherigen Abschnitt weist dieser hier eine zusätzliche Schwierigkeit auf: Einige der Operationen haben mehrere Eingaben. In einem solchen Fall müssen wir den lokalen Gradienten für jedes Eingabe-Ausgabe-Paar berechnen. Damit benötigen wir hier fünf lokale Gradienten:



Jetzt können wir die Kettenregel anwenden. Als Erstes berechnen wir $\partial L/\partial w_1$. Genau genommen, ist dies der Gradient von `L` nach w_1, aber weil das ziemlich umständlich klingt, sprechen ML-Experten meistens nur vom »Gradienten von w_1«.

Um diesen »Gradienten von w_1« zu bekommen, multiplizieren wir alle Gradienten zwischen `L` und w_1:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1} = -6 \cdot 1 = -6$$

Mit den aktuellen Gewichten erhalten wir –6 für den Gradienten von w_1. Für den nächsten Schritt im Gradientenverfahren müssen wir diesen Gradienten mit der Lernrate multiplizieren und das Ergebnis von w_1 subtrahieren.

Für $\partial L / \partial w_2$ gehen wir auf die gleiche Weise vor, wobei sich uns allerdings eine zusätzliche Schwierigkeit stellt: Es führen zwei Pfade von w_2 nach L, von denen der eine die Multiplikation durchläuft und der andere nicht. Wenn mehrere Pfade vorliegen, müssen wir deren Gradienten addieren:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2} + \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{\partial w_2} = -6 \cdot 1 + -6 \cdot 1 \cdot 3 = -24$$

$\partial L / \partial w_2$ ist also negativ, aber größer als $\partial L / \partial w_1$. Auch hier multiplizieren wir den Gradienten mit der Lernrate und subtrahieren das Ergebnis von w_2.

Sie können sich $\partial L / \partial w_1$ und $\partial L / \partial w_2$ als Maße dafür vorstellen, wie viel jedes Gewicht zum Verlust beiträgt. Beide Gewichte leisten hier einen negativen Beitrag, was bedeutet, dass sie wachsen müssen, um den Verlust kleiner zu machen. Allerdings trägt w_2 mehr bei als w_1, da es in der Berechnung von $\hat{y}$ zweimal vorkommt, nämlich einmal in der Multiplikation und einmal in der Summe.

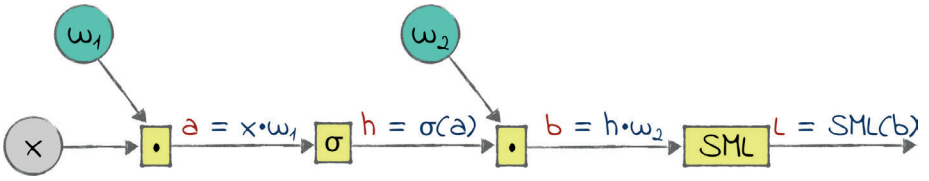
Wenn ein Gewicht einen kleinen Gradienten hat, bedeutet das im Allgemeinen, dass es nicht viel zum Fehler des Netzes beiträgt, sodass es sich nur ein kleines bisschen ändern muss. Ein Gewicht mit einem großen Gradienten dagegen hat großen Einfluss auf den Fehler und muss daher entschiedener geändert werden. Die Backpropagation ist eine Möglichkeit, um zu berechnen, wie stark die Gewichte angepasst werden müssen.

Die Backpropagation ist also ein Algorithmus, bei dem die Gradienten der Gewichte durch Multiplikation der lokalen Gradienten an den einzelnen Operationen berechnet werden. Ihr Name bezieht sich darauf, dass sie in entgegengesetzter Richtung zur Forward-Propagation verläuft. Die Forward-Propagation bewegt sich von den Eingaben zu den Ausgaben und berechnet letzten Endes den Verlust, die Backpropagation dagegen geht vom Verlust zu den Gewichten und erfasst dabei anhand der Kettenregel alle lokalen Gradienten.

Als Nächstes wollen wir die Backpropagation auf unser Netz anwenden.

Backpropagation anwenden

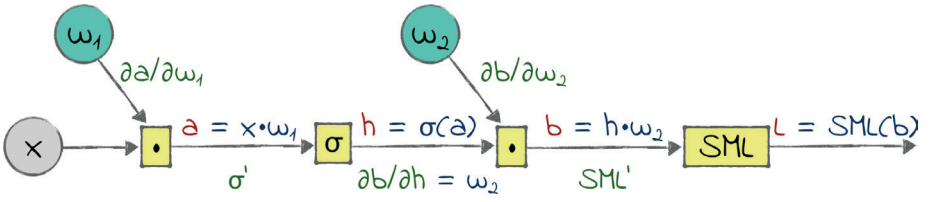
In Form eines Rechengraphen sieht unser dreischichtiges neuronales Netz wie folgt aus:



Mit Ausnahme des Verlusts `L` sind alle Variablen in diesem Graphen Matrizen. Das Symbol σ steht für die Sigmoide. Aus Gründen, die ich in Kürze darlegen werde, habe ich die Softmax-Funktion und die Kreuzentropie zu einer einzigen Operation namens `SML` zusammengefasst. Die Ausgaben der Matrizenmultiplikationen habe ich `a` und `b` genannt.

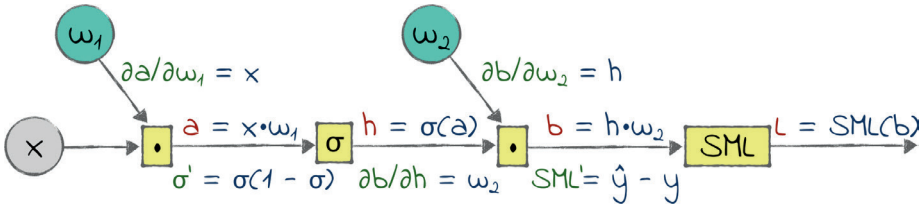
Dieses Diagramm stellt genau das neuronale Netzwerk dar, das wir in den zwei vorausgehenden Kapiteln entworfen und erstellt haben. Von links nach rechts laufen die folgenden Operationen ab: `x` und w_1 werden multipliziert und an die Sigmoide übergeben, woraus die verdeckte Schicht `h` entsteht, und daraufhin wird `h` mit w_2 multipliziert und an die Softmax- und die Verlustfunktion übergeben, was den Verlust `L` ergibt.

Kommen wir nun aber zu dem Vorgang, dessentwegen wir uns überhaupt erst mit der Backpropagation beschäftigt haben: Wir wollen die Gradienten von `L` nach w_1 und w_2 bestimmen. Um die Kettenregel anwenden zu können, brauchen wir die lokalen Gradienten entlang des Wegs von `L` nach w_1 und w_2:



σ' und `SML'` sind die Ableitungen der Sigmoide bzw. der SML-Operation. In diesem Buch verwende ich für Ableitungen meistens Terme wie $\partial\sigma/\partial a$, aber wenn eine Funktion wie hier nur eine Variable hat, kann man die Ableitung auch einfach mit einem Apostroph kennzeichnen.

Nun aber einmal kräftig in die Hände gespuckt, um die lokalen Gradienten zu berechnen. Auch hier müssen Sie die Ableitungen nicht selbst ausrechnen – das habe ich schon für Sie erledigt. Die Ergebnisse sehen wie folgt aus:



Im Abschnitt »*Kreuzentropie*« auf *Seite 148* habe ich schon gesagt, dass die Softmax-Funktion und die Kreuzentropie wie füreinander gemacht sind. Nun zeigt sich auch, warum das so ist. Jede Funktion für sich allein hat eine komplizierte Ableitung, aber wenn Sie sie kombinieren, ergibt sich eine ganz einfache Ableitung. Der gemeinsame Gradient lässt sich kostengünstig als Netzwerkausgabe minus Grundwahrheit berechnen. Wenn Sie sich für den mathematischen Hintergrund dieser Ableitung interessieren, können Sie sich die schrittweise Erklärung auf ProgML[2] ansehen.

Siehe »Killer Combo: Softmax and Cross Entropy« auf ProgML.

Die Ableitungen der Matrizenmultiplikationen lassen sich ebenfalls kurz und schmerzlos bestimmen. Wenn Sie wissen, wie Sie die Ableitung einer Skalarmultiplikation berechnen, sind Sie fein raus, denn die Ableitung der Matrizenmultiplikation sieht genauso aus.

Die Ableitung der Sigmoide habe ich in einem Lehrbuch der Mathematik nachgeschlagen. Sie hat eine merkwürdige Form, da die Sigmoide selbst in dem Ausdruck vorkommt.

Da die lokalen Gradienten nun bekannt sind, können wir die Kettenregel anwenden, um die Gradienten der Gewichte zu berechnen. Diesmal erledigen wir das aber nicht auf dem Papier, sondern schreiben Code dafür.

Auf Kurs bleiben

Im Folgenden schreiben wir eine Backpropagation-Funktion, die die Gradienten der Gewichte in unserem neuronalen Netz berechnet. Vorab aber eine kleine Warnung: Die Funktion ist zwar kurz, aber kompliziert.

Unser Code wendet die Kettenregel unmittelbar an. Wenn Sie mit skalaren Gradienten zu tun haben, lässt sich diese Regel ganz einfach nutzen, aber bei Matrizen wird es kompliziert. Damit die Multiplikationen in einem neuronalen Netz zulässig sind, müssen wir die Operanden so vertauschen und transponieren, dass die Dimensionen der Matrizen zusammenpassen.

2. *https://www.progml.com*

Dieses Jonglieren mit Matrizen lässt sich einfacher durchführen als beschreiben. Wenn Sie Ihren eigenen Code für die Backpropagation schreiben, können Sie die Berechnungen in einem Interpreter überprüfen und die Dimensionen der Matrizen als Orientierungshilfe dafür heranziehen, welche Matrizen Sie transponieren müssen, in welcher Reihenfolge sie zu multiplizieren sind usw. Bei einer schriftlichen Erklärung nehmen die Begründungen für diese Operationen jedoch sehr viel Platz ein.

Sie haben nun folgende Möglichkeiten:

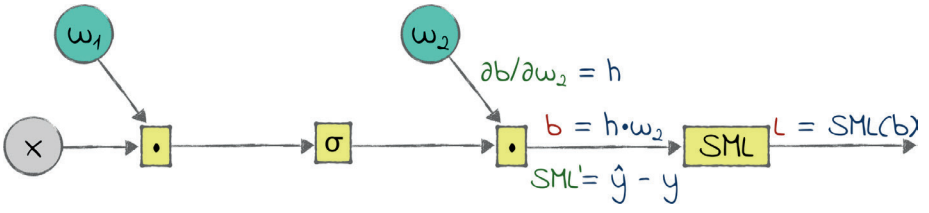
1. Sie können den Abschnitt lesen, um sich anzusehen, wie der Backpropagation-Code im Großen und Ganzen geschrieben wird, die kniffligen Einzelheiten aber überfliegen.
2. Sie können die folgenden Seiten sehr aufmerksam lesen und die Matrizenoperationen dabei selbst auf Papier oder in einem Python-Interpreter nachvollziehen.
3. Sie können den Backpropagation-Code auch komplett selbst schreiben und nur nach Tipps auf den folgenden Seiten Ausschau halten, wenn Sie nicht weiterkommen. Das stellt zwar eine große Herausforderung dar, ist aber die beste Möglichkeit, um den Code in allen Einzelheiten zu begreifen.

Welche Möglichkeit Sie auch immer wählen, sollten Sie sich nicht entmutigen lassen, wenn Sie auf den folgenden Seiten mit einigen Grenzfällen zu kämpfen haben. Diese Sonderfälle sind nicht das, worum es in diesem Kapitel geht. Der Schwerpunkt liegt darauf, das Prinzip der Backpropagation zu verstehen und eine Vorstellung davon zu bekommen, wie der Code dafür zustande kommt.

Damit hätten wir die Vorbemerkungen aus dem Weg geschafft. Schreiben wir nun den Code zur Berechnung des Gradienten von w_2.

Den Gradienten von w_2 berechnen

Zur Berechnung von $\partial L/\partial w_2$ benötigen wir die folgenden lokalen Gradienten:



Die Anwendung der Kettenregel ergibt hier Folgendes:

$$\frac{\partial L}{\partial w_2} = \mathrm{SML}' \cdot \frac{\partial b}{\partial w_2} = (\hat{y} - y) \cdot h$$

Als Code sieht die Formel wie folgt aus:

11_training/backpropagation.py

```
w2_gradient = np.matmul(prepend_bias(h).T, y_hat - Y) / X.shape[0]
```

Ich musste hier die Operanden der Multiplikation vertauschen und einen von ihnen transponieren, damit das Ergebnis dieselben Dimensionen bekommt wie w_2. Da wir den Gradienten anschließend mit der Lernrate multiplizieren und das Ergebnis von w_2 subtrahieren wollen, müssen die beiden Matrizen von derselben Form sein.

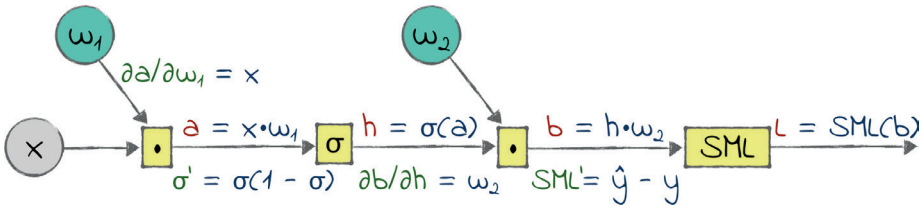
In dieser einen Codezeile stecken jedoch noch zwei weitere Details, die nicht unmittelbar einsichtig sind. Das eine davon ist der Aufruf von `prepend_bias()`. Da wir aber in der Funktion `forward()` eine Biasspalte zu `h` hinzufügen, müssen wir das auch bei der Backpropagation tun.

Der zweite verwunderliche Vorgang ist die Division durch `X.shape[0]` am Ende. Dieser Term gibt die Anzahl der Zeilen in `X` an, also die Anzahl der Beispiele in den Trainingsdaten. Die Matrizenmultiplikation liefert uns den akkumulierten Gradienten über alle Beispiele. Da wir aber am *durchschnittlichen* Gradienten interessiert sind, müssen wir das Ergebnis dieser Multiplikation durch die Größe des Trainingsdatensatzes teilen.

Diese eine Codezeile hatte es in sich! Weiter geht es mit dem Gradienten von w_1.

Den Gradienten von w_1 berechnen

Zur Berechnung von $\partial L/\partial w_1$ benötigen wir die folgenden lokalen Gradienten:



Mit der Kettenregel ergibt sich Folgendes:

$$\frac{\partial L}{\partial w_1} = \mathrm{SML}' \cdot \frac{\partial b}{\partial h} \cdot \sigma' \cdot \frac{\partial a}{\partial w_1} = (\hat{y} - y) \cdot w_2 \cdot \sigma \cdot (1 - \sigma) \cdot x$$

Da diese Formel ziemlich lang ist, habe ich sie auf zwei Codezeilen und eine Hilfsfunktion aufgeteilt:

```
def sigmoid_gradient(sigmoid):
    return np.multiply(sigmoid, (1 - sigmoid))

a_gradient = np.matmul(y_hat - Y, w2[1:].T) * sigmoid_gradient(h)
w1_gradient = np.matmul(prepend_bias(X).T, a_gradient) / X.shape[0]
```

Die vorletzte Zeile berechnet `(ŷ - y) · w₂ · σ`. Dabei sind einige Feinheiten zu beachten. Erstens verwenden wir `h` diesmal im ursprünglichen Zustand, also ohne Biasspalte. Diese Spalte wird erst *nach* der Berechnung von `h` hinzugefügt, weshalb deren Gradient nicht so weit zurückläuft wie der von w_1. Anders ausgedrückt: Diese Spalte hat keinen Einfluss auf $\partial L/\partial w_1$.

Hier heißt es aber aufgepasst: Wenn wir die erste Spalte von `h` entfernen, müssen wir auch die zugehörigen Gewichte loswerden, also die erste *Zeile* von `w2`, da bei der Matrizenmultiplikation Spalten mit Zeilen multipliziert werden. Daher wird im Code `w2[1:]` verwendet, was »`w2` ohne die erste Spalte« bedeutet.

Ich muss gestehen, dass ich diesen Code beim ersten Versuch falsch geschrieben habe. Als sich NumPy wegen unpassender Matrixdimensionen beschwerte, ging ich die Dimensionen aller beteiligten Matrizen durch, um den Fehler zu beheben. Das war eine ziemlich mühselige Erfahrung.

In der betrachteten Codezeile geht es mit `sigmoid_gradient()` weiter. Diese Hilfsfunktion berechnet den Gradienten der Sigmoide aus deren Ausgabe. Da wir bereits wissen, dass diese Ausgabe die verdeckte Schicht `h` ist, können wir einfach `sigmoid_gradient(h)` aufrufen.

In der letzten Codezeile bringen wir die Aufgabe dann zum Abschluss, indem wir das bisherige Zwischenergebnis mit `X` multiplizieren. Dabei sind die gleichen Tricks erforderlich wie bei der Berechnung des Gradienten von w_2: Wir müssen die Reihenfolge der Operanden vertauschen, eine der Matrizen transponieren, die Biasspalte wie bei der Forward-Propagation vorn an `X` anhängen und den Durchschnitt des Gradienten über alle Trainingsbeispiele bilden.

Ich hatte Sie vorgewarnt, dass dieser Code ziemlich knifflig sein würde, aber jetzt sind wir auch damit fertig. Nun können wir die Einzelteile zusammenfügen.

Die Funktion back() erstellen

Den Code für die Backpropagation können wir nun in eine bequeme dreizeilige Funktion einbinden:

```
def back(X, Y, y_hat, w2, h):
    w2_gradient = np.matmul(prepend_bias(h).T, (y_hat - Y)) / X.shape[0]
    w1_gradient = np.matmul(prepend_bias(X).T, np.matmul(y_hat - Y, w2[1:].T)
                            * sigmoid_gradient(h)) / X.shape[0]
    return (w1_gradient, w2_gradient)
```

Sie können jetzt aufatmen, denn damit haben wir bereits den schwierigsten Code im ganzen Buch geschrieben.

Mit der Funktion `back()` sind wir dem Ziel eines voll funktionsfähigen Algorithmus für das Gradientenverfahren in neuronalen Netzen schon einen großen Schritt näher gekommen. Wir müssen uns nur noch um eine letzte Einzelheit kümmern: Bevor wir die Gewichte optimieren können, müssen wir sie zunächst initialisieren. Diese Aufgabe ist einen eigenen Abschnitt wert.

Das Problem der lokalen Minima

Ebenso wie Perzeptrone werden neuronale Netze mithilfe des Gradientenverfahrens trainiert. Allerdings gibt es dabei einen bedeutenden Unterschied. Bei Perzeptronen wählen wir die Verlustfunktion sorgfältig aus, um eine konvexe Verlustoberfläche zu bekommen, also eine ohne lokale Minima. Bei einem neuronalen Netz können wir das aber nicht garantieren. Die Verlustfunktion eines neuronalen Netzes ist im Allgemeinen nicht konvex und kann daher lokale Minima enthalten.

In »*Probleme beim Gradientenverfahren*« auf *Seite 53* haben Sie gelernt, dass das Gradientenverfahren in einem lokalen Minimum stecken bleiben kann. In der Praxis heißt das, dass Sie nicht sicher sein können, tatsächlich den minimalen Verlust erreicht zu haben, selbst wenn Sie das neuronale Netz sehr lange trainieren.

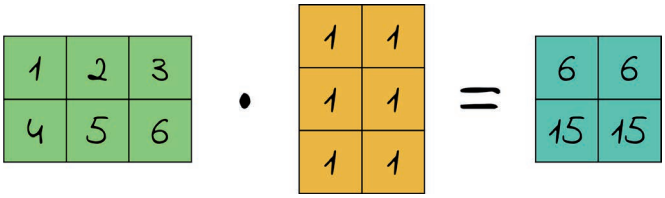
Wie gehen wir also in neuronalen Netzen mit lokalen Minima um? Die gute Nachricht lautet, dass Sie meistens gar nichts tun müssen. Forscher haben bewiesen, dass problematische lokale Minima in der Praxis ziemlich selten vorkommen. Außerdem wird heutzutage auch nur selten das einfache Gradientenverfahren genutzt, sondern Varianten, die lokalen Minima geschickt ausweichen können. In Kapitel 18, »*Tiefe Netze zähmen*«, werden wir uns einige davon ansehen. Während die Verlustfunktionen neuronaler Netze durchaus lokale Minima aufweisen, sorgt das in der Praxis daher nur selten für ernsthafte Probleme.

Die Gewichte initialisieren

Bei unserem Perzeptron in Teil I waren wir mit der Initialisierung der Gewichte schnell fertig: Wir haben sie einfach auf 0 gesetzt. Bei einem neuronalen Netz lauert bei der Initialisierung der Gewichte dagegen eine schwer auszumachende Fallgrube. Sehen wir uns an, was es damit auf sich hat und wie wir sie umgehen können.

Gefährliche Symmetrie

Die wichtigste Regel lautet: Initialisieren Sie niemals alle Gewichte eines neuronalen Netzes mit demselben Wert! Der Grund dafür ist die Matrizenmultiplikation in dem Netz. Betrachten Sie zum Beispiel die folgende Operation:



Während die erste Matrix lauter unterschiedliche Zahlen enthält, hat die resultierende Matrix zwei identische Spalten – weil die zweite Matrix völlig gleichförmig ist. Allgemein gilt: Wenn bei der zweiten Matrix in einer Multiplikation alle Zellen denselben Wert aufweisen, dann sind in jeder Zeile des Ergebnisses alle Werte identisch.

Stellen Sie sich vor, die erste und die zweite Matrix in dem obigen Beispiel wären `X` und `w1`, also die Eingaben und die Gewichte der ersten Schicht eines neuronalen Netzes. Im Anschluss an die Multiplikation wird die resultierende Matrix an die Sigmoide übergeben, um die verdeckte Schicht `h` zu bilden. Nun stehen in allen Zeilen von `h` aber lauter identische Werte, weshalb alle verdeckten Knoten im Netzwerk denselben Wert haben. Wenn wir also alle Gewichte mit demselben Wert initialisieren, verhält sich unser Netz, als ob es nur einen einzigen verdeckten Knoten hätte.

Wenn wir nun `w2` ebenso gleichförmig initialisiert, setzt sich dieser symmetrieerhaltende Effekt in der zweiten Schicht und sogar bei der Backpropagation fort. In der praktischen Übung am Ende dieses Kapitels können Sie selbst ausprobieren, wie einheitliche Gewichte dazu führen, dass sich das Netz wie eines mit einem einzigen Knoten verhält. Wie Sie sich denken können, ist ein solches Netz nicht sehr genau. Schließlich haben wir die ganzen verdeckten Knoten nicht ohne Grund angelegt.